Atomic Dataflow Model

Vladimir Gajinov

Department of Computer Architecture

Universitat Politècnica de Catalunya

Thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

Advisors:

Prof. Eduard Ayguadé

Dr. Osman S. Unsal

Dr. Adrián Cristal

Barcelona, October 2014

Abstract

With the recent switch in the design of general purpose processors from frequency scaling of a single processor core towards increasing the number of processor cores, parallel programming became important not only for scientific programming but also for general purpose programming. Inevitably, this also stressed the importance of programmability of existing parallel programming models which were primarily designed for performance. It was soon recognized that if the parallel programming is to become a mainstream, new programming models are needed that will make parallel programming possible not only to experts, but to a general programming community.

Transactional Memory (TM) is an example which follows this premise. It improves dramatically over any previous synchronization mechanism in terms of programmability and composability, at the price of possibly reduced performance. The main source of performance degradation in Transactional Memory is the overhead of transactional execution. Our work on parallelizing Quake game engine is a clear example of this problem. We show that Software Transactional Memory is superior in terms of programmability compared to lock based programming, but that performance is severely hindered due to extreme amount of overhead introduced by transactional execution.

In the meantime, a significant research effort has been invested in overcoming this problem. Mainly, the target of this effort was to improve the implementation of transactional memory. Our approach is different and aimed towards improving the performance of transactional code by reducing transactional data conflicts. The idea is based on the organization of the code in which highly conflicting data is promoted to dataflow tokens that coordinate the execution of transactions.

The main contribution of this thesis is Atomic Dataflow model (ADF), a new task-based parallel programming model for C/C++ that integrates dataflow abstractions into the shared memory programming model. The ADF model provides language constructs that allow a programmer to delineate a program into a set of tasks and to explicitly define data dependencies for each task. The task dependency information is conveyed to the ADF runtime system that constructs a dataflow task graph that governs the execution of a program. Additionally, the ADF model allows tasks to share data. The key idea is that computation is triggered by dataflow between tasks but that, within a task, execution occurs by making atomic updates to common mutable state. To that end, the ADF model employs transactional memory, which guarantees atomicity of shared memory updates.

The second contribution of this thesis is DaSH - the first comprehensive benchmark suite for hybrid dataflow and shared memory programming models. DaSH features 11 benchmarks, each representing one of the Berkeley dwarfs that capture patterns of communication and computation common to a wide range of emerging applications. DaSH includes sequential and shared-memory implementations based on OpenMP and TBB to facilitate easy comparison between hybrid dataflow implementations and traditional shared memory implementations based on work-sharing and/or tasks. We use DaSH not only to evaluate the ADF model, but to also compare it with other two hybrid dataflow models in order to identify the advantages and shortcomings of such models, and motivate further research on their characteristics.

Finally, we study applicability of hybrid dataflow models for parallelization of the game engine, which is the problem that motivated the work summarized in this thesis. We show that hybrid dataflow models decrease the complexity of the parallel game engine implementation by eliminating or restructuring the explicit synchronization that is necessary in shared memory implementations. The corresponding implementations also exhibit good scalability and better speedup than the shared memory parallel implementations, especially in the case of a highly congested game world that contains a large number of game objects. Ultimately, on an eight core machine we were able to obtain 4.72x speedup compared to the sequential baseline, and to improve 49% over the lock-based parallel implementation based on work-sharing.

Keywords: Computer Architecture, Parallel Programming, Dataflow, Transactional Memory, Atomic Dataflow Model, Game Engine, Benchmark

To my family, my girlfriend, my friends and colleagues.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Eduard Ayguadé and my co-advisors Osman Unsal and Adrián Cristal for their continuous support of my Ph.D study and research, for their guidance, enthusiasm, and expertise. I would also like to thank Mateo Valero for his dedication and continuous effort in making Barcelona Supercomputing Center such a great place for conducting quality research. Special thanks to Prof. Veljko Milutinović for helping me get to Barcelona in the first place.

A very special gratitude goes to my thesis committee: Dr. Paolo Meloni, Dr. Xavier Martorell and Dr. Miquel Pericas for their time, insightful comments and effort that they invested into making this thesis better.

I would like to thank Tim Harris, who kindly mentored me during my three-month stay in Microsoft Research Cambridge. I had a great and productive time there thanks to Tim's positive attitude and enthusiasm.

I would also like to acknowledge all my friends and colleagues from Barcelona Supercomputing Center that helped me throughout my PhD; for their insights and expertise in technical matters, and for their support that has been crucial to keep this part of my life memorable. Many thanks go to Adriá Armejach, Ana Jokanović, Azam Seyedi, Bojan Marić, Branimir Dickov, Chinmay Kulkarni, Cristian Perfumo, Daniel Nemirovsky, Erdal Mutlu, Ferad Zyulkyarov, Gokcen Kestor, Gülay Yalçin, Ivan Ratković, Javier Arias, Maja Etinski, Milan Pavlović, Milan Stanić, Milovan Djurić, Miloš Milovanović, Miloš Panić, Mladen Slijepčević, Nehir Sonmez, Nikola Marković, Nikola Rajović, Nikola Vujić, Oriol Arcas, Oscar Palomar, Paul Carpenter, Saša Tomić, Timothy Hayes, Uglješa Milić, Vasilis Karakostas, Vesna Smiljković, Vladimir Subotić, Vladimir Marjanović, Vladimir Čakarević and many others. I sincerely thank you all for your help and all the great moments we have had together.

These acknowledgments would not be complete without mentioning all the people that I have meet during my life in Barcelona: Ognjen Obućina who have shown me hidden gems of Barcelona and thought me parts of the history of this great city, Uros

Savićević who was always full of positive energy and readiness for having a good time, Neda Konstandinović who helped me see things from a different perspective when I was feeling negative, Miloš Čakarević and Rade Stanojević who were always there to have a drink or spend a night out looking for fun, Irina Blagojević, Nemanja Vučević, Jou Melo, Megan Heinley, Chesney McKenzie, Jelena Lazarević, Jelena Nikolić, Larry Cassius Araw-Roe, Desanka Bošković and many others whom I have had a pleasure to call my friends all these years. Thank you all!

Last but not the least, I would like to thank my family for the love and support they have provided me through my entire life. My deepest thanks to my girlfriend, Jelena Koldan, without whose love, encouragement and understanding I would not have finished this thesis.

In conclusion, I recognize that this research would not have been possible without the financial assistance of Barcelona Supercomputing Center and BSC-Microsoft Research Center, Government of Catalunya through FIE scholarship, European Commission through the HiPEAC-3 Network of Excellence and the VELOX FP7 project, and the Spanish Ministry of Education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The advent of multicore processors pushed concurrent programming into the focus of recent research. While concurrency has been a research topic for past few decades, it is widely recognized today that if parallel programming is to become mainstream, the research community has to provide new parallel programming models with high level concurrency constructs. In this thesis, we present Atomic Dataflow model, which purpose is to enable easier and more efficient way to write parallel programs on current and future multicore systems. We start this chapter by explaining the motivation for this work and organization of this thesis, and then we provide the background on main topics relevant to our work.

1.1 Motivation and Thesis Organization

This thesis starts by evaluating the programmability of transactional memory (TM) for parallelization of the Quake game engine that represents a complex sequential application. Since the main purpose of TM is to make parallel programming easy by abstracting away the complexities of using locks, the goal of this work was to discover if it is possible to obtain good performance with a coarse-grained parallelization approach using TM. Although resulting parallel implementation scales, we show that a coarse-grained approach is not applicable due to a high overhead and abort rate of the underlaying transactional memory system. Furthermore, we also describe a fine-grained parallel implementation based on TM that provides better performance, but overall the TM related problems still remain. While the instrumentalization overhead of TM systems is a more pressing problem, however one that is related to the implementation of a TM system, we were primarily concerned in ways to decrease the abort rate of parallel TM implementations by reducing the number of conflicts between concurrent transactions. In particular, our motivation was to decrease synchronization requirements of parallel TM implementations by explicitly coordinating the execution of dependent transactions. The idea is based on the organization of the code in which highly conflicting data is promoted to dataflow tokens that orchestrate the execution of transactions. The gain is two-fold: a transaction is differed until the input dependency necessary for its execution has been satisfied, and at the same time the probability that the execution will fail is decreased due to a reduction in a number of accesses to the data that can cause a conflict. However, providing such support as an extension to TM systems is not a good solution since it breaks composability, which is one of the most important characteristics of transactional memory. Therefore, we were inspired to look for better ways to solve the problem at hand.

Accordingly, in Chapter 3, we describe a more general solution that provides coordination between transaction on a higher level of abstraction. Specifically, we present Atomic Dataflow model (ADF) – a new task-based parallel programming model that integrates dataflow abstractions into the shared memory programming model. The ADF model provides language constructs that allow a programmer to delineate a program into a set of tasks and to explicitly define data dependencies for each task. The task dependency information is conveyed to the ADF runtime system that constructs a dataflow task graph that governs the execution of a program. Additionally, the ADF model allows tasks to share data. The key idea is that computation is triggered by dataflow between tasks but that, within a task, execution occurs by making atomic updates to common mutable state. To that end, the ADF model employs transactional memory, which guarantees atomicity of shared memory updates. Chapter 3 describes the language support for the ADF model in the form of C/C++ pragma directives, introduces the ADF model application programming interface and provides an overview of the ADF execution model as well as example on how to use the ADF model for the implementation of the multiple producer, multiple consumer bounded buffer.

Important characteristic of a successful programming model is that is general enough, which means that it must enable easy expression of a range of different problems while providing good performance of corresponding implementations. This type of task can not be properly fulfilled unless the evaluation is general enough. Therefore, in Chapter 4 we present DaSH - the first comprehensive benchmark suite for hybrid dataflow and shared memory programming models. DaSH features 11 benchmarks, each representing one of the Berkeley dwarfs that capture patterns of communication and computation common to a wide range of emerging applications. We provide implementation details of each of the benchmarks from DaSH, accompanied with extensive programmability and performance evaluation that proves the value of the ADF model in comparison with other similar models, as well as traditional shared memory models.

Finally, in Chapter 5, we show that using dataflow, and the ADF model in particular, drastically minimizes synchronization requirements of the parallel game engine implementation by eliminating or restructuring the explicit synchronization that is necessary in shared memory implementations. By utilizing inter-task dependencies we were able to decrease the length of transactions that are used for the synchronization of the remaining critical section. This not only decreases the probability of data conflicts between concurrent transactions, and thus the abort rate, but also it decreases the TM overhead by reducing necessary TM instrumentalization. Thus, we were able to successfully achieve our goal.

1.2 Programming Model

A computing model is a method that describes how a program is to be evaluated. It is tied to a particular model of programming that has primacy. It is abstract in the sense that it assumes an idealized implementation. For sequential computers, the most common computing model is the "von-Neumann" control-flow computing model. This model assumes that a program is a series of addressable instructions, each of which either specifies an operation along with memory locations of the operands or specifies transfer of control to another instruction unconditionally or when some condition holds. The method for executing this program is to start at the first instruction of the program, execute it, and proceed to the next instruction unless the executed instruction requires transfer of control. In the latter case, instruction execution continues from the instruction to which control is transferred. The amount of concurrency available in this model is relatively small. Computers having more than one processor allow several multiprocessor von-Neumann program execution models, such as shared memory or distributed memory models.

In multiprocessor systems, two main issues must be addressed: memory latency, which is the time that elapses between issuing a memory request and receiving the corresponding response, and synchronization, which is the need to enforce the ordering of instruction executions according to their data dependencies. The two issues cannot be properly resolved in a von-Neumann context since connecting von-Neumann processors into a very high speed general purpose computer often results in a synchronization bottleneck.

Nevertheless, with the emergence of multicore processors, the shared memory architecture became prevalent. This is mainly because shared memory programming resembles the sequential style of programming since it provides a single view of data to all program threads. Unfortunately, shared memory programming can be quite difficult. A programmer has to reason about not one, but many threads of execution that concurrently access potentially shared data. Regardless of the constant evolution of the shared memory models and a rich support for programming shared memory systems, some of the problems remain. Specifically, such a complex coordination of threads can introduce subtle and difficult-to-find bugs due to interleaving of processing on data shared between threads. For example, programming with mutual exclusion locks, which is a conventional method for controlling accesses to shared data, is associated with various synchronization problems such as deadlocks, race conditions, priority inversions and non-composability, that make programming with threads difficult. Transactional Memory (TM) [10,12] is a synchronization mechanism that was proposed with the aim to solve these problems by abstracting away the complexities associated with concurrent accesses to shared data. Each thread optimistically executes transactional code in isolation and relies on the TM runtime system to detect conflicting memory changes. If no conflicts were detected during its execution, a transaction can safely commit its tentative changes to the memory. Otherwise, it has to roll back and re-execute from the beginning until it succeeds. However, the scalability and performance of TM programs are often hindered by excessive data conflicts and overheads of transactional execution. Overall, synchronization costs often prevent shared-memory programs from scaling to high core counts, and solutions to this problem require significant programmer effort. These problems have caused a renewed interest into stateless programming models, such as dataflow.

Contrary to the shared memory model, a dataflow computing model [Johnston *et al.* (2004), Silc *et al.* (1998)] is not based on flow of control; instead, it is based on flow of data. Unlike the control-flow computing model, it assumes that a program is given by a data-dependency graph, or dataflow graph whose nodes denote operations and whose edges denote dependencies between operations. The essential difference between control-flow and dataflow computing models is that in control-flow, program execution corresponds to instructions in motion operating on data at rest whereas in dataflow, program execution corresponds to data in motion being processed by instructions (more accurately, operations) at rest. Dataflow graphs, as opposed to conventional machine languages, specify only a partial order for the instruction execution and thus provide opportunities for parallel and pipelined execution at the level of individual instructions.

In the remainder of this chapter, we provide more details on the shared memory model and the transactional memory in particular, as well as the dataflow model, since the Atomic Dataflow model, that is the theme of this thesis, is a hybrid programming model built as a combination of these two models.

1.3 Shared Memory Model

The dominant parallel model for programming current processor architectures is the shared memory model. In this model, all processors can access all machine memory as global address space. Changes in a memory location affected by one processor are visible to all other processors. Global address space provides a user-friendly programming perspective to memory and both fast and uniform data sharing between concurrent program tasks. On the other hand, it does not scale with number of processors and requires a programmer to provide explicit synchronization in order to ensure consistent view of global memory.

Programming models that rely on shared memory favor the use of the program state to represent the problem. However, maintaining the shared state consistent between different threads of execution that access the data concurrently requires some form of synchronization. The conventional choice of the synchronization mechanism for enforcing limits on access to the shared data is to use mutual exclusion locks. Nevertheless, lock based synchronization has many disadvantages. Problems like deadlocks, race conditions, priority inversions and non-composability make programming with locks difficult. Transactional Memory (TM) [Harris & Rajwar (2010), Herlihy & Moss (1993), Shavit & Touitou (1995)] is a synchronization mechanism that aims to solve lock-related problems by abstracting away the complexities associated with concurrent access to shared data. Next, we give an overview of Transactional Memory.

1.3.1 Transactional Memory

Transactional memory (TM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent programming. Compared to lock-based synchronization, transactional memory raises the level of abstraction for programmers that write shared memory parallel programs. The main idea can be summarized in the following: each program thread optimistically executes transactional code in isolation and relies on the TM runtime system to detect conflicting memory changes. If no conflicts were detected during a transaction execution, the transaction can safely commit its tentative changes to the memory. Otherwise, it has to roll back and re-execute from the beginning until it succeeds.

Transactional memory borrows proven concurrency-control concepts from database transactions, but since it manages memory and not a database, it has to satisfy different requirements. In the context of databases, a transaction specifies a program semantics in which a computation executes as if it was the only computation accessing the database. Other computations may execute simultaneously, but the model restricts the interactions among the transactions by serializing their execution. As a consequence, a programmer who writes code for a transaction lives in the simpler, more familiar sequential programming world and only needs to reason about computations that start with the final results of other transactions.

A database transaction has four specific attributes: failure atomicity, consistency, isolation, and durability – collectively known as the ACID properties.

- ***Atomicity*** refers to the ability of the database to guarantee that either all of the tasks of a transaction are performed or none of them are. For example, the transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account will not be debited if the other is not credited. Atomicity states that database modifications must follow an all or nothing rule. Each transaction is said to be atomic. If one part of the transaction fails, the entire transaction fails. A transaction that completes successfully commits and one that fails aborts.
- ***Consistency*** ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not). Consistency states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the databases consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.
- ***Isolation*** refers to the constraint that transactions do not interfere with each other while they are running. This constraint is required to maintain the performance as well as consistency between transactions in a DBMS system.
- ***Durability*** refers to the guarantee that once a transaction commits, its result will be permanent (e.g. stored on a durable media such as disk) and available to all subsequent transactions.. Many databases implement durability by writing all transactions into a log that can be played back to recreate the system state right before the failure. A transaction can only be deemed committed after it is safely stored in the log.

From the perspective of Transactional memory, only atomicity and isolation properties are required to provide correct synchronization of program data. Consistency property, as is defined in database system, is not necessary for the correct execution of memory transaction. However, most transactional system provide automatic re-execution of failed transaction using the retry operation, which allows implementation of additional consistency rules required by the program. Finaly, since Transactional memory programs work with program memory, the durability property is irrelevant.

Essentially, a memory transaction is a piece of code that executes a series of reads and writes to shared memory. These reads and writes logically occur at a single instant in time (atomicity); intermediate states are not visible to other concurrent transactions (isolation).

We refer to the set of locations that a transaction has read from as its *read-set* and the set of locations that it has written to as its *write-set*. Read and write set can overlap or they can be disjunctive.

Transactional Memory (TM) systems can be subdivided into three flavors: Hardware TM (HTM), Software TM (STM) and Hybrid TM (HyTM) (the mix of hardware and software transactional memory).

The idea of providing hardware support for transactions originated in paper by Tom Knight [Knight (1986)]. The idea was popularized by Maurice Herlihy and J. Eliot B. Moss [Herlihy & Moss (1993)] for hardware TM. In 1995 Nir Shavit and Dan Touitou extended this idea to software-only transactional memory (STM) [Shavit & Touitou (1995)]. STM has recently been the focus of intense research and support for practical implementations is growing.

1.3.1.1 Transactional Memory Design Properties

Many alternative TM implementations are possible which can vary on a number of characteristics such as how the tentative updates made by transactions are managed and how conflicts between transactions are detected. The main design choices are: optimistic versus pessimistic concurrency control, eager versus lazy version management, and eager versus lazy conflict detection.

Concurrency Control

- ***Pessimistic concurrency control*** refers to the TM implementation in which the system detects and resolves a conflict at the moment when a transaction is accessing a memory location. This type of concurrency control allows a transaction to claim exclusive ownership of data prior to proceeding, preventing other transactions from accessing it.
- ***Optimistic concurrency control*** refers to the TM implementation in which conflict detection and resolution can happen after a conflict occurs. This type of concurrency control allows multiple transactions to access data concurrently and to continue running even if they conflict, so long as the TM detects and resolves these conflicts before a transaction commits.

Version Management

- ***Eager version management*** – also known as direct update – means that the transaction directly modifies the data in memory. The transaction maintains a private undo-log holding values that it has overwritten. This log allows the old values to be written back if the transaction subsequently aborts.

- ***Lazy version management*** – also known as deferred update – means that the updates are delayed until a transaction commits. The transaction maintains its tentative writes in a transaction-private redo-log. When a transaction commits, it updates the actual locations from these private copies. If it aborts, the redo-log is discarded.

Conflict Detection

- ***Eager conflict detection*** means that a conflict is detected as soon as a transaction references the data.
- ***Lazy conflict detection*** means that a conflict is detected at commit time, i.e. when the first of two or more conflicting transactions commits.

1.3.1.2 Strengths and Weaknesses of TM

The main advantage of transactional memory is that it greatly simplifies conceptual understanding of multithreaded programs and results in more maintainable programs by working in harmony with existing high-level abstractions such as objects and modules. This is because transactions are composable which greatly improves programmability compared to locks. Lock-based programming has a number of well-known problems that frequently arise in practice:

- The programmers have to think about overlapping operations in distantly separated and seemingly unrelated sections of code. This task is often very difficult and error-prone.
- The programmers have to adopt a locking policy to prevent deadlock, livelock, and other failures to make progress. These policies are often informally enforced and error prone, and when the previous issues arise they are very difficult to reproduce and debug.
- Locks can lead to priority inversion, a situation where a high-priority thread is forced to wait on a low-priority thread holding exclusive access to a resource that it needs.

Conversely, a memory transaction is much simpler concept, because each transaction can be viewed in isolation as a single-threaded computation. Deadlock and livelock are either prevented entirely or handled by an external transaction manager; the programmer doesn't need to worry about it. Priority inversion can still be an issue, but high-priority transactions can abort conflicting lower priority transactions that have not already committed.

Optimistic approach inherent to TM results in increased concurrency: no thread needs to wait for access to a resource, and different threads can safely and simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock.

On the other hand, the need to abort failed transactions places limitations on the behaviour of transactions: they cannot perform any operation that cannot be undone, including system calls and I/O.

Moreover, TM systems also suffer a performance hit primarily due to the overhead associated with maintaining the log and the time spent committing transactions. In Section 2.3.1 we show that this overhead can impede the performance of the game server parallelized using transactional memory.

1.4 Dataflow Model

The fundamental principles of dataflow were developed by Jack Dennis [Dennis & Misunas (1974)] in the early 1970s. The dataflow model avoids the two features of von-Neumann model, the program counter and the global updatable store, which become bottlenecks in exploiting parallelism. The computational rule, also known as the firing rule of the dataflow model, specifies the condition for the execution of an instruction. The basic instruction firing rule, common to all dataflow systems is as follows: *An instruction is said to be executable when all the input operands that are necessary for its execution are available to it.* The instruction for which this condition is satisfied is said to be enabled. The effect of firing an instruction is the consumption of its input values and generation of output values. Due to the above rule the model is asynchronous. It is also self-scheduling since instruction sequencing is constrained only by data dependencies. Thus, the flow of control is the same as the flow of data among various instructions. As a result, a dataflow program can be represented as a directed graph consisting of named nodes, which represent instructions, and arcs, which represent data dependencies among instructions. Data values propagate along the arcs in the form of data packets, called tokens. The two important characteristics of the dataflow graphs are functionality and composability. Functionality means that evaluation of a graph is equivalent to evaluation of a mathematical function on the same input values. Composability means that graphs can be combined to form new graphs.

In a dataflow architecture the program execution is in terms of receiving, processing and sending out tokens containing some data and a tag. Dependencies between data are translated into tag matching and transformation, while processing occurs when a set of matched tokens arrives at the execution unit. The instruction which has to be fetched from the instruction store (according to the tag information) contains information about what to do with the data and how to transform the tags.

Depending on the way of handling data, several types of dataflow architectures emerged in the past: single-token-per-arc dataflow [Dennis & Misunas (1974)], tagged-token dataflow [Arvind & Nikhil (1987)], and explicit token store [Papadopoulos & Culler (1990)].

1.4.1 Static (Single-token-per-arc) Dataflow Model

The single-token-per-arc (or static) model was first proposed by Dennis at MIT describing the Static Dataflow Architecture [Dennis & Misunas (1974)]. At the machine level, a dataflow graph is represented as a collection of activity templates, each containing the operation code of the represented instruction, operand slots for holding operand values, and destination address fields, referring to the operand slots in subsequent activity templates that need to receive the result value. The static dataflow approach allows at most one token to reside on any one arc. This is accomplished by extending the basic firing rule as follows: *A node is enabled as soon as tokens are present on its input arcs and there is no token on any of its output arcs.* To implement the restriction of at most one token per arc, acknowledge signals, travelling along additional arcs from consuming to producing nodes, are used as additional tokens. Thus, the firing rule can be changed to its original form.

The major advantage of the single-token-per-arc dataflow model is its simplified mechanism for detecting enabled nodes. Unfortunately, this model of dataflow has a number of serious drawbacks. Since consecutive iterations of a loop can only partially overlap in time, only a pipelining effect can be achieved and thus a limited amount of parallelism can be exploited. Another undesirable effect is that token traffic is doubled. There is also a lack of support for programming constructs essential to any modern programming language.

The static computing model can exploit structural parallelism (as in different unrelated operators executing at the same time) and pipeline parallelism (as in different parts of the graph consuming different tokens of a stream of tokens at the same time). Since each edge can only hold one token at a time only one iteration or one function invocation can be active. Thus, it cannot exploit dynamic forms of parallelism such as loop parallelism (from simultaneously executing different unrelated iterations of a loop body) or recursive parallelism (from simultaneously evaluating multiple recursive function calls). It also cannot deal with intermittent streams, streams in which some of tokens are undefined. The model is thus well suited for applications with regular numerical computational structures such as signal processing and image processing applications which do not make heavy use of iterative or recursive program structures.

The main source of inefficiency in the classic static architecture is due to the use of tokens not just as data carriers but as implicit signal carriers. This has been addressed by refining the classic model in the following manner. The edges of a graph no longer represent flow of actual data in tokens; instead they are simply paths for signals that indicate availability of data values. When a node has all its signals present, it causes the corresponding instruction to execute in a conventional sense, by retrieving operands from memory and storing the result in memory.

1.4.2 Dynamic (Tagged-token) Dataflow Model

The performance of a dataflow machine significantly increases when loop iterations and subprogram invocations can proceed in parallel. To achieve this, each iteration or subprogram invocation should be able to execute as a separate instance of a re-entrant subgraph. However, this replication is only conceptual. In the implementation, represented by Manchester Dataflow Computer [Gurd *et al.* (1985)] and MIT Tagged-Token Dataflow Machine [Arvind & Nikhil (1987)], only one copy of any dataflow graph is actually kept in memory. Each token includes a tag, consisting of the address of the instruction for which the particular data value is destined, and other information defining the computational context in which that value is used. Each arc can be viewed as a bag that may contain an arbitrary number of tokens with different tags. Now, the firing rule is: *A node is enabled as soon as tokens with identical tags are present at each of its input arcs.* Dataflow architectures that use this method are referred to as tagged-token (or dynamic) dataflow architectures.

The major advantage of the tagged-token dataflow model is the higher performance it obtains by allowing multiple tokens on an arc. Thus, the reason this computing model is called dynamic is because tagging enables both loop parallelism and recursive parallelism that arise (dynamically) at runtime, to be exploited. The price for exploiting such parallelism is that any architecture embodying the model must support efficient implementation of the matching unit that collects and matches tokens with the same tags. Ideally, associative memory could be used for this purpose to improve performance. Unfortunately, this is not cost-effective since the amount of memory needed to store tokens waiting for a match tends to be very large. Therefore, all dynamic dataflow machines have used some form of hashing techniques which are typically not fast enough compared with associative memory.

The most common approach in which dynamic dataflow model manipulates complex data structures is using what is known as I-structures (for incremental structures) [Arvind *et al.* (1987)]. The basic idea is to associate each atomic unit of a data structure with status bits and with a queue of deferred reads. The status of the atomic unit can be one of PRESENT, which means that it can be read but not written, ABSENT, which means that read has to be deferred but it can be written, and WAITING, which means that at least one read has been deferred and nothing has been written. When an atomic unit of the data structure is defined (which it can be precisely once), all deferred reads are immediately satisfied. Thus, it is possible to use a data structure before it is fully defined and it is not necessary for a data structure to be fully defined before a subsequent data structure is incrementally defined to deal with a modification to the original data structure.

1.4.3 Explicit Token Store Dataflow Model

One of the main problems of tagged-token dataflow architectures is the implementation of an efficient token matching. To eliminate the need for associative memory searches, the concept of an explicit address token store has been proposed [Papadopoulos & Culler (1990)]. The basic idea is to allocate a separate memory frame for every active loop iteration or a subprogram invocation. Each frame slot is used to hold an operand for a particular activity. Since access to frame slots is through offsets relative to a frame pointer, no associative search is necessary. To make this concept practical, the number of concurrently active loop iterations must be controlled. Hence, the constraint condition of k-bounded loops was proposed, allowing the number of concurrently active loop iterations to be bounded by a constant.

The explicit token address store principle was developed in the Monsoon project [Papadopoulos & Culler (1990)]. The nodes of the Monsoon dataflow processor are coupled by a packet-communication network with each other and with I-structure storage units. The main objective of the Monsoon dataflow processor architecture was to alleviate the waiting/matching problem by explicitly using address token store. This is achieved by a dataflow processor using an eight-stage pipeline. The first pipeline stage is the instruction fetch stage which is arranged prior to the token matching, in contrast to dynamic dataflow processors with associative matching units. The new arrangement is necessary, since the operand fields in an instruction denote the offset in the memory frame that itself is addressed by the tag of a token. The explicit token address is computed by the composition of frame address and operand offset. This is done in the second stage, which is the first of three pipeline stages that perform the token matching. In the third stage a presence bit store access is performed to find out if the first token of a dyadic operation has already arrived. If that is not the case, the presence bit is set and the current token is stored of the frame slot in the frame store in the fourth stage. Otherwise the presence bit is reset and the operand retrieved from the frame store in the fourth pipeline stage. The next three stages are execution stages in the course of which the next tag is also computed concurrently. The eighth stage forms one or two new tokens that are sent to the network, stored in a user token queue, a system token queue, or directly recirculated to the instruction.

1.5 Combining control-flow and dataflow

Pure dataflow computers based on the single-token-per-arc or tagged-token dataflow model usually perform quite poorly with sequential code. This is due to the fact that an instruction of the same thread can only be issued to the dataflow pipeline after the completion of its predecessor instruction. A further drawback is the overhead associated with token matching. Before a dyadic instruction is issued to the execution stage, two result tokens are necessary. The first token is stored in the waiting-matching store, thereby introducing a bubble in the execution stage(s) of the dataflow processor pipeline. Only when the second token arrives is the instruction issued. Finally, since a context switch occurs in fine-grain dataflow after each instruction execution, no use of registers is possible.

One solution of these problems is combining the dataflow and control-flow mechanisms. The symbiosis between dataflow and von-Neumann architectures is represented by a number of research projects developing von-Neumann/dataflow hybrids [Iannucci (1988), Nikhil (1989), Papadopoulos & Traub (1991), Jagannathan (1995), Bohm *et al.* (1993)]. The spectrum of such hybrids is very broad, ranging from simple extensions of a von-Neumann processor with a few additional instructions to specialized dataflow systems attempting to reduce overhead by increasing the execution grain size and employing various scheduling, allocation, and resource management techniques developed for von-Neumann computers. These developments illustrate that dataflow and von-Neumann computers do not necessarily represent two entirely disjoint worlds but rather are the two extreme ends of a spectrum of possible computer systems.

1.5.1 Threaded dataflow

In a dataflow program each subgraph that exhibits a low degree of parallelism can be identified within a dataflow graph and transformed into a sequential thread. By the term threaded dataflow we understand a technique where the dataflow principle is modified so that instructions of a sequential instruction stream can be processed in succeeding machine cycles. A thread of instructions is issued consecutively by the matching unit without matching further tokens except for the first instruction of the thread. Data passed between instructions from the same thread is stored in registers instead of being written back to memory. These registers may be referenced by any succeeding instruction in the thread. Thereby single-thread performance is improved. The total number of tokens needed to schedule the instructions of a program is reduced thus saving hardware resources. Pipeline bubbles are avoided for dyadic instructions within a thread. The Moonson [Papadopoulos & Culler (1990)] and Epsilon-2 [Culler *et al.* (1991)] are machines that represent this approach.

1.5.2 Large-grain dataflow

This technique, also referred to as coarse-grain dataflow, advocates activating macro dataflow actors by the dataflow principle while the sequences of instructions that constitute an actor are executed in the von-Neumann style. Large-grain dataflow machines, such as TAM - Threaded Abstract Machine [Culler *et al.* (1993)] and *T [Nikhil *et al.* (1992)], typically decouple the matching stage (sometimes called signal stage, synchronization stage, etc.) from the execution stage by using FIFO-buffers. Pipeline bubbles are avoided by the decoupling. Of-the-shelf microprocessors can be used for the execution stage. For example, *T has a scalable computer architecture designed to support a broad variety of parallel programming styles including those which use multithreading based on non-blocking threads. A *T node consists of the data processor (executing threads), the remote memory request coprocessor (for incoming remote load/store requests), and the synchronization coprocessor (for handling returning load responses and join operations), which all share local memory. This hardware is coupled with a high performance network having a fat-tree topology with high cross-section bandwidth.

1.5.3 RISC dataflow

Another stimulus for dataflow/von-Neumann hybrids was the development of RISC dataflow architectures, notably P-RISC [Nikhil (1989)], which allow the execution of existing software written for conventional processors. Using such a machine as a bridge between existing systems and new dataflow supercomputers made the transition from imperative von-Neumann languages to dataflow languages easier for the programmer. The basic philosophy underlying the development of the P-RISC architecture can be characterized as follows: use a RISC-like instruction set, change the architecture to support multithreaded computation, add fork and join instructions to manage multiple threads, implement all global storage as I-structure storage, and implement the load/store instructions to execute in split-phase mode. The architecture based on the above principles, developed at MIT, consists of a collection of processing elements and heap memory elements, interconnected through a packed-switching communication network.

Chapter 2

QuakeTM - Parallelizing a Quake game engine using TM

This chapter presents QuakeTM, a complex real life TM application that was parallelized from the sequential version with TM-specific considerations. QuakeTM consists of 27,600 lines of code spread across 49 files and exhibits irregular parallelism for which a task parallel model is hte most suitable. Two implementations are compared: a coarse-grained TM implementation characterized with eight large transactional blocks as well as a fine-grained implementation which consists of 58 different critical sections. In spite of the fact that QuakeTM scales, we show that more effort is needed to decrease the overhead and the abort rate of current software transactional memory systems to obtain a good performance. This chapter provides insights into development challenges, suggests techniques to solve them and provides extensive analysis of the transactional behavior of QuakeTM. Special emphasis is put on the discussion of the TM programmability.

