2009-05-29

An Internal Representation for Adaptive Online Parallelization

Koy D. Rehme
Brigham Young University - Provo

AN INTERNAL REPRESENTATION FOR ADAPTIVE ONLINE PARALLELIZATION

by

Koy D. Rehme

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

August 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Koy D. Rehme

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	David A. Penry, Chair
Date	Michael J. Wirthlin
Date	James K. Archibald

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Koy D. Rehme in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

David A. Penry
Chair, Graduate Committee

Accepted for the Department

Michael J. Wirthlin
Graduate Coordinator

Accepted for the College

Alan R. Parkinson
Dean, Ira A. Fulton College of
Engineering and Technology

ABSTRACT

AN INTERNAL REPRESENTATION FOR ADAPTIVE ONLINE PARALLELIZATION

Koy D. Rehme

Department of Electrical and Computer Engineering

Master of Science

Future computer processors may have tens or hundreds of cores, increasing the need for efficient parallel programming models. The nature of multicore processors will present applications with the challenge of diversity: a variety of operating environments, architectures, and data will be available and the compiler will have no foreknowledge of the environment until run time. Adaptive Online Parallelization (ADOPAR) is a unifying framework that attempts to overcome diversity by separating discovery and packaging of parallelism. Scheduling for execution may then occur at run time when diversity may best be resolved.

This work presents a compact representation of parallelism based on the task graph programming model, tailored especially for ADOPAR and for regular and irregular parallel computations. Task graphs can be unmanageably large for fine-grained parallelism. Rather than representing each task individually, similar tasks are grouped into *task descriptors*. From these, a *task descriptor graph*, with *relationship descriptors* forming the edges of the graph, may be represented. While even highly irregular

computations often have structure, previous representations have chosen to restrict what can be easily represented, thus limiting full exploitation by the back end. Therefore, in this work, task and relationship descriptors have been endowed with *instantiation functions* (methods of descriptors that act as factories) so the front end may have a full range of expression when describing the task graph. The representation uses descriptors to express a full range of regular and irregular computations in a very flexible and compact manner.

The representation also allows for dynamic optimization and transformation, which assists ADOPAR in its goal of overcoming various forms of diversity. We have successfully implemented this representation using new compiler intrinsics, allow ADOPAR schedulers to operate on the described task graph for parallel execution, and demonstrate the low code size overhead and the necessity for native schedulers.

ACKNOWLEDGMENTS

Any work of sufficient size requires the help of many people. This thesis is no different. Many hours have been spent researching, programming, and writing, with each hour supported by someone else. My thanks go to all friends, family, and coworkers for their assistance in getting this thesis done. My name may go under the title, but each of theirs deserves to be there as well.

I transferred to BYU without knowing what the future would hold for me. Without prior contacts with professors, I needed some way of starting up my research. Dr. Penry, being a new professor here, made for a perfect match: he needed students for his research, I needed a advisor for mine. I have expanded my knowledge under his guidance, and I hope this thesis shows that.

I wondered if I would ever finish this thesis, especially after each of my four computer failures. I am very thankful for backups ... and for the loaner from Dawn and Gary Goldwasser that got me through to the end.

And most of all I want to thank my wife Becky. She made sure there was always a carrot to motivate me and didn't spare the whip when gentle reminders failed. She even served as an editor, receiving but a peanut in return. Without her encouragement I would still be working on the first chapter.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advances in the fields of processor architecture, compiler research, and process development have contributed to exponential performance improvements of the microprocessor. The resulting performance improvements strive to satisfy the ever-growing processing requirements of consumers, industry, and researchers. Although one cannot reasonably expect advances to continue indefinitely, we have yet to see the end of Moore's Law [1]. Transistor manufacturing improvements will continue to provide more raw processing capabilities in the future.

Both research and commercial products have transitioned from using additional transistors to improve performance of a single processor to a "step and repeat" replication of many processors on a single die. These Chip Multi-Processors (CMPs) [2] give a significant potential improvement for software. Unfortunately, software cannot necessarily take advantage of the extra processing power without being rewritten or at least recompiled. Even then, new techniques need to be developed to utilize all cores to full potential.

Irregular computations, such as those found in sparse linear algebra and molecular dynamics, are especially difficult for software. The irregularity of these computations impede the static compiler analyses normally used to optimize parallelization; however, optimizations are possible once the dynamic nature of the system is resolved and communication patterns are determined. Current compiler research is developing dynamic frameworks which can optimize at run time as long as the parallelism is represented appropriately.

Compilers and optimizers rely on an Internal Representation (IR), or the specific method for representing the structure of a program. The representation deter-

mines how optimizations and analyses are performed and impacts the performance of the compiler and final executable. Many IRs are possible, depending on the goal of the system: Chapter 2 describes several that are currently used. The *task graph* representation for parallelism provides a mechanism for describing irregular parallel computations and communications. As the task size decreases and the number of tasks increases, more parallelism can be found in the graph; however, smaller task sizes also dramatically increase the amount of computation needed to operate on the graph and increase the final run-time overhead (given the larger number of tasks). While algorithm researchers are trying to reduce the complexity of scheduling algorithms, it is not uncommon to see algorithms from $O(n \log n)$ in the best case to $O(2^n)$ and worse.

Working within a dynamic environment may relieve the difficulties of creating parallel code while providing specialized optimization opportunities for the application. Dynamic optimization of serial programs may use information only available during execution; the hope is that parallel programs may receive the same benefit. Since the operating environment may only be known when executing the program, these same optimizations may be specialized for the immediate situation. As such it is important to include as much information as possible into the parallelism representation so it can be accessed at run time.

This work presents two modifications of the task graph representation specific to a dynamic environment. The first is the addition of highly generic *task descriptors*, a conceptual "factory" for tasks. Task relationships are formed with *relationship descriptors*, which form the edges of the task graph. The second modification is the concept of *instantiation functions*, where control of task and relationship descriptors is governed by a section of code which describes the tasks procedurally. Instantiation functions provide flexibility to represent irregular computations with arbitrary patterns while also providing a mechanism to reduce the complexity of compiler analysis and scheduling. Embedding the algorithmic descriptions of tasks and relationships into instantiation functions can improve performance by increasing the effectiveness of task scheduling. Combined, descriptors and instantiation functions provide a mech-

anism to concisely represent the structure and communication patterns of tasks of varying granularity while still operating with overhead appropriate to a dynamic framework.

1.1 Motivation

1.1.1 The Manycore Era

Trends in microprocessor design and production point to a new era where processors have increasing numbers of cores rather than improved single-core designs. This trend has been prompted by the ever-increasing transistor budgets available for additional logic and the matching challenges to fully profit from it. Practical limits have been reached for clock speeds and exploitable instruction-level parallelism. Signal propagation time around the die creates timing limitations. The density of transistors increases the power use and heat dissipation required. CMPs offer the possibility of significantly improving performance potential, as long as the software can adapt to this new era.

Indeed, software presents the greatest challenge of the manycore era: how to fulfill user expectations for performance improvements while managing the variety of architectures and environments that are sure to exist. We already see great variations in current consumer architectures, from low-speed single-core computers to high-performance servers with 32 processing cores immediately available to a program, with various core designs. Current x86 processors are being shipped with support for up to 8 threads per die, and the SPARC T2 can handle up to 64 threads on a single chip. Eventually industry may even produce CMPs with hundreds (some would speculate thousands) of processors.

This variety is sure to increase as the range of cores widens, the design of cores improves, and the possibility of heterogeneity becomes a reality [3]. Every design choice, from the memory hierarchy to the chip layout to the types of processing elements in any particular machine has impacts on parallel performance, and it will be important for programs to adapt to this diversity in order to have full performance.

Optimizing for the wrong architecture will either under-utilize the available resources or over-estimate the machine's capabilities.

Architectural diversity is not the only variable that can affect parallel program optimization. In fact, any number of environmental factors, even on the same machine but between separate executions of the same program, may also affect performance. For example, desktop computers see a variety of applications attempting to run concurrently, each with different requirements and capabilities for parallel execution. The program itself may present run-time dependencies that affect parallelism, such as an input data set which places constraints on parallel work and communication. Thus, the resources that may be available to a program may vary between executions or even change on the fly. An application must adapt to these constantly changing conditions.

In many cases, proper parallelization is dependent on the data used by the program, creating computation patterns that cannot be determined until run time. The difficulty for a parallelizing system is to account for the many different situations and environments: it may be possible to optimize for a couple specific situations, but the general case is another matter. All of these factors contribute to the difficulty of effectively parallelizing the program, the main consequence of failure being reduced performance in the final executable.

1.1.2 Discovery of Parallelism

The most difficult problem for parallelizing compilers – and, for that matter, programmers – is finding the parallelism inherent in an application. Parallelizing compilers are an active area of research, and there have been many research compilers working on discovering parallelism in a program, e.g. [4, 5, 6, 7]. Parallelizing compilers have better success with data-parallel programs and coarse-grain parallelism through inter-procedural analysis than with other types of parallelism.

Parallelism is a property of the code and data of a program. Code regions might execute on different sections of data and therefore run in parallel. The same code may execute on separate data regions for data-parallel and loop-parallel pro-

grams. Separate code regions working on potentially the same data form task-parallel programs. The compiler attempts to prove that code or data can be parallelized and transforms the program appropriately. An important observation is that parallelism is not a property of the architecture or execution environment.

It is not always possible to discover all parallelism at compile time, which may be dependent on the data set or other run-time factors. The compiler can, however, compute the necessary runtime conditions for code to be parallel. Parallelism can be finalized by the run-time system when the conditions can be resolved.

1.1.3 Packaging of Parallelism

Once the parallel properties of an application have been established, the compiler or run-time system must associate data, loop iterations, or tasks with appropriate threads. The result of association is the *packaging* of the parallelism. Packaging of parallelism is highly dependent on the architecture and operating environment: accounting for these differences is key to extracting the full performance from the execution of a program. The process of packaging consists of mapping and scheduling work to the available resources, organizing and optimizing for the architecture and environment.

Discovery and packaging of parallelism are not orthogonal processes: extracting more parallelism has a direct impact on the packaging/scheduling process, and transforming code to find additional parallelism may alter the dependencies that need to be considered when scheduling. In other cases, the relationship is more direct: for example, each iteration of a Monte Carlo [8] algorithm is easily proved to be independent during the discovery process. The packaging is generally simple, but may be complicated by load balancing.

Even algorithms where discovery is simple may have complications at the packaging stage. In general, the task execution time is not known *a priori* and may vary greatly at run time, creating classic problems with load balance. Dynamic load balancing techniques [9] may help in many situations but may be limited by high overhead, especially for fine-grained parallelism. Static packaging systems must rely

on an estimate of the execution time, an estimate which is assumed to be sufficiently accurate for the application in question.

Task dependencies create both communication problems and load balancing issues: not all schedules of tasks are valid, and different schedules have very different effects on the execution time and load balancing. Communication costs make packaging decisions especially important.

1.1.4 The Run-time Advantage

The discovery of parallelism tends to be an expensive process and must be done at compile time to reduce overhead. However, diversity cannot be managed at compile time. Serial programs have dealt with diversity by deferring some optimization to run time through hardware techniques such as out of order execution. Parallel programs require similar run-time techniques to deal with diversity as run-time dependencies preclude full compile-time analysis. We have proposed [10] a solution similar to the out-of-order technique described above: perform general static optimizations and create a dynamic optimization scheme to account for diversity.

Separating discovery and packaging in the parallelism process allows the high overhead of discovery to be managed offline. The run-time system, freed from this overhead, is free to optimize the parallelism using the wide variety of information available.

1.2 The ADOPAR Vision

The technique of separating the packaging of parallelism from its discovery to overcome manycore diversity is what we call Adaptive Online Parallelization (ADOPAR) [10, 11]. As its name implies, ADOPAR adapts the parallelism of an application to diversity through online (run-time) techniques. The BYU Architecture Research, Design, and Description (BARDD) research group is creating a research infrastructure to implement its principles of diversity-independent discovery and diversity-dependent packaging. BARDD is currently researching the techniques required to create effective packaging.

ADOPAR stands in stark contrast to previous methods for parallelizing software. Its process may best be illustrated by examining a program's life cycle.

1. Software is developed by a programmer. The focus at this stage is software behavior and description of parallelism, not the packaging or optimization for a particular system. This simplifies the work of a software developer, whose job is to describe the parallelism, not package and optimize it.

2. The compiler extracts information about parallelism from the programming model and discovers new implicit parallelism.

3. At run time, schedulers present in the run-time system use the results of discovery to package parallelism for best performance in the current operating environment. Generally, this consists of assigning work to processors and accounting for the capabilities and limitations of the available resources. Any program information not available at compile-time (such as the operating data) is also available for packaging optimization.

4. The schedule is executed. For some programs, loops and other forms of reuse allow the packaging overhead to be amortized.

5. The packagers may need to adapt when conditions change, such as an increase or reduction in the number of threads. Information from profiling may yield better performance through repackaging.

In this process, the ADOPAR infrastructure must interact with the operating system. We envision a process where the run-time system and operating system negotiate for resources. The run-time system requests resources according to its needs while the operating system presents what is available. The traditional process scheduling role of the operating system is transferred to the run-time system so it can manage its own threads (as in [12] and [13]).

By splitting the discovery and packaging phases, ADOPAR achieves the full benefits of parallelism discovery and run-time information. But to use run-time information, the ADOPAR infrastructure requires a dynamic compilation environment.

Using a Just-In-Time compiler (JIT) framework offers the opportunity to optimize the code given the task schedule, the specific execution environment, and the run-time data dependencies. Having an on-the-fly compiler also allows for the scheduler to directly work with code primitives such as basic blocks, loops, and functions, as well as run various code analyses as part of the scheduling process. This work uses the Low Level Virtual Machine (LLVM) [14] infrastructure, a compiler framework that includes the ability to add analyses and transformations, create new compiler intrinsics, and optimize and generate machine code for several instruction set architectures at runtime.

1.3 Parallelism Representation

One key element in ADOPAR is a mechanism to communicate the results of discovery to packagers. Since discovery occurs at compile time and packaging occurs at run time, a persistent representation of parallelism is needed. This work describes a representation suitable for an ADOPAR infrastructure that can act as a bridge between discovery and packaging. The goal of the framework's representation make sufficiently detailed information available to the packagers and present it conveniently. More detailed information allows the scheduler to provide a more efficient mapping of tasks to processors; a convenient format reduces the analysis overhead of the program while compiling, scheduling, and executing. ADOPAR requires the following from its parallelism representation:

1. Irregularity

   ADOPAR's specific goal is to parallelize programs with irregular features; the representation needs the ability to conveniently express many types of irregularity. Some applications are completely irregular (random or nearly so) while others may have a complex pattern. The representation must handle either case conveniently and compactly.

2. Regularity

   While ADOPAR attempts to optimize irregular applications, some programs have a mixture of regular and irregular properties. A representation that can handle irregular applications (possibly by enumerating each task) will be able to work with regular applications as well, although the memory usage and processing time may not be optimal. Instead, a direct means of representing regularity in a program may reduce processing time significantly.

3. Communication

   Almost every parallel application requires communication between processors during the computation. Often, the communication time constitutes a significant portion of the total execution time. One of ADOPAR's goals is to consider the communication costs and optimize the scheduling appropriately. The representation should effectively represent communication regardless of whether the application is regular or irregular.

4. Low Run-time Overhead

   ADOPAR may be used with fine-grained task models, which leads to increased interaction with the representation. Lowering the overhead of the representation is critical when dealing with many tasks.

We refer to the parallelism representation as an Internal Representation (IR), in analogy to the internal representations used by compiler frameworks, since the representation is internal to the infrastructure, communicating between its separate components. The IR presented in this work will be evaluated according to the requirements of ADOPAR.

1.4 Contributions and Objectives

This work contributes a new way of viewing internal representations for parallelism. While previous methods have been constrained by the programming model or

arbitrary schemes, this new representation is very flexible and expressive. This representation forms the connective infrastructure to bridge the gap between the static discovery and the dynamic packaging that comprise the ADOPAR infrastructure. The exposed interface allows for a variety of programming models to integrate with the system while still using their native format within ADOPAR IR.

The following mechanisms comprise the IR:

Task Descriptors Similar tasks are bundled together and represented as a single task descriptor. Using descriptors potentially simplifies static analysis of parallelism and can represent both regular and irregular computations. On the surface, task descriptors act as task "factories", but might be more accurately viewed as a description of a group of tasks.

Relationship Descriptors Because tasks are condensed in the representation, the relationships between them must also be represented in a similar manner. Both regular and irregular relationships may be represented.

Instantiation Functions The key contribution that makes task and relationship descriptors possible is an algorithmic representation of parallelism and relationships as a method of the descriptors. In other words, instead of specifying task parameters directly, the front end provides the computation to specify these parameters. In the case of constant parameters, there is little difference; however, irregular and complex properties may be expressed algorithmically. Having the capability to express complex patterns algorithmically is especially important for irregular computations: the computations may have a pattern that may not be possible to define with any one regular system, but an arbitrary sequence of code may describe it. Instantiation functions may be analyzed to find regular patterns at compile time or executed at run time to instantiate the tasks themselves.

This IR presents flexible and descriptive information about parallelism to an ADOPAR infrastructure. Instantiation functions allow flexibility that stands in stark

contrast with the strict constraints used by other representations. Overall, the representation adds a significant amount of additional parallelism information while increasing the code size by less than 20% for even fine-grained parallelism and negligible additional compilation time. We will show that while it is possible to produce a traditional task graph for schedulers to use, it will be important that schedulers operate on the native representation to minimize overhead.

1.5 Outline

The rest of this work is organized as follows:

Chapter 2 presents related work in the area of parallelism representations and other background information. Various compilers and run-time environments for parallel systems use an internal representation appropriate for the techniques used by the same. These representations are compared to the requirements of ADOPAR.

Chapter 3 describes ADOPAR's representation. The description includes several examples of task graphs that a front end might create. Also, various optimizations and operations that may be performed on that task graph are discussed.

Chapter 4 details the implementation of the representation, along with continuations of the examples in Chapter 3. This chapter also describes the programming API and the back-end programming model of the ADOPAR IR.

Chapter 5 evaluates the representation by the overhead that it incurs while building and operating on the task graph. It also shows how well this new representation integrates with various front ends.

Finally, Chapter 6 presents future work that can be performed in this area of research and some final thoughts on this thesis.

Chapter 2

Background and Related Work

The proliferation of multicore processors has stimulated continuing and extensive research in optimizing compilers and other methods of increasing parallelism. Much of this research has required development of a method to express parallelism. In some cases, parallelism is presented as part of the programming paradigm; in others it is defined as the compiler's or tool's internal representation for the parallelism.

This chapter summarizes the various paradigms that have been developed and the internal representations used and presented in literature. The remainder of this chapter is organized as follows: first, scheduling and execution is examined in context of parallel programming. Next, an overview of relevant programming paradigms is presented, followed by a discussion on how data-parallel, loop-parallel, and task-parallel programs are represented in other systems.

2.1 Scheduling and Execution

There are several methods for scheduling and executing the parallelism of a program. *Static scheduling* is the process of taking tasks, loop iterations, or data (depending on the type of parallelism) and assigning them to a processing element all at once. While generally performed at compile time, static scheduling may also be a single step before execution, introducing a one-time execution overhead. Static scheduling is especially effective for regular programs but can be inefficient if the parallelism has a dynamic or irregular nature.

Other applications simply cannot be scheduled statically. For example, some irregular applications delay task creation until after other tasks have been executed. When tasks are created dynamically, a common solution is to have a "bin" of ready

tasks, from which work is assigned to available processors. *Dynamic scheduling* is a mechanism which operates on the collection of tasks and makes task assignments during the execution of a program. Dynamic and irregular programs tend to match the paradigm of dynamic scheduling (where static scheduling would be inappropriate or inefficient), but the run-time assignments introduce critical execution overhead that must be avoided as much as possible.

A compromise for those cases where static scheduling may be appropriate if the dynamic nature can be resolved is to schedule at run time but before the actual execution. Being a form of static scheduling, it involves examining the parallelism first and then making assignments. The Inspector-Executor (IE) model, first proposed by Saltz [15], has been proposed as a method of dealing with irregular communication and data layout in many applications, but has found support in many other research areas as well. The CHAOS/PARTI compiler framework [5] contains an implementation of the IE model.

The core concept is to separate execution into two phases: an *inspector* and an *executor*. The inspector preprocesses the data to determine the necessary communication pattern, passing this information on to an executor which does the actual computations. The preprocessing may also consist of optimizing the execution order for cache effects, etc. Separating the inspection of the loops from the execution gives opportunities for scheduling and optimization. Any overhead incurred may be mitigated by repeatedly reusing the results of the inspector phase.

IE has been used for sparse linear algebra. Sparse matrix execution and other similar computational problems are a case where the indexes to an array may be expressed as functions (e.g. $a[f()] = g(b[h()])$). The basic task is defined by a loop (or loop nest) that operates on the matrix, but the number of tasks and their dependencies are not known until the indexes are calculated. Thus, the IE model breaks execution into two phases: an inspector that calculates the loop indexes and then an executor that then uses those indexes. The model usually operates as a loop, the iterations of which are scheduled to threads in a simple fashion.

Although reusing the results of the inspection phase is helpful in amortizing overhead costs, it is still important to reduce overhead as much as possible. To that end, the original IE model uses a simple *wavefront* scheduling process, a scheme similar to As Soon As Possible (ASAP) schedulers. In wavefront scheduling, the tasks to schedule are represented as a Directed Acyclic Graph (DAG). All nodes without predecessors are scheduled in parallel as the first wavefront. These nodes are removed from the graph and the new set of nodes without predecessors is scheduled in parallel as the second wavefront. This procedure continues until all nodes are scheduled. Computing the wavefronts is a low-overhead task for the run-time environment. Creating this schedule is simple and may itself be parallelized with low overhead; however, the schedule produced is not optimal, especially for fine-grained tasks.

IE can be used for more than just parallel programming: the same concept can be applied to serial programming where data access needs to be analyzed. The result of the inspector is a sequence of memory references that could then be reorganized to maximize cache efficiency. The abstract concept of inspecting the data access patterns (as a preprocessing step) and determining a work schedule is a generic concept that may be applied in many situations. Many of the programming representations presented in the following sections use the IE concept by providing a mechanism for partitioning parallelism and then reusing this partitioning repeatedly during the actual execution. This is also the paradigm used by ADOPAR; this work presents the communication mechanism between ADOPAR's inspector and executor concepts.

2.2 Parallel Programming Paradigms

There are many paradigms for programming parallel systems, with more being developed constantly. Each attempts to simplify the specification of parallelism, increase the amount of parallel work, decrease overhead, or optimize for specific hardware. These paradigms may be expressed as language constructs, library calls, or compiler directives, often placed on the top of traditional, serial languages.

Some newer programming languages have the built-in ability to perform parallel operations. Single-assignment languages, functional languages, and process-based

languages all fall under this category. Traditional languages, however, form the mainstay of benchmarks and legacy code. Many developers would prefer to parallelize existing code or at least use a language with similar constructs. For this reason, High Performance FORTRAN (HPF), FORTRAN 95, and others were made as extensions of FORTRAN. C and C++ (as well as FORTRAN) have been extended with OpenMP and various libraries to transform them into parallel languages.

For the purposes of this work, we define *parallelism* as the properties of program segments that allow them to run concurrently. Shared data between program segments limit, prevent, or otherwise constrain parallel execution.

2.2.1 Types of Parallelism

Most parallel programming paradigms fall into the following categories:

Data-Parallel (Section 2.3) This type of parallelism is defined by the data being operated on. In general, a thread "owns" a portion of the data and operates on it. Irregular data divisions create imbalance in the load balance and communication in the system. Parallelism is intrinsically defined by the data division.

Loop-Parallel (Section 2.4) For many applications, loop iterations can be independently executed in parallel. A compiler may have to transform the loop and prove independence to parallelize programs. Loop-parallel programs may be irregular if the amount of work in each iteration varies or if not all loop indexes should be performed. Parallelism is defined by the division of loop iterations.

Task-Parallel (Section 2.5) A third class of parallel programs are defined by tasks: portions of the program that may be executed simultaneously and, depending on the model, atomically. There are generally more tasks than threads or processors in a system: tasks must be scheduled or mapped to a thread or processor at the appropriate times while not violating dependencies. Scheduling often involves some sort of *task graph* or similar representation. Task graphs show irregular tasks and communication directly, and can usually represent an equivalent form of both the data-parallel and loop-parallel paradigms by defining

loop iterations and portions of the data to be the individual tasks. Parallelism is defined by the task graph.

2.3 Data-Parallel Representations

Data-parallel programming recognizes that the same set of operations may be applied to a block of data. The block can then be divided between processors for parallel operation. Data-parallel programs may be specified by:

1. control-flow of operations on a data set,
2. partitioning of the data to processors, and
3. communication and synchronization.

Kernel Lattice Parallelism (KeLP) [16, 17] is a set of C++ libraries designed to ease the specification of irregular block-wise data layouts, where sections of matrices are segmented into individual blocks to be assigned to different processors. Blocks are partitioned among processors through geometric operations. Communication is specified as sequences of data motion between these blocks. The motion may be set up and then reused throughout execution of the program. A simple example of KeLP is a simple dimension-wise decomposition of the data into rectangular sections. Each block then receives "ghost cells" around the border, representing the communicated data between processors. The data motion for each iteration of the program copies the data in these cells between processors.

KeLP succeeds in representing regular applications and some irregular applications, although irregularity must be described in blocks. Some very irregular programs see a high overhead from the block-wise decomposition. In addition, KeLP is a strictly runtime-only environment, eliminating compile-time analysis of the representation.

The ParaScope compiler tool suite has a special representation for interprocedural side-effect analysis using Regular Section Descriptors (RSDs) [6]. Large-granularity parallelism can only be found by determining that code sections are independent of others, especially code that operates on an array. The compiler may

separate code sections for execution if the compiler can prove that different code blocks or iterations of a loop operate on independent sections of an array. ParaScope works on FORTRAN input and creates parallelism by transforming sequential DO loops into parallel DO loops.

ParaScope approaches proving independence through regular sections: an analysis of the access patterns across an array. A *regular section* is a commonly used access pattern over a portion (*section*) of an array, such as a column or row. A regular section defines a set of actual data accesses, including the operation performed that data. Set intersection reveals dependencies between two code blocks; set union combines regular sections. There are many possible ways of representing regular sections: some of these methods will be analyzed below as variations on the RSD. The easiest regular section representation is a vector of indexes. This could contain a single element, row, column, or diagonal. Triangular sections and discontinuities have no representation in RSDs. While irregular patterns may require many RSDs, regular patterns potentially have very low overhead. Working with RSDs is mainly a compile-time process but might also be applied in a run-time system.

As a refinement of the RSD, the Data Access Descriptor (DAD) [18] defines three properties for a task's access to a portion of an n-dimensional array. The main contribution is the *simple section*, a simple boundary around the accessed portion of an array. By limiting the boundaries to 45° angles a maximum of $2n^2$ boundaries are available to describe sections, sufficient for most standard applications. Constant-time operations are also available to define the intersection and union of simple sections, with the complication that the union may not result in a completely convex region. As a solution, the conservative approach expands the region so it satisfies simple boundary conditions. In addition, a *traversal order* and *reference template* (the definitions for how the code accesses descriptors) are provided for each DAD as it is defined for an array. DADs have many of the same benefits and drawbacks as RSDs, but also have the ability to describe other shapes of regular sections and attach additional access patterns for communication.

Processor Tagged Descriptors (PTDs) [19] further extend the RSD and the DAD. The bounded area is more precisely defined, allowing for concave sections and other complex shapes, while requiring more processing to perform arithmetic on these sections. Instead of bounding an area by known lines, the boundaries are parameterized by the processor number and a linear equation. These restrictions still allow for simple mathematical operations to manipulate the descriptors. The parameterizations automatically partition the data among processors, the number of which may vary from run to run with the same binary and descriptors. As with RSDs and DADs, PTDs are mainly used for analysis by the compiler, rather than an inherent description of the parallelization. However, the results may be used later as the descriptors reveal the end data partitioning. Of the three descriptors (RSD, DAD, and PTD), the last has the greatest flexibility for describing irregular accesses.

A different approach is used in Pilar's internal representation, Communication Pattern Internal Representation (CPIR) [20, 21]. Many applications show a mixture of regular and irregular communication patterns. Three basic constructions are used by the Pilar compiler to represent data access patterns: *intervals*, *enumerations* and *cyclics*. Intervals are similar to the descriptors previously described: a set of bounds on the index ranges for data access. CPIR intervals are actually a collection of such ranges. Enumerations allow for completely irregular access patterns and are a simple list of all individual accesses. Cyclics provide a specific access pattern: for some applications, access is regular but an interval is not sufficient. Instead, the list may be strided like the diagonals of a matrix. Such a list may be inefficiently represented as an enumerated list, but cyclics allow for a smaller memory footprint and faster set operations. An access schedule (meaning, the ordering of element access within the construct) is also created in addition to the data pattern. This schedule is then used to better analyze the relationships between different patterns. Since CPIR is a message-passing system, the schedule consists of the sequence of messages to communicate CPIR primitives, which are then translated to the actual local addresses.

Pilar has the ability to describe very regular and irregular systems (a feature important to ADOPAR) and some capability to describe simple patterns when nec-

essary. The pattern is limited to simple, strided array accesses, which results in lower overhead for those applications that it describes well.

These various versions of regular sections are a useful concept for ADOPAR, but are constrained to specific formats for these sections. They are also constrained to array accesses, a limitation that ADOPAR must avoid. The ability to describe various patterns, however, is a concept that is useful in a more advanced parallelism representation.

2.4 Loop-Parallel Representations

Most automatic parallelizing compilers attempt to find coarse-grained loop parallelism. Loop parallelism is easy to express: simply mark loops as being parallel and map individual loop iterations to threads. Representations for loop-parallel programs generally have attributes such as:

1. special parallel loops (such as DOALL or DOACROSS),
2. mapping of iterations to processors, and
3. communication and synchronization.

The Polaris compiler's Internal Representation (IR) [7] has a 1-1 relationship with basic FORTRAN constructs using a high-level format. Expressions, statements, and symbol tables are basic classes. Parallel loops, after appropriate analysis, become doall loops, where each iteration may be executed in parallel. Variables may be declared as private to the loop.

Polaris is similar to the OpenMP [22] standard. OpenMP is a set of compiler directives for C and FORTRAN programs, including automatic partitioning of loops and parallel sections to processors, synchronization through barriers, etc. The parallelism described by OpenMP is implicit, specifying which variables are private and which loops may be executed in parallel. Some hints as to scheduling may also be specified, but such control is limited.

In both cases, irregular applications are not well-supported: iterations that do nothing are generally assigned to processors along with those that perform useful

work. Loop-based representations have few facilities to express communication and dependencies beyond the use of barriers and other simple primitives. These representations are often translated to a simple back end where loop iterations are simply divided between processing elements, reducing the optimizations possible at the back end.

The Stanford University Intermediate Format (SUIF) compiler [4, 23] is an extensive project to increase the ability of compilers to automatically extract parallelism from existing benchmarks. Internally, programs are represented on a relatively high level: loops, conditional statements, and array accesses (as well as other helpful information from the front end) are part of the IR, as well as the more common low-level information used by a compiler. High-level information is in a canonical form for the compiler passes to use. After the high-level passes have completed, the high-level information is transformed into lower-level compare/branch instructions, etc.

The higher-level IR of SUIF allows for easy analysis, but is not strictly meant for use by a unifying back end such as ADOPAR. Higher-level information is useful (as little information is lost when lowering to machine instructions), but needs to be in a form that can be scheduled and executed. In the end, SUIF has many of the same limitations as other loop-parallel representations, which are all limited by the implicit nature of loop parallelism.

2.5 Task-Parallel Representations

Task graphs are another way to represent parallelism. Individual units of work (tasks) are separated and reconnected by their dependencies and ordering constraints. The sequence of dependencies determine which tasks may be executed simultaneously; therefore, the dependencies are critical to determining ordering, scheduling, and partitioning. Analyzing each dependency reveals the constraints to the available parallel

work. The requirements for task-parallel representations are different than those for other types of parallelism. Task-parallel representations require:

1. code and data for task execution,
2. task dependencies, and
3. additional synchronization between tasks.

Johnson proposed a task graph specially designed for dynamic task scheduling [24]. This task graph assigns specific states to each task: unexecuted, executing, finished, and not ready. As tasks become available they are processed by threads. A task structure handles dynamic creation and execution of tasks. Tasks are never explicitly scheduled to reduce overhead or increase locality as some applications have unpredictable and non-repetitive behavior. Use of this task graph requires good heuristics appropriate to the problem in order to properly and efficiently execute tasks.

The nature of this task graph makes both regular and irregular applications easy to represent. Communication and dependencies are represented as well as part of the task graph. The execution overhead, however, may be too large for fine-grained applications: tasks are dynamically assigned, so the overhead is not amortized by reusing information. In addition, little or no static analysis is possible.

The Hierarchical Task Graph (HTG) [25] was made as part of the autoscheduling project headed by Polychronopoulos [26]. As its name implies, the HTG describes a task graph in a hierarchical manner: loops in the control-flow graph are collapsed into a single node (similar to a strongly-connected component). These nodes may be further collapsed to change the granularity of the task graph. The edges in the graph represent the communication and dependencies between tasks. The overall preconditions necessary to execute a task are represented by a set of *execution tags*. Redundant tags may be optimized for run-time efficiency.

The HTG satisfies many of ADOPAR's requirements: it may express irregular applications and communication with minimal task overhead. Indeed, this thesis

borrows several techniques from the HTG and may be considered an extension of it in many ways: the basic task graph and the hierarchical structure being key examples. However, the HTG makes no provisions for regularity as its focus is on dynamic, rather than static, scheduling. ADOPAR, however, requires a more flexible representation for the scheduler to work with and using the HTG with many fine-grained tasks will exaggerate overhead issues; some method of condensing and summarizing tasks is a step to mitigate this problem.

2.6 Summary

Various representations exist to represent parallelism, each with its own implementation infrastructure. Many representations follow the general idea of the IE model: perform some sort of analysis before running the program in some reusable fashion. The needs of the infrastructure tend to determine the properties of its respective representation. For example, the need to analyze dependencies specifies a data-based representation with set operations, while iterating algorithms call for a loop-based representation. Dynamic and irregular problems may require a task-based representation.

ADOPAR's representation has specific requirements: the representation must handle irregular and regular tasks and communication patterns in both compile-time and run-time environments, all with low overhead. Many of the representations used in other parallelizing environments have been presented here, but none completely fulfills the requirements of ADOPAR, especially the requirement that a representation can cross the boundary between a compile-time environment and a run-time environment.

The following chapter describes a new IR for ADOPAR. The IR uses a task graph described in the program's code with compiler intrinsics. The IR handles irregularity and regularity using executable code as part of the representation itself, making a very flexible system within a compact representation that may be analyzed within the compiler or run-time system.

Chapter 3

The ADOPAR Representation

ADOPAR is an execution framework designed to improve parallel performance by overcoming environmental and architectural diversity and by increasing the efficiency of parallelization. ADOPAR has a special focus on applications with irregular communication and data access patterns. To this end, ADOPAR uses the following concepts:

- Fine-Grained Task Model

  Task parallelism is a flexible method to express both regular and irregular parallelism, but most research has been limited to coarse-grained tasks. Large tasks generally have the advantage of fewer inter-task dependencies, simplifying scheduling and reducing overhead. As the task granularity becomes more fine, however, more parallelism may be extracted from the program. The additional parallelism, in turn, increases potential for a better-balanced load and tighter overall schedule. ADOPAR attempts to mitigate the overhead while still extracting sufficient parallelism to justify the extra work of scheduling.

- Front-End Agnostic

  ADOPAR does not rely on any one specific programming paradigm in the front end but may be used as an optimizing back end to many different languages and systems. Any model that can be represented as a set of tasks (which include most data and loop parallel systems) may then use ADOPAR for an optimization and execution environment. No front-end yet exists that produces the ADOPAR Internal Representation (IR), a problem outside the scope of this work.

- Dynamic Execution Environment

  Dynamic code compilation offers many advantages for any system trying to overcome diversity challenges (like the architecture differences and environment changes described in Chapter 1). Run-time compilation may take the architecture diversity into account automatically; changes in the number of available processors or additional overhead in the communication on the system imply that portions of the program may even need recompilation for optimal performance.

- Run-time Optimization System

  The ability to compile code dynamically allows for run-time optimizations. Dynamic optimization is especially important with irregular applications, but also can be effective for serial applications as well [27]. In addition, hardware and operating systems provide performance counters – useful for refining the compilation and schedule of a parallel system. Lastly, run-time optimization can potentially reduce the run-time overhead of fine-grained tasks significantly by intelligently merging the tasks together.

ADOPAR requires an internal representation in order to take advantage of each of these concepts. This representation must include the dependencies between each task and the synchronization necessary to execute correctly. In addition, the representation needs to be as compact as possible while still providing rapid querying for scheduling and analysis.

Note that this representation is significantly different from many of those described in Chapter 2. Rather than being used as a compiler analysis technique, storing the data access patterns for the various function calls, the ADOPAR representation is a way of integrating a scheduling and executing back end with arbitrary front-end analyses to define a set of executable tasks. Integration is accomplished by including code as a part of the representation, as will be described in Section 3.2. The ADOPAR IR makes a bridge between the static discovery representations and the run-time packaging systems.

3.1 Fine-Grained Task Model

The goal of all parallelizing compilers is to increase the amount of parallelism, balance the load between processors, and reduce the amount of communication. Ideally, this goal will be accomplished with as little overhead as possible. Diversity in operating environments complicates each goal. ADOPAR's solution is to separate the process of parallelization into three phases:

Discovery Analyze the program for possible parallelism. ADOPAR allows the use of very fine-grained tasks to maximize the parallelism available in the program. Discovery might be done at compile time or at run time, but compile-time analysis would reduce execution overhead. Discovery is partly the function of a front end, but can also be done by analysis of the IR.

Packaging Map and schedule tasks to threads. Packaging is a well-studied problem (see [28, 29, 30, 31]) but is still an active area of research. Packaging would ideally be done at compile time to reduce overhead, but irregular problems may not be scheduled until the task structure has been resolved at run time.

Execution Perform the scheduled tasks in the specified order. Execution must, of course, be done at run time, and may be augmented by performance monitoring for schedule tuning.