2.1 Quake Description

QuakeTM is based on Quakeworld, which is the multi-player mode of Quake I – the first person shooter game released under the GNU general public license by ID Software. It is a sequential application, built on a client-server architecture, where the server maintains the game world and handles coordination between clients, while the clients perform graphics update and implement user-interface operations. Players connect to the server, join the game and interact with other players until they leave the game willingly, or due to connection or server problems. During the gameplay there is no direct communication between clients; all the communication is done through the server that executes client actions and sends all relevant updates to the clients.

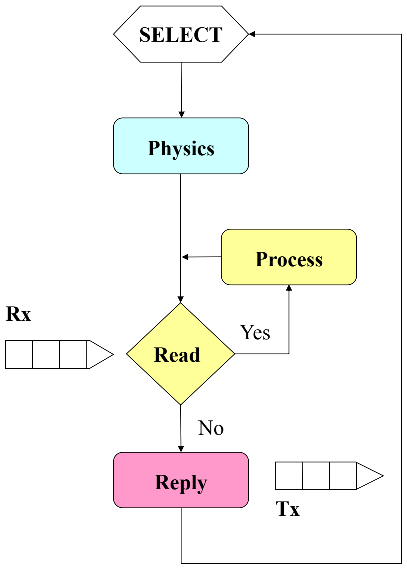


Figure 2.1: Frame execution algorithm.

2.1.1 Game Algorithm

The server executes an infinite loop, where each iteration performs the calculation of a single game frame. The frame execution algorithm is presented in Figure 2.1. The server blocks on the *select* system call waiting for client requests. If requests are present on the receiving port, it starts the execution of the new frame. It is possible to distinguish three stages of the frame execution: world physics update (P), request receiving and processing (R) and reply stage (S). When all three stages are done, the server frame ends and the process is repeated. Generally, the server sends replies only to clients that were active in the current frame; namely those who have sent a request. All replies are sent after all requests have been processed. This clear separation of the frame stages simplifies the parallelization.

A client and the server exchange various types of messages that can be divided into two categories:

- *connection related messages*, sent by the client in order to connect to the server, download missing files and join the game, and
- *gameplay messages* that carry player's actions.

Since connection related messages are not common, the focus of the parallelization is on gameplay messages. Two main types of game messages are string commands, which are strings that relate to specific functions of the game, and move commands, which are by far the most common messages.

Quake game world is a polygonal representation of the 3D virtual space in which all objects are referred to as entities. Each entity has its own specific characteristics and actions it can

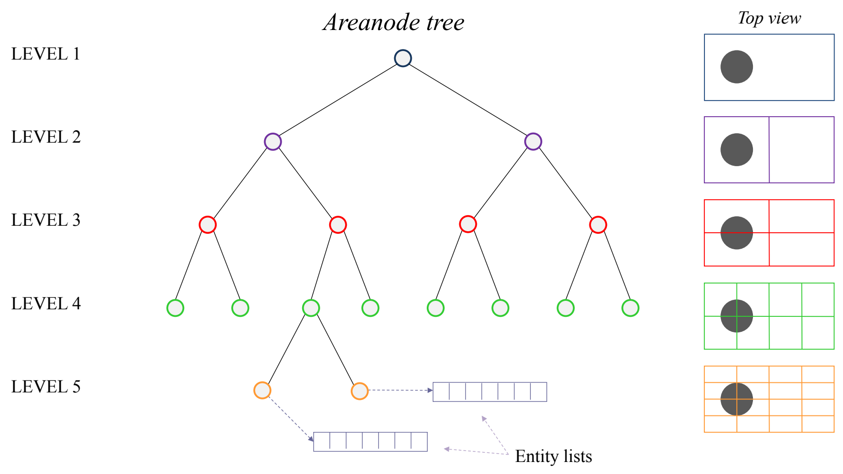


Figure 2.2: Constructing areanode tree from the BSP map volume.

perform. During the update, the server sends information only for those entities which are of interest to a client. Nevertheless, the server has to simulate and model, not only player's actions, but also effects induced by these actions, e.g. when the player hits some object. In such a case, it is necessary to determine the applied force, the weight of the object, its shape, the original position, the environment etc. in order to present as realistic view as possible. Thus, server processing is a complex, compute intensive task, in which the computational complexity increases super-linearly with the number of players [Abdelkhalek *et al.* (2003)].

2.1.2 Map Description

A map of the Quake world is represented with a file that holds the *binary space partition* (BSP) implementation of the 3D world with all the details relevant to draw and position the objects in the world such as planes, vertices, nodes, visibility data, texture information, models, brushes etc. [Fuchs *et al.* (1983), McGuire (2000)]. The level of details contained within the BSP tree is large; therefore BSP trees are hard to maintain for dynamic scenes. If the server wants to generate a quick list of the objects that an entity may interact with, traversing the BSP tree is inefficient, and since this is a common operation involved in each move command, the server constructs and maintains a secondary binary-tree structure, called *areanode tree*. This is a 2D representation of the BSP tree, constructed during the server initialization by dividing the 3D volume in the x-y plane. Figure 2.2 demonstrates the building process. Starting from the root, which corresponds to the entire world, the space is divided in two equal parts to form nodes on the next level of the tree. Each division is done using the alternate plane, and after four divisions, the final areanode tree is constructed, with total of 5 levels, 31 areanodes, and 16 leaves in the lowest level. The size of the tree can be changed by redefining the maximum depth, but using bigger trees has minor effect on the performance of the server [Abdelkhalek &

Bilas (2004)]. The structure of the areanodes has no correlation with rooms in the world, and likewise division planes do not correlate with walls or ceilings.

Each entity in the game world is either contained within an areanode that is a leaf, or it intersects more than one leaf which means that it crosses the division plane, in which case the entity is contained within an areanode on the higher level of the tree. Each areanode has an associated list of entities contained within the space defined by that areanode. When an entity moves, it is necessary to update the areanode tree to reflect the new position of the entity. This is done by removing the entity from the current list, and inserting it into the list of the areanode that corresponds to the destination of the entity.

2.1.3 Move Execution

Clients influence the gameplay by sending the move command which specifies various parameters related to the player's state, actions and intentions. Using the data extracted from the command (motion indicators, origin of the player and time to run the command), the server constructs the bounding box of the player's motion, thus defining the region of the world that it can affect. Then, it traverses the areanode tree to find all the entities contained within this bounding box and to associate them with the move command. It then simulates the move, and upon completion, removes the player's entity from the old position in the game world and links it to the new one.

2.2 Parallelization

The fine-grained lock-based parallelization of the Quake server has been previously done using POSIX threads (pthreads) [Abdelkhalek & Bilas (2004)]. The work took about 15 man-months of development. The server starts with a fixed number of threads that synchronize globally on barriers and utilize locks to synchronize shared data accesses. Global synchronization is used to separate three stages of the frame execution: physics update, request processing and reply stage. Sequential parts of the frame execution are executed by a single "master" thread, while other threads proceed directly to the barrier. Parallel regions are executed by all threads using static assignment of the clients to threads. This approach is similar to the fork-join parallelism, which is well supported by the OpenMP programming model. In general, Quake server is highly suitable for fork-join parallelism since the stage separation is clear, and sequential parts consume a negligible execution time. Moreover, one of the main goals of the OpenMP model, as well as transactional memory, is to make parallel programming easier. Therefore, in theory, the combination of OpenMP and transactional memory should provide an easy way to exploit parallelism, which is the reason why this approach was chosen for the parallelization the Quake game server presented in this thesis.

The pthreads version is the base for the work on Atomic Quake [Zyulkyarov *et al.* (2009)], whose main objective was to evaluate the effort of replacing locks with transactions. The process was not straightforward, as many specific challenges were encountered that have increased the development effort considerably. First, in general locks are not block structured, so the code had to be reorganized to adapt to the TM model. The second problem was to avoid I/O operations, which is not an issue in a lock based system. Finally, a big fraction of the development time was spent to determine how locks are associated with variables and to understand the locking strategy.

On the other hand, QuakeTM does not build on top of the pthreads version, but implements a different parallelization strategy. The intention was to start from the sequential implementation and to parallelize it using OpenMP and transactional memory. Moreover, a coarse-grained parallelization approach has been chosen in order to test one of the promises of transactional memory, which is to make multithreaded programming easier by providing good performance scaling coupled with a coarse-grained parallelization effort.

Section 2.4 provides a comparison between these two implementations and shows that both approaches suffer from the problems related to high transactional memory instrumentation overhead, though the fine-grained version is less affected. Semantically comparing these two solutions, it is clear that a coarse-grained implementation, is characterized with atomic blocks and read and write sets that are an order of magnitude larger. Overall, there are only eight transactional blocks in QuakeTM compared to 58 in the fine-grained implementation.

The main goal of this work was to test the primary objective of transactional memory that is to make multithreaded programming easier. With no prior knowledge of the application itself, it was necessary to invest some time to understand the code. The next step was to identify parts of the application suitable for parallelization by studying the program structure and profiling the execution of the sequential code. The process of adding OpenMP parallelization pragma directives and transactions was then straightforward and simple. The real challenge was to identify global data structures and variables that must be shared and those that can be private to a thread. From a performance perspective this is crucial, since excessive data sharing hinders the performance.

2.2.1 Shared Data

During the gameplay, there are three types of shared data structures: message buffers, the areanode tree and game objects (entities). Among the buffers we further distinguish the global state buffer and per-player reply buffers. The global state buffer holds the updates that reflect the actions of all players involved in the game session. It is updated in the physics update stage and the request processing stage.

Accesses to the areanode tree are in the form of linked list operations on the entity lists associated with each areanode. The access pattern for the request stage has already been covered in the explanation of the move command. A similar pattern is observed in the physics update stage since physical influences, which may affect an entity, can change its position and hence its areanode container.

Game entities are updated in the physics update stage and the request processing stage. During the move execution, each entity that is touched is updated in the global shared part of the memory that is statically allocated and populated during the server initialization. Quake I is the first game that has used the concept of interpreted core of the game. This means that the essence of the game is coded in a special interpreted language called *QuakeC* [Montanuy (1996)], which is a C like language developed specifically for the purpose of the game, primarily to achieve modularity. Using *QuakeC*, a programmer can customize Quake to great extents by adding weapons, changing game logic and physics, and programming complex scenarios. *QuakeC* source code is compiled using a tool called *qcc* into a byte code kept in a special file *progs.dat*. During the server initialization this file is loaded into the memory, and appropriate program pointers are set according to the layout of the file. It is possible to distinguish following regions of the file: (1) strings, (2) functions, (3) statements, (4) field definitions, (5) global definitions (6) globals and (7) entities. Once loaded into the memory, specific parts of the engine are accessed using pointers and statically defined offsets. As an example this is the usual way to execute *PutClientInServer* program function, which is called to spawn the player into the game:

```
pr_global_struct->time = sv.time;
pr_global_struct->self = EDICT_TO_PROG(sv_player);
PR_ExecuteProgram (pr_global_struct->PutClientInServer);
```

PR_ExecuteProgram function acts like an interpreter. It determines the program function, and accesses the part of the program memory that defines the start address for function arguments and local variables, number of parameters and their sizes, and the address of the first statement. These are all integer values representing previously mentioned offsets. It allocates the space for parameters and local variables on the internally defined stack and starts the execution of the first statement. Statement execution is carried out by reading the source and

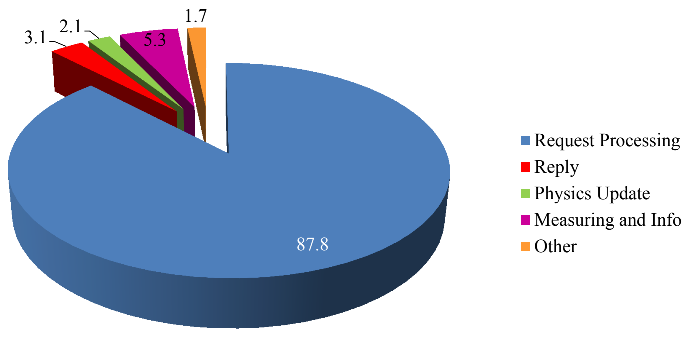


Figure 2.3: Execution breakdown of the sequential server.

destination operands and performing the specified operation. Any level of function calls is allowed while there is a space on the stack for a new function. However, only entity part of the program memory has to be shared. Other parts, regardless of their misleading names, can be thread-private.

2.2.2 Parallelization Strategy

An execution breakdown of the sequential Quake server is given in Figure 2.3. It is clear that the majority of time is spent in the request processing phase, and although physics update is characterized with similar shared data access patterns, its operations are significantly less involved. Therefore, the parallelization efforts were concentrated to the request processing stage only.

The algorithm for this stage is presented in Listing 2.1. Each iteration of the loop executes a single client request. A tasking model is the most suitable for parallelization of this loop for two reasons: (i) given the diversity of the client requests, load balancing is an important factor to consider and (ii) the number of requests that are pending on the receiving port are not known beforehand. Profiling shows that the time needed to receive all packets is negligible compared to the time to process the requests. Therefore, in the parallel implementation these two phases of the request processing stage are separated in order to make the code suitable for task execution. First, the server receives all packets and stores them into a temporary packet list. Then, it traverses the packet list and creates a task for each packet from the list. Listing 2.2 illustrates this approach.

The QuakeTM coarse-grained approach consists of eight large atomic blocks, but here we describe only the four which synchronize the processing of the client's move command, the most common action performed by the Quake server. The execution of the move command is illustrated by the diagram in Figure 2.4 in which transactional blocks are clearly marked. The execution begins with the client's physics update (*ClientPhysics*), followed by the so called

```

while (NET_GetPacket ()) {
  // Filter packets
  if (connection related packet) {
    SV_ConnectionlessPacket ();
    continue;
  }

  // game play packets
  for (i=0 ; i<MAX_CLIENTS ; i++) {
    // Do some checking here
    SV_ExecuteClientMessage ();
  }
}
```

Listing 2.1: Pseudocode for the request processing stage. Sequential implementation.

```

while (NET_GetPacket ()) {
  // Filter packets
  if (connection related packet){
    SV_ConnectionlessPacket ();
    continue;
  }
  AddPacketToList();
  CopyBuffer();
}

#pragma omp parallel shared(packetlist, ...)
{
  #pragma omp single
  {
    while (packetlist != NULL) {
      #pragma omp task firstprivate(packetlist)
      {
        NET_Message_Init(..);
        // Do some checking here
        for (i=0 ; i<MAX_CLIENTS ; i++){
        // Do some checking here
        SV_ExecuteClientMessage ();
        }
      }
      packetlist = packetlist->next;
    }
  }
}
```

Listing 2.2: Pseudocode for the request processing stage. Parallel implementation.

”think” function (*ClientThink*) that is used to register actions that exceed the duration of a single frame and should influence the state of an entity in successive frames. For example if some object falls from a high altitude, the server requires more than one frame to simulate its fall. Along with the *pmove* (player move) structure initialization – (*PmoveInit*), these actions form the preparation for the actual move execution, and can be contained inside a single transactional block.

The next phase in the execution *AddLinksToPmove*, determines which entities could be affected by the current move command by constructing the bounding box around the player's entity and traversing the areanode tree in order to discover other entities located within this area. Links to all affected entities are added to the *pmove* structure entity list.

Further processing is carried on with the execution of the *PlayerMove* function. First, a model box is assigned to the player's entity and each entity from the *pmove* entity list. Then, using the parameters from the move command, extracted from the received message,

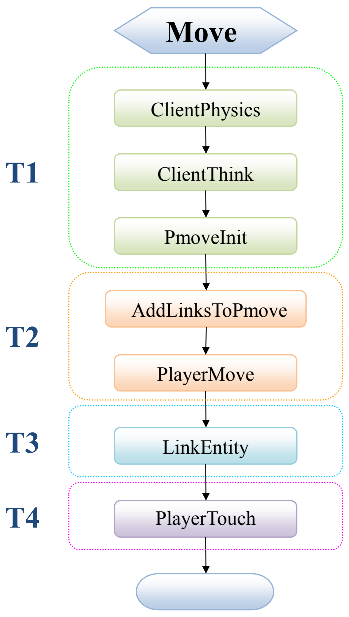


Figure 2.4: The diagram of the move command execution.

a trajectory is followed from the player's original position to its potential destination. If the player's model box clips a model box of the other entity, moving along the trajectory line, there is a collision between them. Depending on the various parameters of the collided entities and the environment that surrounds them (air, water, solid area, etc.) the player's final position is calculated.

The next phase of the move execution represents the *LinkEntity* function that re-links the player's entity to the new position in the areanode tree. In the end, the player's influence during its movement is applied to each entity that was "touched" along the trajectory. This action is denoted as *PlayerTouch* in Figure 2.4.

2.2.3 Programming with Transactional Memory

The conceptual simplicity of transactional memory requires only that a block of code that represents a critical section is contained within the *__tm_atomic* block. For example, the following code inserts a node into a doubly-linked list atomically:

```
__tm_atomic {
    newNode->prev = node;
    newNode->next = node->next;
    node->next->prev = newNode;
    node->next = newNode;
}
```

Within each atomic block, the compiler instruments shared-memory accesses so that the consistency of the block is managed by the TM runtime. The TM runtime tracks all memory reads and writes within a transaction and detects conflicting accesses by concurrent transactions.

The previous example is rather simple. In general, an arbitrary code block can have a complex structure that must be respected by the atomic construct. Therefore, entry and exit points of the atomic block must be on the same level.

With a coarse grained approach one can expect to have a block of code that includes function calls. Since function calls transfer the control of the program, to guarantee the correctness of the execution, the function code must be instrumented in the same way as the block from which the function was called. To ensure this, functions called from within a transactional block must be annotated with proper TM function annotations that serve as a hint to the compiler. For example, if a programmer intends to call a function from within a transaction it must be annotated as *tm_callable*. The compiler generates two versions of the code for such functions, one with instrumentation and one without. The compiler uses a mangled name for the transactional clone of a *tm_callable* function. The name mangling simply adds a transactional suffix to the function name and is designed to work correctly with the template name mangling for *tm_callable* function templates. During the execution, the runtime determines if the call comes from the transactional block or not, and decides which version of the function to call.

Our experience in programming with TM is that it is an iterative process. Once we understood the code, we were able to determined the boundaries of transactional blocks. We then used various techniques to gain performance. Since there are no debuggers for transactional memory, the only option that we had in the beginning was to print relevant information using the *printf* function. This would normally cause the TM runtime system to serialize transactions (switch to a single thread execution). Therefore, after any change in the implementation, it was necessary to first switch to the execution based on a global lock, to test it, and then to remove the *printf* calls to be able to run transactions. Fine tuning was done using ***ReachPoints***, described in Section 2.2.7, mostly to discover data that may not be shared. Especially the last step was iterative, since making one data structure threadprivate typically unmasked other data that was unnecessarily causing conflicts.

2.2.4 Consistency Violations

Transactions in the coarse-grained implementation are large and synchronize accesses to shared data that are part of a bigger functional unit. On the other hand, in the fine-grained implementation those accesses are decoupled and it is necessary to take special care not to violate consistency of the program. In Quake, a naive approach is to parallelize C part of the code without taking special care of the execution that happens in the interpreted part of the Quake runtime.

When the *ProgramExecute* function is called, Quake runs the simulation by maintaining the stack for function parameters and local variables, as well as the program counter that is used to transfer the control of the execution when the interpreter encounters a function call. Most functions are defined in the *QuakeC* language and their calls are interpreted, but some functions are defined in regular C code. Therefore, the execution jumps back and forth from the interpreted environment to the regular C environment. This has devastating effects on consistency if it is not handled in the correct way. Our solution was to buffer all the changes to the shared data that happen in C code while the interpreted program executes and then write them once the program terminates. Additionally, program execution can be nested, meaning that another program can be called from the currently executing program. In that case, it is necessary to empty the buffers in order to prepare them for the new call by updating the corresponding shared data with current contents of the buffers.

2.2.5 Handling I/O operations

An obvious limitation for TM is its inability to handle side-effecting operations. Entirely precluding I/O from transactions is unlikely to be a realistic solution in general as I/O regularly occurs within critical sections. Since most of the I/O in QuakeTM are file operations, our solution was to buffer all I/O that occurs in transactions. However, not all I/O operations can be handled correctly in this way. For example, there are cases when the data read from a file within a transaction is used for further calculations in the same transaction. The problem is that the same file from which a transaction reads the data might be simultaneously written by some other transaction, which can lead to file corruption. Fortunately, we were able to disable functions that have this behavior and avoid the problem, but in some other applications that might not be possible.

2.2.6 Rewriting library functions

I/O operations are not the only operations that can produce side effects and break atomicity of transactions. System calls and library functions are also considered unsafe, since transactional runtime system doesn't have control over their execution or knowledge about the implementation of these routines. Hence, it must be conservative and invoke irrevocable mode to handle these cases. To avoid frequent serialization, we had to redefine library functions that are used in transactions. Most of the redefined functions are mathematical routines and string handling functions. From the performance perspective, it was imperative to handle *sprintf* function because it is called very often during the execution. Specifically, Quake execution is based on string messages that exchange commands between the server and the clients. These commands are often written one byte or word at the time, and each of these writes is based on a call to *sprintf* function.

2.2.7 Parallelization Issues and ReachPoints

Although we have dedicated a significant time to manually identify global data that could be thread-private, that is not sufficient for unmanaged code written in a sequential programming style in which a vast amount of data is global, as in the Quake case. During the attempt to boost performance, we came up with a solution, refereed to as ***ReachPoints***, that helped us identify the rest of the global variables that could be thread private and also to avoid the problems related to the TM cache-line granularity conflict detection.

```

int reachpoints[NumThreads][x*16]

TM_PURE
void PointReached(int check) {
  reachpoints[ThreadId][check]++;
}

int main () {
  . . .
  TRANSACTION
  PointReached (1);
  statement_1;
  PointReached (2);
  TRANSACTION_END
  . . .
}
```

Listing 2.3: ReachPoints: a simple solution for discovering conflicting regions of the transactional code.

The *ReachPoints* solution, shown in Listing 2.3, consists of allocating an array of counters for each thread, taking cache line granularity into account (x*16 integers for each thread where x=1,2,... and cache line size is 64 bytes). At the end of execution, when we print the state of counters, the difference between two counters pinpoints the region of the code where transactions abort. Analyzing that regionallows easier discovery of the causes for the aborts.

Simple as it may be, we have found *ReachPoints* very valuable and useful. Beside their main objective, which is to identify global variables that could privatized, *ReachPoints* can be used to discover sources of false conflicts referred to as false sharing [Yoo *et al.* (2008)] that occur mostly within structured data when two different variables or structure fields reside in the same cache line. Assuming that only one of them is written, say variable X, under the cache line granularity conflict detection system, a read-only variable Y is also causing conflicts, since the writes to X are treated by the TM runtime system also as the writes to Y. We have applied cache padding as a solution for such cases, which resulted in a significant performance improvement due to a decrease in the number of aborts. This solution is specific for eager conflict detection, since in TM systems with lazy conflict detection transactions can be aborted at any time, regardless of the read or write issued in the moment of the abort. This simple solution was later incorporated in the proposal for the TM-aware debugger [Zyulkyarov *et al.* (2010)].

2.3 Evaluation

In order to test QuakeTM, we have used existing Quake client code to build *TraceBot* – an automatic trace client whose behavior is controlled by a finite state machine. We also had to implement changes on the server side to be able to synchronize clients' actions with the server response. Essentially, *TraceBot* is sending messages at the server frame rate until it dies, as a result of actions of other connected players, or until the end of the trace when it commits a suicide. When it dies, *TraceBot* sends another special string command whose function is to respawn the client into the game world and restart the process. Traces are recorded using *VideoClient*, which is similar to *TraceBot* with the addition of graphics. To record traces we use the original, sequential Quake server, and connect *VideoClient* to play the game, producing traces that represent recorded human actions.

We run the server on one machine and the clients on another to simulate the real game environment, given that network latency and bandwidth are not critical [Abdelkhalek & Bilas (2004)]. The server and client frame rates are synchronized and set to 100ms, which is sufficient to handle the worst case transactional frame length. Both machines are PowerEdge 6850, with four dual-core 64-bit Intel Xeon processors running at 3.2 GHz, with 16MB L3 cache memory per processor unit, running SUSE LINUX 10.1.

The QuakeTM code has been compiled using the prototype Intel STM C/C++ compiler version 3.0 [Adl-Tabatabai *et al.* (2006), Ni *et al.* (2008)] with all the optimizations enabled. This compiler provides new language extensions to support transactional memory and STM runtime library that implements both optimistic and pessimistic concurrency control. Serial execution mode is also provided to support system calls and I/O operations inside transactions. The runtime can switch between execution modes dynamically to optimize performance. Atomic blocks provide single lock atomicity semantics meaning that the program behaves as if a single global lock guards each atomic block. It provides weak atomicity guarantees which means that it does not enforce isolation between a transactional and non-transactional code. Function annotations: *tm_callable*, *tm_pure* and *tm_unknown* that instruct the compiler wether to generate a transactional clone of the function or not. Nesting is supported in a closed nesting fashion via flattening; a data conflict rolls back to the outermost level and re-executes the transaction. The runtime algorithm relies on a time-stamp mechanism for the optimistic concurrency control. It uses cache-line granularity conflict detection and implements strict two-phase locking for writes. Writes update values in place and generate undo log entries. Transactions validate the read set at commit time, and if necessary during the read operation, which means that transaction can abort any time during the execution when it encounters a conflict.

In the first part of this evaluation, the performance of the coarse-grained QuakeTM implementation is compared with the performance of the sequential implementation, as well as the parallel implementation in which critical regions are protected with the global lock. We vary the number of threads from one to eight and the number of connected clients from one to sixteen, and report the average frame execution time obtained by different configurations. Although only the request processing stage is parallelized, this evaluation takes into account the time to execute entire game frame. Each experiment executes 2000 game frames (about 200s of real time). We use the *rdtsc* instruction to measure the number of cycles between two events and then translate that value into milliseconds. The results are collected for the last 1000 frames in order to avoid cold start that consists of the server initialization and the time that it takes for all the clients to connect and join the game session. Note that in this evaluation it is not necessary to stress the server by running a large number of clients; if the server is able to service given number of clients faster, it is able to service more clients in a desired frame length.

2.3.1 The Overhead of the Transactional Execution

Figure 2.5 presents normalized average frame execution times of the parallel implementations for the single-thread execution. The baseline is the average frame execution time of the sequential server for a given number of clients. Since there is no contention, the lock version introduces almost no overhead. On the other hand, the overhead for the transactional version goes from 3.5x for a single client up to 6x for the execution with sixteen clients. These results exceed the findings of Wang et al. [Wang *et al.* (2007)] for non optimized version of STM. For microbenchmarks the authors report an overhead of non optimized STM code from 2.4x to 4.5x over the fine-grained lock implementation. For the *SPLASH-2* benchmarks the reported overhead does not exceed 20%, but it is a measure across the entire execution, which hides the fact that little time is spent in critical sections. In our case more than 85% of the time is spent in critical sections and evidently this adds extensive overhead in the execution of the coarse-grained QuakeTM implementation based on STM.

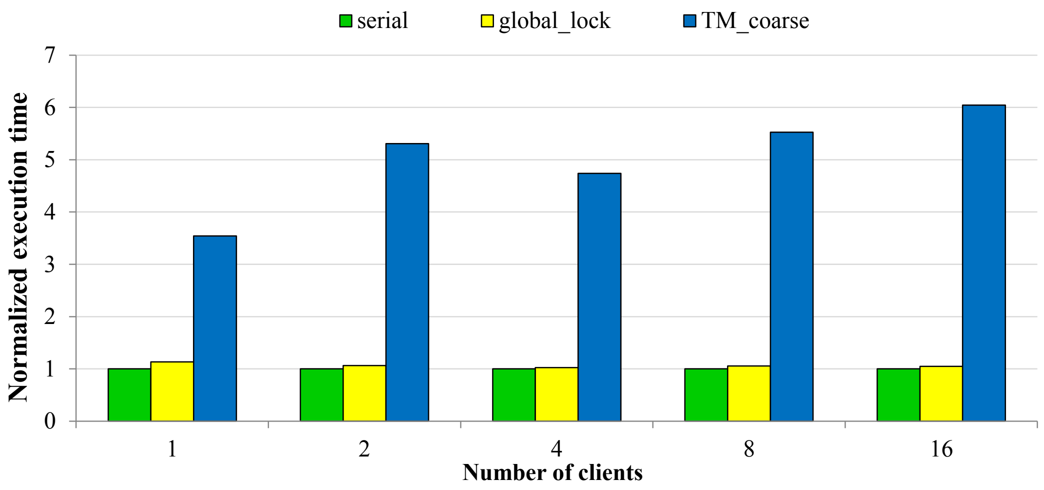


Figure 2.5: Normalized average frame execution times of the QuakeTM parallel implementations for the single-thread case.

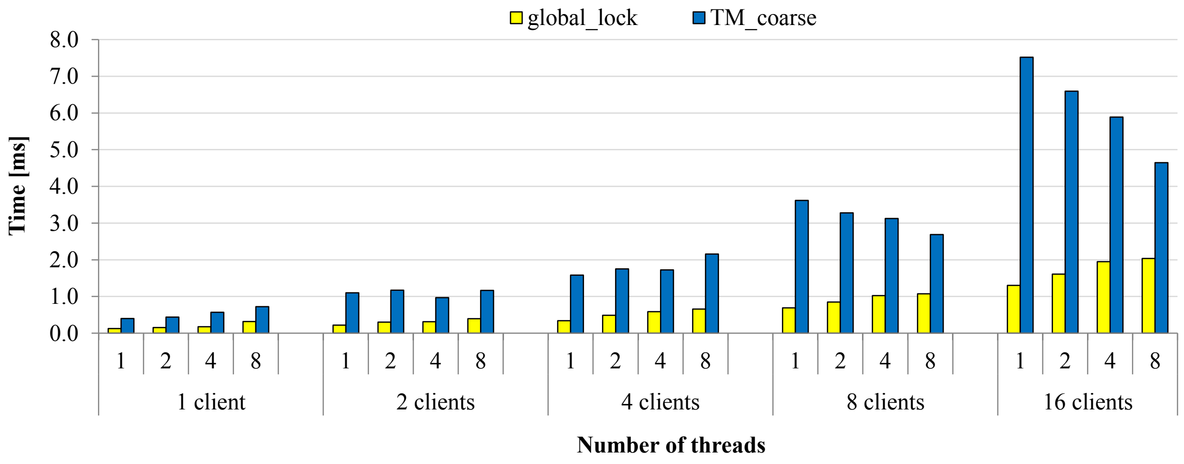


Figure 2.6: Comparative performance of parallel configurations.

2.3.2 QuakeTM Speedup and Scalability

Figure 2.6 shows the comparative performance of parallel implementations for different numbers of connected clients. As expected, the global lock version does not scale, while the QuakeTM implementation scales only when the workload becomes sufficient, or in this case, when eight clients are involved in the game. The transaction overhead remains approximately 4x-6x. When we run the application with sixteen clients we start to notice a considerable speedup. Figure 2.7 provides a better overview of this case. The values are normalized to a single-thread execution time. The speedup for eight threads is 1.62 which is a reasonable result, but it is insufficient to cover the costs of transactional instrumentalization. Figure 2.8 shows the scalability of the QuakeTM server running with sixteen clients. Evidently the TM version scales, but it still performs worse than the global lock version, even though it is able to compensate for almost 50% of the transactional overhead.

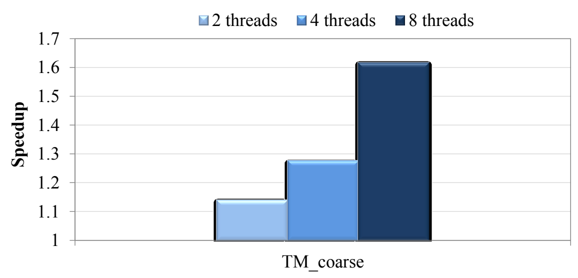


Figure 2.7: Speedup of the parallel server running with 16 clients.

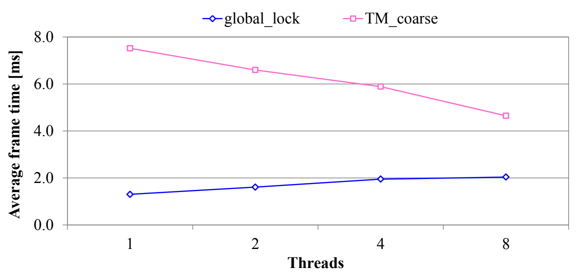


Figure 2.8: Scalability of the parallel server running with 16 clients.

2.3.3 QuakeTM Transactional Statistics

To discover the reasons why the transactional version performs poorly, it is necessary to look at the statistical data that is provided by the Intel compiler. Table 2.1 presents the statistics for the QuakeTM implementation running with eight threads. All statistical values increase when we increase the number of clients connected simultaneously, but from a performance perspective, the most important is the transaction abort rate. In the case of sixteen connected clients 35.3% of transactions abort, causing a high amount of wasted work. There are examples when a transaction aborted 136 times before it eventually committed. This leads to a significant waste of processor cycles to re-execute the transactional code. Table 2.1 also shows that although the mean value of the amount of data read by a transaction is about 5.1 KB there are cases when it grew up to 1.7 MB. This is an important factor which could stress the design of any hardware transactional memory system that typically can support only transactions with relatively small read sets.

Clients	Transactions	Aborts	Abort rate [%]		Mean [KB]	Max [KB]	Total [MB]
1	34754	0	0.0	Reads	3.0	104	105
				Writes	0.6	17	20
2	95980	1970	2.1	Reads	2.8	863	263
				Writes	0.6	164	55
4	179241	10820	6.0	Reads	3.4	1413	570
				Writes	0.6	269	108
8	364305	76560	21.0	Reads	4.2	1478	1207
				Writes	0.8	251	216
16	524561	184992	35.3	Reads	5.1	1704	1725
				Writes	0.9	262	296

Table 2.1: Transactional statistic of the QuakeTM server running with 8 threads.