Notice that these phases are very similar to the Inspector-Executor (IE) model: the discovery phase inspects the program for parallelism and the execution phase performs the parallel calculations. An intermediate phase (packaging and scheduling) optimizes the parallelism and reduces communication. Fine-grained tasks, while allowing for additional parallelism, introduce more communication and overhead; the scheduling phase offers a chance to reduce overhead. IE is limited by a very simplistic packaging algorithm (wavefront scheduling) and has only limited static capabilities.

The extra overhead of fine-grained tasks requires that compile-time analysis must be done when possible. Irregular communication patterns will usually prevent scheduling at compile time, but discovery may still be done. Thus a mechanism must

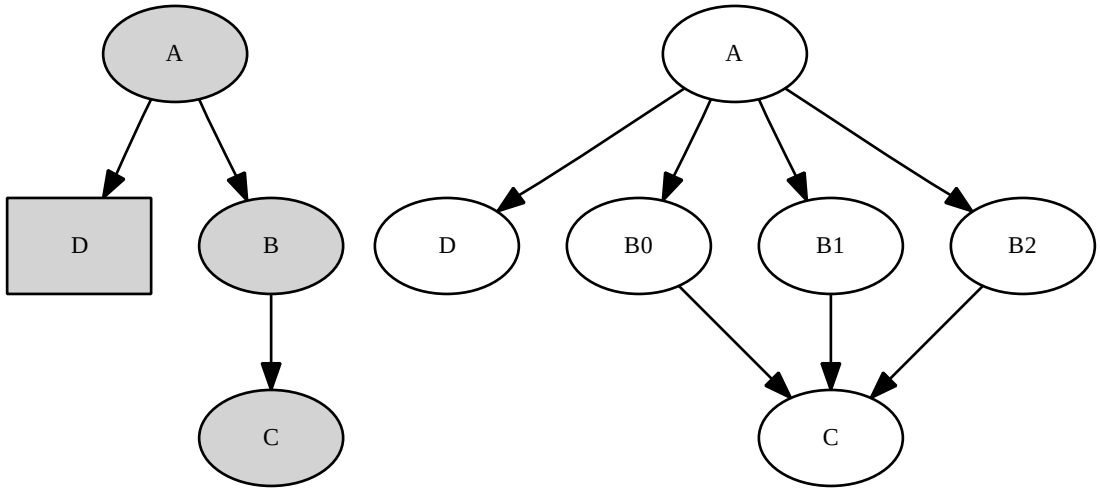
exist to transfer the parallelism between the static environment and the dynamic environment.

3.2 Static Task Graph

The main feature of the ADOPAR IR is a *task graph*, similar to those described in Chapter 2. A task graph offers a simple, well-studied representation and is the underlying data structure used by the ADOPAR scheduling system. A goal of the IR is to work with a variety of programming paradigms; task graphs offer a universal mechanism to represent many types of parallelism.

The underlying data structure is similar to the Hierarchical Task Graph (HTG) but with the addition of highly generic *task descriptors* and *relationship descriptors*: nodes and edges in the task graph that represent an unspecified number of tasks, subtasks, and relationships. Nodes often take the place of loop iterations, where each iteration may be considered a task. Using these nodes, a front end may represent a large number of similar tasks in a very compact form. Figure 3.1 demonstrates how this works in a trivial graph. The program defines a set of task descriptors (a), where node B in fact represents a set of nodes. At run time, node B is instantiated to create nodes B0, B1, and B2 (b). The instantiated nodes may be actually created or just implied in the implementation. As will be seen in Chapter 4, it may be more efficient to do one or the other, depending on the task graph structure and how the instantiation is specified.

In addition to representing sets of instantiated tasks, task descriptors may be a hierarchy of other task descriptors, similar to the hierarchy mechanism in the HTG. A hierarchy is the grouping of task descriptors rather than of individual tasks and serves a different purpose. Rather than containing loops, the hierarchy consists of additional task descriptors, as seen in Figure 3.2. Of course, the hierarchy is not strictly necessary as the inheritance of dependencies could be carried out manually. A hierarchy provides the framework for a set of transformations to convert the representation between these different forms: more will be given on these operations in Section 3.5.



(a) Task Descriptor Graph (b) Task Instantiation Graph

Figure 3.1: A visualization of the tasks made by the task descriptors. Node B is a task descriptor for three instantiated tasks. (a) shows the static TDG, (b) shows the instantiated task graph represented by (a).

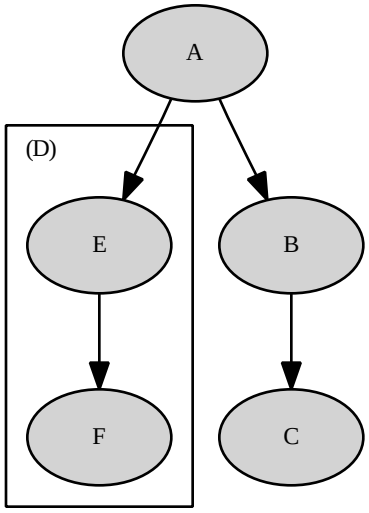


Figure 3.2: Example of a hierarchy of tasks. Nodes represent task descriptors, with sequencing and conflict relationships represented by the edges in the graph. Nodes E and F form a hierarchy within node D.

The basis of the representation is a graph of task descriptors, which we designate as the Task Descriptor Graph (TDG). The task descriptors may form loops

or may be hierarchical. The scheduling back end for ADOPAR, however, operates on a simple Directed Acyclic Graph (DAG) of individual (*instantiated*) tasks, called the Task Instantiation Graph (TIG). The TIG no longer contains this hierarchy and cannot contain loops (which have no meaning to the scheduler). The TDG may be converted to an TIG by instantiating out each task and relationship represented by the descriptors.

The figures in this work distinguish between TDGs and TIGs by shading the nodes of task descriptors. Because the IR is based on a graph, most of the figures will use standard graph notation consisting of nodes (ovals) and edges (lines and arrows). The nodes of the graphs represent the tasks while edges represent the relationships between them. Relationships may be directed (arrows) or undirected (dotted lines). Directed edges are used for sequencing relationships while undirected edges are more appropriate for mutually exclusive and read sharing relationships where no direction is implied by the relationship itself. Hierarchical nodes (only in TDGs) are represented by including a sub-graph of the contained tasks within a separate box annotated with the parent node's name. The parent's node is a rectangle instead of an oval to reflect this hierarchy.

3.2.1 Task Types

To support the front-end programming model, the IR includes several task types. Some task types are vital to the functioning of the TDG while others are for convenience. The types of tasks may be expanded in the future; the currently supported tasks are:

Root Task The root of the task structure. It contains no body (like an empty task) and cannot be contained within another task (but see also Substitution Task below).

Empty Task A task which has no body: its only purpose is to be a hierarchical placeholder for task descriptors.

Block Task A set of basic blocks for the task body, where the control flow through the blocks has a single entry and exit point; loops and branches may exist between these two points.

Function Call Task A single function call, with subtasks defined within the function call itself.

Substitution Task A mechanism for creating dynamic task structures (an entire task structure may be substituted in its place).

3.2.2 Synchronization Primitives

The ADOPAR representation does not have any specific synchronization primitives beyond those implied by task relationships. The scheduler is required to insert the appropriate synchronization between tasks to enforce serial or exclusive execution where it is required. Of course, synchronization need not occur when two tasks have been scheduled in the appropriate order on the same processor. The duty of a front end is to describe relationships: synchronization is then a property extracted from those relationships.

The relationships between tasks determine how they must be scheduled: dependencies on shared variables may force two tasks to become serialized (task dependencies) or simply prevent them from executing simultaneously (mutually exclusive tasks). Relationships are described as edges on the graph. Serial relationships in the graph are enforced at run time through binary semaphores; the latter of the related tasks does not execute until the earlier task signals its completion. A semaphore is signaled once the task ordering is satisfied. Synchronization is not necessary when ordered tasks are scheduled onto the same thread, but they must be scheduled in the correct order if the schedule is to be correct. Conflict relationships require that tasks cannot execute simultaneously. The execution order of two tasks is not important as long as they are not executed simultaneously on separate processors. In this case, the scheduler must either schedule exclusive tasks to the same thread or to separate threads with appropriate locking synchronization to prevent simultaneous execution.

Barriers, where a set of threads must wait until all have reached the same point, exist but not as a specific function call or other primitive; rather, the TIG may be organized so that a barrier is simulated. Figure 3.3 shows an example of such a barrier. By forcing all control flow to go through a single empty task (node E in the figure), all threads must wait on the completion of that task. Partial barriers consist of only a portion of the graph passing through the controlling node. Shaping the graph through a barrier allows the scheduler to control which threads must wait on the barrier task.

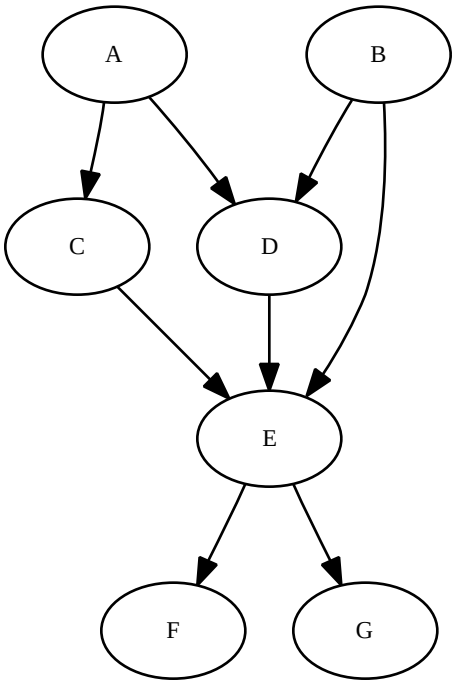


Figure 3.3: The implementation of a barrier in the ADOPAR internal representation. All tasks must go through node E, which acts as the barrier.

3.3 The ADOPAR Internal Representation

ADOPAR uses schedulers that operate on a well-defined task graph, but tries to not be constrained by any single front-end programming paradigm. The ADOPAR internal representation is formed by two major principles:

1. Tasks are defined in the program executable, but not instantiated until run time. Delayed instantiation is accomplished through *task descriptors* and *relationship descriptors*, parameterized task graph primitives defined in the executable.

2. Few restrictions should be placed on the tasks to provide compatibility for a wide variety of front ends. ADOPAR runs within a Just-In-Time compiler (JIT) environment, enabling executable code to be part of the representation. *Instantiation functions* of descriptors contain this code and perform the task and relationship instantiation.

3.3.1 Task and Relationship Descriptors

Task descriptors are a powerful concept that can represent a class of tasks. However, descriptors need some mechanism for specifying how many tasks are to be created, what the parameters are for each task, and how the tasks relate to each other. Different paradigms force the representation to be very generic, especially within applications that have irregular communication patterns. Previous work has limited the user to a specific constraint system to simplify operations on the sets (such as unions or intersections), with anything that does not fit these constraints defaulting to singleton sets (an enumeration). ADOPAR takes a significantly different approach by allowing instantiation functions of any form.

The instantiation functions are actual code provided by the front end, rather than a predefined form, such as a linear equation, where the user only has control over the constants that control the representation. The ADOPAR IR has the capability of executing instantiation functions in a JIT compiler or at compile time if the necessary parameters are available. In fact, there are normally no calls to the representation functions outside of the representation implementation. In those cases when static analysis is not possible, the representation function is not called until the tasks are actually instantiated and the TIG built during execution.

An instantiation function for a task descriptor has access to the parameters of the instantiated parent task in the task hierarchy. The instantiation function creates a set of subtasks using the task parameters. In addition, the parameters of each

subtask are initialized with data specific to the task, such as array indexes to access when executed. The ADOPAR schedulers have the capability to schedule based on a task cost: the instantiation function may provide an estimate of execution cost as part of instantiation. This estimation takes precedence over that provided by the definition of the task descriptor.

Any number of schemes are possible due to the flexibility of instantiation functions. For instance, a task descriptor may represent the contents of a DOALL loop by instantiating a task for each iteration and assigning the loop index as the parameter of each task. Tasks using sparse data may analyze the matrices and determine the specific elements that need to be computed, either creating a task for each item or storing information about the element as a task parameter. Examples will be seen in Section 3.4. Regular access patterns may operate as a special case: analysis of constants and partial evaluation [32] will provide the compiler with enough information to perform static analysis.

Providing instantiation functions supplies the user with immense flexibility in their programming model: any number of front-end representations may be used and then easily integrated into the ADOPAR representation by providing a suitable translation function. In a sense, instantiation functions work as the inspector of the communication and tasks.

Beyond the actual instantiation of tasks, it is also necessary to provide a generic means to represent the relationships between these instantiated tasks. Relationship instantiation is similar to instantiating tasks, but edges in the TIG are created rather than nodes. A relationship instantiation function takes an instantiated task and produces a set of predecessors and successors to that task, as well as the exact relationship and cost associated with any communication involved. The definition of a relationship descriptor also contains an estimate of the communication cost, an estimation for the type of relationship, and even association with a variable (so alias analysis results and other information may be encoded). There are five possible relationships between tasks:

Read After Write (RAW) A "true dependency" – the second task will read the results of the first. Ordering must be enforced between these two tasks.

Write After Read (WAR) An "anti-dependency" – the second task writes to a data location read by the first. Renaming the data location for one or the other can break this dependency; otherwise, ordering must still be enforced.

Write After Write (WAW) An "output dependency" – both tasks write to the same data location. As with WAR, renaming the data location can break the dependency.

Exclusive The ordering of the two tasks does not matter, but they cannot occur at the same time.

Read Share The tasks read the same data, but no ordering needs to be enforced between them.

None No relationship exists, meaning that the scheduler has full control over the execution order of the tasks and no edge exists between instantiated tasks in the TIG. This is the default relationship if none is specified.

The first three relationships specify directed edges between nodes in the TIG, forcing execution orderings. In some cases, the scheduling system might break WAR and WAW edges by copying data, removing the dependency at the expense of additional memory usage. The fourth relationship type adds a conflict(an undirected edge). The scheduler must then add a lock to maintain mutual exclusivity, but only if the tasks are scheduled to different threads. The scheduler may eliminate or avoid locks if the task and communication costs are accurately known [33, 34]. Read sharing relationships are not strictly for correctness and the scheduler may even ignore them. However, the scheduler may use the knowledge about read sharing to improve the cache locality. Specifying more than one relationship between tasks is allowed as long as they are not contradictory.

Care must be taken that relationships do not form loops in the TIG. Loops may exist between task descriptors in the TDG, but loops must be resolved when instantiated to form a DAG[1].

3.3.2 IR Creation Process

Any front end to ADOPAR, whether it be a parallelizing compiler, programming model, or manually-supplied by the programmer, must include the following elements of the IR as part of the IR building process:

Task Descriptors The basic element of the IR is the task descriptor, which consists of a task body, instantiation function, and an estimation of the task cost.

Task Hierarchy The hierarchical structure of the task descriptors is specified inline where the code should be executed. OpenMP uses a similar "fork-and-join" concept.

Relationship Descriptors Relationship descriptors are specified between two task descriptors, associating two tasks and the appropriate relationship instantiation function. Relationship descriptors reference two task descriptors, have an instantiation function, and an estimate of the relationship type and communication cost.

Execution Directives Calls to schedule, execute, and control the task graph must be placed in appropriate locations of the code.

3.4 Examples

Some simple examples illustrate how standard algorithms translate into the ADOPAR internal representation. These examples will be expressed in pseudocode and graphs to avoid the implementation details that will be discussed in Chapter 4.

[1]It may be possible to support loops that have WAR and WAW relationships if these loops can be broken as described. Future work may look at using dependency elimination for optimization and additional parallelization opportunities.

The IR does not attempt to regulate the programming model or granularity used. The examples in this section show only a small variety of possibilities, but the IR can mold to whatever style the front end supplies. Read sharing relationships will not be shown in these examples for simplicity and because they are an optional component of the task graph. In practice, the addition of read sharing is a simple process in the IR.

3.4.1 Linear Algebra

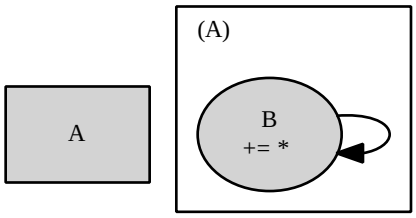
While ADOPAR attempts to target irregular applications, it is informative to show how regular algorithms may be expressed. Using regular applications also simplifies the representation for the purposes of this section. A simple representative algorithm with few data dependencies is matrix-vector multiplication. The basic algorithm can be seen in Figure 3.4.

```
for ( i = 1 to m )
  for ( j = 1 to n )
    result[i] += matrix[i][j] * vector[j];
```

Figure 3.4: Pseudocode for a dense matrix-vector multiply.

There are several ways of representing the matrix-vector multiply algorithm with various trade-offs in simplicity and effectiveness. The most straightforward is to start with a base task descriptor that corresponds to the outermost loop. Contained within the outer descriptor is another task descriptor representing the inner loop and its body (see Figure 3.5). The leaf tasks have a simple body: multiply two items together and add the product to the result. These leaf tasks may execute in any order, but any two tasks that would store in the same result location must maintain mutual exclusivity. If the tasks are scheduled appropriately, the tasks may be able to execute with minimal blocking on the necessary synchronization, but only if iterations

are scheduled and distributed correctly. The resulting fine-grained TIG can be seen in Figure 3.6.




```
Task A: Instantiate m tasks
Task B: Instantiate n tasks
Rel(B,B): Exclusive operations on result[i] if in same row
```

Figure 3.5: Representation for a dense matrix-vector multiply algorithm. Each inner loop body accesses the same element as those on the same row, making an exclusive relationship.

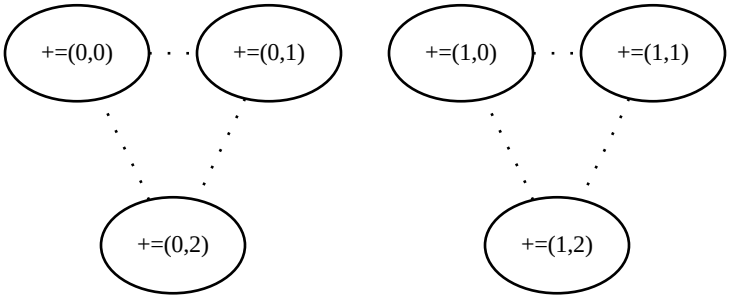
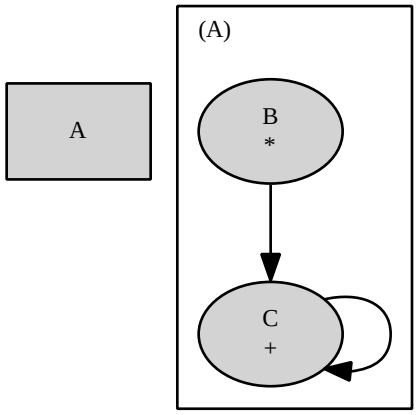


Figure 3.6: The TIG created when multiplying a 2x3 matrix by a 3x1 vector where the sum is reduced using tasks that are exclusive to each other. See Figure 3.5. Dotted lines in this figure represent the exclusivity relationship between nodes accessing elements (i, j).