TM block	Multithread execution - 8 threads, 16 clients					
	Total [10^9 cycles]	Instrumentation overhead		Abort overhead		Abort rate [%]
		10^9 cycles	%	10^9 cycles	%	
1	13.5	10.3	75.8	3.3	24.2	19.5
2	9.5	9.0	94.1	0.6	5.9	18.0
3	17.2	15.1	87.9	2.1	12.1	52.7
4	11.6	10.9	94.3	0.7	5.7	22.4
5	5.9	3.2	53.7	2.8	46.3	61.1
overall	57.9	48.5	83.8	9.4	16.2	35.2

Table 2.2: QuakeTM transactional execution breakdown.

2.3.4 QuakeTM Overhead Breakdown

Table 2.2 presents the execution breakdown of the TM server running with eight threads and sixteen clients. Out of eight atomic blocks implemented in QuakeTM only the five presented in the table contribute considerably to the overall performance. It can be seen that the abort overhead is not significant even for atomic blocks which have a high abort rate. Since Intel compiler is not open source, we can only speculate about the reasons for such results. A possible explanation could be that transactions are aborted early during the execution or that the contention handler and the abort mechanism are efficient. On the other hand, the instrumentation overhead may be high as a result of the STM runtime operations associated with locks.

2.4 Comparison with the Fine-grained Implementation

The fine-grained QuakeTM implementation is based on our work on Atomic Quake [Zyulkyarov *et al.* (2009)], which goal was to study the usability of transactional memory for transforming existing lock-based parallel implementation of the Quake server [Abdelkhalek & Bilas (2004)] and assess possible gains and problems associated with this kind of transformation. Therefore, Atomic Quake inherits existing parallelization structure based on pthreads in which program threads are explicitly controlled. Conversely, In QuakeTM we have started from the sequential version and were free to choose our parallelization strategy, which eventually resulted in the choice of fork-join parallelism supported by OpenMP. Finally, the fine-grained QuakeTM implementation is similar to Atomic Quake, except that parallelization is done using OpenMP tasking model, in addition to some minor differences in the implementation of atomic regions of the code.

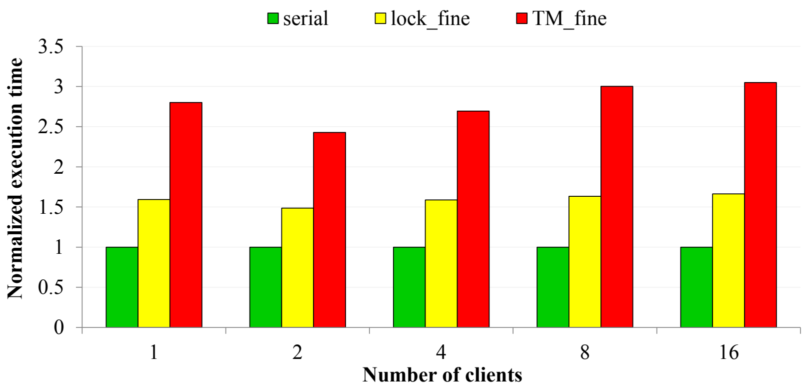


Figure 2.9: Normalized average frame execution times of the fine-grained parallel implementations for the single-thread case.

2.4.1 The Overhead of the Fine-grained Implementation

Figure 2.9 shows the overhead graph of the fine-grained implementation. As expected, the overhead of the fine-grained lock implementation is higher than in the case of the parallel implementation based on the global lock. This is partially due to data copying from global to private buffers and vice versa, and partially due to the programming patterns associated with the use of locks that are necessary to avoid lock related problems such as deadlocks and livelocks. This overhead is approximately 50%. The overhead of the fine-grained QuakeTM implementation is 2.4x – 3x, which is approximately 50% decrease compared to the coarse-grained QuakeTM implementation. The reason is that in this implementation the transactions are shorter and their read sets are an order of magnitude smaller than in the coarse-grained implementation.

2.4.2 Speedup and Scalability of the Fine-grained Implementation

The comparative performance of the fine-grained lock and TM implementations is shown in Figure 2.10, while the speedup and scalability are presented in Figure 2.11 and Figure 2.12. Speedup of the fine-grained lock version is 1.63 while the speedup of fine-grained TM implementation is 1.5. For completeness, Figure 2.12 also shows the performance of the global lock and the coarse-grained QuakeTM implementations. The fine-grained lock version performs the best followed by the global lock version. The fine-grained transactional version comes close to the global lock version, while the coarse grained implementation of QuakeTM falls behind. It is now clear that both transactional versions pay a high performance cost associated mainly with instrumentation overhead, which is supported by the fact that the abort rate of the fine-grained TM server running with eight threads and sixteen clients is only 4.1%. Table 2.3 provides the summary of TM statistics of the fine-grained QuakeTM server running with 8 threads.

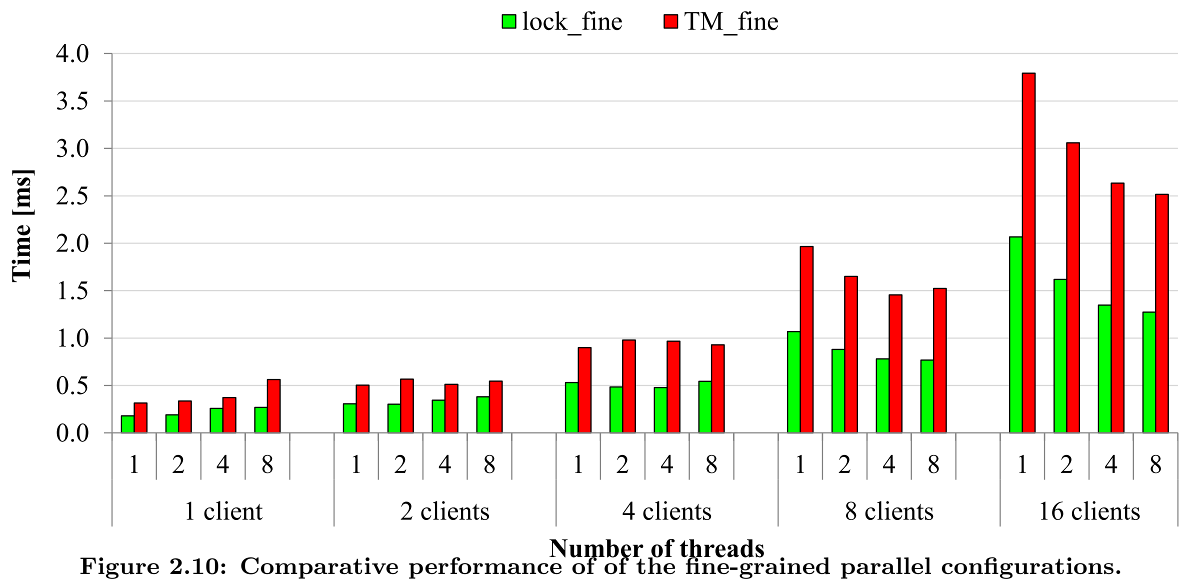


Figure 2.10: Comparative performance of of the fine-grained parallel configurations.

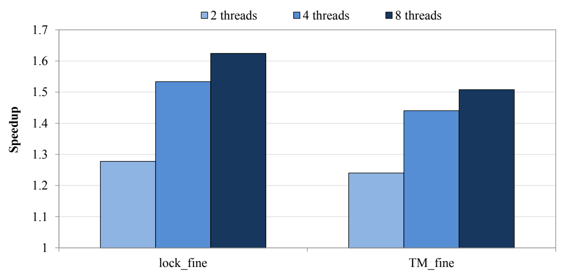


Figure 2.11: Speedup of the parallel server running with 16 clients.

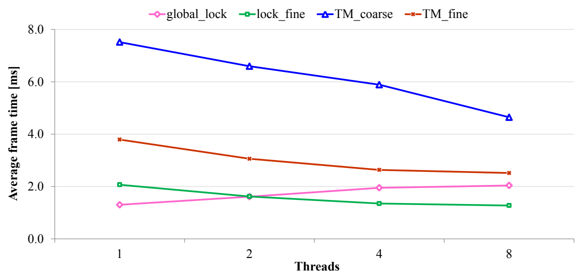


Figure 2.12: Scalability of the parallel server running with 16 clients.

Clients	Transactions	Aborts	Abort rate [%]		Mean [B]	Max [B]	Total [MB]
1	190206	0	0.0	Reads	65.1	58511	12
				Writes	5.2	20102	1
2	367118	826	0.2	Reads	66.0	62728	25
				Writes	5.7	24397	2
4	655020	4165	0.6	Reads	83.7	80275	55
				Writes	8.2	39726	5
8	1439874	20593	1.4	Reads	102.5	102470	145
				Writes	9.6	57552	14
16	3226759	131814	4.1	Reads	133.3	231593	192
				Writes	15.5	211651	22

Table 2.3: Transactional statistic of the fine-grained server running with 8 threads.

2.5 Lessons Learned

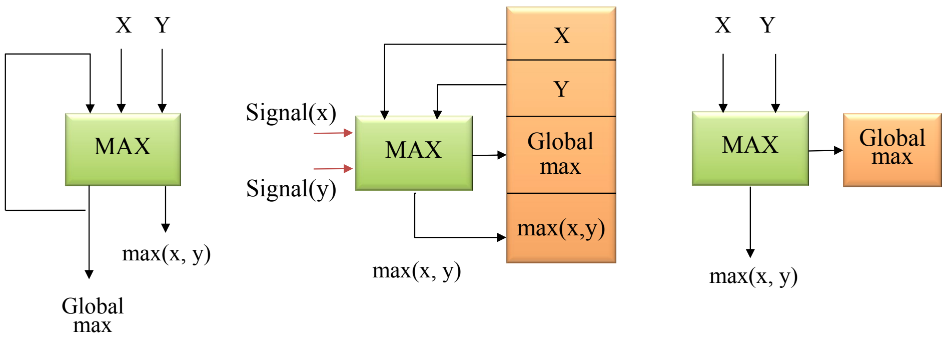
The importance of this work is that it has introduced QuakeTM – the first complex real-world TM application that was developed directly from a sequential version using transactional memory. We have made a detailed description and commented on the challenges involved in the development process. The emphasis was on testing the TM programmability, especially using a coarse-grained parallelization approach. As a result, QuakeTM is characterized with large atomic sections that put too much pressure on the underlying STM system. The evaluation clearly shows that the transactional overhead that results in 6x slowdown and the abort rate that goes up to 35.3% are excessive and cannot be compensated with the speedup from parallel execution. Consequently, a coarse-grained approach is not a viable option for the current STM systems. Moreover, we have shown that the read and write set sizes are significant, which could impose serious problems for hardware TM systems. Finally, we were surprised by the amount of the programmer time investment. The results suggest that the only option left for a programmer is to take a fine-grained parallelization approach in order to avoid instrumentation overheads of transactional memory. Moreover, this approach almost entirely eliminates data conflicts between concurrent transactions, which decreases the number of transaction aborts. However, a fine-grained implementation of such a complex system as the Quake game engine is quite difficult and requires considerable programmer effort that exceeds significantly the effort required to implement a coarse-grained approach based on transactional memory. Therefore, the conclusion is that some other approach, based on fine grained transactions, is necessary for the efficient parallelization of such a complex system as a game engine. In Chapter 5 we show how using dataflow can minimize synchronization requirements of parallel game engine and allow easier and more efficient implementation.

Chapter 3

Atomic Dataflow Model

The main argument of this thesis is that dataflow and shared memory programming can be integrated into a more versatile programming model that can provide a user with more expressive concurrency abstractions that are sufficiently easy to reason about. Accordingly, this section introduces the Atomic Dataflow Model (ADF) - a programming model for shared-memory systems that combines aspects of dataflow programming with the use of explicitly mutable state. The key idea is that computation is triggered by dataflow between ADF components (tasks) but that, within a component, execution occurs by making atomic updates to the common mutable state. This enables a programmer to delineate a program into a set of tasks and to define explicit dependencies between them. However, the key aspect of the proposed model is that it does not require a programmer to explicitly state all of the task's dependencies, but only those that are inherently present in an algorithm and thus easy to comprehend. The remainder of the inter-task dependencies, in particular those that are not easily observable and are often hard to reason about, are automatically managed by the ADF model. Specifically, a body of a task is enclosed inside an implicit memory transaction, which provides an easy-to-program optimistic concurrency substrate and enables a task to safely share data with other concurrent tasks in the system. This leads to a flexible form of parallelism in which computation can be carried out in a pure dataflow fashion, in a common task-based style of shared memory programming, or in a combination of these two methods. Therefore, compared to each of the aforementioned techniques alone, the ADF model can represent a wider range of problems naturally.

Furthermore, the ADF model is able to overcome some deficiencies specific to each of these techniques by combining their positive aspects. For example, synchronization related problems specific to shared memory programming could be avoided by promoting highly conflicting shared data into dataflow variables. This is especially important when optimistic concurrency technique, such as transactional memory, is used to maintain shared data consistency. The



```
in(x);
in(y);
in(g_max);
{
  z=max(x, y);
  g_max=max(g_max,z);
  out(z);
  out(g_max);
}
```

a)

```
cond_wait(x);
read(x);
cond_wait(y);
read(y);
z=max(x, y);
atomic {
  if (z>g_max)
    g_max=z;
}
```

b)

```
in(x);
in(y);
{
  z=max(x, y);
  atomic {
    if (z>g_max)
      g_max=z;
  }
  out(z);
}
```

c)

Figure 3.1: Max function with the global maximum: a) dataflow, b) shared memory and c) the ADF implementation.

previous chapter shows one such example. Specifically, it shows that a coarse grained approach to game engine parallelization using transactional memory is characterized with high abort rate of memory transactions due to a high level of data sharing between atomic regions, whereas a fine grained approach, which is not affected with the high level of data conflicts, may be too restrictive from the programmability point of view. Therefore, the idea behind the ADF model is that promoting highly conflicting data into dataflow variables and orchestrating execution of dependent tasks based on their mutual dependencies may decrease transactional memory conflicts and aborts. On the other hand, due to single-assignment semantics, a pure dataflow model requires that a new copy of the data is produced each time the data is updated. Consequently, the problem occurs when the updated data is some complex data structure such as an array, which effectively makes updates expensive. Previously, I-structures [Arvind *et al.* (1987)] were proposed to circumvent this problem. Alternatively, the ADF model allows for the complex data to be treated as a mutable state and updated in-place by executing a task inside an implicit memory transaction.

To illustrate the advantages of the ADF model let us consider a simple example from Figure 3.1. Given two input streams of integers, x and y, the goal is to discover which of the current values of these two streams is larger, but we would also like to find the global maximum of all values received on both streams. Dataflow model handles the input streams naturally, but in order to maintain the global maximum the state of the previous execution has to be provided as an input for the next execution - Figure 3.1a. This means that the state has to be copied after each invocations of the MAX node. On the contrary, shared memory model treats the global maximum as the mutable state which can easily be updated in-place - Figure 3.1b. Still, we need to synchronize the access to this data in order to allow safe updates by concurrent threads. The issue with the shared memory implementation is how to deal with input data streams. A typical solution is to rely on some form of conditional waiting, i.e. conditional variables or transactional memory retry mechanism [Harris *et al.* (2005)], through which we implement coordination between different threads of execution. Ultimately, the ADF model, being the combination of the previous two models, can handle both of the above problems naturally. Figure 3.1c illustrates the idea: we handle the streams and find the current maximum in a dataflow fashion, while the global maximum is treated as a mutable state.

We are not the first to recognize the potential of including the state into the dataflow model. The work on M-structures [Barth *et al.* (1991)] was one of the early attempts in this direction. Similarly, the concept of Monads [Peyton Jones *et al.* (1996)] introduced the state into the pure functional programming language, Haskell.

Recently, transactional memory researchers have investigated ways to provide more robust cooperation between concurrent transactions. Luchango and Marathe introduced transaction communicators [Luchangco & Marathe (2011)], as special objects through which concurrent transactions can communicate and interact. The system tracks dependencies between transactions that access communicators and assures that mutually dependent transactions commit or abort together as a group. Similarly, transactional events [Donnelly & Fluet (2006)] combine properties of synchronous message-passing events with the isolation property of transactions. Central to this abstraction is a sequencing combinator which enables sequential composition of events. Dependence-aware transactional memory (DATM) [Ramadan *et al.* (2008)] is an implementation of transactional memory based on conflict serializebility. The runtime system tracks data dependencies between transactions and serializes the commit of dependent transactions. Finally, a model of Communicating Memory Transactions (CMT) [Lesani & Palsberg (2011)], combines the Actor model with the software transactional memory in order to allow transactions to coordinate across actors or threads. In this model, concurrent transactions that cooperate by exchanging tentative messages form a set which effectively assures that either all transactions commit successfully or they all abort. Compared to these proposals, in the ADF model we decouple the dataflow and transactional execution: data dependency information is explicitly defined by a programmer and used to construct the dataflow graph that governs the scheduling of tasks based on the availability of task input data. Transactional execution is managed by the

TM runtime system and it does not influence the dataflow execution. It is used only to protect mutually shared data.

The ADF model enforces high level abstractions that provide portable programming model for focused set of applications. We are primarily concerned with problems that exhibit irregular parallelism, but the model is applicable to other domains as well. The characteristic of irregular problems is that the distribution of the work and data they use cannot be determined a priori, because these properties are input-dependent and evolve with the computation itself. Programming these problems in the shared memory model using transactional memory is possible but results in poor performance [Kulkarni *et al.* (2007)]. In addition, previous research on programming scientific irregular applications in a distributed environment using message passing model [Nikolopoulos *et al.* (2001)] demonstrates that irregular task parallelism can be effectively exploited in such environment. This thesis will show that the ADF model presents a new approach in dealing with the irregular parallelism and that, beside its ability to efficiently support programming of dataflow-like and stream-like applications, it may as well be used for implementation of conventional shared memory applications. On the other hand, ADF is not intended for forms of parallelism that can be efficiently exploited automatically (i.e. loop level parallelism).

3.1 Model Description

This section describes the language support for the ADF model in the form of C/C++ pragma directives. It further introduces the ADF model application programming interface and provides an overview of the ADF execution model.

3.1.1 Language support

The ADF model is based on execution of atomic units of work called ADF tasks. It provides language constructs that allow a programmer to delineate a program into a set of tasks and to explicitly define data dependencies for each task. The ADF runtime system coordinates the execution of tasks based on the flow of data, and guarantees that each task accesses shared data atomically.

At the beginning of a program, the main thread initializes the runtime system and creates all ADF tasks defined by a programmer. Then, it initiates the dataflow execution using the following ADF pragma directive:

```
#pragma adf_start
```

The execution of ADF tasks that follows is self-scheduled and governed only by the production of data. The main thread waits for the end of the dataflow execution using the second ADF directive:

```
#pragma adf_taskwait
```

A task is defined using the *adf_task* pragma directive with the following syntax:

```
#pragma adf_task    [ trigger_set (list) ]
                    [ instances (integer-expression) ]
                    [ until (exit_condition) ]
                    [ execute (integer-expression) ]
                    [ pin (integer-expression) ]
                    [ relaxed ]
{ < task_body > }
```

Data dependencies for a task are defined using the `trigger_set` clause. The trigger set is a set of data that controls firing of an ADF task. In particular, the ADF runtime system schedules tasks based on the availability of the trigger set data. A consumer task blocks if its input dependencies are not satisfied and waits for a producer task to commit new data values. When the input is ready, the consumer ADF task atomically processes the data, commits the changes and returns to the beginning of the task to wait for new data. During commit, the consumer task may produce the data needed by some other ADF task, which then processes the data further. Since each ADF task is triggered only when a new data value is produced, the execution of an ADF program is coordinated by a flow of data.

Usually, dataflow execution stops once it achieves the goal of the calculation; e.g. when the product of two matrices has been found. However, there are cases in which the execution should proceed until some external signal is received. For example, a game server runs indefinitely processing clients requests and terminates only when it is explicitly shutdown. In such case, the exit condition for a task can be defined using the `until` clause.

Similarly, the `execute` clause terminates the task after a given number of executions. This clause is useful for controlling the execution of tasks that need to execute only once, or when the number of executions is otherwise restricted by the algorithm.

By default, the ADF runtime system creates a single instance of a task. However, the `instances` clause may be used in cases when many instances of the same task are necessary. For example, given a bounded buffer problem, there are only two possible tasks: Produce and Consume. Still, we can have many instances of these tasks that are simultaneously accessing the bounded buffer.

```
/* programmer API */
void adf_init (int num_threads);
void adf_terminate();

/* internal API */
void adf_start ();
void adf_taskwait();
void adf_create_task ( int num_instances, int num_tokens, void *tokens[],
                       std::function <void (token_t *)> fn );
void adf_pass_token ( void *addr, void *token, size_t token_size );
void adf_task_stop();
```

Figure 3.2: The ADF model API.

The main purpose of the `pin` clause is to bind a task to a given thread in order to preserve cache locality between successive task executions. In addition, it simplifies implementation of certain programming tasks. For example, in a game engine implementation, the render task is usually executed by a dedicated thread to avoid unexpected interleaving of graphics library callbacks.

Finally, by using the `relaxed` option it is possible to switch off the implicit TM synchronization of the task body. The implicit transaction guarantees the atomicity of the task execution in regard to the shared memory operations, but in some cases a programmer can provide more optimal synchronization by inserting transactions manually within the body of the task.

3.1.2 Application Programming Interface

The atomic dataflow model API is shown in Figure 3.2. Only the first two routines are exposed to a programmer, while the remaining five runtime calls serve as the support for the expansion of the ADF pragma directives.

Every ADF program has to call `adf_init` at the beginning of the main function to initialize the ADF runtime system. This function initializes the task scheduler and transactional memory support, and creates a pool of worker threads.

At the end of the program, the main thread calls the `adf_terminate` routine. This sends the signal to the worker threads to stop the execution and exit. When the last worker thread has terminated, the runtime system destroys the task graph, all remaining tasks and the runtime support for task scheduling. Then, it calls STM exit routines and optionally prints the ADF runtime statistics.

The first two internal runtime calls serve as a support for `adf_start` and `adf_taskwait` pragma directives. The final three routines from Figure 3.2 are used for the transformation of the `adf_task` directive. To create a task the program calls `adf_create_task` routine. This call accepts four parameters: 1) the number of task instances that should be created for this task, 2) the number of input tokens, 3) the pointer to the list of consumed input tokens which is provided by the ADF runtime system, and 4) the name of the outlined procedure that represents the task. During the dataflow execution, when the task instance consumes a set of input tokens, it becomes their exclusive owner. The tokens are used for the single execution of the task instance, after which they are destroyed. Finally, when a worker thread finds a ready task, it executes the task by calling an outlined procedure `fn`, passing the list of tokens as the parameter to the call.

If the task generates output tokens, it calls `adf_pass_token` routine to pass the tokens to their consumer tasks. The runtime system uses `addr` parameter to access the map that associates token addresses with input token buffers of consumer tasks. A copy of a token is provided for each of these token buffers.

If the `until` clause is used in the definition of the task, and the exit condition of the clause is set, the task calls the `adf_task_stop` routine to stop its execution. This means that the task stops consuming input tokens and removes itself from the further dataflow execution.

3.1.3 The ADF Execution Model

The ADF program is based on the dataflow execution of ADF tasks. An ADF task processes one set of data, produces output data and then waits for the new matching set of input data. As such, the ADF task operates as a macro dataflow actor [Iannucci (1988), Silc *et al.* (1998)] which repeats the execution as long as the tokens are present on its inputs or until the exit condition is met. The exit condition acts as a control input for the task and can be set either by the task itself, or by some other program thread.

The ADF program execution proceeds as follows. The main program thread

- initializes the ADF runtime system,
- creates all ADF tasks defined by a programmer,
- generates initial tokens that are necessary to start dataflow execution,
- and starts the dataflow execution.
- Then, it waits for the end of the dataflow execution after which it
- destroys all constituent tasks,
- stops the ADF runtime and
- terminates the program.

Each ADF task is defined by its context, a reference to the outlined procedure that represents the body of the task, and a number of input data tokens on which the task depends. All instances of a given task share the same context and reference the same outlined procedure, but their execution is based on different sets of input token values. For each input token, the ADF runtime system creates an implicit buffer to store different token values, thus enabling asynchronous execution of producer and consumer tasks. Conceptually, this resembles a dynamic dataflow model [Arvind & Culler (1986)] which allows any number of tokens to reside on a given arc. Since different tasks can depend on the same token, the runtime system maps the address of a token with the list of its corresponding token buffers. When a new token value is produced, the runtime system uses this map to pass the value to the correct token buffers. Therefore, this map represents the data dependency graph between all the tasks in a given program.

The ADF runtime system also creates a pool of worker threads that spin in a loop executing ready tasks. If a thread cannot find a ready task it blocks until a new task is enabled. A thread executes a task by calling an outlined procedure with a given set of matched input tokens. Once the execution of the task starts, the input tokens are consumed and they cannot be referenced by any other task. Thus, the task can access token variables safely, without the need for synchronization. Upon the end of a single execution of the task, the thread tries to consume a new set of input tokens. If all input tokens are already present in input buffers, the new set is matched and the task is enabled. Otherwise, the thread enqueues the task into the waiting queue associated with the buffer that corresponds to the missing token. There, the task remains blocked until the missing token is generated by some other task.

By default, the body of the task is executed atomically. This allows a task to safely share the state with other running tasks. In case of a conflict, the task transaction aborts and starts a new execution. However, this does not affect the dataflow execution since the tokens are privatized by the task. Thus, the task simply restarts the transaction with the initial state of the input tokens. On the other hand, the shared state is certainly changed. Therefore, the atomic dataflow model doesn't impose any ordering between the dataflow execution and shared memory operations of different tasks.

During the execution, the task can generate new values for output tokens. These values are passed to the runtime system, which uses the address of each token to access the token-to-buffer mapping and acquire the list of token buffers that correspond to the given token. For each buffer from this list, the runtime system creates a copy of the token. Further, if the waiting task queue associated with a buffer is not empty, the runtime system dequeues the first task and passes the copy of the token directly to it. Then, the runtime system checks the remaining data dependencies for the task and, if all dependencies are satisfied, enables the task. Otherwise, it moves the task into the waiting queue for the next missing token. Therefore, the token is passed directly to the waiting task and put into the buffer only if all consuming tasks are busy.

Figure 3.3 illustrates the execution of the ADF model on the example of multiple producers and multiple consumers that synchronize on data x. Shaded circles denote busy tasks, while

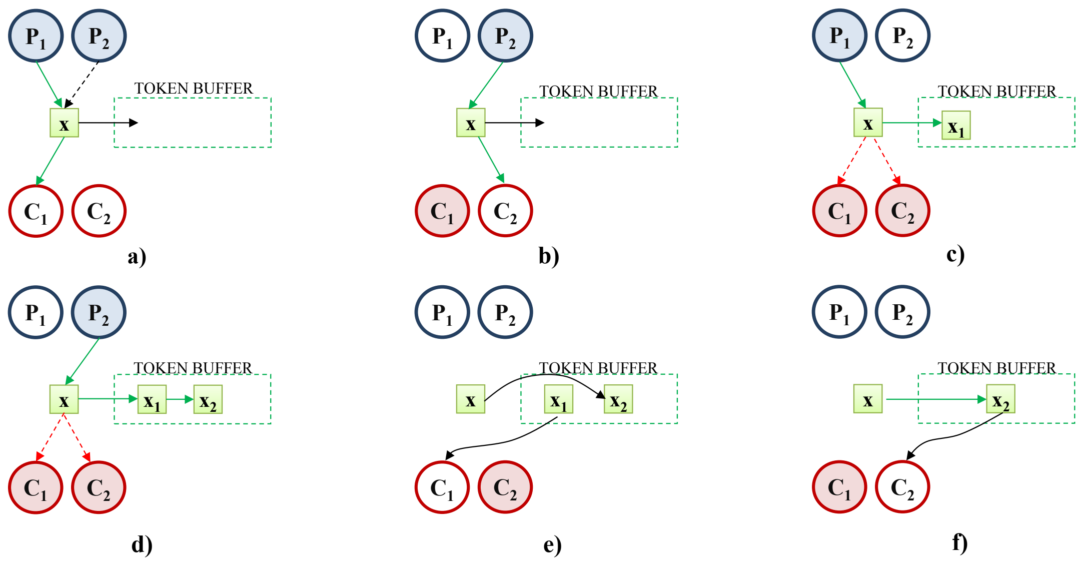


Figure 3.3: Illustration of the ADF execution model on the example of multiple producers (P) and multiple consumers (C) that synchronize on data x: a) P1 produces a token for C1, b) P2 produces a token for C2, c) P1 stores a token x_1 into the buffer, d) P2 stores a token x_2 into the buffer, e) C1 consumes token x_1 and f) C2 consumes token x_2.

white circles are idle tasks. Figure 3.3a shows two producers that are concurrently producing a new value for x. Producer P1 commits first and passes the new token directly to the consumer C1. Next, the producer P2 commits a new token and passes it to the consumer C2 – Figure 3.3b. Then, in Figure 3.3c the producer P1 produces a new token, but this time all consumers are busy. Therefore, the ADF runtime system puts the new token x_1 into the buffer. The same action is performed for the second token produced by the producer P2 – token x_2 in Figure 3.3d. In the meantime, the consumer C1 has finished the previous execution and is ready to start a new one. Hence, it checks the buffer and finds the token x_1 ready, takes it from the buffer (thus consuming the token) and immediately continues with the execution – Figure 3.3e. Finally, the consumer C2 consumes the token x_2 and continues the execution as well – Figure 3.3f. Thus, a consumer only needs to block if there are no input tokens in the buffer.

3.1.4 Scheduling

Analogous to the dataflow model in which enabled instructions are scheduled to available execution units, in the ADF model enabled tasks are scheduled to worker threads. The core task scheduler is based on the LIFO version of the work-stealing lock-free queues [Chase & Lev (2005)]. When the task produces tokens that enable consumer tasks, the thread pushes these tasks into its own task queue. The push operation always succeeds and it does not require synchronization. When it needs to get a new task, the thread checks its own queue first. If the queue is empty, the thread attempts to steal a task from another thread in a random fashion.

```

void producer() {
  while (true) {
    transaction {
      if (itemCount == BUFFER_SIZE)
        retry;
      item = produceItem();
      putItemIntoBuffer(item);
      itemCount = itemCount + 1;
    }
  }
}

void consumer() {
  while (true) {
    transaction {
      if (itemCount == 0)
        retry;
      item = getItemFromBuffer();
      itemCount = itemCount - 1;
      consumeItem(item);
    }
  }
}

int main() {

  #pragma omp parallel
  {
    #pragma omp single
    for (int i=0; i<num_tasks; i++) {
      #pragma omp task
        producer();
      #pragma omp task
        consumer();
    }
  }
}
```

Listing 3.1: The OpenMP implementation of the bounded buffer.

The problem with work-stealing is that it has negative effects on cache locality, which can be particularly detrimental in a dataflow model [Kavi *et al.* (1995)]. Specifically, a thread that executes a task that produces data which then enables a consumer task (i.e. producing thread) is the best candidate for the execution of the consumer task because the data is already in the cache. However, in the work-stealing environment, some other thread can steal the consumer task from the producing thread, thus hindering the benefits of cache locality. Therefore, the core task scheduler is extended with a per-thread task reservation slot, in which the producing thread can store a single task from a set of enabled tasks. This task is not visible to other threads and cannot be stolen. Hence, the producing thread can immediately execute the reserved task and benefit from cache locality.

Both scheduler implementations also support task binding using the `pin` clause. When a thread needs to find a new ready task, it first checks its bind task queue, then the reservation slot, then its own work-stealing queue, and finally it tries to steal a task from other threads.

```

void producer() {
  item = produceItem();
  putItemIntoBuffer(item);
  consumeToken = 1;
}

void consumer() {
  item = getItemFromBuffer();
  consumeItem(item);
  produceToken = 1;
}

int main() {

  #pragma adf_task trigger_set(produceToken) instances(num_producers)
    producer();

  #pragma adf_task trigger_set(consumeToken) instances(num_consumers)
    consumer();

  /* create_initial_tokens */
  #pragma adf_task execute(BUFFER_SIZE)
    produceToken = 1;
}
```

Listing 3.2: The ADF implementation of the bounded buffer.

3.2 Programming with Atomic Dataflow Tasks

This section illustrates the use of the ADF model for the implementation of the multiple producer, multiple consumer bounded buffer. Listing 3.1 shows the OpenMP implementation in which the main thread creates a number of producer and consumer tasks. Each task executes an infinite loop, in which an iteration represents a single operation on a bounded buffer. The task has to check the buffer condition before it can proceed with the operation. Hence, producers check if the buffer is full before they produce a new item and store it into the buffer, while the consumers check if the buffer is empty before they try to take the item from it. These operations are executed inside a transaction that also utilizes retry mechanism in order to avoid unnecessary re-executions until a given operation is possible.

In the ADF implementation – Listing 3.2 – the main thread creates a number of instances of *Producer* and *Consumer* tasks, which operate on a shared bounded buffer. All instances of the *Producer* task share a single token buffer to store *produce* input tokens. Similarly, all instances of the *Consumer* task share the same *consume* token buffer. Next, the main thread creates a number of initial *produce* tokens to initiate the dataflow execution. The execution that follows is self-scheduled and coordinated by the production of *consume* and *produce* tokens by corresponding *Produce* and *Consume* tasks. For example, a *Producer* task produces an item, stores it into the buffer and generates the *consume* token that in turn enables a single *Consume* task. Since the tasks are enabled only when the tokens are available, there is no need to check the state of the buffer. For example, if the buffer is empty, there will be no *consume* tokens available and all *Consume* tasks will be blocked. As a result, in the ADF implementation there is no need to maintain *itemCount* counter, which effectively eliminates transactional memory conflicts caused by this counter.