Alternatively, a summing tree could be used instead of mutual exclusivity, as illustrated in Figure 3.7. In this case, the task body of the inner loop only performs the multiplication. The product is then used in a sequence of summations that reduce

the result to a single value to be stored in the proper location. Using a reduction privatizes the summation to each processor and increases the amount of parallel work available, but requires that the scheduler reduces the amount of communication. The TDG is represented in Figure 3.8.




```
Task A: Instantiate m tasks
Task B: Instantiate n tasks
Task C: Instantiate n−1 tasks
Rel(B,C): RAW in a tree
Rel(C,C): RAW in a tree
```

Figure 3.7: An alternative TDG for matrix-vector multiply, but using a summing task instead of mutual exclusivity.

The third method is a compromise between the two: the summing tree is condensed into a single task for each row, an overall coarser granularity. The resulting TDG of Figure 3.9 is nearly the same as Figure 3.7, but with different instantiation functions, making the TIG as seen in Figure 3.10. In reality, there are a near-infinite number of combinations between a single summation and a full summation tree to modify the task granularity and improve performance: using summing tasks with more than one input and then placing these tasks in a tree is an effective solution. However, the instantiation functions must be carefully designed to support this com-

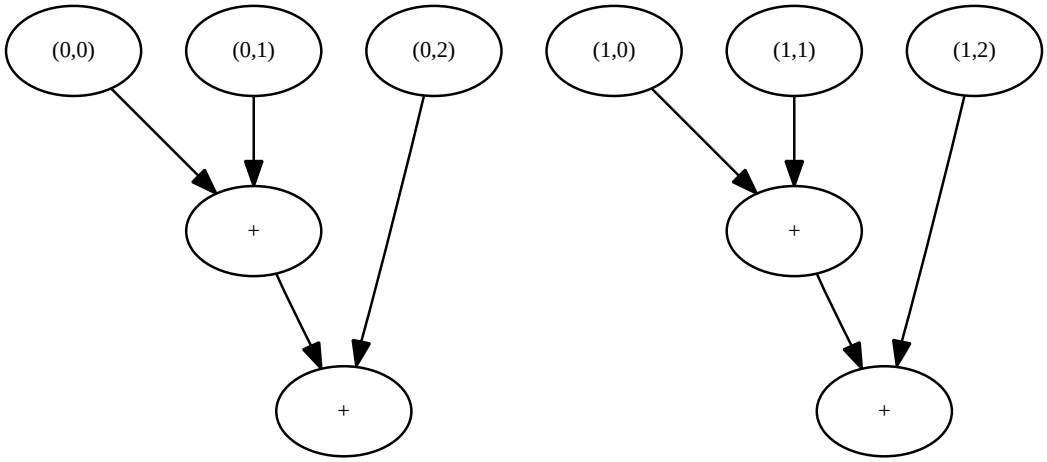


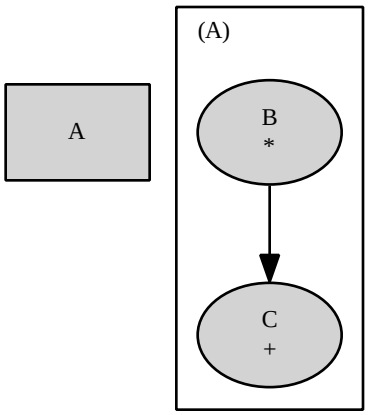
Figure 3.8: An equivalent TIG to Figure 3.6 but using a summing task instead of conflicts.

plex structure. A similar result can also be achieved by merging tasks to adjust the granularity. The chosen solution must consider the number of tasks that must be scheduled, the communication cost between each of these tasks, and increasing the amount of parallelism available for the scheduler to use.

Lastly, we can increase the task size by combining the hierarchy (see Section 3.6) into the parent row task. The granularity modification forces each row to be computed as a single task (Figure 3.11) so that no synchronization is needed. The load will be well-balanced in this case as there are many rows in the matrix, there are no locks to deal with, and the task granularity is larger (further reducing overhead). See Figure 3.12 for the TIG created by the granularity reduction method. Granularity adjustments (as presented in Section 3.5) provide a mechanism for manually or automatically improving the performance by reducing the per-task overhead.

A transformation compiler pass may convert between these different but equivalent representations. The details behind this procedure are largely beyond the scope of this work, but task combining and descriptor splitting (as described in Section 3.5) are specific TDG transformations supported to assist in the transformation process.

One goal of ADOPAR is to work with irregular access patterns like those produced with sparse matrices. Interestingly, the TDGs (Figures 3.5, 3.7, and 3.9) for




```
Task A: Instantiate m tasks
Task B: Instantiate n tasks
Task C: Instantiate n−1 tasks
Rel(B,C): RAW within the same row
```

Figure 3.9: Using a single summation task for each row rather than a summing tree.

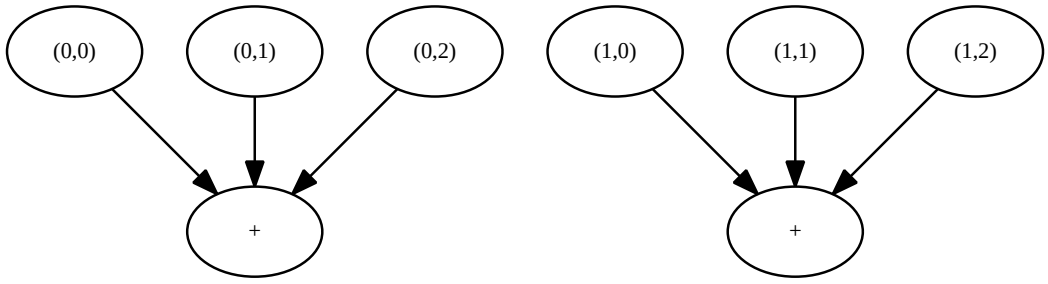
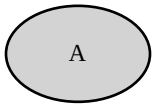


Figure 3.10: An equivalent TIG to Figure 3.8 but reducing the summation into a single step rather than using a tree.




```
Task A: Instantiate m tasks
```

Figure 3.11: Flattening the hierarchy to change the granularity of computation.

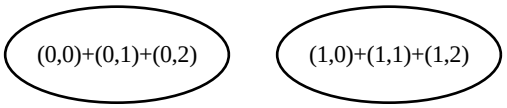


Figure 3.12: An equivalent TIG to Figure 3.6 but with the granularity changed by merging the hierarchy so that each row is computed in a single step.

sparse matrix access are generally equivalent: the only differences are the instantiation functions. Instead of simply creating a regular pattern of tasks, the sparse versions of the instantiation functions are implemented as an inspection of the sparse matrix and an enumeration of the tasks necessary to access it. The TDG, however, remains unchanged.

3.4.2 Iterative Linear System Solvers

Both the Jacobi and Gauss-Seidel methods for solving linear systems have more interesting properties to demonstrate with the IR. It is also interesting to examine the differences in the representations for each and compare with the simple matrix-vector multiplication algorithm.

The Jacobi method is very simple and easy to parallelize in general. The basic pseudocode for these solvers is outlined in Figure 3.13. The calculations in each iteration are dependent on the results of the previous iteration, and not on any results obtained in the current iteration. After the current iteration has finished, the new results must become the old results for the next iteration.

The entire process resembles that of matrix-vector multiplication, but with additional steps involved (compare the Jacobi TDG of Figure 3.14, which creates a resulting TIG in Figure 3.15). First, the summation for each row is used in calculating the new estimate for X in that iteration. Also, after all the rows have completed, the new and old X vectors must be swapped. Other than the swap task, the Jacobi algorithm creates a task graph with an overall tree-like structure. Lastly, the Jacobi algorithm is iterative, so the algorithm checks for convergence – the algorithm has

```
while( notdone )
  for ( i = 1 to m )
    for ( j = 1 to n )
      if ( i != j )
        sum += A[i][j] * X_old[j];
    X_new[i] = ( b[i] - sum ) / A[i][i];
    notdone ||= chk(X_new[i], X_old[i]);
  swap(X_new, X_old);
```

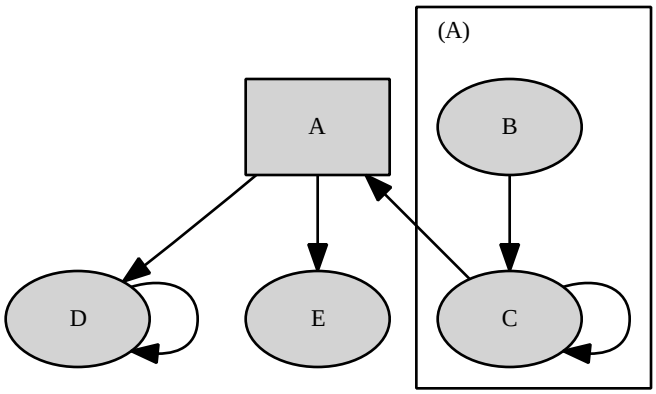
Figure 3.13: Pseudocode for a dense iterative linear system solver using the Jacobi method.

not converged if any row has not converged. For visualization purposes, the TIG can be seen in Figure 3.15.

Instantiation functions must be created for all relationship and task descriptors to form the TIG. Chapter 4 describes the creation process with each instantiation function implemented with the appropriate API calls. Briefly, the functions must create the trees for the summation and convergence checks. In addition, sparse matrices have the same underlying task structure, but with modified instantiation functions to create only those tasks which are necessary.

Notice that the outermost control loop is not part of the task structure. In fact, the task structure is not dependent on this loop, allowing the schedule results to be reused on each iteration. The performance of the loop may be improved by partially unrolling the WHILE loop (potentially increasing the amount of parallel work or allowing for task combining opportunities), but these techniques are outside the scope of this work.

The Gauss-Seidel method is different from the Jacobi method in that calculations may depend on the results of the current iteration (Figure 3.16). The intra-iteration dependencies create complex relationships, but the convergence is much quicker. Unfortunately, it is harder to parallelize due to these additional dependencies. The representation for the Gauss-Seidel method is nearly the same as the Jacobi method, but with an additional relationship descriptor. The following discussion relies on Figures 3.14 and 3.17.




```
A: instantiate m tasks -- (b-sum)/diagonal, chk
B: instantiate n tasks -- scalar multiplication
C: instantiate n-1 tasks -- summation reduction
D: instantiate m-1 tasks -- logical or reduction
E: instantiate one task -- swap X vectors
Rel(A,D): RAW (tree)
Rel(A,E): RAW
Rel(B,C): RAW (tree)
Rel(C,C): RAW (tree)
Rel(C,A): RAW
Rel(D,D): RAW (tree)
```

Figure 3.14: The TDG for the Jacobi algorithm presented in Figure 3.13. Generally, each step in the algorithm becomes a task or task descriptor. Using a reduction tree to determine convergence eliminates a WAR dependence.

The first difference to note is the lack of a swap step, an obvious change given how the algorithm works. The second difference is an additional dependence function between the multiplication step and the result calculation for each row. While the additional dependence appears to create a loop in the task graph, it only exists in the TDG; relationship instantiation functions prevent a loop in the TIG. Figure 3.18 shows that the additional dependencies do not create any loops, but they do make the graph a general DAG rather than a simple tree (which the Jacobi algorithm produces, with the exception of the swap task).

Resolving the apparent loops in a TDG relies on appropriately implemented relationship instantiation functions that correctly describe the algorithm without in-

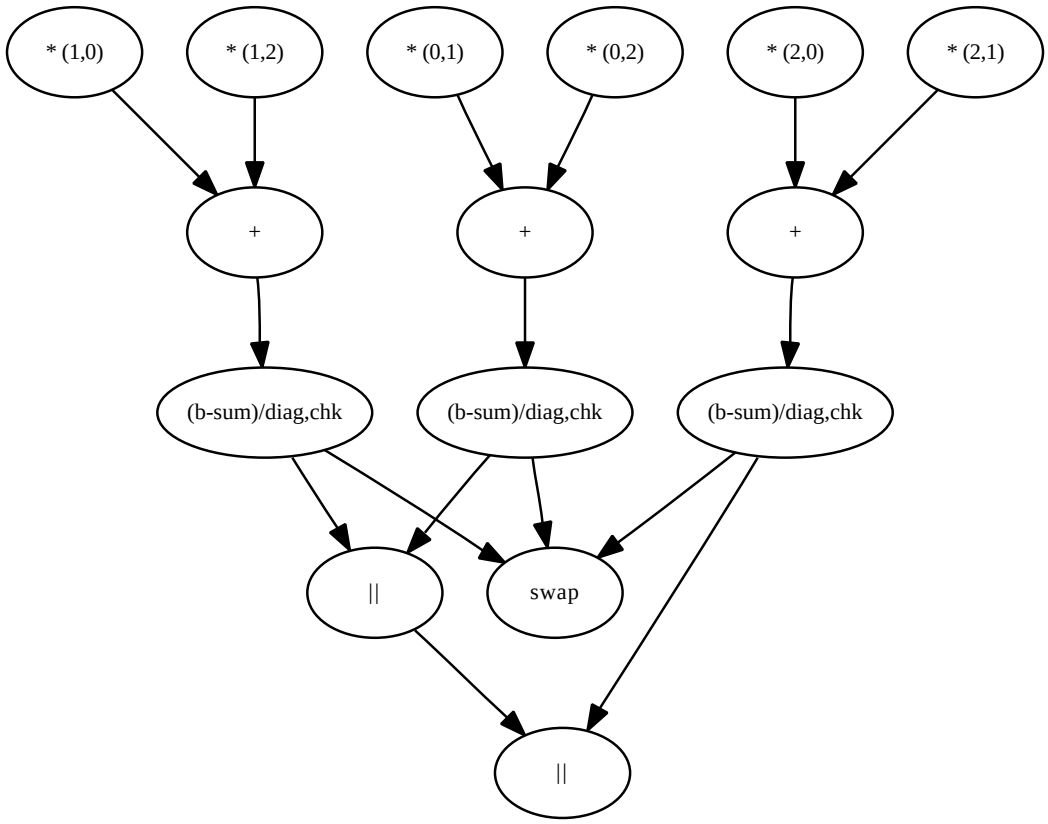
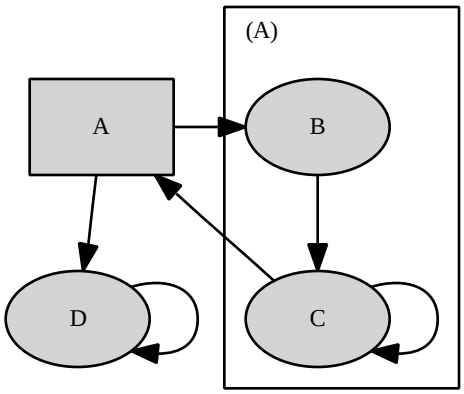


Figure 3.15: The instantiated task graph for a dense 3x3 Jacobi iterative solver using the TDG in Figure 3.14.

```
while( notdone )
  for ( i = 1 to m )
    for ( j = 1 to n )
      if ( i != j )
        sum += A[i][j] * X[j];
    old = X[i];
    X[i] = ( b[i] - sum ) / A[i][i];
    notdone ||= chk( X[i], old );
```

Figure 3.16: Pseudocode for a dense system solver algorithm using the Gauss-Seidel method.

troducing loops into the TIG. The proper implementation for the instantiation functions will be described in Chapter 4.




```
A: instantiate m tasks -- (b-sum)/diagonal, chk
B: instantiate n tasks -- scalar multiplication
C: instantiate n-1 tasks -- summation reduction
D: instantiate m-1 tasks -- logical or reduction
Rel(A,D): RAW (tree)
Rel(A,B): RAW
Rel(B,C): RAW (tree)
Rel(C,C): RAW (tree)
Rel(C,A): RAW
Rel(D,D): RAW (tree)
```

Figure 3.17: The TDG for the Gauss-Seidel algorithm presented in Figure 3.16. The TDG differs from Figure 3.14 by removing the swap task and inserting the additional intra-iteration edge.

As stated previously, the examples shown here are only one way of transforming the Gauss-Seidel and Jacobi algorithms into TDGs. It is also possible to use coarser granularities (by splitting along each row or separating the upper and lower triangular portions of the matrix). The goal of this representation is not to designate what granularity is appropriate, but rather to support whatever granularity that the front end chooses.

3.5 Future Transformations on the TDG

A task graph offers a convenient representation for task manipulation, allowing for optimizations at both the front and back ends. Likewise, the ADOPAR internal

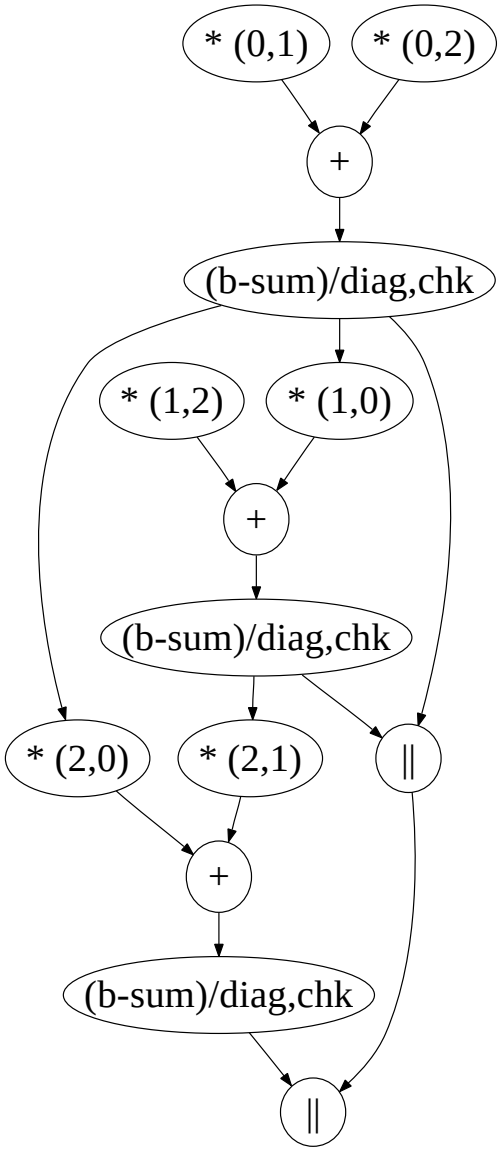


Figure 3.18: The instantiated task graph for a dense 3x3 Gauss-Seidel iterative solver using the TDG in Figure 3.17.

representation, being a task graph, is capable of supporting a wide variety of transformations and analyses. The purpose of this section is to describe some possible future transformations that may be performed on the ADOPAR IR. Hierarchical nodes, task and relationship descriptors, and the variety of task types offer additional manipulations that may be useful for optimization. Improved performance can come through better parallel scheduling, increased parallelism, reduced dependencies, or

lower task overhead. The transformations described here may be used to simplify the task graph, adjust the granularity, or modify it to fit a scheduling model.

The representation's task graph has hierarchical nodes, simplifying analysis as they can be treated as a single node. Task descriptors complicate standard analyses, but other transformations (as presented below) are possible. In addition, it may be possible to improve the run-time performance of the resulting schedule by providing hints that can improve the schedule itself.

This section describes the various transformations that could be performed on the task graph, including splitting task descriptors, flattening the task graph, and more.

3.5.1 Inlining and Extraction

Function call tasks and block tasks have a relationship analogous to the same relationship in code: functions may be inlined into the control flow graph where they are called. It is possible that a compiler will want to inline a function that is a function call task. When inlining occurs to the task graph, a function call task will become a block task with the body of the function being the body of the task. All task relationships remain the same. Also, all subtasks of the function task become subtasks of the block task.

The inverse of inlining is the extraction of a set of basic blocks to create a function. The opposite transformation of the inlining process must occur: first, the blocks must be extracted to form a new function (for the task body); second, an appropriate function call task must be created in the place of the block task; finally, all contained subtasks and relationships must use the new function task. This is a necessary operation so that the task body may be extracted for later execution.

3.5.2 Descriptor Splitting

Task descriptors represent a similar class of tasks, where each has the same code to execute but different inputs. At compile time, we may not know in general the number of tasks that will be instantiated, nor do we know the inputs of each.

However, in those situations that the number of tasks is known or the instantiation function is simple enough to analyze the task descriptor may be replaced with multiple descriptors, each representing a portion of the instantiation space.

Descriptor splitting is analogous to loop unrolling, where individual iterations of a loop are explicitly stated rather than implied within the loop iteration space. Just as a loop may be partially or completely unrolled, a descriptor may be split into "singleton descriptors" (where only one task is instantiated) or a few descriptors which cover a specific portion of the original instantiations.

Splitting task descriptors potentially saves run-time processing at the expense of memory overhead, in much the same way that the run-time system may instantiate tasks or simply analyze the task descriptors with many of the same trade-offs. Splitting may be necessary to permit other cross-iteration task optimizations. This process is not always possible as it relies heavily on the implementation of the instantiation functions for both task descriptors and relationship descriptors. When a task descriptor is split, the task instantiation functions must be rewritten and appropriate relationship descriptors must be inserted to preserve semantics.

3.5.3 Flattening

The task graph is hierarchical in nature, with a subgraph of nodes possibly contained within each node. A hierarchy is not strictly necessary, but makes granularity adjustments rather easy since only the parent node needs to be considered in many cases. On the other hand, some task descriptors can only be split if there is no hierarchy involved, making it necessary to flatten child nodes (removing the hierarchy) in some situations.

Figure 3.19 shows how the simple TDG in Figure 3.2 may be flattened. Task D implicitly has all the dependencies of its child tasks; it retains its edge. Nodes E and F are no longer contained as subtasks within node D. Note that we cannot simply remove node D: it still may have a task body to execute. Of course, a "dead node elimination" transformation could remove node D if it is no longer needed.

In addition, the instantiation functions for nodes E and F need to be extended to complete the flattening process.

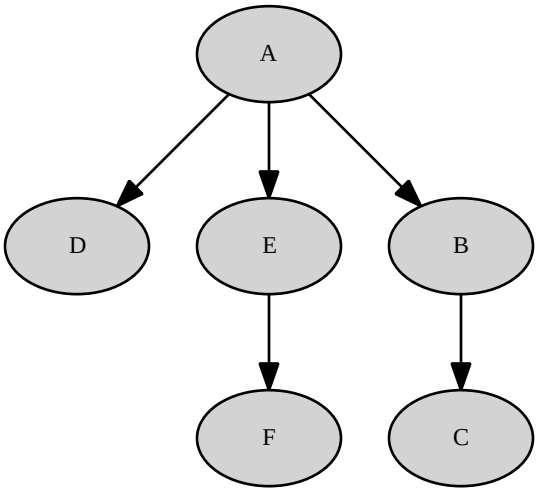


Figure 3.19: A flattened version of Figure 3.2. Nodes E and F have been removed as subnodes of D, which gains the edge to node E.

Flattening may be done statically for certain instantiation functions. A singleton parent task descriptor is the simplest case, so splitting the parent task descriptor simplifies the flattening process. Each of the new descriptors can then be flattened individually.

3.6 Granularity Adjustments

The main disadvantage of fine-grained tasks is the extra overhead and communication involved. The overhead expresses itself at run time as each task is executed and at scheduling time as additional nodes are considered. The latter is especially problematic with higher complexity scheduling and analysis algorithms which may have a complexity of $O(n^2)$ or worse.

It is possible to reduce the number of tasks by changing the granularity: make tasks larger at the expense of parallelism, load balancing, and scheduling options. Adjustments should be programmable: the user and scheduling system must decide

the appropriate level of granularity. Reducing the task size is generally more difficult, but is closely related to compiler-performed parallelism discovery. There are three methods for adjusting the granularity: *discrete task combining*, *task descriptor combining*, and *hierarchy combining*.

3.6.1 Discrete Task Combining

The simplest way of increasing task size is to combine individual tasks. Combined tasks are then considered as a single unit by the scheduling algorithm and will be scheduled sequentially to the same processor as if they were a single task. Execution overhead is also reduced by combining the actual task bodies into the same unit.

Not all tasks may be combined; the following conditions must hold in order to combine a task:

- The transformation may not form loops when combining two tasks. Standard graph algorithms can enforce this rule on the instantiated task graph.
- The tasks must be the same type (block with block, function with function, etc.).
- The tasks must share the same parent task: combining tasks at different levels of the hierarchy is not possible.

In order the combine two tasks, the transformation must:

1. create a single node for both tasks;
2. combine task bodies in an order that satisfies internal dependencies;
3. perform a union on the predecessors, successors, and conflicts;
4. combine the hierarchical subtasks for each task; and
5. calculate cost of combined node by summing the cost of the individual tasks.

Performing these transformations is convenient in the representation. Joining the contained basic blocks combines block tasks. Creating a new function that calls each other task body combines function tasks (the old functions might be inlined for efficiency). The most difficult part of combining tasks lies in modifying the instantiation and representation functions. Chapter 4 shows how these functions are formed.

3.6.2 Task Descriptor Combining

All of the instantiated tasks from a single descriptor satisfy the conditions for discrete task combining. Therefore, it is possible to perform a simple combination of all instantiated tasks from a descriptor into a single task body so that the instantiation function only produces a single task. It is not even necessary to have these tasks enumerated statically in many situations.

As long as there are no subtasks nor circular dependencies that would be created, it is always possible to simply combine the instantiated tasks together: the body of the task must be duplicated the same number of times as the number of tasks instantiated. A transformation may accomplish the combination by joining the instantiation function with the task body (the inverse of what an automatically parallelizing compiler would do). To merge the instantiated tasks of a simple task descriptor with no subtasks, the transformation must create a software loop wrapping the task body. The cost of the resulting node is nc, where n is the number of tasks that were created and c is the cost of any one task. Combining dependencies slightly more difficult: if any of the instantiated tasks use the dependency, the combined task uses the same dependency. The new representation functions must be modified appropriately with a loop (or other appropriate control structures) to combine the resulting dependencies.

Dependencies shared between combined tasks require special consideration. The transformation must combine task bodies in a way that satisfies the dependencies as any scheduling algorithm would be required to do. Following the dependencies resolves loops in the TDG when combining task bodies. In general, however, it is

sufficient to completely split the task descriptor and then iteratively combine tasks in the order necessary to prevent loops in the final TIG.

Hierarchical subtasks also have special requirements: a task descriptor that creates n tasks would also create n copies of the subtask graph. If it is possible to analyze the number of tasks and their parameters, it is also possible to statically replicate the subtasks as well. Flattening the hierarchy aids in task descriptor combining.

3.6.3 Hierarchy Combining

The simplest method to increase task size is to only consider tasks at a certain level of the hierarchy, with any child tasks being part of its parent. Polychronopolous' HTG uses this same idea but is simplified by its nature [25]. Dependencies reaching child nodes complicate the process, as these are implicit dependencies to the parent as well.

Combining tasks hierarchically is not a transformation in itself; rather, it is a property of the representation. The implicit dependencies given by child tasks must be found when examining a parent task. Combining the hierarchy folds these dependencies in so the edges become explicit.

3.7 Summary

This section has described ADOPAR's internal representation. This representation is both flexible and compact: classes of tasks may be represented by *task descriptors*, capable of describing irregular quantities of tasks in any manner required by the front end. In addition, communication is described by *relationship descriptors* that can describe data dependencies as the front end chooses.

There are also many transformations that may be performed on the task graph: nodes may be combined, task descriptors may be split, hierarchies may be flattened, and more. These transformations allow for optimizations to occur, reducing the overhead by adjusting granularity. Static hints may be supplied to improve the scheduling abilities of the representation. The next chapter describes the implementation and

API that the front end and scheduling system may use to build and access the representation, including several examples.

Chapter 4

Implementation

Chapter 3 showed how a task graph can be condensed into a graph of task and relationship descriptors. This chapter presents a method of implementing descriptors in software. Once the Task Descriptor Graph (TDG) can be represented, the compiler, scheduler, and run-time environment have access to the information that can overcome the diversity inherent in manycore environments. Analyses and transformations have the advantage of working with a compressed data set (and the smaller memory footprint that accompanies it). The end goal, shown by several examples of descriptors and TDGs, is to provide support for as much information as possible in a convenient format for the ADOPAR environment.

This chapter is organized as follows: first, we will present the representation's operating environment, including the compiler framework (LLVM) and the scheduler. Next, this chapter describes the API that implements the Internal Representation (IR) along with examples. The method for creating TDGs using the API will be discussed, as well as the code transformations that have been implemented to use the IR. Last, this chapter implements the examples from Chapter 3 based on algorithms for solving linear systems of equations.

4.1 Environment

ADOPAR must have access to all the details of a program to fully adjust for diversity. Thus, ADOPAR needs hooks into a compiler framework to access the code structure, a dynamic execution framework to access variable values, a task representation framework to access task relationships, and a scheduling framework to execute the program given all the available information.

This chapter presents the representation framework that serves as a connecting link between the code and scheduler. To assist integration with the code, we have elected to define the representation implementation within a compiler framework. Using an existing compiler framework has several advantages: reduced development time, access to static code information, and optimization of the representation itself. In addition, we can integrate with the execution environment, making dynamic task structures available within the representation. The goal is to define a common interface for run-time systems and static analyses while building on the backbone of a compiler IR which provides similar capabilities.

ADOPAR is currently being augmented with a variety of schedulers. The IR must cleanly integrate with the scheduling framework as well, providing a translation of the Task Instantiation Graph (TIG).

4.1.1 LLVM

The ADOPAR representation uses the LLVM compiler framework to integrate with both front ends and schedulers. LLVM provides an expressive, flexible, and extensible code representation and has several features to assist with the implementation.

- Compiler *intrinsics* (a custom bitcode instruction in the LLVM IR) extend the compiler's capability to represent parallelism.
- Custom compiler passes provide analyses and transformations.
- A Just-In-Time compiler (JIT) compiler to generate code and run instantiation functions.

This work uses a set of compiler intrinsics as a TDG-creating interface for an arbitrary front end. These intrinsics will be described in detail later. Some basic passes currently exist to serve as a basic front end to the representation, promote block tasks to function tasks, and prepare the code for execution. Passes may also be written to perform optimizations on the TDG. The JIT compiler is used to compile

and execute the final schedule and also allows instantiation functions to be compiled and executed as part of the IR.

4.1.2 Scheduler

The representation must provide an interface to the ADOPAR schedulers. The task graph representation must provide these schedulers with:

- the number of tasks,
- the directed relationships of each task,
- the undirected relationships of each task, and
- the priority and cost of tasks and communication.

The current implementation of the representation explicitly instantiates the full task graph. Iterators over the instantiated tasks and relationships are then provided to the scheduler. Iterators are generally seen as an abstract way of accessing a collection of data; this abstract nature works well with representation functions which can be seen as such a collection. Instead of actually instantiating relationships, iterators may be provided that call the relationship instantiation functions intelligently and only as needed. Using abstracted iterators cuts out the instantiation step and reduces memory usage at the expense of additional computation for each use.

4.2 ADOPAR Intrinsics

The core of the ADOPAR IR is a set of *compiler intrinsics* that allow the front end to specify the TDG. A compiler intrinsic is a built-in function, instruction, or operation of a compiler (although LLVM models intrinsics as special function calls), used by the compiler to provide additional functionality and to aid analysis.

The intrinsics presented in this section allow the front end to specify task and relationship descriptors. A later section will examine how these intrinsics are used to create descriptors.

A user will not generally see the API presented here; rather it is the interface generated by an ADOPAR front end. However, a set of C functions and type definitions have been created that translate directly into LLVM intrinsics and types. The examples will use these functions to clarify the implementation of the IR.

The ADOPAR intrinsics provide three functions to the program:

1. bounds and specifications for tasks and their hierarchical structure,
2. functionality to instantiate tasks and relationships, and
3. querying of task information.

Providing a specific API in the form of intrinsics gives the compiler a standard tool for analyzing the program. How each intrinsic is *lowered* (transformed from an intrinsic to a set of known instructions) depends on the use of the intrinsic. Most of the task specification intrinsics are simply removed when the TIG is created and serve only for analysis and specification purposes. Others are translated into some LLVM instructions to implement functionality. The remaining intrinsics are translated into calls within the compiler itself to be executed inside the JIT environment. The following sections describe the various intrinsics and their purposes, uses, and semantics. The names for intrinsics and types follow the pattern established by LLVM.

4.2.1 Parameters and Types

One focus of the implementation is to take the abstract components of the IR and assign concrete definitions. Task IDs, relationships, and task descriptor variables all fall under this focus.

Task ID A unique signed integer assigned to each instantiated task. Negative integers represent invalid tasks. Root tasks (as they are not a "true" task and are not instantiated) are not assigned an ID. Task IDs are also called *global IDs* to distinguish from *local IDs*.

Task Group ID Task descriptors instantiate groups of tasks. Each grouping is assigned an unsigned integer identifier. This identifier is not necessarily unique

except within the set of task groups of a task descriptor, where they are consecutive by the order created.

Local Task ID Within the task group, each instantiated task is assigned a unique identifier. Together with the task descriptor and group ID, the local ID uniquely specifies a task within the hierarchy and is mapped to the appropriate global ID. The pair of task group and local ID create a scoped identification that is consistent regardless of the overall task structure.

Relationships An enumeration value that describes the relationship between two tasks. The relationship can be any of those described in Section 3.3. The implementation defines the values in Table 4.1 for each of these relationships. Directed and undirected relationships also have an unknown state signifying that the relationship cannot yet be determined (due to run-time constraints). Read sharing is represented by a Read After Read (RAR) relationship.

Table 4.1: Possible relationships for relationship instantiation functions.

Edge Type	Relationship	Value
	None	0
Predecessor	Read After Write	1
Predecessor	Write After Write	2
Predecessor	Write After Read	3
Predecessor	Unknown	4
Successor	Read After Write	5
Successor	Write After Write	6
Successor	Write After Read	7
Successor	Unknown	8
Undirected	Read After Read	9
Undirected	Exclusive	10
Undirected	Unknown	11

Task Descriptors A global variable for each task descriptor. The task descriptor variable allows the front end to define the TDG hierarchy and relationships. It

also associates various properties with the task descriptor (such as the instantiation function and execution cost).

4.2.2 TDG Creation Intrinsics

A large portion of the ADOPAR intrinsics are devoted to designating the hierarchical structure of task descriptors and associating each with their task bodies. Table 4.2 lists the intrinsics involved in creating the TDG. For examples, see Section 4.5. We chose to have instantiation functions create groups of tasks rather than single instances. Creation of task groups is an optimization that recognizes the commonality between the instantiations from a single descriptor. As an added benefit, tasks receive a natural internal naming which can aid in the instantiation functions and task bodies in lieu of a nonstandard task parameter.

To start the ADOPAR-scheduled parallel work, a program simply inserts a TDG inline where execution should take place. In a dynamic environment, the inline definition guarantees that all variables are defined before the TIG needs to be created and before the run-time environment performs the parallel computations. The start of the TDG is the root task descriptor, which then has a hierarchy of subtasks. Figure 4.1 shows how the TDG is embedded in the general function `f()`, which may be called wherever appropriate in the executing program. The example uses the C functions that are directly translated to intrinsics. Note, however, that a user will rarely, if ever, deal with either the intrinsics or equivalent C function calls, which are tools that a front end will use to interface with ADOPAR.

The current implementation does not perform much static analysis (beyond the simple passes described in Section 4.4). To dynamically execute the TIG, the back end does the following:

1. instantiates the TIG,
2. schedules the TDG for the number of available cores (or as specified), and
3. executes the schedule a single time.

Table 4.2: Intrinsics to create the TDG, defining task and relationship generators and their properties.

Intrinsic	Description
`void task.root( task.type* task, void* data )`	Starts a task tree with the designated descriptor given a pointer to arbitrary data. There is no explicit parent, but one may be specified via `task.sub`. The data argument will be provided to the instantiation functions.
`void task.rootend( task.type* task )`	Ends the task tree. For the top root of the TDG, the TIG is created, scheduled (if necessary), and executed.
`void task.empty( task.type* task, task.type* parent, gen.type( )* inst )`	Begins an empty task descriptor. There is no task body (and therefore, no cost for execution).
`void task.block( task.type* task, task.type* parent, i32 cost, gen.type( )* inst )`	Begins a block task descriptor, with a task body designated by the code between this intrinsic and the corresponding `task.blockend`.
`void task.blockend( task.type* task )`	Ends a block task, matching to the last `task.block`. All control flow must encounter a `task.blockend` before any other task is created.
`void task.func( task.type* task, task.type* parent, i32 cost, gen.type genfunc( )* inst, (function), ... )`	Begins a function call task descriptor. The function and any arguments are given. The arguments are passed to every instantiation of this task descriptor when executed.
`void task.sub( task.type* task, (function), ... )`	Placeholder for a portion of the TDG. The function is called with each of the arguments specified on each execution of the TIG, and any root task is "spliced" into the current TDG.
`void rel.basic( task.type* t1, task.type* t2, i32 cost, i32 rel, relfunc.type* inst )`	Defines a relationship descriptor between two tasks. The communication cost and relationship type are estimated, and the instantiation function is given. No knowledge of shared memory is known.
`void rel.var( i8* mem, task.type* t1, task.type* t2, i32 cost, i32 rel, relfunc.type* inst )`	Defines the shared variable of a relationship. Otherwise, acts the same as `rel.basic`.

```
task_type root;
void f( int *array, int size ) {
  task_root( &root );
  /* Task Generator Hierarchy */
  /* Relationsip Generators */
  task_rootend( &root );
}
```

Figure 4.1: Example of the creation of the root task for a hierarchy and the appropriate task descriptor.

Instantiation and scheduling only occur once. However, if the TDG is embedded within a loop, the schedule can be reused each time (although the user may explicitly invalidate a schedule, causing these steps to be performed again). Instantiation and scheduling may also occur statically with some programs. Analysis may determine the task structure at compile time, and the schedule may be inserted directly, rather than allow the run-time environment to perform these (generally difficult) steps.

Task descriptors, the basic building blocks of the TDG, are defined by a few basic properties:

- the task descriptor,
- the parent task,
- a execution cost estimate, and
- its instantiation function.

Function tasks may also be given an additional set of parameters that will be passed to every created task and updated for each reuse of the schedule. These *automatic parameters* allow communication of state outside of the parallelized code. Automatic parameters also exist for block tasks and are implied by the variables that are accessed by the task body.

Relationship descriptors (edges in the TDG) are created in a similar fashion. Basic properties of relationships include:

- two task descriptors,
- a communication cost estimate,
- the relationship type, and
- its instantiation function.

If the relationship type is not known statically, it may be defined programmatically in its instantiation function. In this case, one of the "unknown" relationship types would be used. Variable relationships may also specify the memory location that created the relationship. The front end provides a base address for the memory (such as the start of an array) and then the combination of relationship and task descriptors describe the access pattern. Task and relationship descriptors are significantly different than Regular Section Descriptors (RSDs), Data Access Descriptors (DADs), and Processor Tagged Descriptors (PTDs), where the description is limited to a dense matrix and a prescribed pattern of access.