Although in a dataflow implementation of a pure producer consumer problem it is not necessary to share the state because the items are stored as tokens in an implicit buffer for the *Consume* task, in this example the requirement is that all items have to be stored in a bounded buffer allocated as an array of memory locations. Therefore, in the ADF implementation shown in Listing 3.2 the accesses to the buffer from the *producer* and *consumer* functions have to be synchronized. While in the OpenMP implementation these acccesses are synchronized with explicit transactions, in the ADF implementation the executions of both task, and consequently the accesses to the bounded buffer, are protected by implicit transactions. Evidently, this improves the programmability as it provides a programmer the guarantee that data accessed from within a task will be consistent. However, as the work on QuakeTM has shown, it is important for the performance that the duration of a task and the amount of data that it accesses are not unreasonably large, in order to limit the overhead of transactional memory execution. Still, this example clearly shows the benefits of dataflow, and the ADF model in particular, for decreasing the number of transaction aborts. Specifically, dataflow eliminates the need for the shared counter, *itemCount*, which is the main source of conflicts in the OpenMP implementation.

3.2.1 The Runtime Characteristics of the ADF model

This section starts with the analysis of the overhead introduced by the ADF model using the bounded buffer example. The experiments were conducted on two different machines: the one with four dual-core 64-bit Intel Xeon processors running at 3.2 GHz, and the other with two 6-Core 64-bit Intel Xeon CPU X5650 Westmere processors running at 2.67GHz that are two-way SMT capable, giving a total number of 24 hardware threads. The overhead is measured using the built-in per-thread support for statistics gathering based on *rdtsc* instruction.

Since all ADF tasks are created only once, before the beginning of the dataflow execution, the ADF overhead can be broken into the task creation overhead and the task handling overhead. The profiling shows that the overhead of creating a single task is 6250 cycles or 1.95us, while the time needed to create a task instance is 2000 cycles or 0.62us. The next value measured is the task handling overhead that consists of the time wasted on all other task related operations except the task creation. This is done by comparing the time it takes for the single threaded execution of the ADF implementation to execute 10000 tasks with the time required by the optimized sequential implementation to executes the same task inside the for loop with 10000 iterations. The result of this comparison is that the ADF task handling overhead is

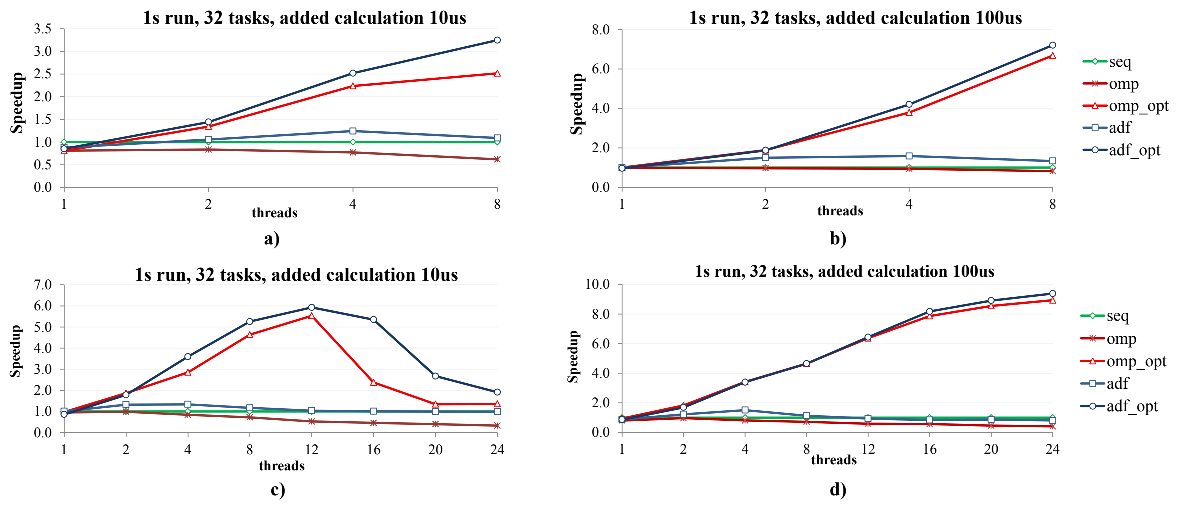


Figure 3.4: Bounded Buffer: results for default and relaxed ADF and OpenMP implementations normalized against the results of the sequential implementation, when the added calculation time is 10us and 100us.

approximately 6700 cycles or 2.1us per task. This value provides a lower limit for the duration of ADF tasks, since the performance of an ADF program that is characterized with very fine-grained, short tasks would be dominated by the ADF task handling overhead.

Moreover, the bounded buffer application is also suitable for testing the throughput of the ADF model. For that purpose, the number of task instances for both *Producer* and *Consumer* tasks is set to 32. The program starts by creating 32 initial *produce* tokens. The diagrams in Figure 3.4 show the speedup of non-optimized and optimized ADF and OpenMP implementations normalized against the results of the sequential execution. Non-optimized ADF implementation is the one with implicit task transactions, while in the non-optimized OpenMP implementation the whole task body is enclosed in a transaction. In both the ADF and OpenMP optimized implementations the transactions are optimally placed to avoid useful calculation, simulated by adding an arbitrary calculation time to *produceItem* and *consumeItem* functions. This useful calculation is implemented using an unoptimized for loop in order to prevent the threads to yield the processor.

Compared to the sequential execution, concurrent tasks in parallel implementations have to synchronize the access to the bounded buffer. Producer tasks synchronize on the tail pointer, while consumer tasks synchronize on the head pointer. Inevitably this adds contention overhead to parallel execution. Also, parallelism depends on the amount of time that each task spends in calculation before it accesses the buffer. From Figure 3.4 it is clear that the contention for the buffer access is the performance bottleneck in non-optimized parallel implementations, because the useful calculation is part of the task transaction. This means that calculation has to be repeated each time the transaction aborts. On the other hand, optimized parallel implementations avoid repeating the calculation and are thus able to deliver better performance.

The synchronization bottleneck in the bounded buffer application is best illustrated by the diagram in Figure 3.4c. Although the optimized versions perform the calculation outside of the transaction, the contention for the buffer access becomes high as the number of threads starts increasing. For the small number of threads, the 10us useful calculation is sufficient to hide performance penalties of buffer synchronization and runtime task manipulation, but after 12 threads the performance starts to drop as a result of increased contention. The 100us useful calculation can effectively hide more performance penalties, but the same behavior as in the previous case is expected, only for larger thread counts.

Importantly, the ADF implementations outperform their OpenMP counterparts. This is due to ability of the ADF to avoid the contention involved in testing the empty and the full conditions of the bounded buffer by scheduling tasks only when their input data is ready. This further supports the premise that by incorporating dataflow principles, the ADF model offers better support for producer-consumer class of problems than the pure shared memory model.

Chapter 4

The DaSH Benchmark Suite

The value of a programming model is usually judged on its generality: how well a range of different problems can be expressed and how well they execute on a range of different architectures. This section is primarely dedicated to the evaluation of the Atomic dataflow model, but it also provides a more general evaluation of the current hybrid dataflow models. This type of task can not be properly fullfiled unless the evaluation is general enough, and since there are no benchmark suites that cover a wide-enough range of problems at the moment, we have implemented the first comprehensive benchmark suite for hybrid datafow models – DaSH. This section provides the implementation details of each of the benchmarks from DaSH, accompanied with extensive programmability and performance evaluation.

While developing DaSH, we have followed the approach from Berkeley that has identified a set of 13 dwarfs that capture patterns of communication and computation common to a wide range of emerging applications [Asanovic *et al.* (2009)]. Dwarfs are algorithmic methods that *constitute equivalence classes where membership in a class is defined by similarity in computation and data movement. The dwarfs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future.*

The decision to develop DaSH by implementing a single application from each dwarf category characterizes DaSH with comprehensive variety of algorithms and application domains. Currently, DaSH consists of 11 benchmarks. For each benchmark DaSH provides sequential implementation, two shared memory implementations based on work-sharing and/or tasking (one that uses OpenMP and the other that uses TBB) and three dataflow implementations realized using the ADF, OmpSs [Perez *et al.* (2008),Duran *et al.* (2011)] and Intel TBB Flow

Graph [Intel (2007-)] hybrid dataflow models. Two dwarfs are not supported. Combinatorial logic dwarf describes problems that exploit bit-level parallelism by performing simple operations on very large amounts of data (computing checksums or CRCs). Since the execution is heavily dominated by the time to read the data, the benefits of parallelization are quite limited and so is the usefulness of this dwarf for programming model evaluation. Graph traversal dwarf should be covered in the next version of the DaSH benchmarks suite.

Therefore, DaSH is designed for the evaluation of hybrid dataflow models in general, and not specifically for the ADF model, which facilitates the comparison of the ADF model with other similar models. Previous research has shown that hybrid dataflow models provide a good support for parallelization of dense linear algebra problems [Kurzak *et al.* (2010)]. The characteristic of such implementations is that they avoid global synchronization and improve resource exploitation. Similarly, a dataflow implementation of the Fast Multipole Method [Amer *et al.* (2013)] provides up to 22% better performance than highly optimized OpenMP implementation. Further, hybrid dataflow models have been successfully used for the implementation of P-means parallel clustering algorithm [Foina *et al.* (2011)], integral histogram [Bellens *et al.* (2011)] and Lee's routing algorithm [Seaton *et al.* (2012)]. Using DaSH for the evaluation of hybrid dataflow models shows that the main strength of these models is the ability to eliminate unnecessary barriers and thus expose more parallelism. Morevoer, in two DaSH benchmarks dataflow provides barrier-free implementation even when the algorithm inherently depends on barriers. Accordingly, most of the performance gain comes as the result of the increased parallelism and decreased thread idle time. This chapter provides comprehensive evaluation of the DaSH benchmarks that compares hybrid dataflow implementations and shared memory implementations, both in terms of programmability and performance.

4.1 Hybrid Dataflow Models

This section provides an overview of hybrid parallel programming models that combine dataflow and shared memory programming. Similarly to ADF, these models are based on execution of tasks that are scheduled, according to the dataflow principles, when their input dependencies are satisfied. Typically, a programmer explicitly defines input and/or output dependencies for each task. This information is then used by the runtime system to construct dependency graph that governs the execution of a program. As tasks execute, they produce data on which other program tasks depend on. Once all input dependencies for a given task are satisfied, the task becomes enabled and can be scheduled for execution. The tasks are executed by worker threads and scheduling is typically based on work-stealing.

Contrary to a pure dataflow model, which assumes side-effect free execution of dataflow tasks, a hybrid dataflow model can benefit from the underlying shared memory architecture by allowing dataflow tasks to share data. Naturally, accessing shared state may require synchronization. In this thesis transactional memory (TM) [11] is used for the shared state synchronization because it integrates seamlessly into the dataflow model. In particular, both abstractions exhibit isolation property: dataflow in terms of execution of data dependent tasks and transactional memory in terms of concurrent accesses to the shared state by different sharers.

4.1.1 OmpSs

OmpSs [Duran *et al.* (2011)] extends OpenMP with a support for asynchronous parallelism that is based on execution of data-dependent tasks. Specifically, OpenMP task directive is extended with three additional clauses - `in`, `out` and `inout` - that a programmer may use to explicitly declare data dependencies for a task. When a new task is created, the runtime system matches its *in* and *out* dependencies against dependencies of all existing tasks. If the match is found, the new task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime. Tasks are scheduled for execution as soon as their predecessors in the graph have finished, or at creation, if they have no predecessors. The task construct is further extended with the `concurrent` clause that allows a number of instances of the task to execute simultaneously. However, a programmer must ensure that additional synchronization is used inside a task, if necessary.

4.1.2 Intel TBB Flow Graph

Starting from version 4.0, Intel Threading Building Blocks (TBB) framework [Intel (2007-)] includes Flow Graph extension that enables construction of task dependency graphs. The flow graph consists of three primary components: a graph object, nodes, and edges. The graph object provides methods to run the graph and to wait for its completion. Nodes generate, buffer, or transform messages and data. In particular, functional nodes execute user code. Buffering nodes implement different forms of data buffering. The rest of the nodes support various forms of control and communication, such as split, join, broadcast etc. Concurrency of functional nodes can be explicitly controlled, which means that a number of instances of the same node can be executed in parallel if multiple data items exist on the node's input port. Edges connect the nodes and represent channels through which nodes communicate and exchange data. In a nutshell, programmers explicitly create nodes and edges that express computations and dependencies between these computations.

```

SomeType Produce();
void Consume(SomeType);
SomeType consumeToken;

void main() {
  while ( itemsToProduce-- ) {
    #pragma omp task out(consumeToken) concurrency(nthreads)
    SomeType item = Produce();
    #pragma omp task in(consumeToken) concurrency(nthreads)
    Consume(item);
  }
}
```

Listing 4.1: OmpSs implementation of multiple producer – multiple consumer problem.

```

void main() {
  #pragma adf task instances(nthreads) execute(numItemsToProduce)
  consumeToken = Produce();

  #pragma adf task trigger_set(consumeToken) instances(nthreads)
  Consume(consumeToken);

  #pragma adf start
  #pragma adf taskwait
}
```

Listing 4.2: ADF implementation of multiple producer – multiple consumer problem.

```

void main() {
  graph g;
  function_node<continue_msg, SomeType> producer(g, concurrency=nthreads, [=] () -> SomeType {
    return Produce();
  });

  function_node<SomeType> consumer(g, concurrency=nthreads, [=](SomeType input)->void {
    Consume(input);
  });

  make_edge(producer, consumer)

  for (int i = 0; i < numItemsToProduce; i++)
    producer.try_put(continue_msg());
  g.wait_for_all();
}
```

Listing 4.3: Intel TBB Flow Graph framewrok implementation of multiple producer – multiple consumer problem.

4.1.3 The Differences Between Models

The main difference between these three models is the type of the dependency graph that the model relies on: OmpSs constructs a task dependency graph, while the other two models build a data dependency graph. In the next section, we will show how this choice may affect the programmability of a given model. Further, OmpSs builds a graph dynamically while the tasks are being created, whereas the ADF model and the TBB Flow Graph framework build their graphs statically, before the start of the dataflow execution. However, in these two models, tasks are being reused as many times as their input dependencies are satisfied. On the contrary, in OmpSs each task executes only once, after which it is discarded, thus maintaining similar semantics with OpenMP tasks.

The models also differ in a programming interface through which they allow a user to express dependencies. Listings 4.1–4.3 illustrate the use of dataflow constructs of these three models for the implementation of multiple producer multiple consumer problem. OmpSs uses task directive data directionality clauses to instruct the runtime how to automatically connect tasks in existing task graph – Listing 4.1. In this model both input and output dependencies must be defined, and the runtime system constructs a task dependency graph. Conversely, in the ADF model, a programmer expresses only input dependencies using dataflow tokens – Listing 4.2. The runtime system uses input task dependency information to construct a data dependency graph between tasks automatically. On the contrary, using TBB, a programmer has to construct entire graph explicitly by connecting the output port of a given node with an input port of a consumer node – Listing 4.3. The TBB framework provides `make_edge` template function for this purpose. Thus, both the nodes and the edges must be defined explicitly.

4.1.4 Related Work

Beside the three hybrid dataflow models that we use in this thesis, there are a number of alternatives. Microsoft TPL Dataflow Library [Microsoft (2007-)] promotes actor-based programming by providing in-process message passing for coarse-grained dataflow and pipelining tasks in a similar way as the TBB flow graph framework.

Charm++ [Kale & Krishnan (1993)] supports directed acyclic graph execution and utilizes implicit message-passing to coordinate execution of actors across system nodes. Charm++ was later used to study the memory-related issues of large-scale LU factorization [Dooley *et al.* (2010)]. The authors propose a memory-aware scheduler in order to restrict the memory usage by limiting the concurrency. Data-Driven Multithreading (DDM) [Stavrou *et al.* (2007)] is a model that provides dataflow scheduling of threads. In DDM a programmer is responsible for finding parts of the code that can execute in parallel and to explicitly express dependencies of each part. The programming interface is based on DFScala [Goodman *et al.* (2012)], which is a dataflow library for Scala. TIDeFlow [Orozco *et al.* (2011)] is a parallel execution model that is designed for efficient development of HPC programs for many-core architectures. The execution in TIDeFlow relies on shared memory to transfer data between actors that represent parallel loops and uses queues to distribute work among processors. Software data-triggered threads (DDT) [Tseng & Tullsen (2012)] is a programming model that relies on C pragma directives to declare data triggers for program threads. Using these pragmas, a programmer attaches a support thread to a variable that becomes a data trigger, upon which any changes to this variable spawn a new thread to execute associated support function. The evaluation of the SPEC2000 and Parsec benchmarks adapted to DDT shows promising results of this programming model.

To the best of our knowledge, DaSH is the first comprehensive benchmark suite for hybrid dataflow models. To prove the concept, existing hybrid dataflow models provide implementations of a few common benchmarks. Typically, these are matrix multiplication, Cholesky decomposition, LU and QR factorization, and other algebra applications. In addition, Amer et al. [Amer *et al.* (2013)] have studied differences between data-driven and fork-join task parallelism execution of the Fast multipole method, suggesting that data-driven implementation can provide up to 22% better performance by avoiding the barriers and reducing the memory-bandwidth. Furthermore, Seaton et al. [Seaton *et al.* (2012)] have extended the DFScala with TM support and showed that such model can be efficiently used in parallelization of Lee's algorithm for circuit routing. Still, these applications are standalone and not part of a larger suite. Therefore, their coverage of algorithms and application patterns is limited. On the other hand, DaSH covers a wide range of algorithms and application domains and features not only the applications that are suitable for dataflow, but also those for which other forms of parallelism may be more appropriate.

4.2 The DaSH Benchmarks

This section describes characteristics and provides extensive evaluation of each DaSH benchmark. All benchmarks are compiled using the gcc compiler version 4.7.2 that supports transactional memory (we have used default GCC-TM runtime configuration), except OmpSs implementations that are compiled using Mercurium compiler version 1.99 [BSC (2004-)]. We also had to adapt the gcc implementation of the STL containers that we use in DaSH (vector, list and map) to avoid a negative performance impact of serialized transaction execution. DaSH provides two synchronization macros, $DASH\_SYNC\_BEGIN(mutex)$ and $DASH\_SYNC\_END(mutex)$, which can be configured to use either transactional memory or mutex locks. In our experiments, we use transactional memory, if not explicitly stated otherwise.

The experiments were conducted on the workstation with two 6-Core 64-bit Intel Xeon CPU X5650 Westmere processors running at 2.67GHz. Each processor unit has 12MB L3 cache memory. In addition, the cores are two way SMT-capable, giving a total number of 24 hardware threads. The machine is running Scientific Linux 6.2 (Carbon). For each experiment we report the average result of ten executions.

4.2.1 Branch and Bound

This dwarf represents a general type of algorithms for finding optimal solutions of various global optimization problems. Branch and bound algorithms work by the divide and conquer principle: the search space is subdivided into smaller subregions (branching), and bounds are found on all the solutions contained in each subregion under consideration. The strength of the branch and bound approach comes when bounds on a large subregion show that it contains only inferior solutions, and so the entire subregion can be discarded without further examination.

The DaSH benchmark for this dwarf implements branch-and-bound algorithm to solve the Traveling Salesman Problem (TSP). Given the number of cities and distances between each pair of cities this algorithm tries to find the shortest round-trip route that visits each city exactly once and then returns to the starting city. In DaSH, the TSP problem is solved by solving equivalent assignment problem (AP) that provides lower bounds for the solution of the TSP problem. Specifically, an assignment problem is less restrictive than the TSP problem, since the route for an assignment problem can contain multiple cycles, whereas the TSP can only have one. As a result, the total distance for the TSP will always be equal to or greater than the total distance for the corresponding AP. The solution is based on two lists: one with solved assignment problems, and the other with newly generated problems derived from branching previously solved APs. Branching is done by imposing a set of mutually exclusive constraints on the AP. If there is one cycle in the APs solution, then no further branching is necessary. Otherwise, the algorithm finds the shortest cycle and for each edge from the cycle, it creates two new assignment problems: one with the constraint that the current edge cannot be part of the solution and the other with the constraint that the solution must include the current edge. For each such problem, a trial solution for the TSP is found in two steps. In the first step, the equivalent assignment problem is solved. If the AP solution has only one cycle, then it is also the solution of the TSP. Otherwise, if the total distance is greater than the current best solution of the TSP the problem is discarded. However, if this distance is lower than the current best solution of TSP, the solution of the AP is *patched* by breaking each cycle and connecting all ends into one big cycle. If the total distance of this patched AP solution is less than the best distance so far, the AP becomes a candidate for the optimal solution. In a nutshell, the algorithm iterates through the problem list, prunes suboptimal problems and solves the rest until the list is empty. A moderate synchronization is necessary to protect list operations and currently found optimal solution in parallel implementations.

Shared memory implementations first create a dedicated task for each problem from the current problem list – line 24 in Listing 4.4. When the worker threads finish executing all these tasks they reach the task barrier – line 55. Next, a single thread removes the first problem from

```

void Solver::Solve() {
  bool done = false;

  best->Solve();
  if (best->getCycleCount() == 1)
    upperBound = best->getLowerBound();
  else {
    upperBound = best->getLowerBound() + best->Patch(infinity);
    best->GenerateSubproblems(problemList, deleteList);
  }


  #pragma omp parallel num_threads(num_threads)
  {
    while (!done) {

      #pragma omp single
      {
        while (problemList.size() > 0) {
          AssignmentProblem *problem = problemList.front();
          problemList.pop_front();

          #pragma omp task firstprivate(problem)
          {
            DASH_SYNC_BEGIN(mutex)
            long currUpperBound = upperBound;
            DASH_SYNC_END(mutex)

            if (problem->getParentLowerBound() < currUpperBound) {
              long lowerBound = problem->Solve();
              if (lowerBound < currUpperBound) {
                if (problem->getCycleCount() == 1) {
                  DASH_SYNC_BEGIN(mutex)
                  best = problem;
                  upperBound = lowerBound;
                  DASH_SYNC_END(mutex)
                }
                else {
                  long penalty = problem->Patch(currUpperBound);
                  long newUpperBound = penalty == infinity ? infinity : lowerBound + penalty;

                  DASH_SYNC_BEGIN(mutex)
                  if (newUpperBound < upperBound) {
                    best = problem;
                    upperBound = newUpperBound;
                  }
                  AddSolvedProblem(problem);
                  DASH_SYNC_END(mutex)
                }
              }
            }
          } /* end task */
        } /* end while */
      } /* end single - implicit barrier */

      #pragma omp single
      {
        if (!solvedList.empty()) {
          AssignmentProblem *problem = NextSolvedProblem();
          if (problem != NULL) {
            if (problem->getLowerBound() < upperBound)
              problem->GenerateSubproblems(problemList, deleteList);
          }
        } else
          done = true;
      }
    } /* end while */
  } /* end parallel */
}
```

Listing 4.4: The OpenMP implementation of the Traveling Salesman problem.

```
void Solver::SolveTask() {
  #pragma adf task trigger_set(problemToken) relaxed instances(num_threads) {
    DASH_SYNC_BEGIN(mutex)
    long currUpperBound = upperBound;   /* privatization */
    DASH_SYNC_END(mutex)

    if (problemToken->getParentLowerBound() < currUpperBound) {
      long lowerBound = problemToken->Solve();
      if (lowerBound < currUpperBound) {
        if (problemToken->getCycleCount() == 1) {
          DASH_SYNC_BEGIN(mutex)
          best = problemToken;
          upperBound = lowerBound;
          DASH_SYNC_END(mutex)
        }
        else {
          long penalty = problemToken->Patch(currUpperBound);
          long newUpperBound = penalty == infinity ? infinity : lowerBound + penalty;
          DASH_SYNC_BEGIN(mutex)
          if (newUpperBound < upperBound) {
            best = problemToken;
            upperBound = newUpperBound;
          }
          AddSolvedProblem(problemToken);
          DASH_SYNC_END(mutex)
        }
      }
      solvedToken = problemToken;
}}}

void Solver::GeneratorTask() {
  #pragma adf task trigger_set(solvedToken) relaxed {
    AssignmentProblem  *problem = NULL;
    if (problemList.empty()) {
      DASH_SYNC_BEGIN(mutex)
      while (!solvedList.empty()) {
        problem = NextSolvedProblem();
        if (problem != NULL) {
          if (problem->getLowerBound() < upperBound) break;
        }
      }
      DASH_SYNC_END(mutex)
      if (problem != NULL)
        problem->GenerateSubproblems(problemList, deleteList);
    }

    /* problemList is accessed only by this task which has a concurrency of 1 */
    long problem_count = 0
    while (!problemList.empty() && problem_count < generator_limit) {
      problem = problemList.front();
      problemList.pop_front();
      if (problem->getParentLowerBound() < upperBound) {
        problemToken = problem;  /* generate tokens */
        problem_count++;
      }
}}}

void Solver::InitialTask() {
  #pragma adf task relaxed {
    AssignmentProblem *problem = NULL;
    while (!problemList.empty()) {
      problem = problemList.front();
      problemList.pop_front();
      if (problem->getParentLowerBound() < upperBound)
        problemToken = problem;  /* generate tokens */
    }
}}

void Solver::Solve() {
  best->Solve();
  if (best->getCycleCount() == 1)
    upperBound = best->getLowerBound();
  else {
    upperBound = best->getLowerBound() + best->Patch(infinity);
    best->GenerateSubproblems(problemList, deleteList);
  }
  SolveTask();
  GeneratorTask();
  InitialTask();
}
```

Listing 4.5: The ADF implementation of the Traveling Salesman problem.

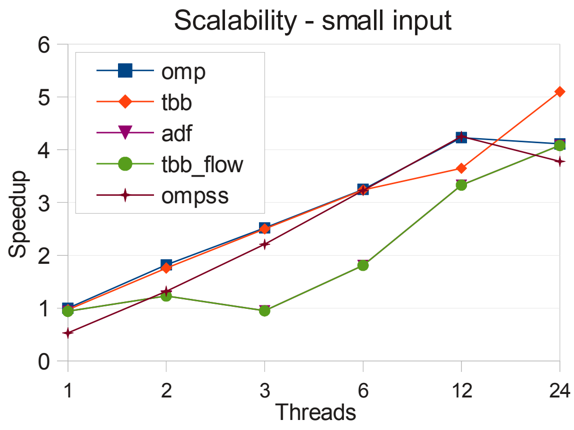


Figure 4.1: Traveling Salesman problem. Scalability with the small input.

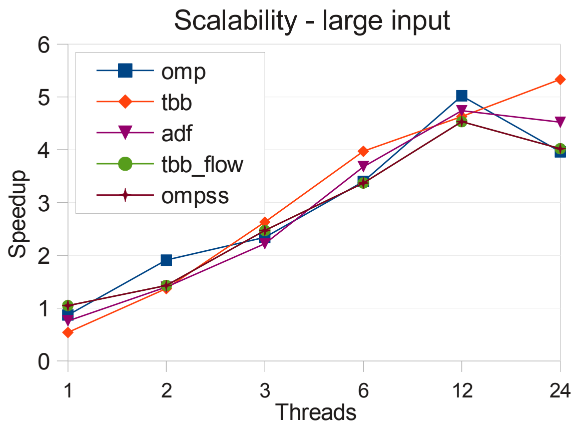


Figure 4.2: Traveling Salesman problem. Scalability with the large input.

the list of solved problems and uses it to generate new subproblems that may lead to an optimal solution, and then restarts the parallel section. This continues until both lists are empty.

The OmpSs implementation is similar to the OpenMP implementation because this is the most natural approach to the problem using this model. The ADF and TBB Flow implementations try to avoid barriers by relying on a *Generator* task that is enabled each time a problem is solved by some *Solve* task. Listing 4.5 shows the ADF implementation of this benchmark. The *Generator* task – line 31 – takes the problems from the start of the solved problem list, generates new problems and stores them into the problem list, and then creates a number of output tokens that carry pointers to the most promising problems from the problem list. These tokens then enable new executions of the *Solve* tasks. The limit on how many tokens will be generated by the *Generator* task can be controlled by a command line parameter. However, as this parameter gets larger the algorithm does more branching then bounding, which leads to a greedy solution. Hence, dataflow implementations reduce to a similar form of fork-join parallelism as in shared memory implementations.

Since the key is to minimize the problem space by minimizing branching, the *Generator* task from our dataflow implementations works the best when only one or two new problems are generated for each task invocation. Using larger values for the parameter that controls the number of newly generated problems increases the problem space and decreases the performance. Thus, dataflow implementations work in a similar way as task-based shared memory implementations, which can be seen in Figure 4.1 and Figure 4.2 that show the scalability of parallel implementations of this benchmark. The figures shows that the parallel implementations have comparable scalability, especially for the large data set, which means that dataflow implementations provide equally good support for the implementation of this dwarf as their shared memory counterparts.

4.2.2 Dense Linear Algebra

This dwarf corresponds to dense vector and matrix operations, traditionally divided into Level 1 (vector/vector), Level 2 (matrix/vector), and Level 3 (matrix/matrix) operations. These operations are often characterized with unit-stride memory accesses to read data from rows and strided accesses to read data from columns. For Level 3 operations, the best performance is typically achieved using block algorithms that result in better load balance and provide more opportunities to overlap computation and communication.

The DaSH benchmark that represents this dwarf is the implementation of the block Cholesky decomposition. The algorithm decomposes positive-definite matrix A into the product of a lower triangular matrix and its conjugate transpose $\mathrm{A} = \mathrm{LL}^{\mathrm{T}}$. Block operations are performed by calling corresponding CLAPACK functions:

spotrf	computes the Cholesky factorization of a real symmetric positive definite matrix A (in our case a single block).
ssyrk	performs one of the symmetric rank k operations, or $C := alpha * A' * A + beta * C$, where $alpha$ and $beta$ are scalars, and A and C are matrices.
strsm	solves one of the matrix equations $op(A) * X = alpha * B$ or $X * op(A) = alpha * B$, where $alpha$ is a scalar, X and B are matrices, A is a unit, or non-unit, upper or lower triangular matrix and $op(A)$ is one of $op(A) = A$ or $op(A) = A$'.
sgemm	performs one of the matrix-matrix operations.

The OpenMP solution in Listing 4.6 is implemented using a combination of work-sharing and tasks and, in each iteration, relies on two barriers that separate operations in order to produce a correct result. Figure 4.3 illustrates the existing parallelism in different steps of execution. At the start of each iteration, a single thread performs *spotrf* operation on a diagonal block — Figure 4.3a and 4.3d, while other threads are idle. Next, *strsm* operations in the corresponding column are executed in parallel — Figure 4.3b and 4.3e. Then, a single thread spawns a single *ssyrk* task and a number of *sgemm* tasks for the blocks bellow the diagonal — Figure 4.3c and 4.3f. These tasks can be performed in parallel; however a single thread has to perform *ssyrk* operation. The alternative is to use synchronization.

The dataflow approach provides a more elegant solution that follows the nature of the algorithm. Listing 4.7 shows the ADF implementation and Figure 4.4 illustrates the idea. First, we do not need to maintain barriers. Rather, we can orchestrate the execution based on data dependencies between block operations. Initially, we produce the *ssyrk[0]* token that enables the *spotrf[0,0]* task, and *sgemm[j,0]* tokens for all blocks from column 0 - Figure 4.3a. When the *spotrf[0,0]* task finishes its execution, it produces the *spotrf[0]* token that enables all *strsm* tasks from column 0 - Figure 4.3b. As these tasks execute, they produce *strsm[j]* tokens.

```
void Solver::Solve() {
  for (long j = 0; j < NumBlocks; j++) {

    /* Cholesky Factorization of A[j,j]  */
    spotrfBlock( A[j][j], BlockSize);

    #pragma omp parallel for firstprivate(j)
    for (long i = j+1; i < NumBlocks; i++) {
      /*  A[i,j] <- A[i,j] = X * (A[j,j])^t  */
      strsmBlock( A[j][j], A[i][j], BlockSize);
    }

    #pragma omp parallel
    {
      #pragma omp single
      {
        for (long i = 0; i < j; i++) {
          /*  A[j,j] = A[j,j] - A[j,i] * (A[j,i])^t  */
          #pragma omp task firstprivate(j)
          ssyrkBlock( A[j][i], A[j][j], BlockSize);
        }

        for (long k= 0; k< j; k++) {
          for (long i = j+1; i < NumBlocks; i++) {
            /*  A[i,j] = A[i,j] - A[i,k] * (A[j,k])^t  */
            #pragma omp task firstprivate(i, j, k)
            sgemmBlock( A[i][k], A[j][k], A[i][j], BlockSize);
          }
        }
      } /* end single */
    } /* end parallel */
  } /* end for */
}
```

Listing 4.6: OpenMP implementation of the Cholesky decomposition.

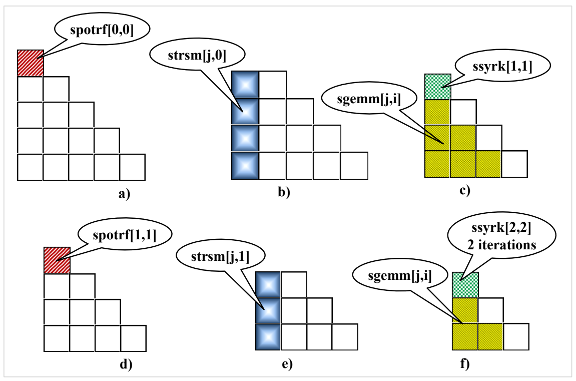


Figure 4.3: An example execution of the OpenMP Cholesky decomposition.

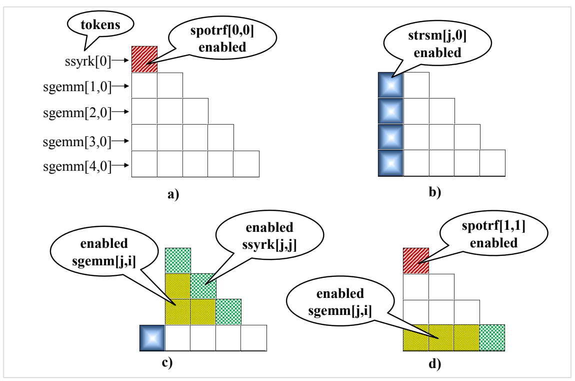


Figure 4.4: An example execution of the ADF Cholesky decomposition.