4.2.3 Instantiation Function Intrinsics

The representation provides a specific set of intrinsics available to task and relationship instantiation functions. These intrinsics are detailed in Table 4.3. The runtime environment will prevent the representation from attempting to call an intrinsic inappropriately (such as calling `gen.maketasks` from an instantiation function). All instantiation function intrinsics work with local IDs to enforce proper scoping. Relationship instantiation functions cannot designate communication between arbitrary tasks, but only those involved in the relationship.

Instantiation functions instruct the representation to create sets of tasks and edges. They also supply the actual cost associated with these instantiations. The method chosen to supply data to the intrinsic call is determined by the front end, and may range from a simple constant to a more complex algorithmic expression, or even a table lookup. In this sense, the instantiation function intrinsics represent the output of the algorithm, while the parameters of the instantiation function represent the inputs.

Table 4.3: Intrinsics used to instantiate tasks and relationships.

Intrinsic	Description
`i32 gen.maketasks( i32 size, i32 count )`	Performs task instantiation given the number of tasks. A storage area for parameters is also created for each task. The task group index for the newly created tasks is returned. This intrinsic must only be called once per call to the task instantiation function. Must be called in a task instantiation function.
`void gen.setcost( i32 ID, i32 cost )`	Sets the execution cost associated with the given task. Using a negative local ID assigns every task in the task group the same cost. If a task is not given a cost, the task descriptor's estimated cost is assumed. Must be called in a task instantiation function.
`void gen.makerel( i32 TG, i32 ID , i32 cost, i32 rel )`	Creates a predecessor, successor, or conflict edge with a task specified by the task group and local ID. The relationship is specified along with the actual cost of communication. Must be called in a relationship instantiation function.

4.2.4 Querying and Execution Intrinsics

This section describes some utility intrinsics that give access to task information and perform scheduling on the TIG. Some of these intrinsics are available to instantiation functions; a few are used in the examples that follow. The complete set of querying and execution intrinsics are described in detail in Table 4.4. This table is divided into three sections: the first section shows the querying intrinsics available to instantiation functions. The second shows parameter access intrinsics available to task bodies. The last section lists intrinsics that control the TDG and TIG.

The equivalence of global and local IDs suggests a set of intrinsics that convert between the two task identifications. One intrinsic provides the parameters for any task. All of the querying intrinsics may be called from any instantiation function.

Task bodies have access to the `exe` intrinsics, providing the parameters of the running task and the parameters of its predecessors and successors. These intrinsics may be called from any task body (such as a function call task or block task). While most intrinsics are implemented as callbacks to the run-time environment, these intrinsics require special performance considerations and are lowered to instructions that directly access the necessary memory locations.

The `sch` intrinsics modify how the TIG is scheduled. They may be used at any point in the program outside of the task bodies and take effect when `task.endroot` is called. The `sch.join` intrinsic is an exception as it terminates the already running threads.

4.3 TDG Creation

Section 4.2 described the intrinsics that create task descriptors, link them together to form the TDG, define task bodies, and then schedule and execute the resulting TIG. This section describes the use of the ADOPAR intrinsics and the process to create a trivial TDG. The front end that creates the TDG may be a parallelizing compiler, a specialized input language, an ADOPAR transformation pass, or even a manual process using the LLVM assembly and ADOPAR intrinsics (although this is rare).

Table 4.4: Intrinsics related to querying the task structure and for scheduling the TIG.

Intrinsic	Description
`i32 query.localid( i32 ID )`	Returns the local ID of a task given its global ID, or a negative integer if an invalid global ID is provided.
`i32 query.taskgroup( i32 ID )`	Returns the task group of a task given its global ID.
`i32 query.globalid( task.type* task, i32 TG, i32 ID )`	Converts a local ID into a global ID.
`i32 query.numtasks( task.type* task, i32 TG )`	Returns the number of tasks created in the specified task group of a task descriptor.
`i8* query.data( i32 ID )`	Returns the parameters associated with a task, NULL if the global ID is invalid. No bounds or type checking is provided for the parameters.
`i8* exe.mydata( )`	Provides the parameters of the current task to the task body. As such, it may only be called within the body of a task.
`i8* exe.mylocalid( )`	Provides the local id of a task to the task body. As such, it may only be called within the body of a task.
`i32 exe.numsucc( )`	Returns the number of successors to the currently executing task.
`i8* exe.succdata( i32 num )`	Returns the parameters of the specified successor to the currently executing task. No bounds checking is provided.
`i32 exe.numpred( )`	Returns the number of predecessors to the currently executing task.
`i8* exe.preddata( i32 num )`	Returns the parameters of the specified predecessor to the currently executing task. No bounds checking is provided.
`void sch.numthreads( task.type* root, i32 num )`	Sets the number of threads to schedule on the specified TDG. The default value is the number of available cores.
`void sch.usepolicy( task.type* root, i32 policy )`	Specifies the policy that should be used when scheduling the TIG.
`void sch.join( task.type* root )`	Signals a return to serial execution, invalidates the schedule, and terminates all threads.

All task descriptors are created within the root task, and follow the same basic form. Figure 4.1 shows the beginnings of our example: a function `f()` that will be called at some point in the program to perform some operation on an array. Let us define this operation to be the serial algorithm in Figure 4.2.

```
void f( int *array, int size )
{
  for( int i=2; i < size; i++ )
    array[i] = array[i-2] + 1;
}
```

Figure 4.2: The serial algorithm used for the examples in this section. Note that there is an inter-iteration dependency that will need to be considered.

The first step is to determine the task body; we'll use the loop body for this example. Figure 4.3 shows how the body may be transformed into a function task body. The task has a single shared array between all tasks. The local ID of the task determines its loop index. The same result could also have been accomplished through a parameter accessed through the `exe.mydata` intrinsic, a more likely option if the data were sparse. The array itself may change on each execution of the TIG, allowing reuse of the schedule as long as the size of the array does not change.

```
void task( int *array )
{
  int i = exe_mylocalid() + 2;
  array[i] = array[i-2] + 1;
}
```

Figure 4.3: The task body extracted from the loop body of Figure 4.2.

The instantiation functions for the task and relationship descriptors are presented in Figure 4.4. The task instantiation function is quite simple: there are `size-2`

tasks to be instantiated (the number of loop iterations). The size variable must be given to the root task descriptor so it may be passed in to the task instantiation function. The relationship instantiation function is only slightly more complex: the results of each iteration will be used in the second iteration afterwords ($0 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 5$). Using this dependency sequence, two chains of even-numbered and odd-numbered iterations are formed. There is no bounds checking in the instantiation function as it is unnecessary (invalid task IDs are simply ignored) and leaving out the additional logic makes the function simpler to analyze.

```
void gen( int, void *size, task_type* )
{
  int num = *(int*) size - 2;
  gen_maketasks( 0, num );
}

void rel( int ID, void*, task_type*,
          task_type* )
{
  int localID = query_localid( ID );
  gen_makerel( 0, localID + 2, 0, succRAW );
}
```

Figure 4.4: The instantiation functions inferred by the algorithm in Figure 4.2.

Finally, the task and relationship descriptors need to be inserted into the TDG. Figure 4.5 shows how descriptors are inserted as part of the root task descriptor presented previously. Admittedly, the total amount of code is larger than the original algorithm. However, the IR captures a considerable amount of information that was only implied in the original serial algorithm. This information will generally come from either compiler analysis or through the programming model; a programmer will

not have to deal directly with the IR's API. The additional information made available includes:

- the task body (`task()`),
- number of tasks (`gen()` and `size`),
- the exact relationship between tasks (`RAW`, `rel()`),
- task-independent data (`array` and `size`), and
- shared variables (`array`).

```
task_type root, func;
void f( int *array, int size )
{
  task_root( &root, &size );
  task_func( &func, &root, 1, &gen, &task, array );
  rel_var( array, &func, &func, 0, succRAW, &rel );
  task_rootend( &root );
}
```

Figure 4.5: Inserting the task and relationship descriptors for the parallel version of Figure 4.2.

Each piece of information is presented separately and algorithmically. The algorithms may be executed to perform the computations or analyzed for additional processing.

4.4 Code Transformations

Using LLVM as a framework for ADOPAR gives the implementation an opportunity to perform code transformations. These transformations serve several functions:

- lower intrinsics to create more efficient code,
- bring the code to a canonical form,

- improve the execution speed,
- bundle data and arguments for efficiency, and
- manipulate the TDG.

We have implemented several of these transformations. Since no front end is currently available for the LLVM IR as it has only recently been implemented, the first transformation of the preceding list allows for a basic C/C++ program to use the intrinsics directly as C function calls. These functions are defined in a special header, along with some useful type definitions. The examples in this chapter use this manual front end for convenience.

Block tasks and function tasks are essentially equivalent in function, if not in form. Internally, a task body is represented as a function call, which means that block tasks should be converted to function tasks, bringing the code to a canonical form. LLVM makes this process simple through a short sequence of steps.

1. Split LLVM basic blocks at the block task begin and task end intrinsics.
2. Find all the LLVM basic blocks of a block task by walking the dominator tree (the control flow within the block task).
3. Use the `ExtractCodeRegion` functionality of LLVM to turn the basic blocks into a function call.
4. Transform the function call and block task intrinsics into a function task intrinsic.

The dynamic instantiation of the TIG also transforms task bodies. Tasks have two sets of parameters. The first is the local parameters which provide essential task-specific information such as the variables on which the task operates. The second set is the automatic parameters which are part of the task descriptor and shared between all instantiated tasks. These parameters are provided as the arguments to the task body. In the case of block tasks, the parameters are inferred (and are automatically translated into function arguments by the LLVM code extractor).

The IR presents a variable-argument intrinsic to support automatic parameters. These parameters are then bundled together and stored in a location specified by the run-time environment; the task body is modified to use the bundle rather than discrete arguments.

Each of the ADOPAR intrinsics is lowered to an internal function call that performs the necessary operations to create the task graph and provide querying information. However, a few exceptions do apply. First, as was already discussed, the block task descriptors are translated into the equivalent function task descriptors, so no internal function is necessary in this case. More significant are the execution intrinsics. Rather than implement these intrinsics as a function call, the intrinsics are translated into arguments of the task body.

4.5 Examples

Chapter 3 presented several examples of TDGs for the use of the ADOPAR IR. The actual implementation of the task graph is presented in this section, using appropriate intrinsics to create tasks and define their relationships.

4.5.1 Sparse Linear Systems

We will first examine the case of the Jacobi algorithm for solving linear systems, showing how the IR describes both sparse and dense matrices. Afterwords, we will present the modifications for the Gauss-Seidel algorithm. We will start from a pseudocode language similar to OpenMP (since no front end has been implemented as of yet) and assuming a dense matrix for simplicity. The algorithm in Figure 3.13 could be annotated as in Figure 4.6 to describe the parallelism. The annotations instruct which tasks to create through the ADOPAR IR and how they are related. Although relationships have been explicitly annotated, it should be trivial for a front end to produce them automatically. Combining the annotations, a front end would produce the TDG in Figure 3.14.

The next step is to encode the TDG, including the nested FOR loops, reduction trees, and synchronization. The code for this task structure is listed in Figure 4.7.

```
while( notdone )
  // ADOPAR: root

  // ADOPAR: task(loop1)
  for ( i = 1 to m )

    // ADOPAR: task(loop2)
    for ( j = 1 to n )
      if ( i != j )

        // ADOPAR: reduction(sums)
        sum += A[i][j] * X_old[j];

    // ADOPAR: rel(RAW, sums, i)
    X_new[i] = ( b[i] - sum ) / A[i][i];

    // ADOPAR: reduction(ors)
    notdone ||= chk(X_new[i], X_old[i]);

  // ADOPAR: task(swap) rel(RAW, loop1, *)
  swap(X_new, X_old);

  // ADOPAR: end
```

Figure 4.6: A pseudocode annotation for the dense Jacobi linear system solver showing how the original version may be parallelized.

Notice that the hierarchy is enclosed in the main iterative loop of the algorithm. An additional task (`final`) has been added to extract the value produced by the `or` task and store it in the variable `notdone`. This value can then be used to control the main loop. Edges are also created as part of the task structure. The TDG for a sparse matrix is the same, but an additional edge is required for the (probably rare) case that only one non-diagonal element exists in a row. In this case, no summing tree is required: the relationship from the inner loop to the outer loop (`loop2TOloop1`) provides the necessary communication. If the matrix is dense, this extra edge is simply never used.

The task instantiation functions are listed in Figure 4.8. The simplest function is for regular accesses, as in the outer loop (`loop1`). This task creates one instantiation

```
sch_numthreads( &root, 16 );
do {
  task_root( &root, &matrix );
    task_func( &loop1, &root, tcost, &gen1, &loop1_body,
               &vector, X_new, X_old, tol );
      task_func( &loop2, &loop1, tcost, &gen2, &loop2_body,
                 X_old );
      task_func( &sums, &loop1, tcost, &genSums, &sum_body );

    task_block( &swap, &root, tcost, NULL );
      std::swap( X_new, X_old );
    task_blockend();

    task_block( &ors, &root, tcost, &genOrs );
      *(bool*)exe_mydata() = *(bool*) exe_preddata( 0 ) ||
        *(bool*)exe_preddata( 1 );
    task_blockend();

    task_block( &final, &root, tcost, NULL );
      notdone = *(bool*) exe_preddata( 0 );
    task_blockend();

    rel_basic(&loop2, &sums,  ccost, succRAW, &seedtree);
    rel_basic(&loop1, &ors,   ccost, succRAW, &seedtree);
    rel_basic(&sums,  &sums,  ccost, succRAW, &formtree);
    rel_basic(&ors,   &ors,   ccost, succRAW, &formtree);
    rel_basic(&sums,  &loop1, ccost, succRAW, &sumsTOloop1);
    rel_basic(&loop2, &loop1, ccost, succRAW, &loop2TOloop1);
    rel_basic(&loop1, &swap,  ccost, succRAW, &loop1TOswap);
    rel_basic(&ors,   &final, ccost, succRAW, &orsTOfinal);
  task_rootend( &root );
} while( notdone );
sch_join( &root );
```

Figure 4.7: The code to produce a fine-grained TDG for the Jacobi iterative system solver in both the sparse and dense cases.

for each row of the linear system. If all rows exist in the matrix (which we will assume for this example), the task is regular and the code in Figure 4.8 is sufficient. The inner loop (`loop2`) is more complex if the matrix is sparse. The number of tasks (as long as the matrix representation allows it) is known by simply examining the matrix,

but the data for the tasks must be filled in by iterating through the structure. The run-time environment uses this data when executing the tasks, when determining relationships, and when instantiating subtasks. The reduction trees are instantiated as well, and are equally simple. Both the summing tree (`genSums`) and the or tree (`genOrs`) instantiate tasks based on the data given to the root task descriptor.

```
void gen1( int parentID, void *data, task_type* ) {
  SparseMatrix *matrix=(SparseMatrix*) data;
  gen_maketasks( /* size */, /* rows */ );
}
void gen2( int parentID, void *data,
           task_type *loop2 ) {
  SparseMatrix *matrix = (SparseMatrix*) data;
  int row = query_localid( parentID );
  unsigned tg = gen_maketasks( /* size */, /* cols */ );

  for ( /* each col */ )
    if ( /* diagonal element */ ) {
      loop1_t* data =
        (loop1_t*) query_data( parentID );
      // Store diagonal in 'data'
    } else {
      int id = query_globalid( loop2, tg, i++ );
      loop2_t *data = (loop2_t*) query_data( id );
      // Store task information in 'data'
    }
}
void genSums( int parentID, void *data,
              task_type* ) {
  SparseMatrix *matrix = (SparseMatrix*) data;
  int row = query_localid( parentID );
  if ( /* need sum tree */ )
    gen_maketasks( /* size */, /* cols-1 */);
}
void genOrs( int parentID, void *data,
             task_type* ) {
  SparseMatrix *matrix = (SparseMatrix*) data;
  gen_maketasks( /* size */, /* rows-1 */ );
}
```

Figure 4.8: Instantiation functions for the tasks shown in Figure 4.7.

The task bodies for the Jacobi solver are shown in Figure 4.9. These tasks are extremely fine-grained with very few instructions in each, making them prime candidates for optimization via the transformations given in Section 3.5.

```
void loop1_body(double* vector, double* X_new,
                double* X_old, double tol)
{
  loop1_t *task = (loop1_t*) exe_mydata();
  size_t row = exe_mylocalid();

  double sum = exe_numpred() > 0 ?
    *(double*)(exe_preddata(0)) : 0.0;

  X_new[row] = ( vector[row] - sum ) / *task->diag;
  task->notdone = fabs(X_new[row] - X_old[row]) > tol;
}
void loop2_body(double *X_old)
{
  loop2_t *task = (loop2_t*) exe_mydata();
  task->result = *task->val * X_old[task->index];
}
void sumbody()
{
  *(double*) exe_mydata() =
    *(double*) exe_preddata(0) + *(double*) exe_preddata(1);
}
```

Figure 4.9: Task bodies for the tasks show in Figure 4.7.

The final piece to the ADOPAR representation for the Jacobi algorithm is shown in Figure 4.10, which lists the relationship instantiation functions for each edge in the TDG. In two cases (`seedTree` and `formTree`), the functions may be shared between two relationship descriptors (`loop2` to `sums` and `loop1` to `ors`, for example). These two instantiation functions create a binary reduction tree. The operation being performed is irrelevant, so other tasks may reuse the code. For

the Jacobi algorithm, the actual task instantiations themselves are not needed to determine any relationship; only the number of instantiations is relevant.

The Gauss-Seidel algorithm strongly resembles the Jacobi algorithm, so the graph is largely the same. Three changes must occur:

1. Add an edge from `loop1` to `loop2` to enforce ordering.
2. Modify the `loop1` task body to use the correct ordering.
3. Remove the swap task and associated edge as it is not part of the algorithm.

The additional edge is specified in Figure 4.11. The instantiation function for the addition differs from those previously shown in that it specifies the *predecessors* of a task, rather than the successors, providing efficiency and convenience when operating with sparse matrices. Specifying successors involves iterating over all possible tasks and selecting the appropriate tasks, which is less efficient than specifying predecessor relationships. Using predecessors is more convenient because of the hybrid regularity/irregularity between the task descriptors. The task body is very similar to the Jacobi algorithm, but operates on a single matrix. The modified TDG construction code is not provided, as it only involves trivial changes.

The Gauss-Seidel algorithm reuses calculated values within the same iteration. A serial algorithm may be written so that the new values are calculated before they are used, and the main loop will look only slightly different than the Jacobi, where this restriction does not exist (compare Figure 3.13 and Figure 3.16). The extra edge within the TDG simply enforces this execution order based on the coordinates within the matrix. The relationship instantiation function in Figure 4.11 simply determines whether the outer loop is in a previous iteration and creates an appropriate edge.

4.6 Summary

The ADOPAR IR has been successfully implemented in the LLVM compiler framework. This implementation retains the flexibility of the task graph through a variety of intrinsics and provides a set of passes to aid in analysis and manipulation

```
void seedtree( int ID, void*, task_type*,
               task_type* ) {
  int localID = query_localid( ID );
  unsigned tg = query_taskgroup( ID );
  gen_makerel( tg, localID / 2, ccost, succRAW );
}
void formtree( int ID, void*, task_type *tree,
               task_type*) {
  int localID = query_localid( ID );
  unsigned tg = query_taskgroup( ID );
  int numtasks = query_numtasks( tree, tg );
  gen_makerel( tg, (localID + numtasks + 1) / 2,
               ccost, succRAW );
}
void sumsTOloop1( int ID, void*, task_type *sums,
                  task_type* ) {
  int localID = query_localid( ID );
  unsigned tg = query_taskgroup( ID );
  if ( /* final sum task */ )
    gen_makerel( 0, tg, ccost, succRAW );
}
void loop2TOloop1( int ID, void *data, task_type*,
                   task_type* ) {
  SparseMatrix *matrix = (SparseMatrix*) data;
  int row = query_taskgroup( ID );
  if ( /* no sum tasks */ )
    gen_makerel( 0, row, ccost, succRAW );
}
void loop1TOswap( int, void*, task_type*,
                  task_type* ) {
  gen_makerel( 0, 0, ccost, succRAW );
}
void orsTOfinal( int ID, void*, task_type *ors,
                 task_type* ) {
  int localID = query_localid( ID );
  unsigned tg = query_taskgroup( ID );
  if ( /* final or task */ )
    gen_makerel( 0, 0, ccost, succRAW );
}
```

Figure 4.10: Relationship instantiation functions for the tasks shown in Figure 4.7.

```
void loop1TOloop2( int ID, void*, task_type*,
                   task_type* ) {
  loop2_t *data = (loop2_t*) query_data( ID );
  unsigned tg = query_taskgroup( ID );
  if ( tg > data->col )
    gen_makerel ( 0, data->col, ccost, predRAW );
}
```

Figure 4.11: An additional relationship is needed to synchronize the Gauss-Seidel calculations. A predecessor relationship is convenient to represent a sparse matrix calculation. The task body also receives some minor adjustments to work with the Gauss-Seidel algorithm.

of the TDG. The following chapter evaluates the ADOPAR representation, how well it works with various programming models, and the overhead of the implementation presented here.

Chapter 5

Evaluation

This work has presented a method for representing a task graph through task and relationship descriptors and their instantiation functions. The Task Descriptor Graph (TDG) is a powerful concept that allows for creation of task graphs using information only available at run time while also providing a framework for static analysis. The TDG is a modification of the well-studied task graph, but the graph is compressed to aid analysis, reduce code size, and minimize execution overhead.

The purpose of this chapter is to examine this new representation and evaluate it against the requirements of ADOPAR as presented in Chapter 1. The overhead is measured for compilation time, code size, and the Task Instantiation Graph (TIG) instantiation time.

This chapter will first explain the experimental methodology, including the criteria, metrics, and benchmarks used in the experiments. The results from these experiments follow, including an analysis of each.

5.1 Methodology

In the absence of a complete toolchain geared toward the ADOPAR representation and a mature back end to schedule and execute the TIG it represents, this work must rely on less direct methods for evaluating the representation. There are several preliminary ADOPAR schedulers. These schedulers will be used to demonstrate that the Internal Representation (IR) functions correctly, although demonstrating overhead relating to the schedulers will not be possible. Also, we must hand-code the representation (as in Chapter 4). We can then evaluate the results. First, we show that the representation fulfills the presented requirements of ADOPAR. Second, we

examine the compiled code size, showing space used by the additional information that is available. Third, we measure the execution overhead involved in instantiating the TDG. Finally, we show the performance of various schedulers that operate directly on the TIG.

5.1.1 Criteria

The internal representation presented in this work has attempted to satisfy the needs of ADOPAR, namely:

- support for irregular structures,
- support for regular structures,
- simple communication description, and
- low overhead.

The IR fully supports regular and irregular structures using task and relationship descriptors. In either case, the number of tasks, their parameters, and their relationships are expressed algorithmically and in whatever way is best suited to the application.

Communication is described through the relationship descriptors as edges in the TDG. Relationship descriptors support regular and irregular communication patterns through the same algorithmic means. Various types of relationships are supported to express more than just the mere presence of a relationship.

The overhead of this representation (compared to a serial version where parallelism is just implied) comes in many forms: additional code to express the TDG structure and instantiation functions, extra compilation time to prepare the IR for execution, and extra time to execute the program. The code size and compilation time overheads are mostly a concern at compile time, but the effects do spill over at run time due to the Just-In-Time compiler (JIT) environment. Chapter 6 will describe several ways of reducing the run-time overhead.

Running within a JIT environment allows the representation to pass dynamic information to ADOPAR. More importantly, any information that is only available at run time (especially the relationships between tasks) may reveal optimizations to the TDG, reducing the overhead of the representation and improving the schedule.

5.1.2 Metrics

This work uses four main metrics to quantify the overhead of the IR. These metrics are:

Code Size LLVM stores a program in a *bitcode* format. The ADOPAR IR describes tasks through additional intrinsics and information overlaid on the bitcode. Using this representation increases the size of the bitcode file. This overhead is measured for each benchmark by comparing the total executable file size to the equivalent serial version.

Compilation Time The IR requires extra time to transform the ADOPAR intrinsics into a form the compiler can use directly. Furthermore, the IR also requires extra time to compile and optimize the extra bitcode. The compilation time is measured with the standard optimizations turned on (`-O2`). As with code size, the extra compilation time is compared to the equivalent serial version.

Absolute Instantiation Time Fully instantiating a task graph can become an expensive process, especially for fine-grained task models (where there may be hundreds of thousands of task relationships). This metric quantifies the effects of the size of the task graph on the instantiation time.

Relative TIG instantiation time The cost of instantiating the task graph must be considered against the execution time of the program. Measuring this ratio shows the speedup and reuse required to overcome the overhead of the representation.

5.1.3 Benchmarks

Since the representation must currently be hand-coded, all measurements are performed on the Jacobi iterative linear system solver described in Chapter 4. This benchmark is a simple kernel with excessively fine-grained tasks. The overheads presented here show a high estimate with a large number of very small tasks and no optimizations. Applications with much coarser task granularities will create fewer tasks and relationships and will spend more time in task bodies (doing real work) while executing. This chapter also uses a more sensible (and therefore, more realistic) coarse granularity version of the Jacobi benchmark for a more complete picture. Future optimizations should be able to make the transformation to the coarse version.

One of the goals of ADOPAR is to support those applications that have regular patterns as well as those that are more irregular, with the scheduler acting appropriately in each case. The fine and coarse solvers have a sparse and a dense version to show how the IR deals with both scenarios. The dense versions allow for more compact representations and faster instantiation time; sparse versions show a partially irregular application that requires more processing to instantiate.

5.2 Measurements

This section presents the experimental data collected to quantify the overhead in the ADOPAR IR.

5.2.1 Code Size Overhead

Using the ADOPAR IR involves inserting additional code into the program. The additional code increases the executable size and has implications on the compilation, analysis, and execution of the program. Figure 5.1 shows the measurements of the additional overhead introduced by the representation.

Two major factors are at work here: the effect of a sparse versus dense matrix representation and the effect of a coarse-grained versus fine-grained computation. Neither of these factors makes much difference in the code size of the serial executable, but the difference is significant for the ADOPAR representation. First, a

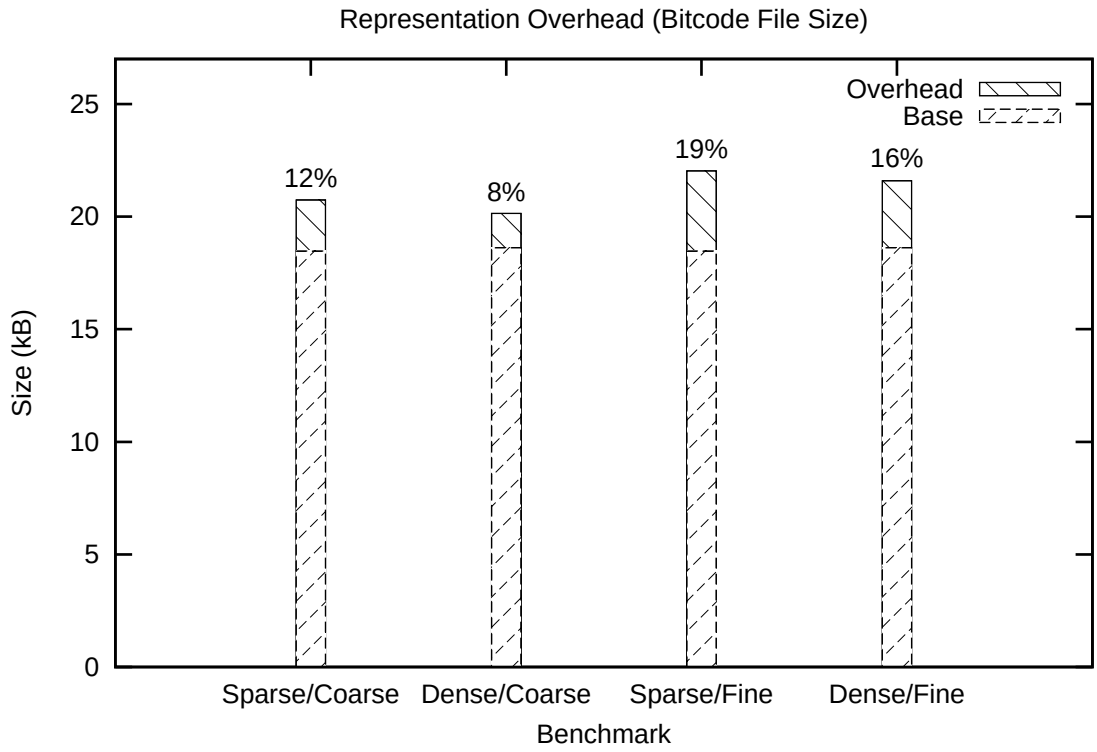


Figure 5.1: The extra code size (as LLVM bitcode) introduced by the ADOPAR IR compared to the equivalent serial implementation.

sparse matrix requires additional logic to instantiate the appropriate TIG. Second, the granularity affects the number of descriptors in the TDG and the complexity of the instantiation functions. Figure 5.1 demonstrates both of these effects clearly, and shows that the effect is cumulative: the overhead for a sparse, fine-grained computation model requires an additional 19% overhead in size.

However, we cannot consider the additional code size as pure overhead. In fact, the representation encodes information either not available or only implicitly found (after extensive analysis) in a serial application. It would be better to compare with a serial version that includes this information, a feature not available.

5.2.2 Compilation Time Overhead

Compiling the extra code and processing the IR takes extra time. As with the code size, processing the IR is mainly a one-time static cost. Figure 5.2 shows the compilation time overhead for the same benchmarks.

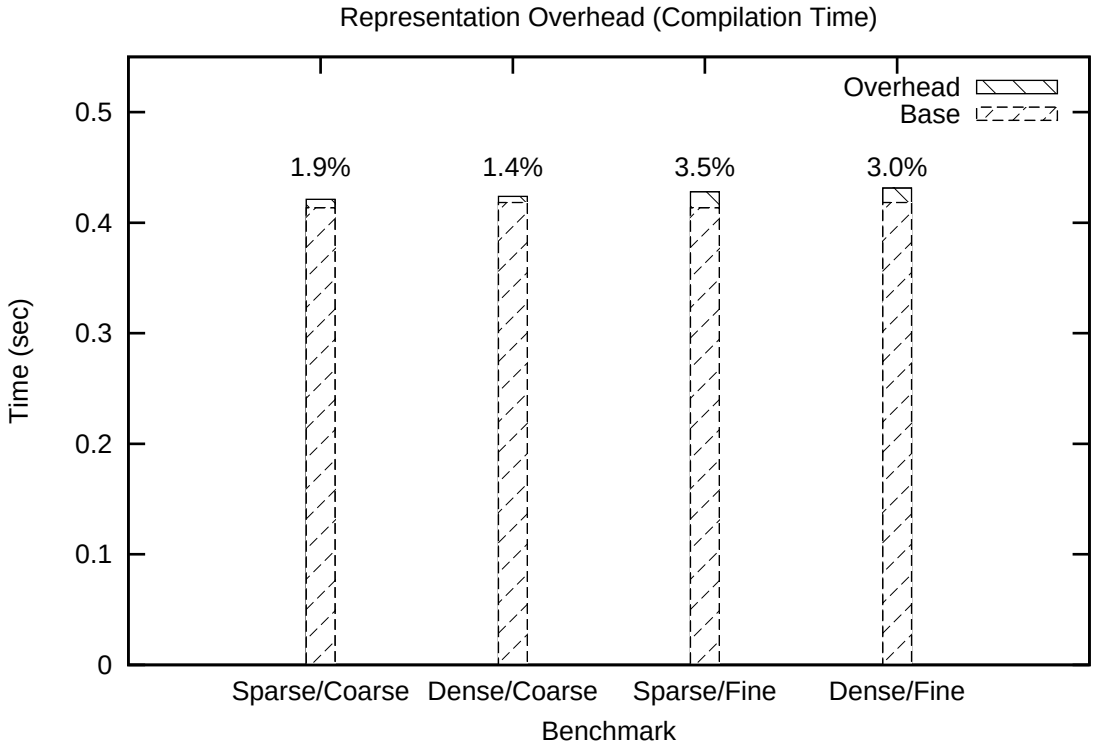


Figure 5.2: The extra compilation time introduced by the ADOPAR IR compared to the equivalent serial implementation.

In addition to the additional code size, the representation also requires some analysis and transformation. Figure 5.2 also indicates the time to perform these transformations. While some of the transformations are not strictly necessary (more specifically, lowering the intrinsics so that TIG instantiation is part of the execution),

we have included them to place an upper bound on the overhead. The three additional steps (implemented as LLVM passes) are:

1. conversion from C function calls to intrinsics,
2. promotion of block tasks to function tasks, and
3. lowering of intrinsics to instantiate the TIG at run time.

For the most part, the IR overhead does not add a significant amount of compilation time. For the coarse-grained model, the extra time is especially minimal. In addition, using sparse matrices does not affect the compile time significantly because the outer task descriptor is the same; only the task body changes significantly. The fine-grained model adds more overhead to the compilation time, especially the sparse version which has more complicated relationships and instantiation functions. In the worse case (fine grained and sparse), the total overhead is a 3.5% increase.

5.2.3 Task Instantiation Overhead

Having both instantiated task graphs and task descriptors imposes a run-time overhead on applications that use the IR. While it may not be necessary to fully instantiate the task graph in the future, the current implementation does so, leaving schedule descriptors and other overhead-reducing options for later work.

The amount of overhead incurred when translating the TDG to a TIG is notable. Figure 5.3 shows how the number of tasks affects the instantiation time. The nature of brute-force TIG instantiation requires at least one examination of each task group, with each task reviewed if any task-specific parameters are required. The instantiation process must also examine all instantiated relationships, which is also a linear process, leading to the $O(n)$ behavior in Figure 5.3. Each benchmark, regardless of the granularity and matrix density, presents this same overall behavior; however, there is a significant additional cost for using a fine granularity or sparse matrix for small problem sizes where the task instantiation, rather than relationship instantiation, has more influence.

Notice that the graph is based on the number of tasks, not the overall problem size: sparse matrices have an inherent algorithmic optimization that varies based on the density. This factor is reduced by comparing equal numbers of tasks. The model used for each benchmark restricts the range of tasks used to produce these figures; the Dense/Coarse model can compute a large problem size with few tasks while the Dense/Fine model requires many more tasks to operate.

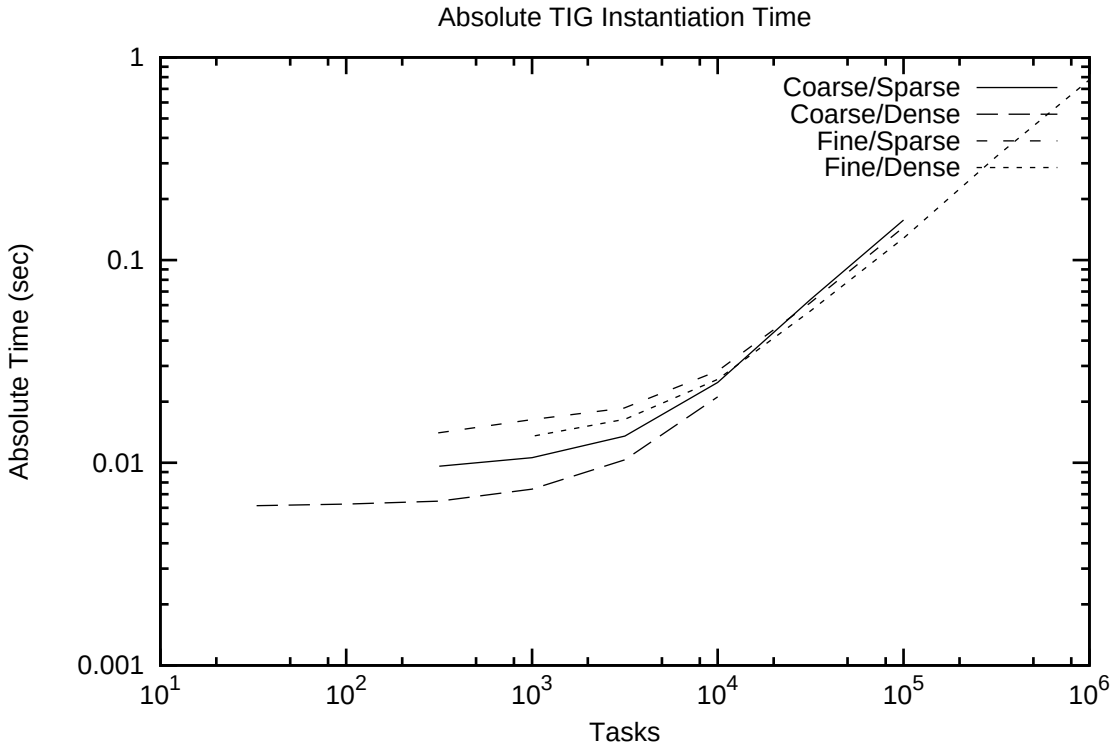


Figure 5.3: The absolute instantiation time for various problem sizes.

We must also examine the cost of instantiating the graph relative to the execution time of the program. Such comparison presents a rough idea of the speedup necessary to overcome the representation overhead. Figure 5.4 shows the instantiation overhead relative to the average execution of a single iteration from the equivalent serial algorithm. The iteration time was averaged over many iteration times to reduce caching effects. Increasing the parallel speedup (through advanced scheduling

techniques and TDG manipulation, for example) and reusing the schedule over many iterations will mitigate this overhead. In fact, we anticipate that many applications will reuse the schedules thousands of times, even for different data sets.