Token *strsm[j]* enables the *ssyrk[j,j]* task, while all *strsm[j]* tokens enable all *sgemm* tasks from the part of the matrix bellow diagonal, excluding column 0 - Figure 4.3c. Task *ssyrk[1,1]* then produces token *ssyrk[1]*, while *sgemm[1,i]* tasks produce corresponding tokens that initiate the same sequence of operations for the next iteration - Figure 4.3d.

Evidently, this approach allows more tasks to overlap and thus results in better concurrency. For example, in the OpenMP solution the *ssyrk[4,4]* block has to wait for all previous iterations before it starts the execution, processing all blocks from rows 3 and 4 and previous four columns. Instead, in the ADF implementation, as soon as *strsm[j,3]* and *strsm[j,4]* tasks are finished *ssyrk[4,4]* task can process these two blocks. Similar reasoning applies to *sgemm* tasks.

```

void Solver::Solve() {
  for (long j=0; j<NumBlocks; j++)
  {
    /* spotrf task */
    #pragma adf task trigger_set(&strsmToken[j]) relaxed execute(1)
    {
      /* Cholesky Factorization of A[j,j]  */
      spotrfBlock( A[j][j], BlockSize);
      spotrfToken = j;
    }

    /* ssyrk task */
    #pragma adf task trigger_set(&strsmToken[j]) relaxed execute(1)
    {
      /*  A[j,j] = A[j,j] - A[j,i] * (A[j,i])^t  */
      ssyrkBlock( A[j][strsmToken], A[j][j], BlockSize);
      if (strsmToken == j-1) ssyrkToken = j;
    }

    /* strsm task */
    for (long i = j+1; i < NumBlocks; i++)
      #pragma adf task trigger_set(&spotrfToken[j], &sgemmToken[i][j]) relaxed
      {
        /*  A[i,j] <- A[i,j] = X * (A[j,j])^t  */
        strsmBlock( A[j][j], A[i][j], BlockSize);
        strsmToken = j;
      }
  }

    /* sgemm task */
  for (long j = 1; j < NumBlocks; j++) {
    for (long i = j+1; i < NumBlocks; i++)
      #pragma adf task trigger_set(&strsmToken[j], &strsmToken[i]) relaxed execute(1)
      {
        long j, k = strsmToken;

        /*  A[i,j] = A[i,j] - A[i,k] * (A[j,k])^t  */
        sgemmBlock( A[i][k], A[j][k], A[i][j], BlockSize);

        if (strsmToken == j -1)
          sgemmToken = j;
      }
  }

  /* initial sgemm tokens */
  #pragma adf task relaxed execute(1)
  for (long i = 1; i < NumBlocks; i++)
    sgemmToken = 0;

  /* initial ssyrkToken token */
  #pragma adf task relaxed execute(1)
  ssyrkToken = 0;
}
```

Listing 4.7: ADF implementation of the Cholesky decomposition.

Figure 4.5 shows the performance sensitivity analysis of parallel implementations of Cholesky decomposition. The size of the input is determined by two parameters: the number of blocks b in a bxb block matrix, and the size of the block. For example, 12 blocks with size 64 represent the smallest matrix used in this analysis, which size is 768x768 elements. The sequential execution with this input takes 64.2 milliseconds, whereas the sequential execution of the largest matrix with 60 blocks of size 256 takes 592.6 seconds. For the smallest number of blocks, 12, the OpenMP implementation performs the best because there are only 12 iterations of the outer loop, which means that barriers have less influence on the performance than when the number of blocks is larger. The overall performance of all parallel implementations is lower than for the

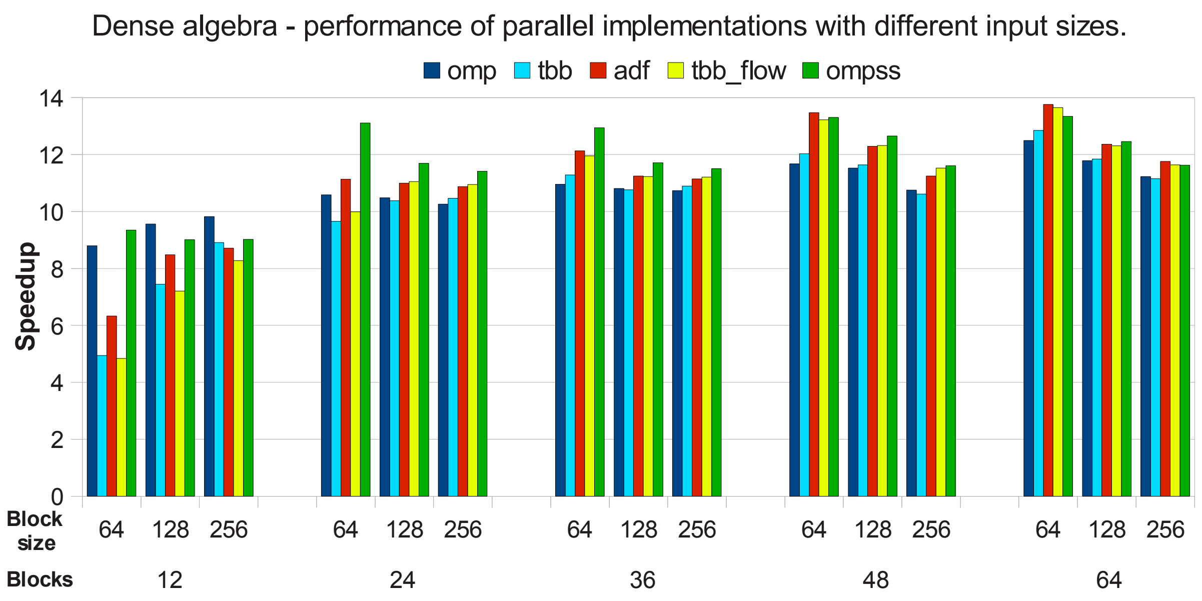


Figure 4.5: Performance sensitivity analysis of parallel Cholesky decomposition implementations.

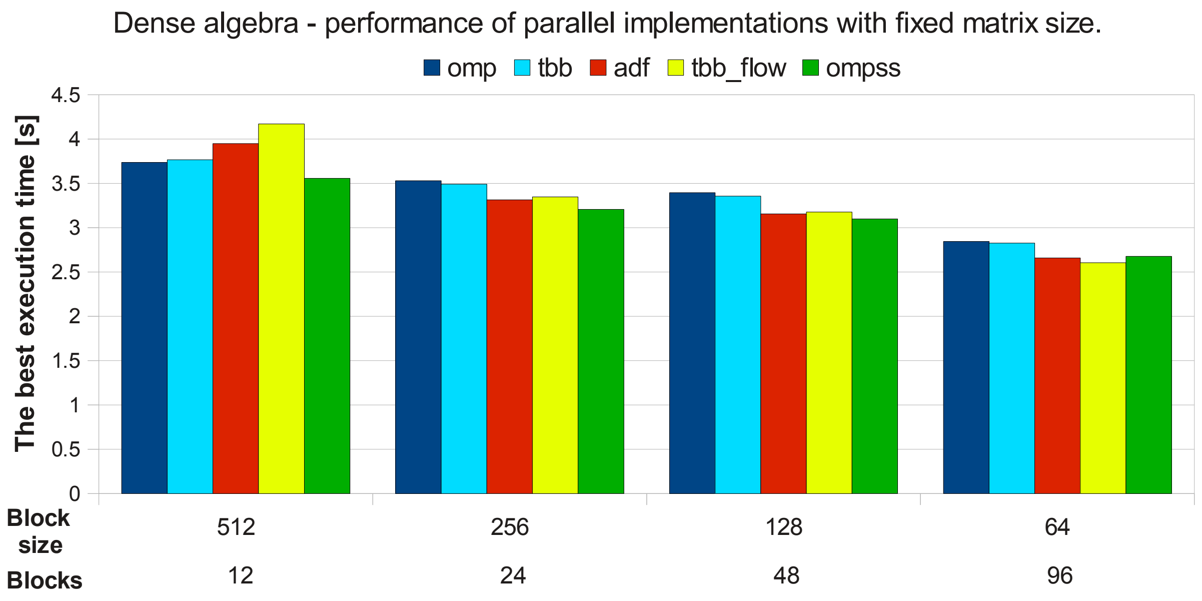


Figure 4.6: Performance sensitivity analysis of parallel Cholesky decomposition implementations with fixed matrix size of 6144x6144 elements.

larger input sizes because partitioning the matrix into 12x12 blocks does not provide enough parallelism to keep all 24 threads busy. As soon as the number of blocks is doubled, parallel performance increases and dataflow implementations start to outperform shared memory implementations moderately, but consistently.

Figure 4.6 shows the sensitivity analysis when the size of the matrix is fixed to 6144x6144. The number of blocks is varied from 12 to 96, adjusting the block size accordingly. Since the Y axes shows the best execution time obtained with each implementation, the lower value is

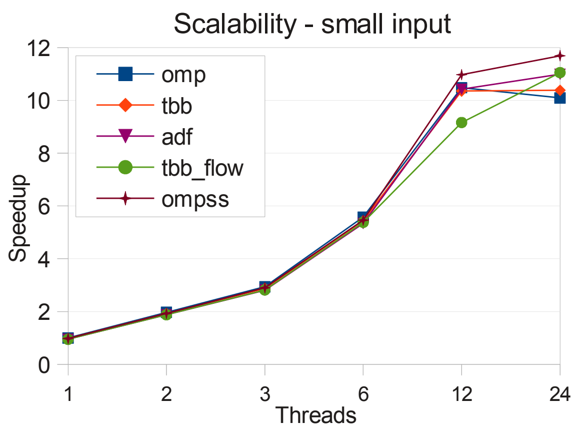


Figure 4.7: Cholesky decomposition. Scalability with the small input.

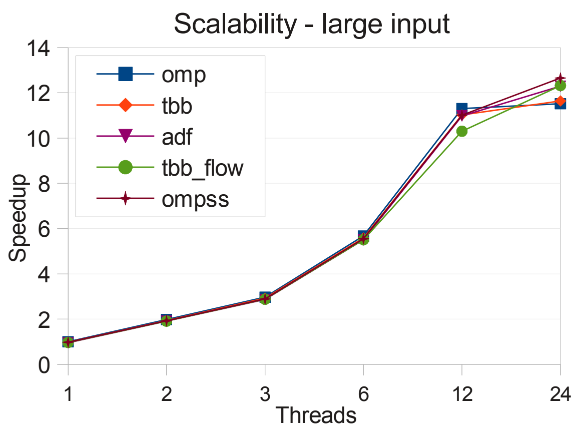


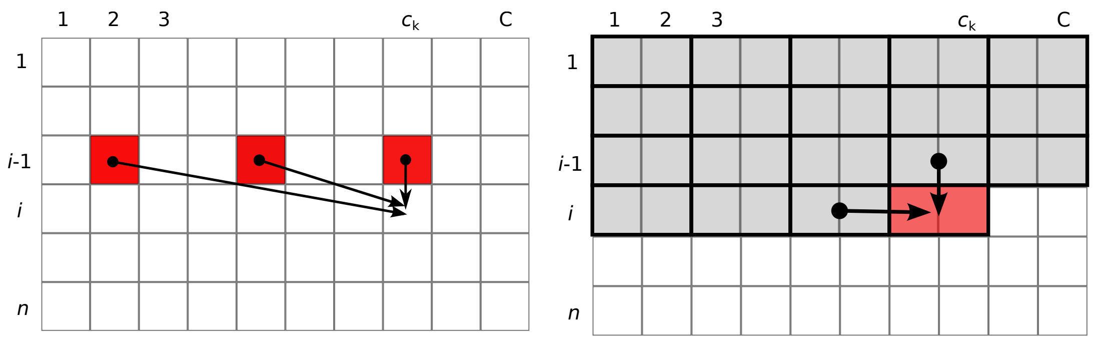
Figure 4.8: Cholesky decomposition. Scalability with the large input.

better. Thus, the best performance provide the experiments in which the blocks size is set to 64, regardless of the increase in the number of tasks due to a larger number of blocks. Also, for the largest block size of 512, the shared memory implementations perform the best as a consequence of dividing the matrix to only 12x12 blocks, which limits the parallelism. However, as the number of blocks increases, the dataflow implementations consistently outperform the shared memory implementations.

Finally, figures 4.7 and 4.8 show the scalability of parallel implementations for the small and large input, respectively. The small input represents 3072x3072 matrix, organized in 24x24 blocks with size of 128x128 elements, while the large input represents 7680x7680 matrix, organized in 60x60 blocks with size of 128x128 elements. Both diagrams show that all implementations scale almost identically until the number of threads reaches 12, when the *tbb_flow* implementation has somewhat lower performance. Moreover, in the experiments with 24 threads that simultaneously share 12 cores, the shared memory implementations do not show any improvement, whereas the dataflow implementations successfully utilize simultaneous multithreading architecture.

4.2.3 Dynamic Programming

Dynamic programming is used in a variety of problems, such as technology mapping in integrated circuit design, longest common subsequence matching of DNA strands and various optimization problems. The essence of this dwarf is that the solution of a problem is based on solving simpler overlapping subproblems. Data dependencies are created between levels of subproblems since an optimal solution to a larger problem depends on an optimal solution to its subproblems. Therefore, these algorithms can be solved naturally using dataflow.



(a) **Per-element dependencies.** (b) **Dataflow task dependencies.**

Figure 4.9: Data dependencies in unbounded Knapsack problem.

Often, subproblems can be grouped into blocks to increase computational granularity. The same method is applied in a solution of the unbounded Knapsack problem that represents this dwarf in DaSH. Given a set of items, each with a weight w_i and a value v_i, and a bag with capacity C, the objective is to choose a subset of the items whose total weight is no greater than the bag capacity while maximizing the profit. The unbounded Knapsack problem places no upper bound on the number of copies of each kind of item and can be formulated as:

$$\text{Maximize } \sum_{i=1}^{n} v_i x_i \text{ subject to } \sum_{i=1}^{n} w_i x_i \leq C$$

Dynamic programming can be used to heuristically solve this problem in pseudo-polynomial time [Gilmore & Gomory (1961)]. The solution is based on a two dimensional matrix M with n rows that represent the items and C columns that represent the knapsack capacities. The algorithm goes through matrix M row by row. For every item x_i, the maximum number of items of type x_i that can be loaded given a knapsack with capacity C is computed as $\lfloor C/w_i \rfloor$. A choice for the number of units of product x_i to load in the knapsack is notated as u. Given a choice of u units for product x_i, a possible value for entry (x_i, C) is calculated adding the previously found maximum value in row x_{i-1} and column $C - (w_i * u)$ to the value from loading u units of item x_i, that is $u * v_i$. After repeating this computation for all possible values of u, $(u = 0..\lfloor C/wi \rfloor)$, the maximum of these values is stored in entry (x_i, C).

Figure 4.9a illustrates the dependency between entries of matrix M, which shows that the most straightforward approach is to calculate the values of entries in a single raw before processing the next raw of the matrix. Thus, shared memory implementations are based on work-sharing. A single row is processed in parallel, after which there is an implicit barrier that separates the processing of successive rows. Listing 4.8 shows the OpenMP implementation of the Knapsack problem.

```

void Solver::Solve()
{
   for (long i = 1; i < numItems; i++) {
      #pragma omp parallel for firstprivate(i) schedule(static,C/(numParts*nthreads))
      for (long j = 0; j < capacity; j++) {
         long maxval = 0.0;
         long u = j/w[i];
         for (long l = 0; l <= u; l++) {
            long newval = l*v[i] + prev_row[j - l*w[i]];
            maxval = (newval > maxval) ? newval : maxval;
         }
         curr_row[j] = maxval;
      }

      long *tmp = prev_row;
      prev_row = curr_row;
      curr_row = tmp;
   }

   maxValue = prev_row[capacity-1];
}
```

Listing 4.8: OpenMP implementation of the unbounded Knapsack problem.

```

int *partToken = new int[numParts];

void Solver::CalculateBlock(int partNum) {
   long row = rowId[partNum];
   long myvalue = v[row], myweight = w[row];
   for (long j = partNum*partitionSize; j < (partNum+1)*partitionSize; j++) {
      long maxval = 0.0;
      long u = j / myweight;
      for (long l = 0; l <= u; l++) {
         long newval = l*myvalue + matrix[(row-1) * capacity +  j - l*myweight ];
         maxval = (newval > maxval) ? newval : maxval;
      }
      matrix[row * capacity +  j] = maxval;
   }
   rowId[partNum]++;
}

void Solver::Solve()
{
   #pragma adf task trigger_set(partToken[0]) relaxed pin(0)
   {
      CalculateBlock(0);
      if (rowId[0] < numItems)
         partToken[0] = 0;
   }

   for (int i = 1; i < numPartitions; i++) {
      #pragma adf task trigger_set(partToken[i-1], partToken[i]) relaxed
                       pin(i%nthreads) until(rowId[i] >= numItems)
      {
         CalculateBlock(i);
         partToken[i] = i;
      }
   }

   for (int i = 0; i < numPartitions; i++) {
      #pragma adf task
      partToken = i;
   }
}
```

Listing 4.9: ADF implementation of the unbounded Knapsack problem.

Dataflow implementations avoid barriers by partitioning each row to *nthreads* partitions and creating the same number of tasks to process these partitions, where *nthreads* is the number of threads in execution. Obviously, coarsening the workload from a single matrix element to

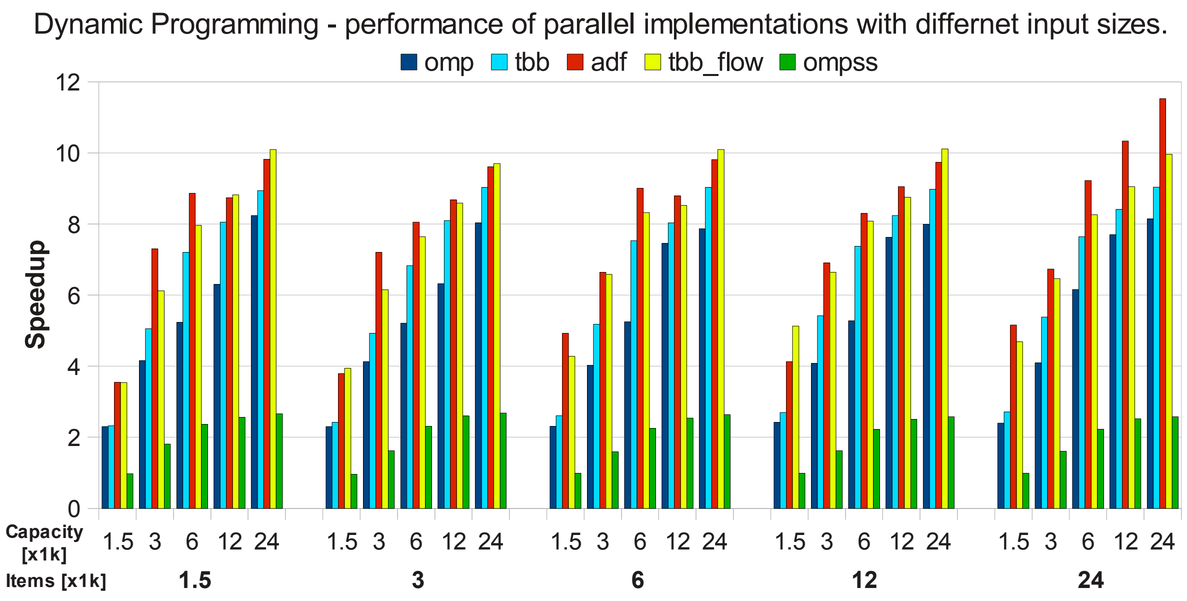


Figure 4.10: Dynamic programming: performance sensitivity analysis of parallel implementations.

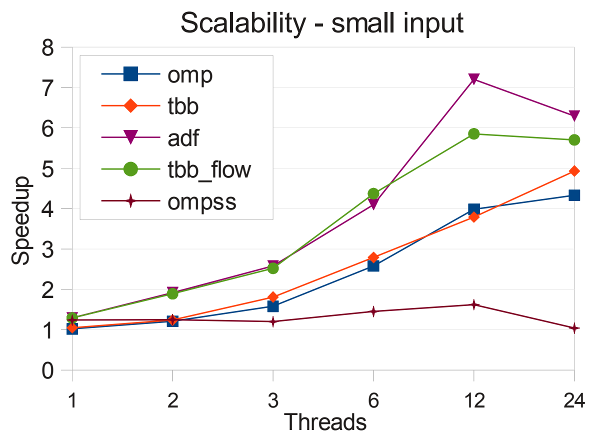


Figure 4.11: Dynamic Programming scalability: small input - 3k items, 3k capacity.

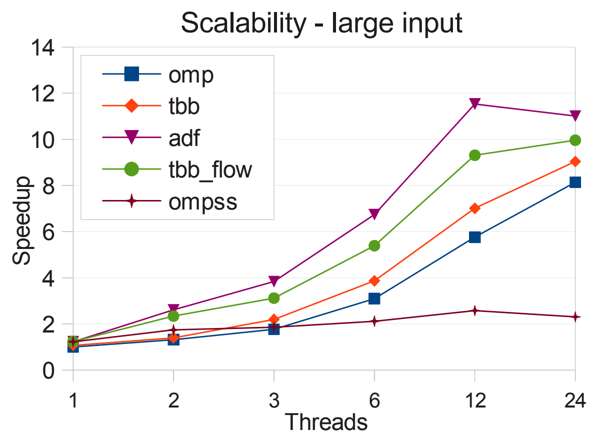


Figure 4.12: Dynamic Programming scalability: Large input - 24k items, 24k capacity.

a block of elements makes tracking dependencies on per-element basis impractical. Thus, we apply a simplified dependency approach illustrated in Figure 4.9b, which effectively resembles a wavefront processing of row partitions. Dependency structure is such that each partition, say $p[i, j]$, can be processed only after partitions $p[i - 1, j]$ and $p[i, j - 1]$ have been processed.

Listing 4.8 and Listing 4.9 show the relevant parts of the OpenMP and the ADF implementations respectively. Evidently, the OpenMP implementation is simpler; however, it relies on an implicit barrier. On the contrary, the ADF implementation eliminates the need for global synchronization by applying a point-to-point communication between tasks. Although the dataflow approach slightly increases the complexity of the code, it provides better programmability by avoiding the global synchronization.

Figure 4.10 shows the performance sensitivity analysis of parallel implementations of the unbounded Knapsack problem. Two parameters of the input examples are varied in this analysis: the number of items and the knapsack capacity. The figure shows that varying the number of items does not affect the performance, whereas increasing the capacity of the knapsack emphasizes the advantages of parallelization. Dataflow solutions consistently outperform shared memory implementations for all combinations of the input parameters, except in the case of the OmpSs implementation that has very poor performance. However, such performance does not reflect the inability of the OmpSs model to support this application efficiently; rather the problem is that the current version of the Mercurium compiler produces suboptimal code even when compiling the sequential version that has no OmpSs directives.

Figure 4.11 and 4.12 show the scalability of parallel implementations for two input sizes using the sequential implementation as a baseline. As we can see, the dataflow implementations have better scalability than the shared memory implementations for all thread counts. Also, the performance of the ADF implementation improves with the size of the input due to the use of the *pin* clause that enables us to bind partitions to threads. As a result, the cache behavior of this implementation is improved.

4.2.4 Finite State Machine

This dwarf represents a system whose behavior is defined by states and transitions between states based on given inputs and events. The state machine can be represented as a tuple:

$$(\Sigma, S, s_0, \delta, F), \text{ where } \quad \begin{array}{ll} \Sigma & \text{is a set of possible inputs.} \\ S & \text{is a finite, non-empty set of states.} \\ s_0 & \text{is an initial state.} \\ \delta & \text{is the state-transition function } \Sigma \times S \to S \text{ that determines} \\ & \text{the next state based on the current inputs and the state.} \\ F & \text{is the set of final states or outputs.} \end{array}$$

Parallelism is often difficult to utilize. However, some state machines can be decomposed into multiple simultaneously active state machines. One such example is a deterministic finite state machine for a text pattern search, which is the problem solved by the DaSH benchmark for this dwarf. The states are integer values from 0 to N, where N is the length of the pattern. The algorithm builds a state transition matrix in which each character has one of the N possible states, based on the search pattern. This transition matrix is used during the content string traversal to change the current state of pattern matching machine. When the machine reaches the final state the index of a new pattern occurrence is recorded and the machine is reset. The result of the algorithm is the list of indexes in the text where the pattern has been found.

```
 1 void Solver::SearchText(long beginPos, long endPos) {
 2     endPos += patternSize;
 3     if (endPos > textSize) endPos = textSize;
 4     int state = 0;
 5     for (long i = beginPos; i < endPos; i++) {
 6         if (text[i] > 0)
 7             state = states[(int) text[i]][state];  /* change the state */
 8 
 9         /* if the pattern is found, push the position to the list */
10         if (state == patternSize) {
11             DASH_SYNC_BEGIN(mutex)
12             positions.push_back(i - patternSize);
13             DASH_SYNC_END(mutex)
14             state = 0;
15         }
16     }
17 }
18 
19 void Solver::Solve() {
20     /* construct the matrix of states  */
21     ConstructStateMatrix();
22 
23     /* search the text using states matrix in parallel  */
24     long beginPos = 0, endPos = partSize;
25 
26     #pragma omp parallel
27     {
28         #pragma omp single
29         {
30             while (beginPos < textSize) {
31 
32                 #pragma omp task firstprivate(beginPos, endPos)
33                 SearchText(beginPos, endPos);
34 
35                 beginPos += partSize;
36                 endPos += partSize;
37             } // end while
38         } // end single
39     }// end parallel
40 }
```

Listing 4.10: OpenMP implementation of the finite state machine pattern search.

```
void Solver::Solve() {
    /* construct the matrix of states  */
    ConstructStateMatrix();

    /* search the text using states matrix in parallel  */
    long beginPos = 0, endPos = partSize;

    /* create ADF tasks */
    while (beginPos < textSize) {

        #pragma adf task relaxed execute(1)
        SearchText(beginPos, endPos);

        beginPos += partSize;
        endPos += partSize;
    }
}
```

Listing 4.11: ADF implementation of the finite state machine pattern search.

Shared memory solutions are implemented using tasks, since we need to adjust partition boundaries after they have been already created to assure that words are not split by these boundaries – line 2 in Figure 4.10. Each task searches a part of the content string by calling the *SearchText* function. Synchronization is necessary when a thread has to update the global list of occurrence indexes – line 11 in Figure 4.10.

Dataflow solutions are reduced to fork-join parallelism since each partition can be processed

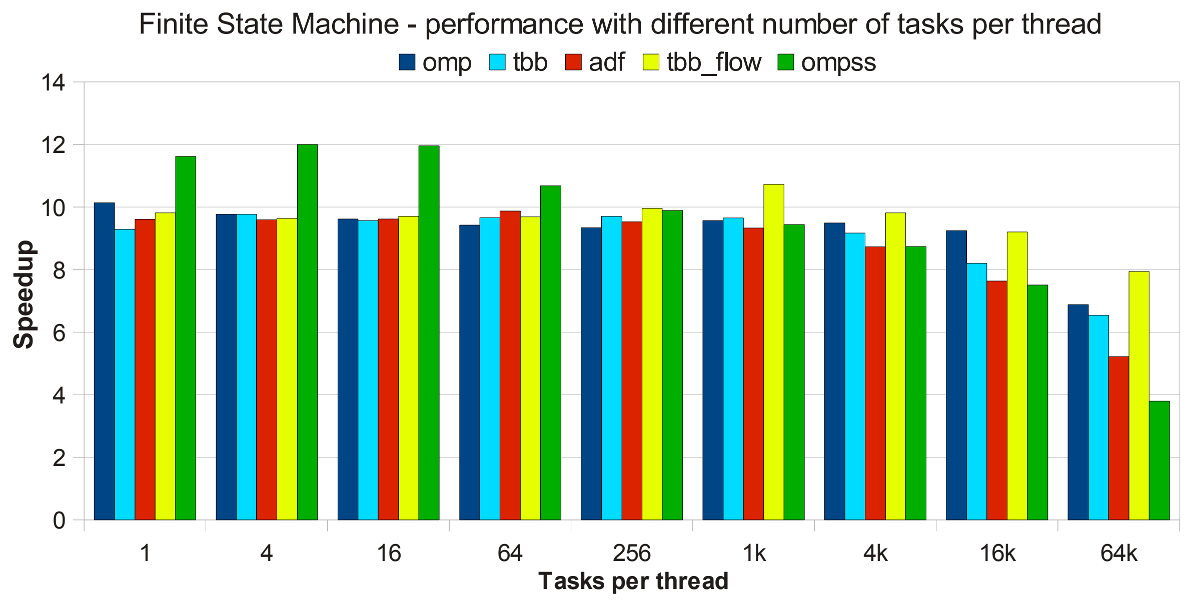


Figure 4.13: Speedup in corallation with different number of tasks per thread.

independently from other partitions. Listing 4.11 shows the ADF implementation of this application that is based on execution of ADF tasks with no input dependencies – line 10. Evidently, there is no difference in programmability of shared memory implementations and dataflow implementations.

Next, Figure 4.13 shows how the speedup of parallel implementations changes in correlation with the number of tasks per thread (i.e the number of partitions of the input text). As expected, the speedup decreases when the number of tasks reaches a limit, after which the overhead of task handling starts to affect the performance. In the case of the large 3GB input file that we use in this analysis, the performance starts to drop when more than 1k partitions are assigned to each thread. All parallel implementations perform almost identically, except the OmpSs implementation that achieves the best overall speedup when the input is partitioned to a small number of partitions. Profiling shows that the OmpSs implementation results with almost two times less L1 and L3 cache misses compared to the other parallel implementations when a single partition per thread is used, while the other hardware events have similar values across all implementations.

Figure 4.14 presents a sensitivity analysis of parallel implementations when varying the size of the input. We can see that the performance of all implementations is stable across a wide range of input sizes. In the case of the smallest input of 100MB, all implementations except the OmpSs version have somewhat lower performance than for the larger inputs, because the cache behavior of the sequential version in this case does not affect the performance as much as in other cases.

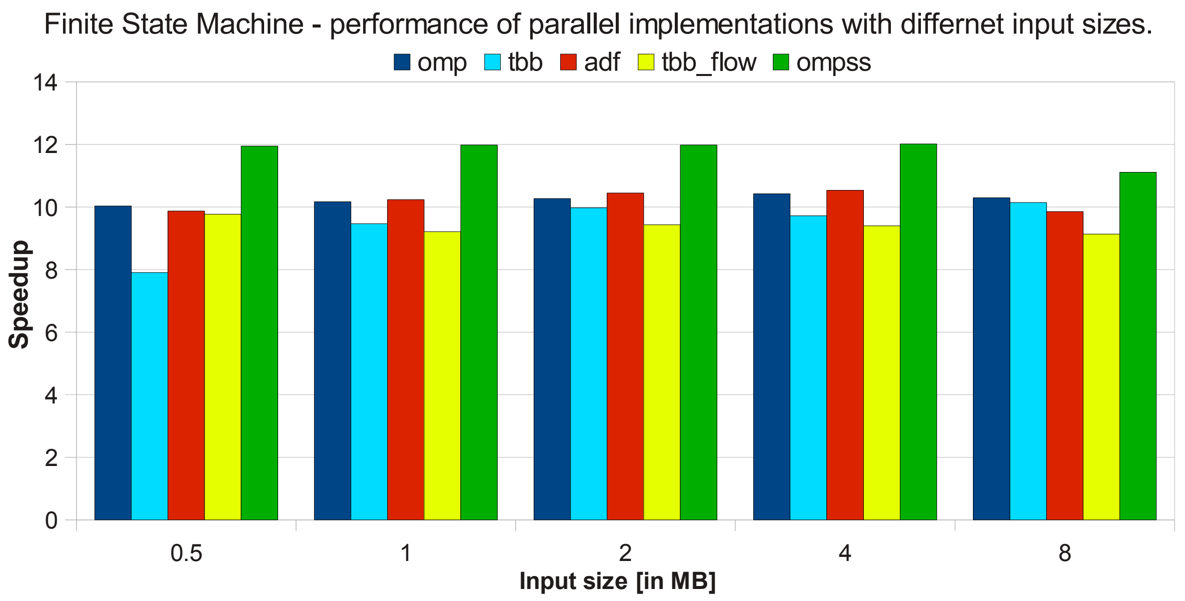


Figure 4.14: Performance of parallel implementations with different input sizes.

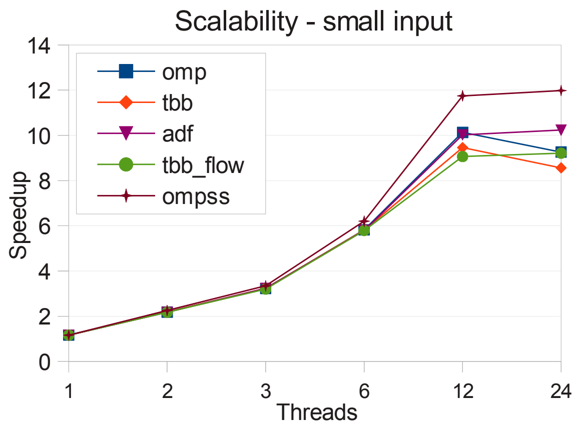


Figure 4.15: Finite state machine. Scalability with the small input (1GB).

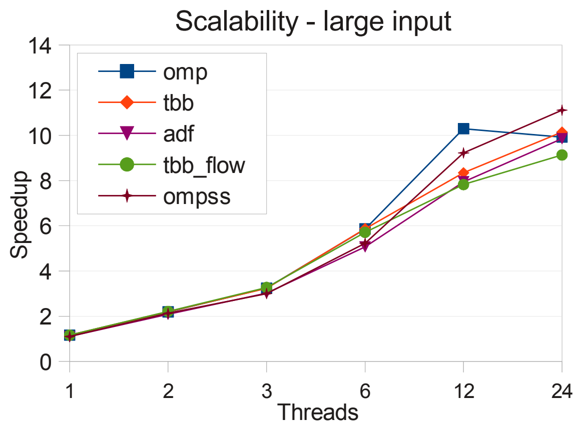


Figure 4.16: Finite state machine. Scalability with the large input (8GB).

Overall, since all parallel implementations of this benchmark implement a simple task-based fork-join parallelism, we can conclude that all models provide equally good support for the implementation of the text pattern search problem using the finite state machine approach. Although the OmpSs implementation performs better than other implementations, this does not mean that dataflow model is superior for this application. Rather, it is the result of better OmpSs runtime support that results in better cache behavior than in the case of other parallel models. Moreover, all implementations perform consistently across a wide range of input sizes, and a simple partitioning that assigns a single partition of the input text to each thread provides the best results.

Finally, figures 4.15 and 4.16 show the scalability of parallel implementations for the small and large input, respectively. The small input represents a 1GB file for which sequential implementations takes 2.5 seconds to execute, while the large input represents a 8GB file for which the sequential execution time is 20 seconds.

4.2.5 Graphical Models

This dwarf describes a group of algorithms based on a graph in which nodes represent variables and edges represent conditional probabilities (dependencies). Examples include Bayesian networks, Hidden Markov Models and neural networks. These applications typically involve many levels of indirection, and a relatively small amount of computation. The DaSH benchmark for this dwarf implements the Viterbi algorithm for the Hidden Markov model. The goal of the algorithm is to answer what is the most likely sequence of states that can explain given observations. The algorithm iterates through a sequence of observations and for each state it calculates the most probable preceding state.

Suppose we are given a Hidden Markov Model (HMM) with state space S, initial probabilities π_i of being in state i and transition probabilities $a_{i,j}$ of transitioning from state i to state j. Say we observe outputs $y_1, .., y_T$. The most likely state sequence $x_1, .., x_T$ that produces the observations is given by the recurrence relations:

$$V_{1,k} = P(y_1|k) * \pi_i$$
$$V_{t,k} = P(y_t|k) * max_{x \in S}(a_{x,k} * V_{t-1,x})$$

where $V_{t,k}$ is the probability of the most probable state sequence responsible for the first t observations that has k as its final state. The Viterbi path can be retrieved by saving back pointers that record the state x that was used in the second equation. Let $Ptr(k, t)$ be the function that returns the value of x used to compute $V_{t,k}$ if $t < 1$, or k if $t = 1$. Then:

$$x_T = argmax_{x \in S}(V_{t,x})$$
$$x_{t-1} = Ptr(x_t, t)$$

Thus, we need to calculate all state probabilities for one observation before we can move to the other, which necessitates a barrier.

Listing 4.12 shows the OpenMP implementation of this benchmark and Figure 4.17 illustrates how the corresponding execution proceeds. The solution is based on a matrix of state probabilities where the number of rows is equal to the length of the observation sequence. Shared memory implementations are based on work-sharing, where each row of matrix is partitioned to a number of partitions equal to the the number of worker threads. Before the execution proceeds to the next row, threads synchronize on an implicit barrier, as required by the algorithm.

Still, using dataflow it is possible to avoid this barrier. Listing 4.13 shows the ADF implementation and Figure 4.18 illustrates the idea. Specifically, we partition each row to n partitions, where n is the number of worker threads, and build dependency graph so that each

```

void Solver::Solve() {
  /* Initialize  delta (the highest probability along a single path at time t)
   * and psi (the index of the state which maximized the probability) */
  ...

  /* Computation */
  for (int t = 1; t < lengthOfObsSeq; t++) {
    #pragma omp parallel for schedule(static, chunk)
    for (int j = 0; j < numberOfStates; j++) {
      int    maxvalind = 0;
      double maxval = 0.0, val = 0.0;

      /* find the maximum */
      for (int i = 0; i < numberOfStates; i++) {
        val = vit->delta[t-1][i]*(hmm->stateTransitionMatrix[i][j]);
        if (val > maxval) {
          maxval = val;
          maxvalind = i;
        }
      }
      vit->delta[t][j] = maxval*(hmm->obsProbability[j][vit->observationSequence[t]]);
      vit->psi[t][j] = maxvalind;
    }
  }

  /* Backtrack to find the most possible sequence of states */
  ...
}
```

Listing 4.12: The OpenMP implementation of the unbounded Knapsack problem.

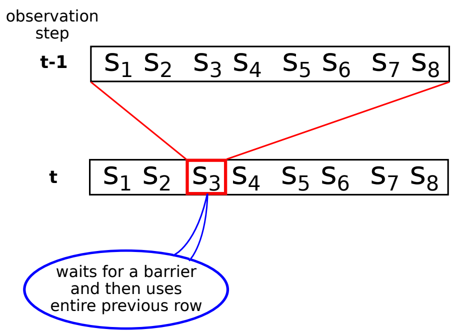

observation step
t-1
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
t
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
waits for a barrier and then uses entire previous row

Figure 4.17: An example execution of the work-sharing Viterbi algorithm.

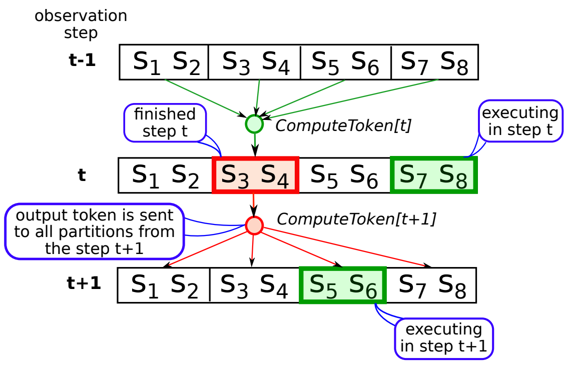

observation step
t-1
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
finished step t
ComputeToken[t]
executing in step t
t
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
output token is sent to all partitions from the step t+1
ComputeToken[t+1]
t+1
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8
executing in step t+1

Figure 4.18: An example execution of the dataflow Viterbi algorithm.

partition from the observation step t is processed by a dedicated task $ComputeTask[t][j]$, where j corresponds to the position of the partition in the row. Each $ComputeTask[t][j]$ has an input dependency on $ComputeToken[t-1]$ that is produced by all $Compute$ tasks from the previous observation step. Importantly, $ComputeTask[t][j]$ should produce output data if and only if it has executed n times, once for each partition of states from the previous observation step. Lines 38 to 53 in Listing 4.13 present the code that governs the correct generation of the output tokens. Specifically, each task maintains a private counter that is increased after each task invocation and reset after n iterations, when the corresponding $ComputeToken$ is generated (the same functionality could be handled by an extension to the ADF task; however, the current implementation does not support such extension). When $ComputeTask[t][j]$ produces the output data, it enables the execution of all $ComputeTask[t+1][k]$, for $k = 1..n$.

```

void Solver::ComputeTask() {
  static int backtrack_count = 0;

  for (int t = 1; t < lengthOfObsSeq; t++) {
    for (int j = 0; j < numStates; j+=state_chunk) {
      #pragma adf task trigger_set(computeToken[t]) relaxed
      {
        /* get the producer partition boundaries */
        int start = computeToken[t];
        int end = computeToken[t] + statePartition;
        end = (end > numStates) ? numStates : end;

        /* finding the maximum */
        int partEnd = ((j+state_chunk) < numStates) ? (j+state_chunk) : numStates;
        for (int k = j; k < partEnd; k++) {

          /* references to the max value and index for delta and psi arrays
           * we are reusing these matrices to store tentative values
           * until all states from the preceding sequence step have been processed */
          int    &maxvalind = vit->psi[t][k];
          double &maxval  = vit->delta[t][k];
          double val = 0.0;

          for (int i = start; i < end; i++) {
            val = vit->delta[t-1][i]*(hmm->stateTransitionMatrix[i][k]);

            if (val > maxval) {
              maxval = val;
              maxvalind = i;
            }
          }
        }

        /* If this task has executed for all partitions of the previous row generate
         * an output token, unless this is the last remaining task in the last row
         * in which case generate backtrackToken to enable Backtrack task */
        exec_count[t][j/state_chunk]++;
        if (exec_count[t][j/state_chunk] >= num_compute_tasks) {
          for (int k = j; k < partEnd; k++)
            vit->delta[t][k] *= hmm->obsProbability[k][vit->observationSequence[t]];

          if (t == lengthOfObsSeq - 1) {
            DASH_SYNC_BEGIN(mutex)
            backtrack_count++;
            if (backtrack_count >= num_compute_tasks) {
              backtrackToken = 1;
            }
            DASH_SYNC_END(mutex)
          }
          else {
            computeToken[t+1] = j;  /* send the start of partition as token value */
          }
        }
      }
    }
  }
}

void Solver::BackTrackTask() {
  #pragma adf task trigger_set(backtrackToken) relaxed execute(1)
  ...
}

void Solver::Solve() {
  /* Initialize  delta (the highest probability along a single path at time t)
   * and psi (the index of the state which maximized the probability) */
  statePartition = numStates/ num_threads;
  ...

  ComputeTask();
  BackTrackTask();

  #pragma adf task relaxed execute(1)
  for (int j = 0; j < numStates; j+=statePartition)
    computeToken[1] = j;
}
```

Listing 4.13: The ADF implementation of the unbounded Knapsack problem

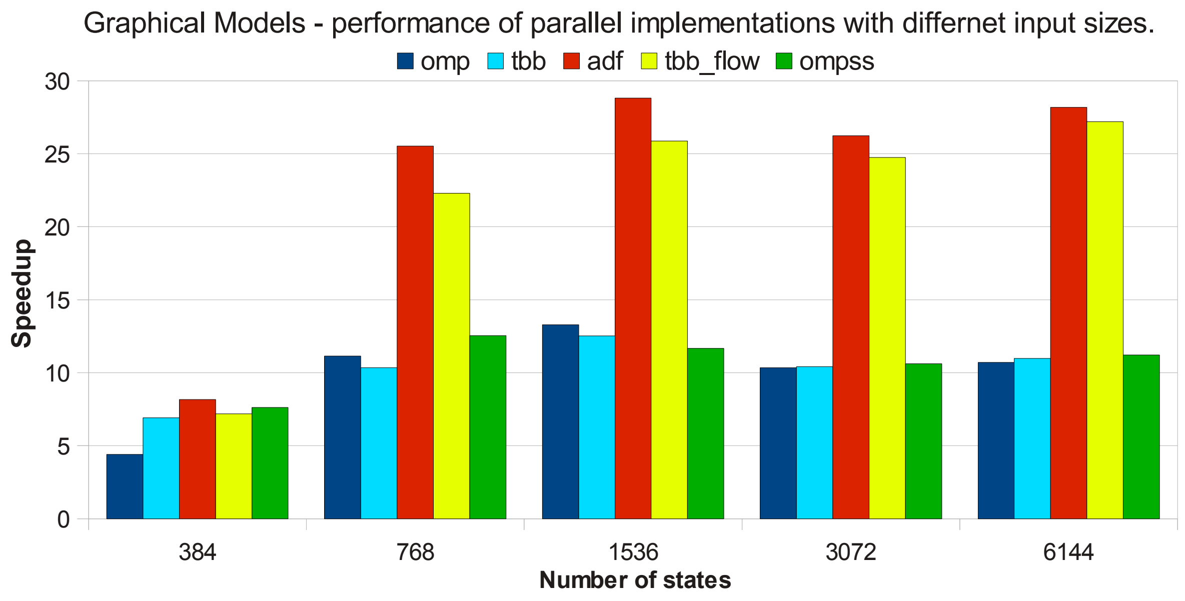


Figure 4.19: Performance sensitivity analysis of parallel implementations of Viterbi algorithm.

Therefore, this benchmark shows an example when dataflow can be useful even if barriers are essential to the algorithm. However, this approach is not feasible using OmpSs because the model builds the *task dependency graph* instead of the *data dependency graph*. For example, assume that task T_1 has an output dependency on data X and that task T_2 has an input dependency on X. Even if T_1 finishes the execution without touching data X, the runtime system assumes that the task has produced a new value for X and thus enables the consumer task T_2. For that reason, the OmpSs implementation is basically the same work-sharing implementation as in the case of the shared memory version, only implemented using tasks.

Figure 4.19 shows the performance sensitivity analysis of parallel implementations of Viterbi algorithm. The size of the input is determined by two parameters: the number of states and the length of the observation sequence. In this analysis, the length of the observation sequence is fixed to 100 and only the number of states is varied. The figure shows no significant difference in the performance of parallel implementations for the smallest input, except in the case of OpenMP implementation. For other state counts, the performance increases and dataflow implementations consistently outperform the shared memory version almost by a factor of two. Partly, this is the due to increased parallelism of these implementations as a result of overlapping the execution of tasks from different steps (rows) and partly due to the better cache behavior. In particular, profiling shows that dataflow implementations have five times less L3 cache references, although all implementations result in almost identical number of L3 cache misses. Such a difference in the number of L3 cache references can be explained by the fact that in dataflow implementations, each task operates only on two row partitions, one from the

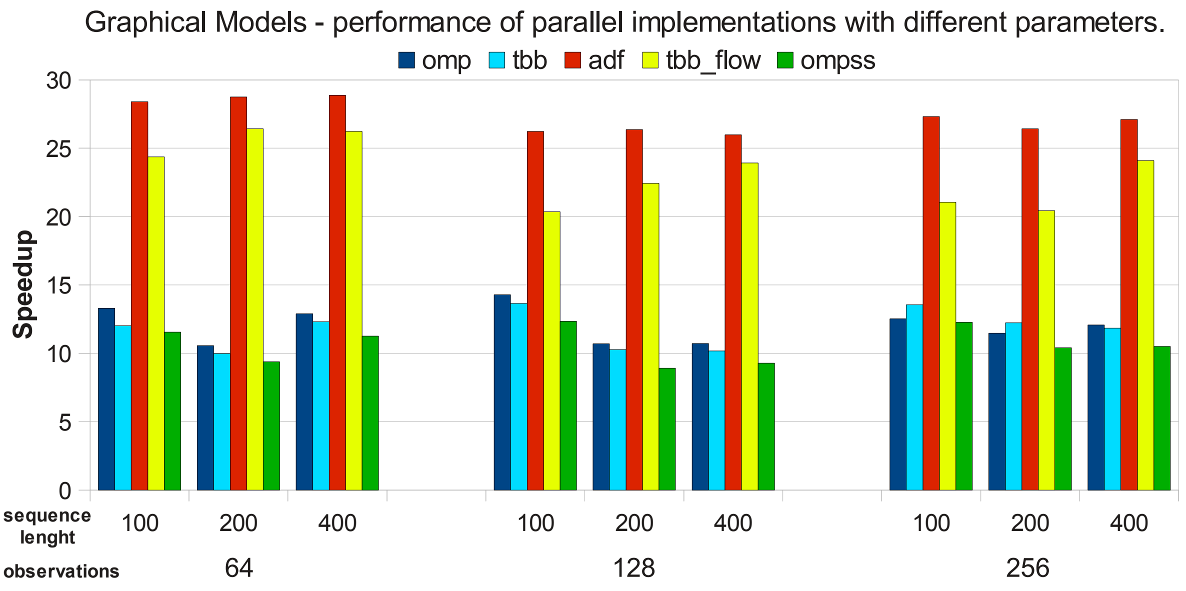


Figure 4.20: Performance sensitivity analysis of parallel implementations of Viterbi algorithm with fixed matrix size of 6144x6144 elements.

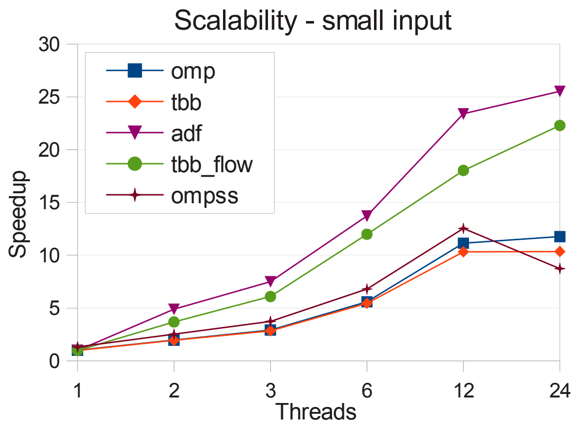


Figure 4.21: Viterbi algorithm. Scalability with the small input.

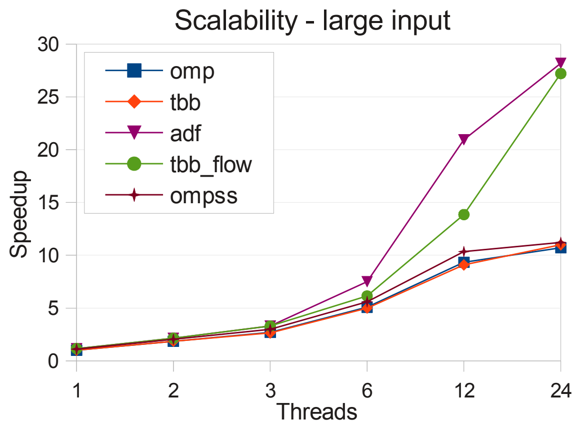


Figure 4.22: Viterbi algorithm. Scalability with the large input.

current step and one from the previous step, while in the work-sharing implementations each task operates on the row partition from the current step and the entire row from the previous step of execution.

Figure 4.20 shows the maximum speedup of parallel implementations when the number of states is fixed to 768 (small input) while varying the length of the observation sequence and the number of observations. As we can see, both parameters have no effect on performance. Thus, for this benchmark, the only parameter that affects the performance is the number of states.

Finally, figures 4.21 and 4.22 show the scalability of parallel implementations for the small input with 768 states and the large input with 6144 states, respectively. The dataflow implementations (with the exception of the OmpSs implementation which is a pure shared memory implementation based on tasks) show better scalability for all thread counts, reaching super-linear speedups due to better cache performance.

4.2.6 MapReduce

This dwarf characterizes the repeated independent execution of a function and the aggregation of the results at the end of the execution, such as in the MapReduce programming model [Dean & Ghemawat (2004)]. Nearly no communication is required between processes. A map function processes a key/value pair to generate a set of intermediate key/value pairs, and then a reduce function merges all intermediate values associated with the same intermediate key. To demonstrate this dwarf, we use an algorithm that processes a text file and maps all distinct words with the number of their occurrences in the text.

Work-sharing cannot be directly applied for this algorithm because partition boundaries have to be properly adjusted to accommodate partitions whose boundaries break the words. Thus, shared-memory implementations are based on tasks. In addition, although there is a straightforward dependency between the *Map* and *Reduce* functions, our experiments have shown that in a shared memory system, given an arbitrary partition, the *Reduce* function needs to be executed immediately after the *Map* function to preserve cache locality. Therefore, our dataflow implementations are functionally equivalent to the fork-join parallelism of shared memory implementations. In addition, for both ADF and OpenMP, there are three different implementations.

Listing 4.14 shows the best performing OpenMP implementation in which the threads that execute tasks reduce the data into their thread-private dictionaries that are merged after the parallel section, in a sequential part of the execution. Thus, synchronization is not necessary. This implementation is notated as *omp_thread*. The second implementation – *omp_part* – differs in that the tasks reduce into partition dedicated dictionaries, instead into thread-private dictionaries, and these dictionaries are later merged into the final dictionary, also in the sequential part of the execution. The third implementation – *omp_global* – is the only one that requires synchronization because the tasks reduce directly into the final dictionary, but it does not require further merging in the sequential part of the program.

Listing 4.15 show the best performing ADF implementation – *adf_thread* – in which the tasks reduce into thread-private dictionaries. This version is equivalent to the *omp_thread* OpenMP implementation. The second ADF implementation – *adf_part* – is similar to the *omp_part* OpenMP implementation in that the tasks reduce into partition dedicated dictionaries. In addition, this implementation creates a number of instances of the *Map* and *Reduce* tasks equal to the number of threads in execution, and a single instance of the *Sum* task, which purpose is to collect all partial reductions generated by the *Reduce* tasks and merge them into a final

```
void Solver::Solve() {
  /* the size of the text per partition in KB */
  partSize  = partSize * 1024;
  int start = 0;
  int end   = partSize;

  #pragma omp parallel
  {
    #pragma omp single
    {
      while (start < textSize) {
        if (end > textSize) {
          end = textSize;
        }
        else {
          while (text[end] != '␣')  /* adjust end to the next space */
            end++;
          end++;
        }

        #pragma omp task firstprivate(start, end)
        {
          std::list<string> partialWordList;
          Map(partialWordList, start, end);
          Reduce(threadReductions[omp_get_thread_num()], partialWordList);
        } /* end task */

        start = end;
        end  += partSize;
      }
    } /* end single */
  } /* end parallel */

  for (int i = 0; i < num_threads; i++)
    SumThreadResult(threadReductions[i]);
}
```

Listing 4.14: The OpenMP implementation of the Map-Reduce problem (*omp_thread*).

```
void Solver::Solve() {
  static int start = 0;

  /* MapReduce tasks */
  #pragma adf task trigger_set(mapToken) relaxed instances(num_threads)
  {
    list<string> *partialWordList = new list<string>();
    Map(*partialWordList, mapToken.start, mapToken.end);
    Reduce(threadReductions[GetThreadID()], *partialWordList);
    delete partialWordList;
  }

  /* Initial task */
  #pragma adf task relaxed until(start >= textSize) pin(0)
  {
    int end = start + partSize;
    if (end > textSize)
      end = textSize;
    else {
      while (text[end] != '␣')
        end++;
      end++;
    }

    mapToken.start = start;
    mapToken.end = end;
    start = end;
  }

  /* start a dataflow execution */
  adf_start();

  /* wait for the end of execution */
  adf_taskwait();

  for (int i = 0; i < num_threads; i++)
    SumThreadResult(threadReductions[i]);
}
```

Listing 4.15: The ADF implementation of the Map-Reduce problem (*adf_thread*).

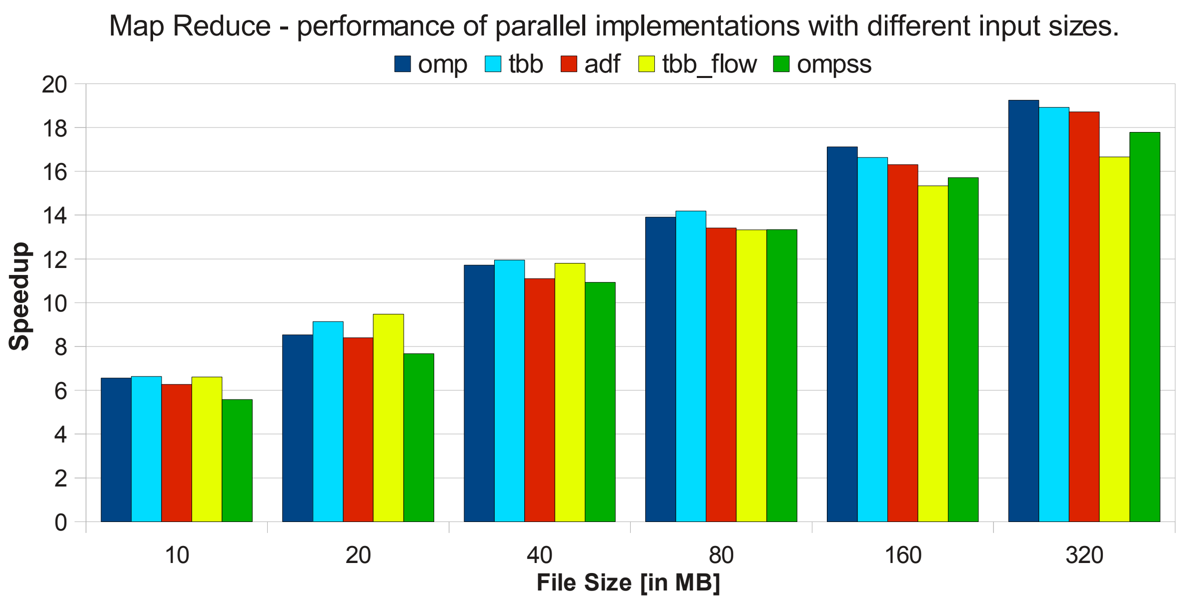


Figure 4.23: Performance sensitivity analysis of parallel implementations of Map-Reduce.

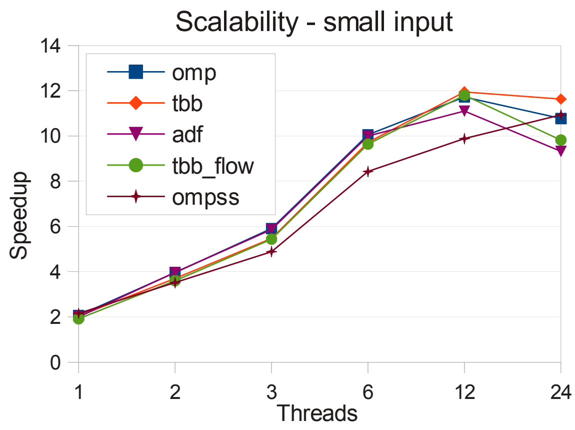


Figure 4.24: Map-Reduce scalability with the small input.

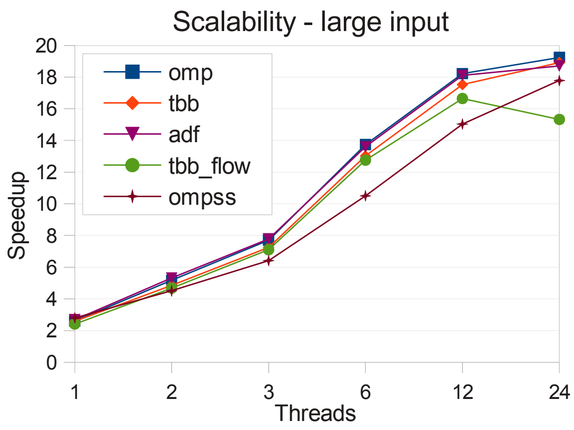


Figure 4.25: Map-Reduce scalability with the large input.

dictionary. The *Sum* task performs merging in the parallel section rather than in the sequential part of the program. By executing only one instance of this task the synchronization is not required while building the final dictionary in parallel. The third implementation – *adf_global* – is equivalent to the *omp_global* implementation.

Performance sensitivity analysis of parallel implementations of the Map-Reduce problem — Figure 4.23 — shows that the performance increases with the input size, which is expected given the pressure that this application has on the cache memory system. Even the single threaded executions of parallel versions have two-fold better performance than the sequential implementation because partitioning the input results in better cache utilization. This can be seen in Figures 4.24 and 4.25 that show scalability of parallel implementations when the small and large inputs are used, respectively. In general, all parallel implementations provide comparable performance with a marginal advantage in favor of the shared memory implementations.

4.2.7 NBody Methods

The NBody methods dwarf represents algorithms that involve interactions between many discrete points in space. There are many variations of the basic concept. In particle-particle methods, every point depends on all others, leading to an $O(N^2)$ calculation. Hierarchical particle methods combine forces or potentials from multiple points to reduce the computational complexity of direct particle-particle methods, resulting in $O(NlogN)$ complexity for the Barnes-Hut algorithm [Barnes & Hut (1986)] that combines forces, and $O(N)$ complexity for the Fast Multipole method [Carrier *et al.* (1988)] that combines potentials of multiple points in space.

The DaSH benchmark for this dwarf implements the Barnes-Hut algorithm [Barnes & Hut (1986)] that applies divide-and-conquer strategy to recursively divide the space into multiple subspaces, thus forming quadtrees or octrees, and treating a distant group of points as a single point, thus reducing the computational complexity of force calculations. The DaSH implementation works in a 3-D space so the algorithm constructs octrees. The parallelization is done by spawning tasks on a desired level of the Barnes-Hut tree that divides the 3-D space until all cubes contain at most one body. Given the Plummer distribution of bodies that are used in the tests, the tree is highly irregular, which means that the subtrees on a given level have different loads. Therefore, spawning tasks on deeper levels of the tree does not help the performance because many tasks end up processing cubes with just a few bodies, while others process the cubes with much larger number of bodies. This introduces an overhead of handling a large number of short tasks, while at the same time providing little in terms of properly partitioning and balancing the work. The Barnes-Hut algorithm essentially depends on barriers that separate tree construction, force calculation and advancing of the bodies in each iteration of the algorithm. Thus, dataflow implementations apply similar approach as the task-based shared memory implementations based on the fork-join parallelism.

Listing 4.16 shows the OpenMP implementation of the *stepsystem* method that represent a single iteration of the algorithm. In the first parallel section – lines 11 to 16 – the worker threads build subtrees that contain at least one body. In the second parallel section –lines 23 to 30 – the threads calculate forces on each body in the underlying subtrees using the approximations based on the distance of the body from the point of interest. Thus, if the center of the masses of a group of bodies is far enough from a body, the force applied is approximated by the cumulative force of such bodies in their center of masses. Finally, in the third parallel section – lines 35 to 41 – the positions and velocities of the bodies are changed according to the current velocity,

```

void Solver::stepsystem() {
  static int firstcall = true;

  if (firstcall)
    InitTree();
  else
    ResetSubtreeMap();

  /* parallel tasks */
  for (int i = 0; i < NUM_SUBTREES; i++) {
    if (tree->subtree[i].nbodies == 0 && tree->subtree[i].nbodies_foreign == 0) continue;
    #pragma omp task firstprivate(i)
    BuildSubtree(i, firstcall);
  }
  #pragma omp taskwait

  firstcall = false;
  FinalizeTree();
  InitGravity();

  /* parallel tasks */
  gravityToken *it = gravityTokenList;
  while (it != NULL) {
    #pragma omp task firstprivate(it)
    GravityCalculation(it);

    it = it->next;
  }
  #pragma omp taskwait

  FinalizeGravity();

  /* parallel tasks */
  for (int i = 0; i < NUM_SUBTREES; i++) {
    if (tree->subtreemap[i].nbodies > 0) {
      #pragma omp task firstprivate(i)
      AdvanceBodies(i);
    }
  }
  #pragma omp taskwait
}

void Solver::Solve() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      while (step++ < num_steps) {
        stepsystem();   /* advance system for a single step */
      }
    }
  }
}
```

Listing 4.16: The OpenMP implementation of the Barnes-Hut algorithm.

the total force applied to each body (calculated in the previous stage of the current step) and the chosen time step (provided as the program parameter).

Listing 4.17 shows the ADF implementation of the Barnes-Hut algorithm. The implementation consist of parallel tasks – *ResetSubtreeMapTask*, *BuildSubtreeTask*, *GravityCalculationTask*, *AdvanceBodiesTask* – that are essentially the same as parallel tasks from the OpenMP implementation, and sequential tasks – *InitTask*, *FinalizeTreeTask*, *FinalizeGravityTask* – that represent sequential code section from the OpenMP implementation with minor differences that are required for the dataflow execution.

```
void Solver::Solve() {
  /* Init task */
  #pragma adf task relaxed execute(1)
  {
    InitTree();
    total_current_tasks = 0;
    for(int i = 0; i < NUM_SUBTREES; i++)
      if (tree->subtreemap[i].nbodies> 0)
        { buildTreeToken[i] = 1; total_current_tasks++; }
  }

  /* ResetSubtreeMap task */
  #pragma adf task trigger_set(resetTreeToken) relaxed
  {
    ResetSubtreeMap();
    total_current_tasks = 0;
    for(int i = 0; i < NUM_SUBTREES; i++)
      if (tree->subtreemap[i].nbodies > 0 || tree->subtreemap[i].nbodies_foreign > 0)
        { buildTreeToken[i] = 0; total_current_tasks++; }
  }

  /* BuildSubtree parallel tasks */
  for(int i = 0; i < NUM_SUBTREES; i++) {
    #pragma adf task trigger_set(buildTreeToken[i]) relaxed
    {
      BuildSubtree(i, buildTreeToken[i]);
      DASH_SYNC_BEGIN(task_lock)
      if (++task_count == total_current_tasks)
        { task_count = 0; finalizeTreeToken = 1; }
      DASH_SYNC_END(task_lock)
    }
  }

  /* FinalizeTree task */
  #pragma adf task trigger_set(finalizeTreeToken) relaxed
  {
    FinalizeTree();
    InitGravity ();
    gravityToken *gt = gravityTokenList;
    while (gt != NULL) { gravToken = gt; gt = gt->next; }
  }

  /* GravityCalculation parallel tasks */
  for(int i = 0; i < NUM_SUBTREES; i++) {
    #pragma adf task trigger_set(gravToken[i]) relaxed
    {
      GravityCalculation(gravToken[i]);
      DASH_SYNC_BEGIN(task_lock)
      if (++task_count == total_current_tasks)
        { task_count = 0; finalizeGravityToken = 1; }
      DASH_SYNC_END(task_lock)
    }
  }

  /* FinalizeGravity task */
  #pragma adf task trigger_set(finalizeGravityToken) relaxed
  {
    FinalizeGravity();
    for (int i = 0; i < NUM_SUBTREES; i++)
      if (tree->subtreemap[i].nbodies > 0) advanceToken[i];
  }

  /* AdvanceBodies tasks */
  for(int i = 0; i < NUM_SUBTREES; i++) {
    #pragma adf task trigger_set(advanceToken[i]) relaxed until(step >= steps)
    {
      AdvanceBodies(i);
      DASH_SYNC_BEGIN(task_lock)
      if (++task_count == total_current_tasks) {
        task_count = 0;
        resetTreeToken = 1;
        step++;
      }
      DASH_SYNC_END(task_lock)
    }
  }
}
```

Listing 4.17: The ADF implementation of the Barnes-Hut algorithm.