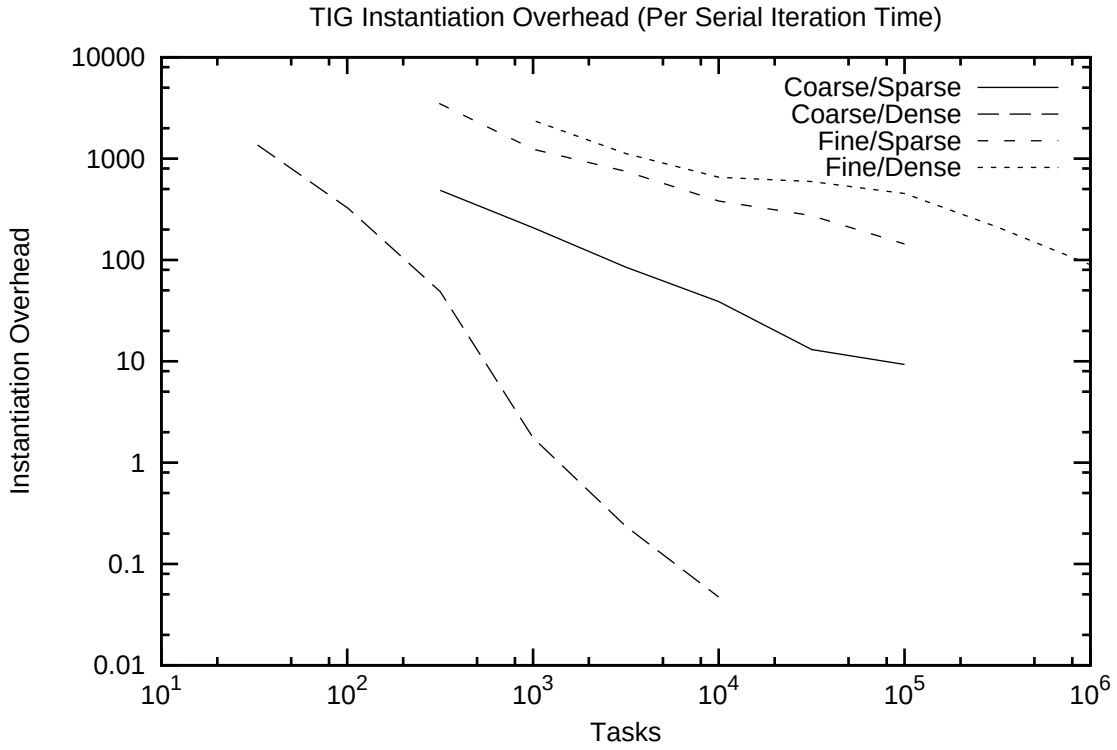


Figure 5.4: The instantiation time relative to the average serial iteration time.

We have already established the instantiation time behavior. The average serial iteration time for the equivalent problem size actually grows at a faster rate as the amount of work increases as well. This is easiest to see in the case of a coarse-grained matrix. The model uses a single task for each row, but each task must compute a matrix-vector multiply for that entire row. As a result, the instantiation time is less significant compared to the amount of work done in each iteration. In the end, less speedup and fewer iterations are required to overcome the full instantiation overhead of the Jacobi algorithm as the problem size increases.

For fine-grained task models, the cost of instantiating the task graph can be quite high relative to the execution time. However, it is expected that ADOPAR will not need to fully instantiate the task graph in the future.

5.2.4 Scheduling Results

ADOPAR currently lacks a front end to create its IR. Current schedulers must use a fully instantiated TIG. However, the hand-coded benchmarks presented here are compatible with this limited setup. Figure 5.5 shows the results of running the sparse, coarse-grained Jacobi solver with various schedulers currently in development.

The schedulers were run on a system with eight 2.2 GHz AMD Opteron 8354 quad-core processors (for a total of 32 cores) with a 512 KByte L2 cache and a 2 MByte L3 cache per processor. The experiment measured the relative speedup of task graph execution time compared to a single-threaded run. The benchmark used is the sparse, coarse-grained Jacobi linear system solver on a 4000x4000 random sparse matrix at 2% density. We can see the improvement that various schedulers can provide to the execution time.

Each scheduler has the same basic properties: as more threads are available, more work can be done in parallel. Speedup can actually be superlinear as the number of processing cores and the available amount of cache increases. Additional cache reduces the number of capacity misses compared to the single-threaded version. As the number of threads increases to very large numbers, the additional inter-thread communication costs outweigh the improved cache behavior and additional threads harm performance. Some schedulers are more able to overcome the effects, but locality and proper scheduling remain critical to parallel performance. Ongoing development work seeks to reduce the number of threads when additional threads harm performance [35].

A discussion of the properties and behaviors of these schedulers is beyond the scope of this work. However, it should be noted that the ADOPAR schedulers use information from the IR to optimize threading. The success of the scheduler depends on many factors. In general, the more information that is available, the better the

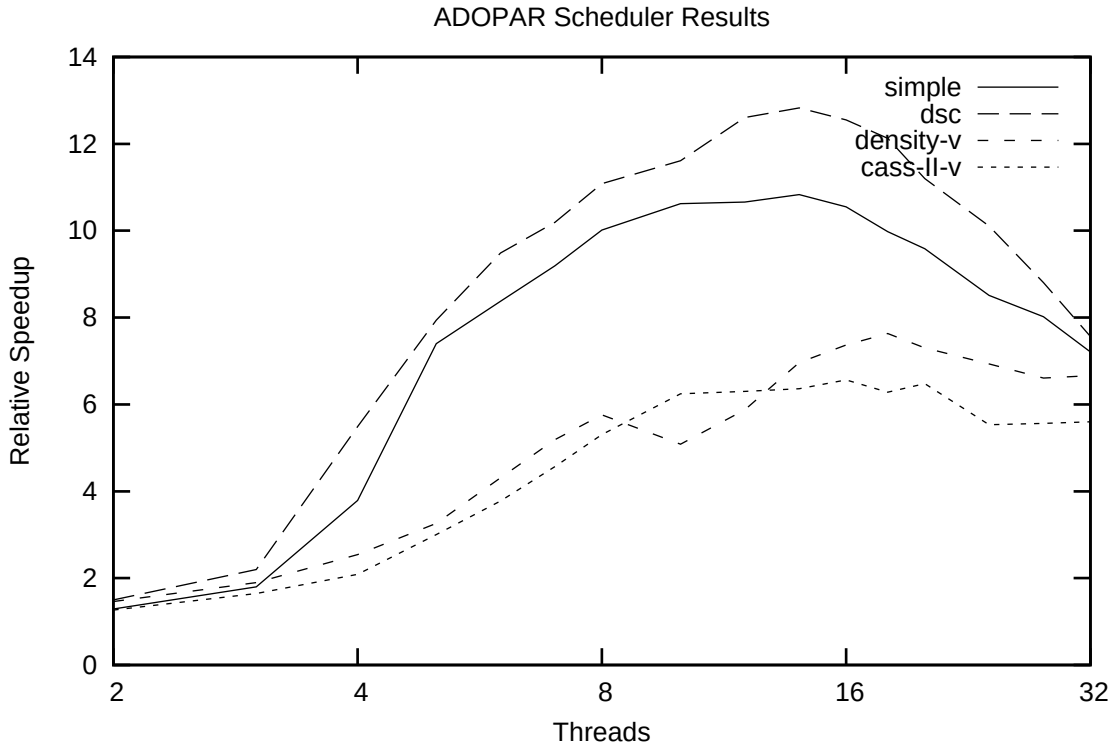


Figure 5.5: The results from preliminary ADOPAR schedulers, demonstrating that the representation may interface with the schedulers to perform parallel calculations.

schedule can be. Specifically, the representation makes the following available to the schedulers:

- task cost (which varies based on the density of the matrix row),
- communication cost between tasks on different processors,
- cost of sharing data, and
- ordering dependencies.

5.3 Summary

Task and relationship descriptors allow a front end to describe task graphs in a complex form with analyses and graph structures summarized in an arbitrary algorithm. While future schedulers may operate directly on the TDG, current schedulers require an explicit TIG. Thus, using a TIG can be an expensive process when

large numbers of fine-grained tasks are represented, it still allows creation of simple schedulers.

The representation also adds additional overhead in terms of code size and compilation time. The amount of additional overhead varies depending on the amount of detail encoded in the representation (such as task granularity and relationships), but is generally low compared to a serial application. Much of this overhead is not only acceptable but is desired: additional information is being provided in the representation that was not available previously.

These data indicate that the IR fills the basic needs of ADOPAR, including support for regular and irregular structures so that even complex fine-grained task graphs may be represented compactly, with information only available at run time made visible to ADOPAR.

Chapter 6

Conclusion

This work has presented a new method for representing parallelism using graphs of task and relationship descriptors. Instantiation functions provide a generic mechanism to arbitrarily specify tasks and their relationships. We have not fully explored the full potential of the ADOPAR Internal Representation (IR). This chapter looks to the future of ADOPAR and the improvements and applications of its IR.

6.1 Summary of Results

The previous chapters have shown the parallelism information that may be represented by ADOPAR's IR. This information is available in the form or a compressed task graph using task and relationship descriptors, enhanced by the flexibility and expressiveness of instantiation functions. Preliminary schedulers have the ability to work with this information to create parallelized code.

The program does see an increase in file size, mainly due to the additional parallelism information that is included in the representation. Compiling the program sees a negligible time increase. However, because current schedulers cannot work with this representation natively there is a significant overhead involved to translate the Task Descriptor Graph (TDG) into a Task Instantiation Graph (TIG). While future work will hopefully eliminate this step, it will be necessary until native schedulers have been written.

6.2 Future Work

This research presents only a portion of how front ends, back ends, schedulers, programmers, and architectures may use task and relationship descriptors. Future

work may investigate the many improvements and extensions that build on to the ADOPAR IR or use its features. Some features are currently under investigation, while others are currently only theoretical. Each is a natural extension made possible by the challenges and features made available by the new IR.

6.2.1 Representation

The representation itself offers several possibilities for future investigation on the use of task descriptors and various extensions:

Additional Task Types This work only presents the basic set of tasks that are useful in the situations we have encountered: empty tasks, block tasks, and function call tasks. We have also added root tasks and substitution tasks for their utility in building the TDG. Some applications may benefit from tasks that represent explicit threads (a task which executes on its own processor without any other tasks and provides its own synchronization) and message tasks (providing a back end to Message Passing Interface (MPI) and similar libraries). New task types may provide for special-case situations that ease construction of the ADOPAR IR for various programming models.

Parallel Instantiation While explicit TIG instantiation is not required in the general case, it may still be prudent to perform this step for highly irregular computations. Multithreading the instantiation function will dramatically reduce the overhead of explicit instantiation. While the TDG may be cyclic, the hierarchy is not. The hierarchy graph can be scheduled for parallel execution using the same ADOPAR schedulers used on the TIG.

Additional Locality Information Current ADOPAR research [10] suggests that locality (especially temporal locality) effects have a critical influence on performance. Incorporating additional information into the representation is vital when optimizing the schedule. The current concept for representing most locality information is through relationship edges in the TDG; however, other schemes may better describe various types of locality information. Data Access

Descriptors (DADs), Regular Section Descriptors (RSDs), and Processor Tagged Descriptors (PTDs) may provide such a mechanism, although it is possible to incorporate them into the relationship descriptor instantiation functions.

Iterative Scheduling The current implementation of the IR assumes a separate scheduling and iteration model: the scheduler runs once while the schedule is reused many times. However, the initial schedule may not be optimal and the run-time environment may find additional parallelism and speedup with continued work. The scheduler may run on an unused processor, if available. Future work may examine adaptations to the representation to support iterative scheduling.

Non-Static Task Graphs The ability of a program to modify its task graph is closely related to iterative scheduling, partly due to the related execution model and the necessary representation extensions. For example, a molecular dynamics simulator might model a chemical reaction that changes the bonds, simulated as a change in the relationships between tasks and possibly the creation of additional tasks. Changes to the task graph are currently supported by completely rescheduling the task graph, a highly inefficient operation considering the potentially small change. Iterative scheduling mechanisms may also improve performance for non-static graphs. The extent of modification could be limited to the TIG rather than the TDG, further simplifying the process.

Schedule Descriptors The nature of task and relationship descriptors implies that other descriptors may help in the scheduling process. Schedule descriptors would create specialized scheduling determined by static analysis but parameterized with run-time information. As with task and relationship descriptors, schedule descriptors provide specific information to the run-time environment. In this case, the exact task ordering and partitioning is provided, rather than instantiations of tasks.

Native Schedulers Schedulers do not necessarily need the run-time environment to create a full TIG, although all the current ADOPAR schedulers do. Future research will create schedulers that can operate directly on the task and relationship descriptors, fully realizing the potential of the representation. Native schedules might run at compile time – creating schedule descriptors – or directly at run time.

6.2.2 Front Ends

There are currently no front ends that use ADOPAR as a back end. An extensive amount of future work is possible in the implementation of various programming models and translating this information to the appropriate internal representation. Some interesting examples include:

OpenMP A simple, widely used programming model [22] that mainly uses loop-based parallelism and simple synchronization. Translating between OpenMP and the ADOPAR IR is relatively simple: the fork/join paradigm is very similar for both, and OpenMP's loop-based parallelism model maps to simple task descriptors and sets of task trees.

High Performance FORTRAN (HPF) and FORTRAN 95 Many extensions have been provided to parallelize FORTRAN. OpenMP is common and may specific keywords exist to describe loop-based parallelism (such as FORALL). HPF [36] also has directives to distribute array data, information that ADOPAR would express as part of the instantiation functions.

C/C++ Many libraries and language extensions exist to parallelize C, C++, and related languages. The ADOPAR IR could be used by various front ends, creating relationship descriptors to summarize the results of alias analysis or other useful information. When analyzing code, the compiler must solve complex sets of equations to determine the relationship between code segments. Run-time dependencies might prevent successful analysis, but the front end can package

the remaining unsolved equations as relationship instantiation functions to be executed to test for aliasing.

MPI Message passing programs will probably require extensions to the representation. A front end might represent MPI and similar paradigms [21] as message task, with messages represented edges between them.

Spreadsheets Spreadsheet calculations translate cleanly into the ADOPAR IR. The dependencies between cell formulas create a Directed Acyclic Graph (DAG) and calculations are often repeated. A cell formula, then, becomes a task descriptor when translating to the ADOPAR IR. The equivalent of duplicating the calculation is the instantiation function, and cell dependencies become relationship descriptors.

6.2.3 Analysis and Optimization

This work presents an internal representation that ADOPAR can analyze and optimize. Future work may examine the utility of the representation in the optimization process. Several possible improvements for analysis and optimization include:

Descriptor Analysis In general, relationship and task descriptors contain a simple algorithmic relationship between tasks. The compiler could analyze relationships using techniques such as partial evaluation [32]. The result of the analysis is a native understanding of the front end's intent. Native schedulers and scheduler generators would require descriptor analysis.

Task Relationships While our research examined some of the properties of task relationships, this work treated Read After Write (RAW), Write After Read (WAR), and Write After Write (WAW) relationships as simple sequencing dependencies and made no particular distinction between them. Exclusivity relationships could likewise be split into read exclusivity and write exclusivity. Future research should explore each relationship type in the process of creating schedulers and optimizers.

Dynamic Optimization Run-time feedback of task execution times provide insight into the actual properties of tasks. Some information is impossible to know when scheduling, such as the actual task execution and data communication costs. Iterative solvers may make use of measured task execution times and communication costs to improve the schedule, updating the estimates already contained in the IR.

6.3 Summary

There are many opportunities for future study of the representation presented in this work. More features will become necessary as ADOPAR continues to develop. These features will provide true front ends, real applications, lower overhead, and tighter integration into the scheduling environment. We expect the representation to grow alongside ADOPAR's scheduling environment.

6.4 A Final Word

The realities of manycore computing create opportunities for many different fields of research, from the design of processors and memory to the creation of programming strategies that use them. The naturally occurring diversity of architectures, programming models, and operating environments should not impede these opportunities. Instead, each difference brings an advantage that improves performance and simplifies the creation of hardware and software.

ADOPAR improves performance and simplifies diversity management through adaptive run-time optimization of software parallelism, but requires more information than is traditionally available. The IR implemented here aids ADOPAR through a flexible and descriptive design. The flexibility supports a wide variety of programming models and applications. A diverse set of operating environments requires that it provide sufficiently detailed information to the ADOPAR schedulers. We have successfully implemented this representation using compiler intrinsics, providing the described task graph to various schedulers and then performing computations in multiple threads.

Bibliography

[1] Semiconductor Industry Association (SIA), "International technology roadmap for semiconductors," 2007. [Online]. Available: http://www.itrs.net/

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proceedings of the 7th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 2–11.

[3] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[4] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.

[5] J. Saltz, R. Ponnusamy, S. D. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, and R. Das, "A manual for the CHAOS runtime library," College Park, MD, USA, Tech. Rep., 1995.

[6] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *Journal of Parallel and Distributed Computing*, vol. 5, no. 5, pp. 517–550, 1988.

[7] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen, "The Polaris internal representation," *International Journal of Parallel Programing*, vol. 22, no. 5, pp. 553–586, 1994.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[9] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979–993, September 1993.

[10] D. A. Penry, "Multicore diversity: A software developer's nightmare," *Operating Systems Review*, vol. 43, no. 2, pp. 100–101, Apr 2009.

[11] D. A. Penry, "You can't parallelize just once: Managing manycore diversity," in *Manycore Computing Workshop*, June 2007.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995, pp. 251–266.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[15] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, May 1991.

[16] S. J. Fink, S. B. Baden, and S. R. Kohn, "Efficient run-time support for irregular block-structured applications," *Journal of Parallel Distributed Computing*, vol. 50, no. 1-2, pp. 61–82, 1998.

[17] J. H. Merlin, S. B. Baden, S. J. Fink, and B. M. Chapman, "Multiple data parallelism with HPF and KeLP," in *HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*. London, UK: Springer-Verlag, 1998, pp. 828–839.

[18] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," *Proceedings of the SIGPLAN '87 symposium on Interpreters and interpretive techniques*, vol. 24, no. 7, pp. 41–53, 1989.

[19] E. Su, D. J. Palermo, and P. Banerjee, "Processor tagged descriptors: A data structure for compiling for distributed-memory multicomputers," in *Proceedings of the 3rd International Conference on Parallel Architectures and Compilation Techniques*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1994, pp. 123–132.

[20] A. Lain, "Compiler and run-time support for irregular computations," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1996.

[21] D. R. Chakrabarti, P. Banerjee, and A. Lain, "Evaluation of compiler and runtime library approaches for supporting parallel regular applications," in *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 1998, p. 74.

[22] *OpenMP Application Program Interface.* OpenMP Architecture Review Board, 2005.

[23] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[24] T. Johnson, "A concurrent dynamic task graph," in *Proceedings of the 1993 International Conference on Parallel Processing.* Washington, DC, USA: IEEE Computer Society, 1993, pp. 223–230.

[25] C. D. Polychronopoulos, "The hierarchical task graph and its use in auto-scheduling," in *Proceedings of the 5th International Conference on Supercomputing*, 1991, pp. 252–263.

[26] C. D. Polychronopoulos, "Toward auto-scheduling compilers," *Journal of Supercomputing*, vol. 2, pp. 297–330, 1988.

[27] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek, "'C: A language for high-level, efficient, and machine-independent dynamic code generation," in *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, 1995, pp. 131–144.

[28] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications," in *Proceedings of the 1999 International Conference on Supercomputing*, 1999, pp. 425–433.

[29] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, December 1974.

[30] R. Anderson, P. Beame, and W. Ruzzo, "Low overhead parallel schedules for task graphs," in *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1990, pp. 66–75.

[31] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, June 1990.

[32] M. Leone and P. Lee, "Dynamic specialization in the Fabius system," in *ACM Computing Surveys Symposium on Partial Evaluation*, vol. 30, no. 3es, Sep 1998.

[33] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006, pp. 29–40.

[34] D. A. Penry, "The acceleration of structural microarchitectural simulation via scheduling," Ph.D. dissertation, Princeton University, Princeton, NJ, November 2006.

[35] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs," in *Proceedings of the 13th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 277–286.

[36] *High-Performance Fortran Specification, Version 2.0.* High-Performance Fortran Forum, 1997.