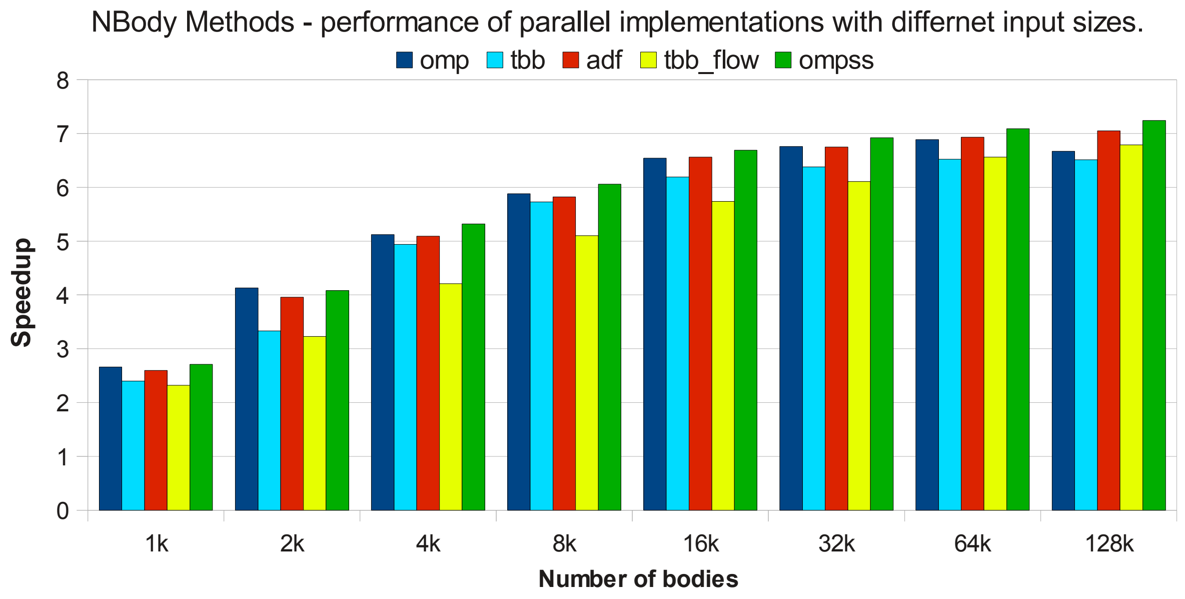


Figure 4.26: Performance sensitivity analysis of parallel implementations of the Barns-Hutt algorithm.

Figure 4.26 shows the sensitivity analysis of the parallel implementations of the Barnes-Hut algorithm for NBody simulation. The number of bodies is varied ranging from 1,000 to 128,000. The parallel tasks are spawned on the second level of the tree, giving 64 cubes in total, out of which around 40% hold at least one body, thus providing enough work for the 24 worker threads that is the maximum used in these experiments. Naturally, in experiments with larger number of threads the tasks would have to be spawned on a deeper level of the tree. As expected, for the small number of bodies, the speedup obtained is modest, but it keeps rising with the increase of the number of bodies until it reaches the maximum for the test machine, which is in a range between 6 and 7. All parallel implementations provide similar performance, except the TBB Flow implementation that performs somewhat worse due to a higher number of cache misses caused by the high level of context switches. Specifically, the TBB and TBB Flow models do not bind the threads to cores, which results in unnecessary thread migrations. Thus, in these two models the number of context switches may reach thousands, especially when using tasks, whereas in other models we may see just a few context switches. Therefore, the TBB version is implemented using work-sharing, which provides 10-15% better performance than the one based on tasks.

Figure 4.27 shows the performance of parallel implementations in a setup with 32,000 bodies, varying the number of steps of the algorithm, which shows that this parameter does not have any effect on speedup. Consequently, the performance gain comes from improvements of the execution of a single step of the algorithm, and not across the steps that must be separated by a barrier.

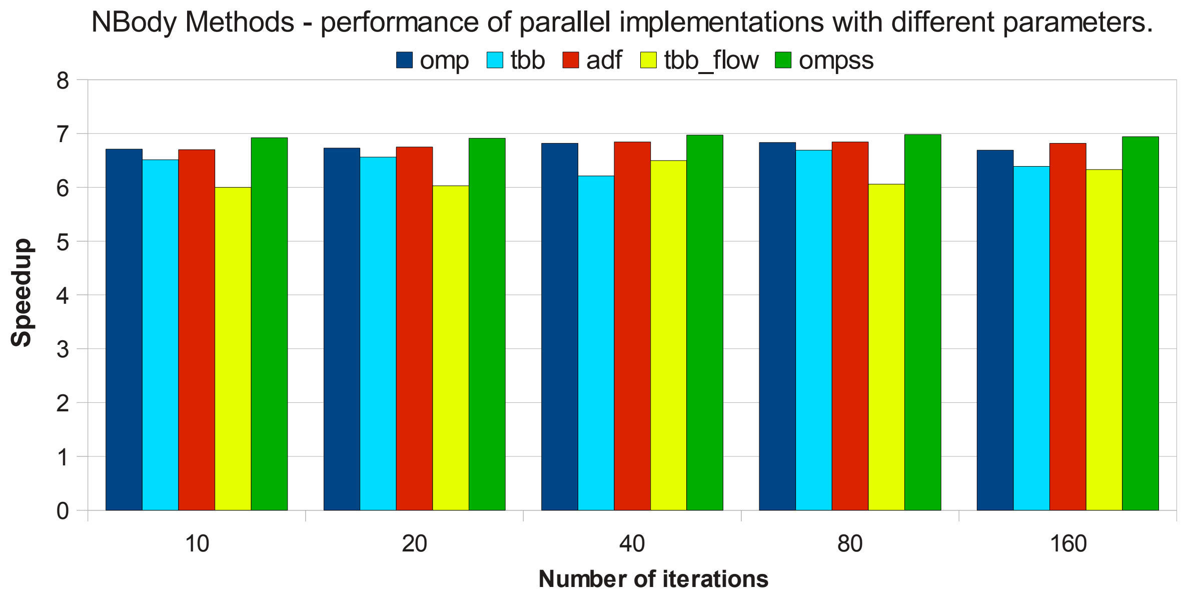


Figure 4.27: Performance analysis of parallel implementations of the of the Barns-Hutt algorithm with 32k bodies, varying the number of algorithm steps.

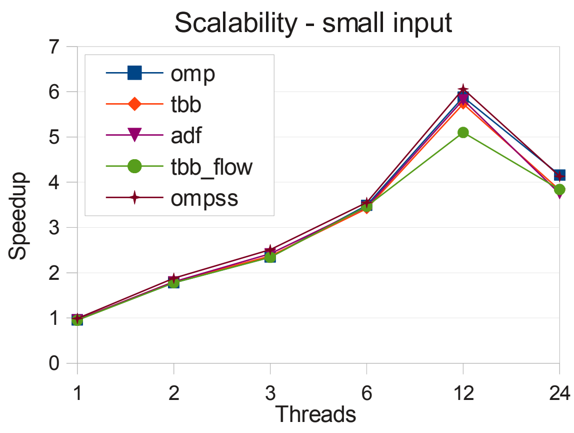


Figure 4.28: Barns-Hutt algorithm. Scalability with the small input.

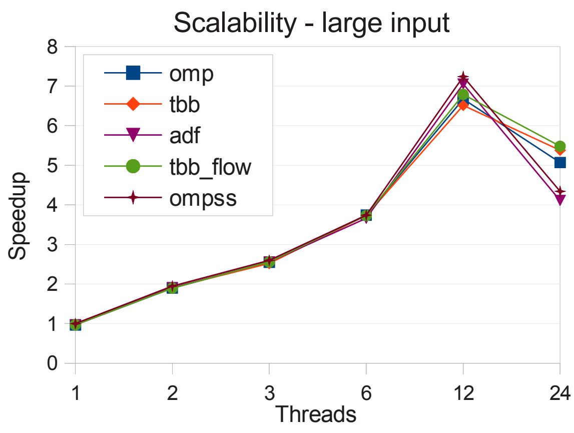


Figure 4.29: Barns-Hutt algorithm. Scalability with the large input.

Figures 4.28 and 4.29 show the scalability of parallel implementations with the small and the large input, containing 8,000 and 128,000 bodies respectively. We can observe two things. First, all implementations suffer performance drop when simultaneous multithreading is used (moving from 12 to 24 worker threads). And second, the negative effects of context switching that characterize the TBB flow version when the small input is used are not observed with the large input because the cache misses have less impact on the performance due to the much larger size of the work that each task needs to perform.

4.2.8 Sparse Algebra

This dwarf represents algorithms that are used when input data sets (vectors and matrices) have a large number of zero entries. The dependence structure of these algorithms tends to be very complex in order to satisfy the goal of avoiding operations on zero entries. Data is usually stored in compressed format to reduce the storage and bandwidth requirements, keeping only the non-zero entries and their indices. Because of the compressed formats, data is accessed with indexed loads and stores.

This dwarf is represented in DaSH with the benchmark that implements a sparse LU decomposition of a matrix that is stored in a block compressed sparse row (BCSR) format. Shared memory implementations are based on work-sharing and implicit barriers are used to separate steps of execution in order to model the complex dependency structure of algorithm operations. Conversely, these dependencies are easily expressed using dataflow, which eliminates the need for barriers and allows a straightforward implementation of the LU algorithm.

The OpenMP solution is shown in Listing 4.18. For each block operation a new task is spawned, but in order to provide the correct result the solution relies on barriers. Specifically, a single thread executes $lu0$ operation on a diagonal block, after which all fwd and $bdiv$ tasks from the corresponding row and column respectively can be executed in parallel. When these tasks finish, the execution of $bmod$ tasks from the lower-right block-matrix can start. Therefore, in each iteration of the outer loop threads synchronize on two barriers: one that separates the execution of $lu0$ task and the execution of corresponding fwd and $bdiv$ tasks, and the other that separates the execution of these tasks and $bmod$ tasks. These operations are applied only to non-zero blocks.

Listing 4.19 shows the ADF implementation of the blocked sparse LU factorization, which enables overlapping execution of different tasks and eliminates the need for the global synchronization. In detail, lu task creates $luToken$ that triggers all fwd tasks in the corresponding row and all $bdiv$ tasks in the corresponding column. fwd task creates $fwdToken$ for the corresponding column. $bdiv$ task creates $bdivToken$ for the corresponding row. $bmod[i, j]$ task waits for $fwdToken$ from column j and $bdivToken$ from row i. It can be triggered from 0 to $min(i, j)$ times depending on its position in the matrix. It creates $bmodToken[i, j]$ that enables either lu, or fwd and $bdiv$ tasks. This task creates initial tokens: $bdiv[0, 1..num_blocks]$ and $bdiv[1..num_blocks, 0]$ for fwd and $bdiv$ tasks in the first row and column of the block matrix, and $bmod[0, 0]$ for the $lu[0]$ tasks. Therefore, the execution of $bmod$ tasks can overlap with the execution of fwd and $bdiv$ tasks from the current step of the algorithm, as well as with the execution of tasks from the next step of the algorithm.

```

void Solver::Solve()
{
  int i, j, k;

  #pragma omp parallel private(k)
  {
    for (k=0; k<num_blocks; k++) {

      #pragma omp single
      {
        lu0(matrix[k*num_blocks+k]);

        for (j=k+1; j<num_blocks; j++)
          if (matrix[k*num_blocks+j] != NULL) {
            #pragma omp task untied firstprivate(k, j)
            fwd(matrix[k*num_blocks+k], matrix[k*num_blocks+j]);
          }

        for (i=k+1; i<num_blocks; i++)
          if (matrix[i*num_blocks+k] != NULL) {
            #pragma omp task untied firstprivate(k, i)
            bdiv (matrix[k*num_blocks+k], matrix[i*num_blocks+k]);
          }
      } /* end single */

      #pragma omp single
      {
        for (i=k+1; i<num_blocks; i++)
          if (matrix[i*num_blocks+k] != NULL)
            for (j=k+1; j<num_blocks; j++)
              if (matrix[k*num_blocks+j] != NULL)
              {
                #pragma omp task untied firstprivate(k, j, i)
                {
                  if (matrix[i*num_blocks+j]==NULL)
                    matrix[i*num_blocks+j] = allocate_clean_block();
                  bmod( matrix[i*num_blocks+k], matrix[k*num_blocks+j],
                        matrix[i*num_blocks+j]);
                }
              }
        } /* end single */
    } /* end for */
  } /* end parallel */
}
```

Listing 4.18: OpenMP implementation of the blocked sparse LU factorization.

Figure 4.30 shows the performance sensitivity analysis of parallel implementations. Similar to the results of the Cholesky decomposition from the dense algebra benchmark — Section 4.2.2 — the parallel implementations provide lower performance when the number of blocks is low due to a lack of available parallelism to keep threads busy. Nevertheless, dataflow implementations perform better than shared memory implementations in all configurations.

Figure 4.31 shows the results of experiments that measure the performance of parallel implementations when the matrix size is constant, while the number of blocks is varied (and consequently the block size). Evidently, the performance increases as the number of blocks is increased. This is partly due to the increased parallelism that keeps threads sufficiently busy, and partly due to better cache utilization as smaller block sizes are used.

Figures 4.32 and 4.33 show the scalability of parallel implementations of the blocked sparse LU factorization, when the small and large inputs are used respectively. The small input represents 3072x3072 matrix divided into 24x24 blocks of size 128, while the large input represents

```

void Solver::Solve() {
  bmod_count = new int[num_blocks*num_blocks];
  for (int i=1; i<num_blocks*num_blocks; i++)
    bmod_count[i] = 0;

  for (int k=0; k<num_blocks; k++) {

    /* lu task */
    #pragma adf task trigger_set(bmodToken[k][k]) relaxed execute(1)
    {
      lu0(matrix[k*num_blocks+k]);
      luToken = k;
    }

    /* fwd tasks */
    for (int j=k+1; j<num_blocks; j++) {
      if (matrix[k*num_blocks+j] == NULL)
        continue;

      #pragma adf task trigger_set(luToken[k], bmodToken[k][j]) relaxed execute(1)
      {
        fwd(matrix[k*num_blocks+k], matrix[k*num_blocks+j]);
        fwdToken = k*num_blocks+j;
      }
    }

    /* bdiv tasks */
    for (int i=k+1; i<num_blocks; i++) {
      if (matrix[i*num_blocks+k] == NULL)
        continue;

      #pragma adf task trigger_set(luToken[k], bmodToken[i][k]) relaxed execute(1)
      {
        bdiv (matrix[k*num_blocks+k], matrix[i*num_blocks+k]);
        bdivToken = i*num_blocks+k;
      }
    }

    /* bmod tasks */
    for (int i=k+1; i<num_blocks; i++){
      if (matrix[i*num_blocks+k] != NULL)
        for (int j=k+1; j<num_blocks; j++)
        {
          if (matrix[k*num_blocks+j] != NULL) {
            bmod_count[i*num_blocks + j]++;
            if (matrix[i*num_blocks+j]==NULL)
              matrix[i*num_blocks+j] = allocate_clean_block();

            #pragma adf task trigger_set(fwdToken[k][j], bdivToken[i][k]) relaxed
            {
              bmod (matrix[bdivToken], matrix[fwdToken], matrix[i*num_blocks+j]);

              bmod_count[i*num_blocks + j] --;
              if (bmod_count[i*num_blocks + j] <= 0)
                bmodToken = i*num_blocks+j;
            }
          }
        } /* end for */
    } /* end for */
  } /* end for */

  /* Create bmod[0,0..num_blocks] tokens */
  for (int i = 0; i < num_blocks; i++) {
    for (int j = 0; j < num_blocks; j++) {
      if (matrix[i*num_blocks + j] == NULL || bmod_count[i*num_blocks + j] > 0)
        continue;

      #pragma adf task relaxed execute(1)
      bmodToken = i*num_blocks + j;
    }
  }
}
```

Listing 4.19: ADF implementation of the blocked sparse LU factorization.

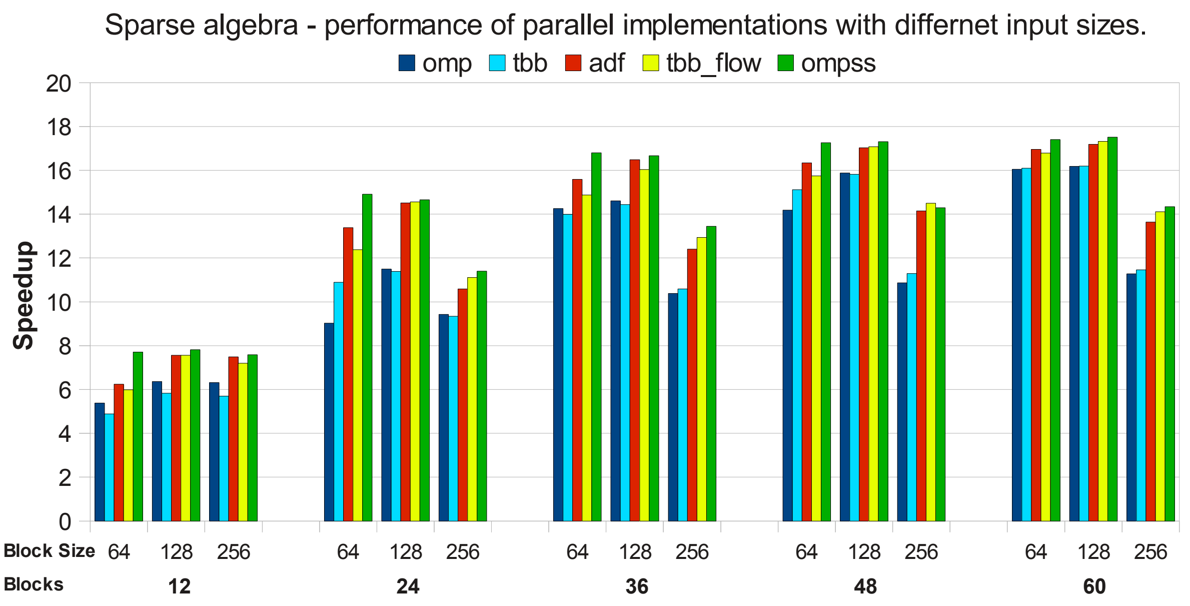


Figure 4.30: Performance sensitivity analysis of parallel implementations of the blocked sparse LU factorization.

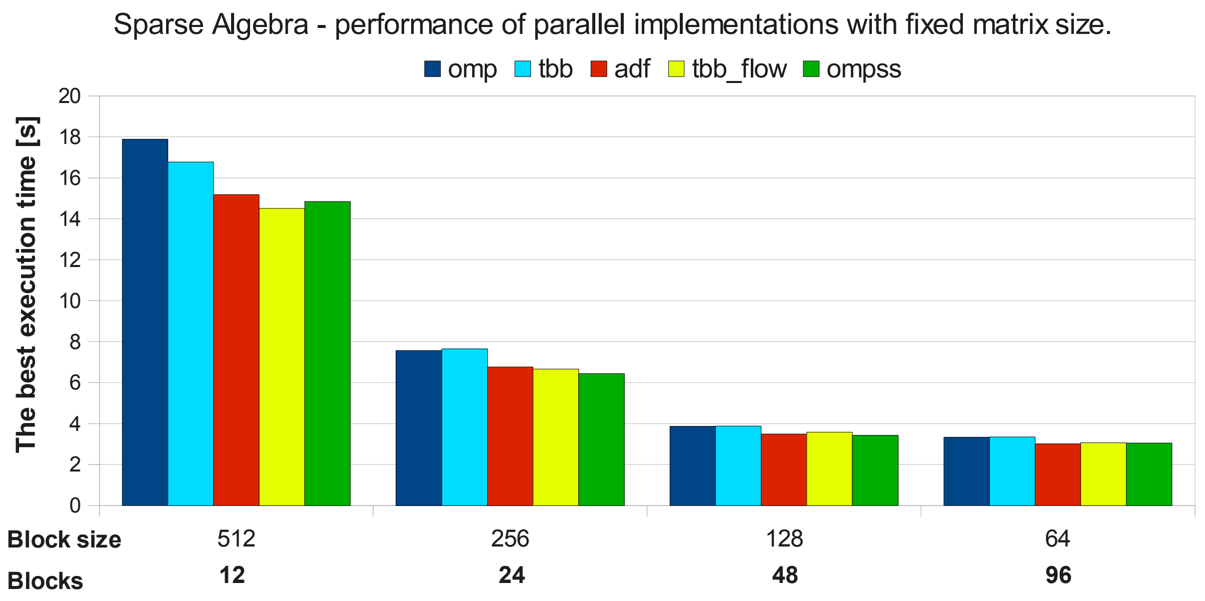


Figure 4.31: Performance analysis of parallel implementations of the blocked sparse LU factorization using the 6144x6144 matrix and varying the number of blocks.

15360x15360 matrix divided into 60x60 blocks of size 256. Both matrices have approximately 20% of non-zero entries. For the small input, the dataflow implementations scale better than the shared memory implementations for all thread counts, whereas for the large input the performance difference is noticeable only for high thread counts. This is due to the increased parallelism when the large number of blocks is used, which is sufficient to keep threads busy even in the case of the shared memory implementations.

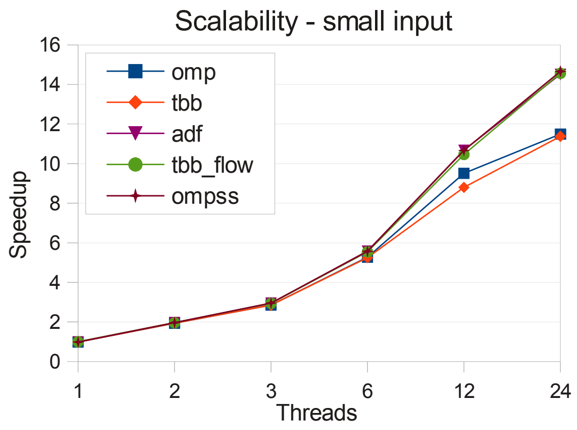


Figure 4.32: Blocked sparse LU factorization. Scalability with the small input.

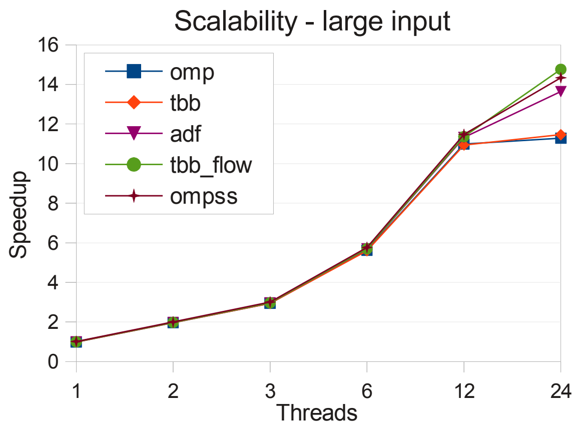


Figure 4.33: Blocked sparse LU factorization. Scalability with the large input.

4.2.9 Spectral Methods

This dwarf features a class of techniques from applied mathematics and scientific computing to numerically solve linear algebra problems that could take an untenable amount of time if done by iterative methods. These techniques utilize Fourier transform to represent a function in terms of its sinusoidal components, such that the resulting transformed function can be used in a way that is much simpler to perform than in the original time or spatial domain. The transform can be discretized in order to perform a transformation on a numerical sequence, such as a single instance of a repeating signal, and limit the resolution to that of the initial input data. The discrete Fourier transform is used in many computational techniques, often to transform the data into a form where a certain operation can be applied more easily, and then perform the reverse transformation to return the data to the former domain.

There are essentially two common mechanisms that can be used: the ordinary, discrete Fourier transform method, which takes $O(n^2)$ time, and a recursive method, known as the fast Fourier transform (FFT) [Cooley & Tukey (1965)], in which the transform is constructed by recursively performing simpler, smaller transforms via a divide and conquer strategy, taking only $O(nlogn)$ time. However, the divide and conquer mechanism leads to complex data dependencies in one-dimensional FFT that form a set of so-called butterfly patterns. These patterns combine multiply-add operations and data permutations, making 1D FFT inefficient to parallelize except for very large data sets or using vector operations. Still, multidimensional forms of FFT are mathematically equivalent to performing 1D FFTs along each dimension in any order. In these algorithms, the stages can be refactored and reordered to improve locality, but this requires one or more steps of global data reordering that can be viewed as transpose operations.

In general, the FFT code is highly vectorizable using unit-stride and constant-stride addressing, although efficiency is greater when multiple transforms are performed simultaneously or the problem is large enough that it can be broken into several smaller sub-problems. These methods have better temporal locality than either structured or unstructured grids and while some spatial locality exists it often requires large strided accesses. The addressing pattern is regular and statically determinable, making hardware and software prefetching fairly effective. Typically, a multidimensional transform is partitioned across multiple processors such that a number of 1D transformations are performed locally on each processor. The algorithms are refactored so that most stages are computed using data local to a node, with a few stages requiring global all-to-all communication for a transpose operation.

The DaSH benchmark for this dwarf implements the 3D FFT algorithm, where basic 1D FFT transformations are handled by the *sfftw* library. Between each of the three 1D FFT stages the data is permuted using transpose operations. Furthermore, the steps of the algorithm are inherently separated with barriers, which means that work-sharing implementations are better suited for this dwarf than dataflow. This can be seen by comparing the OpenMP and ADF implementations in Listings 4.20 and 4.21 respectively. The OpenMP version is based on parallel for loops that share the work in each stage of the algorithm and feature implicit barriers that separate successive stages, whereas the ADF version is based on three type of tasks – *fftxTask*, *transposeXYTask* and *transposeZXTask* – and explicitly implemented barrier that controls the production of the correct type of tokens depending on the previous stage of the algorithm – the first 11 lines in Listing 4.21. Each type of tasks in the dataflow implementation is used twice in the algorithm, in different stages of the execution. For example, *transposeXYTask* is used after the first stage of the algorithm that performs 1D FFT in x dimension to transpose the data between x and y dimensions, and then in the last stage of the algorithm to reverse the data back to its original dimensions.

Figure 4.34 shows the sensitivity analysis of the 3D FFT when the size of the input is varied from 192^3 to 960^3 elements. In general, the difference in performance between parallel implementations is mostly marginal, with the *tbb_flow* implementation performing slightly worse than the other implementations. The overall speedup increases with the size of the input until it stabilizes when the input size reaches 576^3 elements. Figures 4.35 and 4.36 show the scalability of parallel implementations for the small and the large inputs, having 384^3 and 768^3 elements respectively. These diagrams confirm that all parallel implementations have comparable scalability, except the *tbb_flow* implementation that scales worse then other implementations, especially when 12 worker threads are used in execution. However, this implementation seems to profit more than others from the simultaneous multithreading capabilities of the testing platform, finally providing comparable overall performance as the other implementations, as seen in Figure 4.34.

```

void Solver::Solve() {
  /* FFT - X  */
  #pragma omp parallel for
  for (int i = 0; i < num_elements * num_elements; i++)
    fftw_one(xPlan, data[i], NULL);

  /* Transpose X -Y  */
  #pragma omp parallel for
  for (int z = 0; z < num_elements; z++)
    for (int y = 0; y < num_elements; y++)
      for (int x = y + 1; x < num_elements; x++) {
        fftw_complex tmp;
        tmp.re = data[ROW(y,z)][x].re;
        tmp.im = data[ROW(y,z)][x].im;
        data[ROW(y,z)][x].re = data[ROW(x,z)][y].re;
        data[ROW(y,z)][x].im = data[ROW(x,z)][y].im;
        data[ROW(x,z)][y].re = tmp.re;
        data[ROW(x,z)][y].im = tmp.im;
      }

  /* FFT - Y  */
  #pragma omp parallel for
  for (int i = 0; i < num_elements * num_elements; i++)
    fftw_one(xPlan, data[i], NULL);

  /* Transpose Z - X  */
  #pragma omp parallel for
  for (int y = 0; y < num_elements; y++)
    for (int z = 0; z < num_elements; z++)
      for (int x = z + 1; x < num_elements; x++) {
        fftw_complex tmp;
        tmp.re = data[ROW(y,z)][x].re;
        tmp.im = data[ROW(y,z)][x].im;
        data[ROW(y,z)][x].re = data[ROW(y,x)][z].re;
        data[ROW(y,z)][x].im = data[ROW(y,x)][z].im;
        data[ROW(y,x)][z].re = tmp.re;
        data[ROW(y,x)][z].im = tmp.im;
      }

  /* FFT - Z  */
  #pragma omp parallel for
  for (int i = 0; i < num_elements * num_elements; i++)
    fftw_one(xPlan, data[i], NULL);

  /*  At this point Z has the original Y data,
      X has the original Z data, and Y has the original X data */

  /* Reverse Z - X  */
  #pragma omp parallel for
  for (int y = 0; y < num_elements; y++)
    for (int z = 0; z < num_elements; z++)
      for (int x = z + 1; x < num_elements; x++) {
        fftw_complex tmp;
        tmp.re = data[ROW(y,z)][x].re;
        tmp.im = data[ROW(y,z)][x].im;
        data[ROW(y,z)][x].re = data[ROW(y,x)][z].re;
        data[ROW(y,z)][x].im = data[ROW(y,x)][z].im;
        data[ROW(y,x)][z].re = tmp.re;
        data[ROW(y,x)][z].im = tmp.im;
      }

  /* Reverse X -Y  */
  #pragma omp parallel for
  for (int z = 0; z < num_elements; z++)
    for (int y = 0; y < num_elements; y++)
      for (int x = y + 1; x < num_elements; x++) {
        fftw_complex tmp;
        tmp.re = data[ROW(y,z)][x].re;
        tmp.im = data[ROW(y,z)][x].im;
        data[ROW(y,z)][x].re = data[ROW(x,z)][y].re;
        data[ROW(y,z)][x].im = data[ROW(x,z)][y].im;
        data[ROW(x,z)][y].re = tmp.re;
        data[ROW(x,z)][y].im = tmp.im;
      }
}
```

Listing 4.20: The OpenMP implementation of the 3D FFT algorithm.

```
void Barrier(int i) {
  DASH_SYNC_BEGIN(mutex)
  task_count++;
  if(task_count >= num_tasks) {
    task_count = 0;
    barrier_entry++;
    switch (barrier_entry) {
      case 1,6: transposeXYToken = i; break;
      case 2,4: fftxToken = i;        break;
      case 3,5: transposeZXToken = i; break;
    }
  }
  DASH_SYNC_END(mutex)
}

void Solver::fftxTask() {
  for (int i = 0; i < num_tasks; i++) {
    #pragma adf task trigger_set(fftxToken) relaxed
    {
      int startIndex = (num_elements * num_elements * i     ) / num_tasks;
      int endIndex   = (num_elements * num_elements * (i+1)) / num_tasks - 1;
      for (int i = startIndex; i <= endIndex; i++)
        fftw_one(xPlan, data[i], NULL);

      Barrier(i);
    }
  }
}

void Solver::transposeXYTask() {
  for (int i = 0; i < num_tasks; i++) {
    #pragma adf task trigger_set(transposeXYToken) relaxed
    {
      for (int z = i; z < num_elements; z+=num_tasks)
        for (int y = 0; y < num_elements; y++)
          for (int x = y + 1; x < num_elements; x++) {
            fftw_complex tmp;
            tmp.re = data[ROW(y,z)][x].re;
            tmp.im = data[ROW(y,z)][x].im;
            data[ROW(y,z)][x].re = data[ROW(x,z)][y].re;
            data[ROW(y,z)][x].im = data[ROW(x,z)][y].im;
            data[ROW(x,z)][y].re = tmp.re;
            data[ROW(x,z)][y].im = tmp.im;
          }

      Barrier(i);
    }
  }
}

void Solver::transposeZXTask() {
  for (int i = 0; i < num_tasks; i++) {
    #pragma adf task trigger_set(transposeZXToken) relaxed
    {
      for (int y = i; y < num_elements; y+=num_tasks)
        for (int z = 0; z < num_elements; z++)
          for (int x = z + 1; x < num_elements; x++) {
            fftw_complex tmp;
            tmp.re = data[ROW(y,z)][x].re;
            tmp.im = data[ROW(y,z)][x].im;
            data[ROW(y,z)][x].re = data[ROW(y,x)][z].re;
            data[ROW(y,z)][x].im = data[ROW(y,x)][z].im;
            data[ROW(y,x)][z].re = tmp.re;
            data[ROW(y,x)][z].im = tmp.im;
          }

      Barrier(i);
    }
  }
}

void Solver::Solve() {
  fftxTask();
  transposeXYTask();
  transposeZXTask();

  #pragma adf task execute(1) relaxed
  fftxToken = 1;
}
```

Listing 4.21: The ADF implementation of the 3D FFT algorithm.

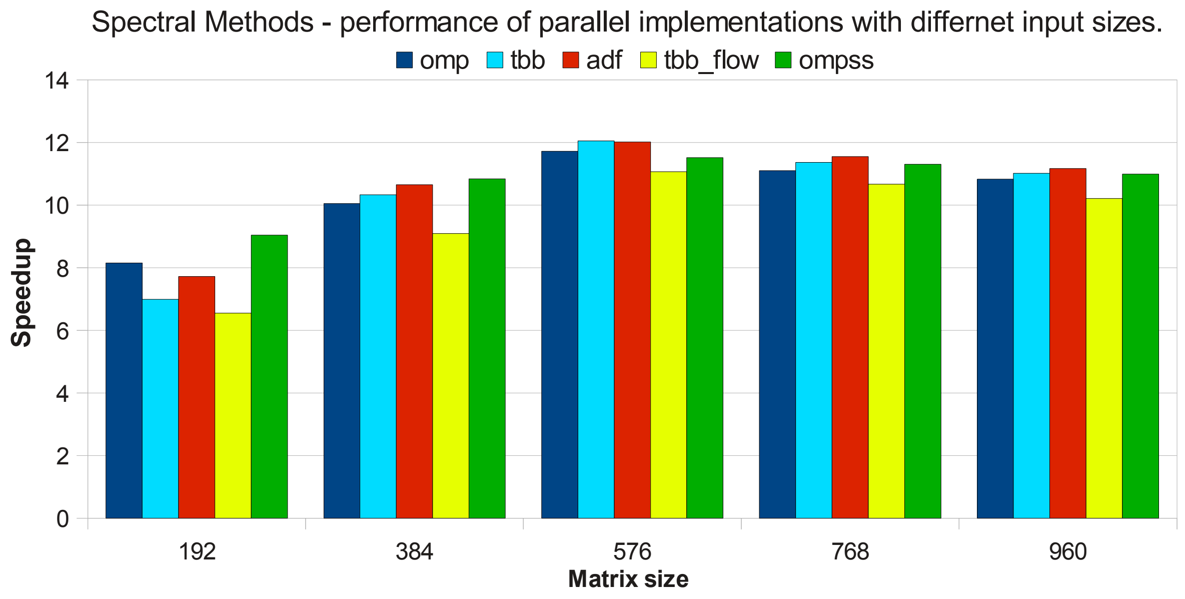


Figure 4.34: Performance sensitivity analysis of parallel implementations of the blocked sparse LU factorization.

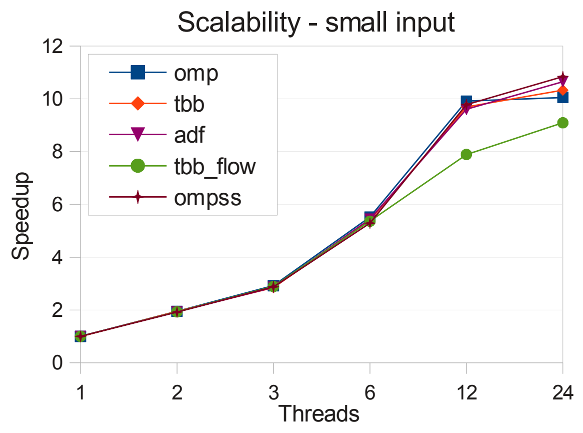


Figure 4.35: 3D FFT scalability. (small input – 384^3 elements).

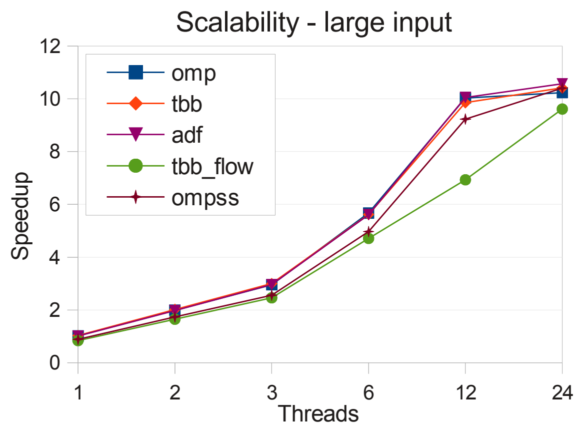


Figure 4.36: 3D FFT scalability. (large input – 768^3 elements).

4.2.10 Structured Grid

The algorithms from this dwarf are characterized by the computation on a regular multidimensional grid. Computation proceeds as a sequence of grid update steps. In each step, a given function is applied in parallel to all nodes in the grid, typically using the data from neighboring nodes. Two types of grids can be distinguished: iterative grids in which a test of convergence is applied after each step of execution until a satisfactory level of correctness has been achieved, and evolutionary grids in which the algorithm proceeds for a certain time interval. Updates may be in place, overwriting the previous version, or use two versions of the grid in which case the updates may be performed in parallel. In some cases updates may alternate (e.g., using a

```
 1
 2  void Solver::Solve() {
 3    for (int cnt = 1; cnt <= iterations; cnt++) {
 4      gerror = 0.0;
 5
 6      /* apply wavefront pattern */
 7      int start, end;
 8      for (int j = 1; j < 2*numblocks; j++) {
 9        if ( j <= numblocks ) {
10          start = 0;
11          end = j;
12        }
13        else {
14          start = j - numblocks;
15          end = numblocks;
16        }
17
18        #pragma omp parallel for shared(cnt, start, end)
19        for (int i = start; i < end; i++) {
20          double lerror = ProcessBlock(i, j-i-1, cnt);
21
22          if (cnt % error_step == 0) {
23            DASH_SYNC_BEGIN(mutex)
24            gerror = fmax(gerror, lerror);
25            DASH_SYNC_END(mutex)
26          }
27        }
28      }
29    }
30  }
```

Listing 4.22: The OpenMP implementation of the Gauss-Seidel stencil computation.

red-black pattern) so that values being updated are not being used by neighbors in the same step. These codes have a high degree of parallelism, and data access patterns are regular and statically determinable.

The structured grid codes are typically amenable to vectorization using unit stride or constant stride memory accesses. Due to a high level of spatial locality, hardware or software prefetching is very effective given the predictable addressing pattern. On the other hand, temporal locality depends on the size of the neighborhood, as each data value is accessed only once by each neighborhood that contains it. Moreover, some neighboring values may not be preserved in cache due to cache-line aliasing when large grids are used.

The DaSH benchmark for this dwarf implements Gauss-Seidel stencil computation on a 2-D square grid. Shared memory implementations apply block-based wavefront parallelization pattern, while dataflow implementations organize block calculations in a dependency graph that resembles a wavefront pattern. However, in shared memory implementations there is an implicit barrier between each step of the wavefront execution as part of the parallel for loop that schedules the execution of blocks from a given step to worker threads – lines 18 to 27 in Listing 4.22. Moreover, there is also a barrier between successive iterations of the algorithm. Dataflow implementations avoid these barriers and only synchronize at every n-th iteration to test for the solution convergence. Given that solutions typically require a few thousand iterations or more to converge to a satisfactory level, we can safely perform convergence test after a certain number of iterations, say ten or twenty, instead of performing the test in each iteration. This

```

void Solver::Solve() {
  /* Block task dependencies:           */
  /*   upper:  forTkn[i-1][j]           */
  /*   left :  forTkn[i][j-1]           */
  /*   lower:  backTkn[i+1][j]          */
  /*   right:  backTkn[i][j+1]          */
  for (int i = 0; i < numblocks; i ++) {
    for (int j = 0; j < numblocks; j ++) {

      if (i == 0 && j == 0)
        #pragma adf task trigger_set(stepTkn, backTkn[i][j+1], backTkn[i+1][j])
                         relaxed
        TaskBody(i,j);

      else if (i == 0 && (j == numblocks - 1))
        #pragma adf task trigger_set(backTkn[i+1][j], forTkn[i][j-1])
                         relaxed
        TaskBody(i,j);

      else if ((i == numblocks -1) && j == 0)
        #pragma adf task trigger_set(backTkn[i][j+1], forTkn[i-1][j])
                         relaxed
        TaskBody(i,j);

      else if ((i == numblocks - 1) && (j == numblocks - 1))
        #pragma adf task trigger_set(forTkn[i][j-1], forTkn[i-1][j])
                         relaxed
        TaskBody(i,j);

      else if (i == 0)
        #pragma adf task trigger_set(backTkn[i][j+1], backTkn[i+1][j], forTkn[i][j-1])
                         relaxed
        TaskBody(i,j);

      else if (i == numblocks - 1)
        #pragma adf task trigger_set(backTkn[i][j+1], forTkn[i][j-1], forTkn[i-1][j])
                         relaxed
        TaskBody(i,j);

      else if (j == 0)
        #pragma adf task trigger_set(backTkn[i][j+1], backTkn[i+1][j], forTkn[i-1][j])
                         relaxed
        TaskBody(i,j);

      else if (j == numblocks - 1)
        #pragma adf task trigger_set(backTkn[i+1][j], forTkn[i][j-1], forTkn[i-1][j])
                         relaxed
        TaskBody(i,j);

      else
        #pragma adf task trigger_set(backTkn[i][j+1], backTkn[i+1][j], forTkn[i][j-1],
                                     forTkn[i-1][j]) relaxed
        TaskBody(i,j);
    }
  }

  /* create inital back tokens */
  for (int i = 0; i < numblocks; i ++) {
    for (int j = 0; j < numblocks; j ++) {
      if (i == 0 && j == 0) continue;

      #pragma adf task execute(1) relaxed
      backTkn[i][j] = 1;
    }
  }

  /* create inital step tokens */
  for (int k = 0; k < error_step; k++)
    #pragma adf task execute(1) relaxed
    stepTkn = 1;
}
```

Listing 4.23: The ADF implementation of the Gauss-Seidel stencil computation. The Solve method

```
 1
 2 void TaskBody(int i, int j) {
 3   step[i][j]++;
 4   double lerror = ProcessBlock(i, j, step[i][j]);
 5
 6   if ((i != numblocks - 1) || (j != numblocks - 1))
 7     forTkn[i][j] = 1;
 8   if ((i != 0) || (j != 0))
 9     backTkn[i][j] = 1;
10
11   if (step[i][j] % error_step == 0) {
12     DASH_SYNC_BEGIN(mutex)
13     if (step[i][j] < total_steps) {
14       barrier++;
15       if (barrier == numblocks*numblocks) {
16         barrier = 0;
17         for (int k = 0; k < error_step; k++)
18           stepTkn = step[i][j];
19       }
20     }
21     else
22       gerror = fmax(gerror, lerror);
23     DASH_SYNC_END(mutex)
24   }
25 }
```

Listing 4.24: The ADF implementation of the Gauss-Seidel stencil computation. The TaskBody method.

has only a negligible effect on performance if large number of iterations is necessary for the solution to converge, so we apply this approximation in all implementations of this benchmark.

Listing 4.23 shows the ADF implementation of this benchmark that is similar to other dataflow implementations. The solution requires a long selection, shown in lines 11 to 54, to appropriately declare dependencies for each task, depending on the position of the block in the grid. In general, a task that processes a block $[i, j]$ from the center of the grid requires two forward tokens from the tasks $[i - 1, j]$ and $[i, j - 1]$ to signal that corresponding tasks have finished the processing in the current step of execution, and two backward tokens from the tasks $[i + 1, j]$ and $[i, j + 1]$ to confirm that corresponding tasks have finished the processing in the previous step of execution – line 52. However, the tasks that process blocks located on the edges of the grid do not require all these dependencies. For example, the task that processes the block at position $[0, 0]$ requires only backward tokens – line 12. In addition, only this task has additional dependency on token *stepTkn* that is used to control the number of executions of each task before the test for convergence is performed. This token is initially produced in the *Solve* method to start the dataflow execution – lines 69 to 71 in Listing and 4.23, and later in the *TaskBody* method to initiate further iterations – line 18 in Listing 4.24.

Figure 4.37 shows the performance sensitivity analysis of parallel implementations of this algorithm. Evidently, the size of the input has little effect on the overall performance of the ADF and TBB Flow implementations, while the performance of other implementations increases with the increase in the grid size. In general, dataflow implementations have much better performance than shared memory implementations due to increased parallelism that comes as a result of barrier elimination. The lower performance of the OmpSs implementation, compared

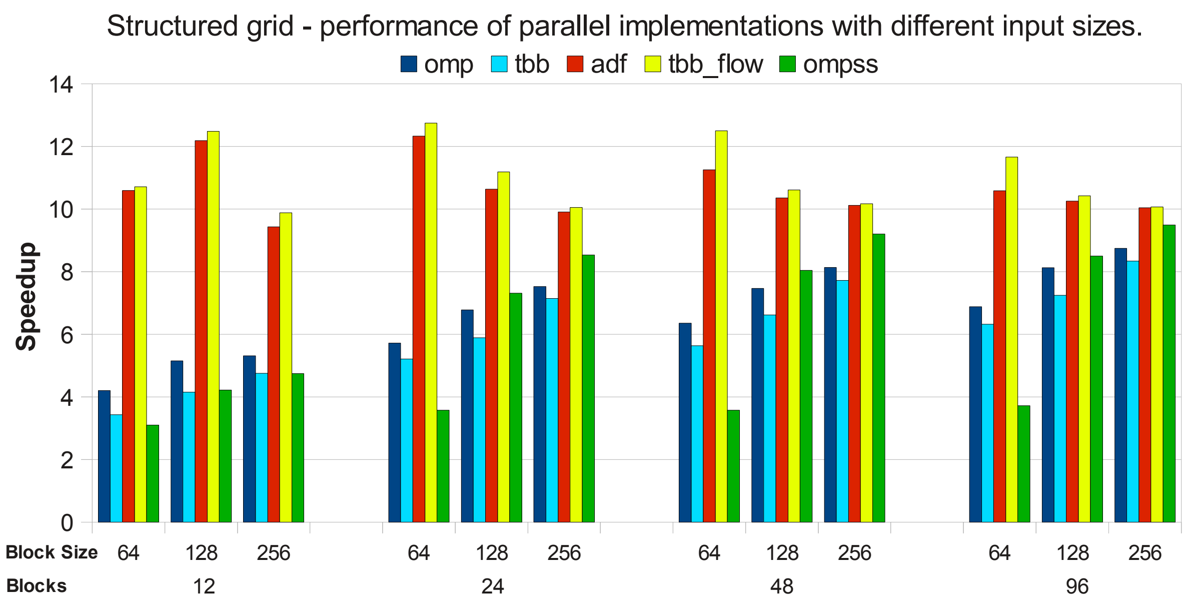


Figure 4.37: Performance sensitivity analysis of parallel implementations of the Gauss-Seidel stencil computation on the structured grid.

to other two dataflow implementations, is a consequence of the fact that this model discards the task after it finishes a single execution, whereas the other models reuse the tasks by providing a new input data values that enable new invocations of the same tasks. While the ADF and TBB Flow models create the number of tasks equal to the number of blocks in the grid once, before the dataflow execution begins, OmpSs creates the same number of tasks in each iteration. Since the convergence test is performed after N iterations (20 in these experiments), the OmpSs implementation creates batches of $N * nblocks * nblocks$ tasks, which evidently has a negative impact on performance.

Figure 4.38 shows the performance of parallel implementations with fixed grid size while varying the number and the size of blocks, accordingly. The best performance is obtained using the smallest blocks with 64x64 elements that result in execution with the most optimal cache behavior. This also shows that dataflow models can support executions with a large number of tasks efficiently (almost 10000 tasks in these experiments).

Figures 4.39 and 4.40 show the scalability of parallel implementations of the Gauss-Seidel stencil computation on the small and large input grids respectively (3072x3072 and 12288x12288 elements). For all thread counts, dataflow implementations provide better performance than the shared memory implementations. Furthermore, the diagrams show that the OmpSs implementation has better scalability for the larger grid, which suggests that the impact of creating a large number of tasks is lower than the impact of the block size on the cache performance.

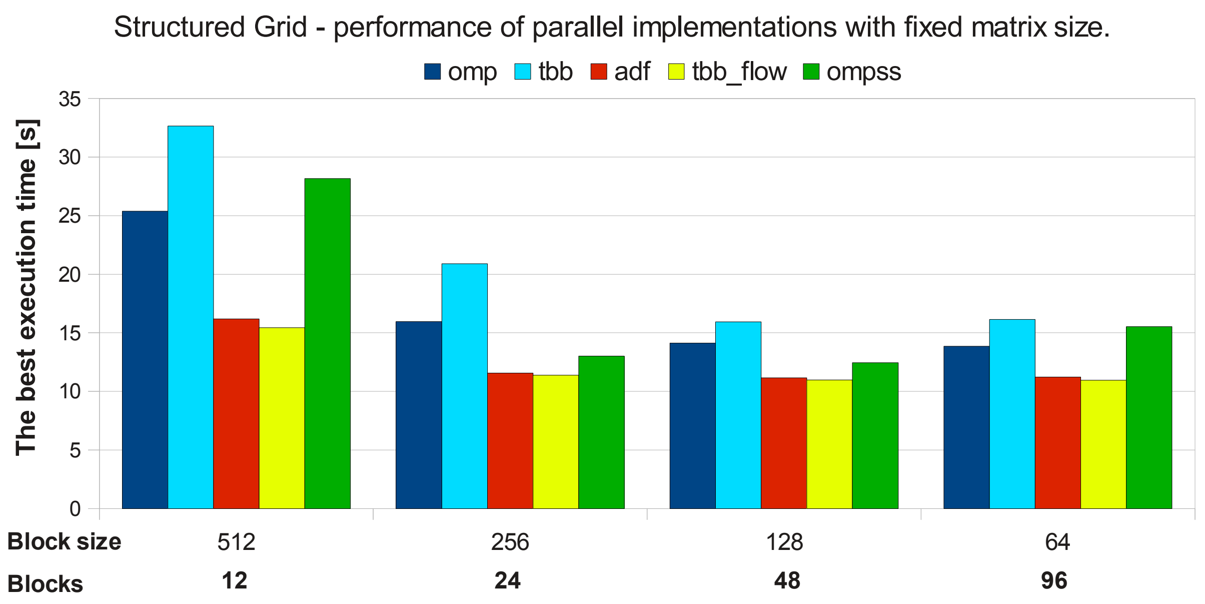


Figure 4.38: Performance analysis of parallel implementations of the Gauss-Seidel stencil computation using the 6144x6144 matrix and varying the number of blocks.

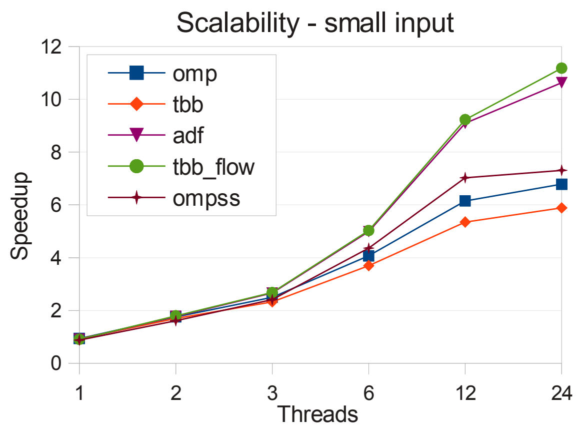


Figure 4.39: Gauss-Seidel stencil computation. Scalability with the small input.

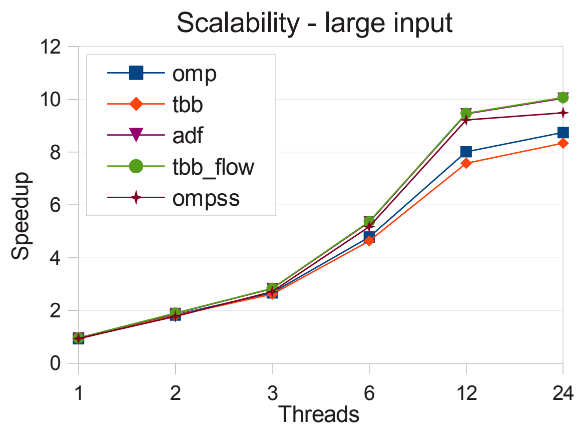


Figure 4.40: Gauss-Seidel stencil computation. Scalability with the large input.

4.2.11 Unstructured Grid

This dwarf contains algorithms that perform updates on unstructured grids. Unlike structured grids that model regularly shaped objects, such as squares and cubes, unstructured grids are typically defined as meshes that model the surfaces or volumes of irregularly shaped objects, where mesh granularity strongly depends on local properties of the problem domain. Entities in the mesh, such as points, edges, faces, and/or volumes, must be explicitly represented and linked by pointers or integer offsets. These entities model different physical values that change over time, such as tension, temperature and pressure. Computation proceeds as a sequence of mesh update steps. Updates typically involve multiple levels of memory references, because the update of each point requires to first determine a list of neighboring points, and then to load values from them.

The code for unstructured grids is often highly vectorizable, but requires scatter-gather memory accesses to retrieve data items. Memory hierarchies are less effective on these codes. Spatial locality is usually poor as data is accessed indirectly with patterns dependent on the mesh structure. Since a given entity is used in the calculations for several different neighborhoods there is potential for some temporal locality, which can be improved by using graph partitioners to place neighboring points nearby in memory. Although the memory access patterns repeat from one mesh update to the next, each step is too long to record as a hardware prefetch pattern, and the indirection in the code makes compile-time analysis difficult. Because the order in which the data items are visited is static within a given iteration, a software prefetch mechanism can be used with some success. However, the prefetch operations in a conventional cache-hierarchy can still suffer from undesired cache-line evictions due to cache-line aliasing.

The main problem in parallel algorithms for this dwarf is efficient partitioning of data that can provide both balanced workloads and minimal communication, which is difficult given the irregular nature of the problem. Graph partitioners are often used to partition the mesh to a number of partitions with approximately equal number of elements, while reducing the number of edges between partitions to minimize the communication. This is usually performed as a preprocessing step before the start of the algorithm, or the data itself may be partitioned beforehand. Several software packages exist for solving this graph partitioning problem, possibly the most notable being *Metis* [Karypis & Kumar (1998)].

The algorithm that represents this dwarf in DaSH is the Jacobi stencil computation on an unstructured triangular mesh grid. The input meshes used in experiments are taken from the repository of the Stanford Computer Graphics Laboratory [Stanford (2013)]. *Metis* partitioner is used to partition a mesh in a preprocessing step of a program. In all implementations, the convergence test is performed after n iterations instead at the end of each iteration. Two meshes are alternately used to enable greater parallelism.

In shared memory implementations, a task is spawned for each partition and all tasks synchronize at the end of each iteration. Thus, there is a barrier between successive iterations of the algorithm. Listing 4.25 shows the OpenMP implementation that is implemented using a work-sharing construct that has an implicit barrier at the end. Adjacency list is compressed as an array that is accessed using a auxiliary index array, *adjacency_start*.

Dataflow implementations avoid barriers by organizing calculations of different partitions based on their mutual dependencies. These dependencies are deduced using adjacency list and partition lists provided by *Metis* to find all edges that connect different partitions. If there is an

```

void Solver::Solve()
{
  for (int cnt = 1; cnt <= iterations; cnt++) {

    gerror = 0.0;

    #pragma omp parallel for schedule(dynamic, 1)
    for (long p = 0; p < num_partitions; p++) {
      double lerror = 0.0;

      vector<long>::iterator it;
      for(it = partition[p].begin(); it != partition[p].end(); it++) {
        double sum = 0.0;
        int index = *it;
        long num_neighbors = adjacency_start[index+1] - adjacency_start[index];

        for (long j = adjacency_start[index]; j < adjacency_start[index + 1]; j++)
          sum += vertex[adjacency[j]];

        tmp_vertex[index] = (num_neighbors > 0) ? sum / num_neighbors : 0.0;

        if (cnt % error_step == 0)
          lerror = fmax(lerror, fabs(tmp_vertex[index] - vertex[index]));
      }

      if (cnt % error_step == 0) {
        DASH_SYNC_BEGIN(mutex)
        gerror = fmax(gerror, lerror);
        DASH_SYNC_END(mutex)
      }
    }

    double *t = vertex;
    vertex = tmp_vertex;
    tmp_vertex = t;
  }
}
```

Listing 4.25: OpenMP implementation of the Jacobi stencil computation.

edge connecting two partitions then there is a dependency between them. Thus, a dependency graph is built between partition tasks, each of which can have different number of dependencies. Listing 4.26 shows the ADF implementation of this benchmark that is based on a selection that checks the number of dependencies for a given partition and creates a dataflow task accordingly using *computeToken* tokens – lines 42 to 67. Similar to the solution of the Gauss-Seidel stencil computation from the Structured grid dwarf, there is an additional *stepToken* token that controls the global synchronization of dataflow tasks in order to test the convergence after a set number of iterations. Lines 22 to 37 implement the control logic for generating output tokens: each tasks generates associated *computeToken* after each step, and a single task that is the last to reach the task barrier generates *stepToken* tokens.

While the OmpSs solution is similar to the ADF solution and also relies on the switch statement for declaring dataflow tasks for each number of possible input dependencies that a partition might have, the TBB Flow implementation is more complicated. Specifically, since the types of the `function` node and the corresponding `join` node that represents a task depend on the number of input dependencies that a partition has, it was necessary to implement a factory construction pattern to build these nodes and provide a class specialization for every possible number of input dependencies that a partition might have – Listing 4.27. Listing 4.28 shows

```

void Solver::ProcessPartition(int pId) {
  double *ingrid   = (step[pId]%2) ? vertex : tmp_vertex;
  double *outgrid  = (step[pId]%2) ? tmp_vertex : vertex;
  double lerror = 0.0;
  step[pId]++;

  vector<long>::iterator it;
  for(it = partition[pId].begin(); it != partition[pId].end(); it++) {
    double sum = 0.0;
    int index = *it;
    long num_neighbors = adjacency_start[index+1] - adjacency_start[index];

    for (long j = adjacency_start[index]; j < adjacency_start[index + 1]; j++)
      sum += ingrid[adjacency[j]];
    outgrid[index] = (num_neighbors > 0) ? sum / num_neighbors : 0.0;

    if (step[pId] % error_step == 0)
      lerror = fmax(lerror, fabs(outgrid[index] - ingrid[index]));
  }

  if (step[pId] % error_step == 0) {
    DASH_SYNC_BEGIN(mutex)
    gerror = fmax(gerror, lerror);
    if (step[pId] < iterations) {
      if  (++barrier == num_partitions) {
        barrier = 0;
        for (int k = 0; k < error_step; k++)
          stepToken = step[pId];          /* generate output */
      }
    } else {
      gerror = fmax(gerror, lerror);
      result = outgrid;
    }
    DASH_SYNC_END(mutex)
  }
  computeToken[pId] = step[pId];     /* generate output */
}

void Solver::Solve() {
  /* create Compute tasks */
  for (int i = 0; i < num_parts; i ++) {
    vector<long> &d = dependencies[i];
    switch (d.size()) {
    case 1:
      #pragma adf task trigger_set(stepToken, computeToken[d[0]]) relaxed
      ProcessPartition(i);
      break;
    case 2:
      #pragma adf task trigger_set(stepToken, computeToken[d[0]],
                                   computeToken[d[1]]) relaxed
      ProcessPartition(i);
      break;
    case 3:
      #pragma adf task trigger_set(stepToken, computeToken[d[0]],
                                   computeToken[d[1]], computeToken[d[2]]) relaxed
      ProcessPartition(i);
      break;
      .
      .
      .
    default:
      printf("Too␣many␣dependencies␣for␣a␣partition!\n");
      exit(0);
      break;
    }
  }

  /* generate initial compute tokens */
  for (int i = 0; i < num_parts; i ++) {
    #pragma adf task execute(1) relaxed
    computeToken[i] = 1;
  }

  /* generate initial step tokens */
  #pragma adf task execute(1) relaxed
  for (int k = 0; k < error_step; k++)
    stepToken = 1;
}
```

Listing 4.26: ADF implementation of the Jacobi stencil computation.

```

struct NodeBase {
  typedef tuple<> node_type;

  static graph *g;
  static Solver *solver;
  static NodeBase *make_node(int num_dependencies, int partId);
  long   partitionId;

  NodeBase(int pId) : partitionId(pId) {}
  virtual ~NodeBase() {}
  /* called by producers to pass a token to a consumer. */
  virtual void PutToken(const int port, ttype token) = 0;
  virtual void Init() = 0;
};

template <long N>
struct Node: NodeBase {
  Node(long pId) : NodeBase(pId) {}
  virtual ~Node() {}
  virtual void PutToken(const int port, ttype token) {
    cout << "Node::PutToken:␣This␣node␣type␣is␣not␣supported!"  << endl;
  }
  virtual void Init() {}
};

/* specialization for the node with three input dependencies */
template <>
struct Node<3>: NodeBase {
  typedef tuple<ttype, ttype, ttype> node_type;

  function_node<node_type> *func;
  join_node<node_type>     *join;

  Node<3>(long pId) : NodeBase(pId) {
    func = new function_node<node_type>(*g, 1 , [=] (node_type input) -> void {
      solver->ProcessPartition(pId);
    });

    join = new join_node<node_type>(*g);
    make_edge(*join, *func);
  }

  virtual ~Node<3>() {
    delete func;
    delete join;
  }

  virtual void PutToken(int port, ttype token) {
    switch (port) {
      case 0: input_port<0>(*join).try_put(token); break;
      case 1: input_port<1>(*join).try_put(token); break;
      case 2: input_port<2>(*join).try_put(token); break;
      default: cout << "TryPut:␣port␣number␣is␣too␣large!" << endl; break;
    }
  }

  virtual void Init() { func->try_put(node_type()); }
};

/* make nodes with N+1 inputs. The first input
 * (port 0) is always for the step token  */
NodeBase * NodeBase::make_node(int num_dependencies, int partId) {
    NodeBase *newnode = NULL;
    switch (num_dependencies) {
      case 0: cout << "The␣node␣must␣have␣at␣least␣one␣input." << endl;
              exit(1); break;
      case 1: newnode =  new Node<2>(partId); break;
      case 2: newnode =  new Node<3>(partId); break;
      case 3: newnode =  new Node<4>(partId); break;
       ...
      default: cout << "Nodes␣with␣" << num_dependencies <<
                       "␣inputs␣are␣not␣supported!" << endl;
                break;
    }
    return newnode;
}
```

Listing 4.27: Factory method for the TBB Flow Graph Jacobi stencil computation.

```

void Solver::Solve()
{
  NodeBase::g = new graph;
  NodeBase::solver = this;

  for (int i = 0; i < num_partitions; i++)
    nodes[i] = NodeBase::make_node((int) dependencies[i].size(), i);

  for (long cnt = 1; cnt <= iterations; cnt+=error_step) {
    gerror = 0.0;

    if (cnt == 1) {
      for (int i = 0; i < num_partitions; i++) {
        /* Send tokens directly to function nodes.
         * This will start task execution. */
        nodes[i]->Init();

        /* That is why we need to generate error_step-1 step tokens. */
        for (int s = 0; s < error_step - 1; s++)
          nodes[i]->PutToken(0, s+1);
      }
    }
    else {
      /* generate error_step step tokens (always port 0).*/
      for (int i = 0; i < num_partitions; i++) {
        for (int s = 0; s < error_step; s++)
          nodes[i]->PutToken(0, s+1);
      }
    }

    NodeBase::g->wait_for_all();
  }

  delete NodeBase::g;
}
```

Listing 4.28: Parallel section in the TBB Flow Graph Jacobi stencil computation.

how this factory method is used to create nodes. With the current gcc implementation of the `tuple` class that supports up to ten tuple arguments, this solution is limited to partitions that have up to ten dependencies, which is obviously quite limiting. In general, this approach is not scalable, even in the case of the ADF and OmpSs implementations. A possible approach in the ADF model could be to supply an arbitrary number of dependencies to a task using C++ vectors. However, this extension is not currently implemented.

Figure 4.41 shows the performance sensitivity analysis of parallel implementations of this benchmark. As we can see, the parallel performance increases consistently with the size of the input, but it is noticeably smaller than in the case of the Structured grid benchmark due to more complex dependencies between partitions and worse cache behavior as a result of indirect memory accesses. Moreover, although dataflow models can express dependencies naturally and eliminate the need for global synchronization, we see that these implementations show a minor performance improvement compared to the shared memory implementations for the larger meshes, whereas for the smaller meshes they fall short. The scalability graphs, shown in Figures 4.42 and 4.43, confirm these findings.

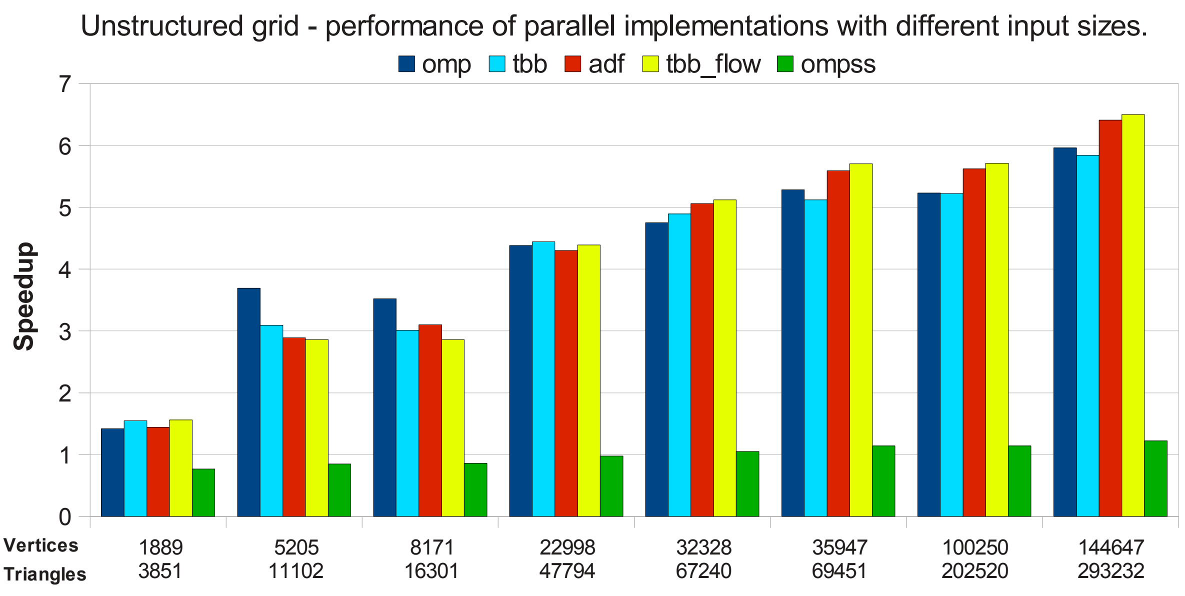


Figure 4.41: Performance sensitivity analysis of parallel implementations of the Jacobi stencil computation on the 2D unstructured grid.

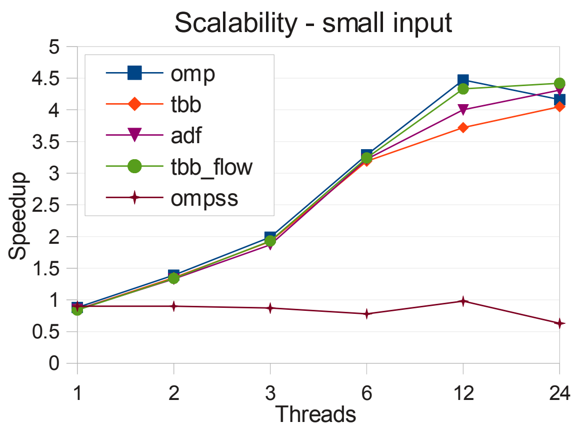


Figure 4.42: Jacobi stencil computation. Scalability with the small input.

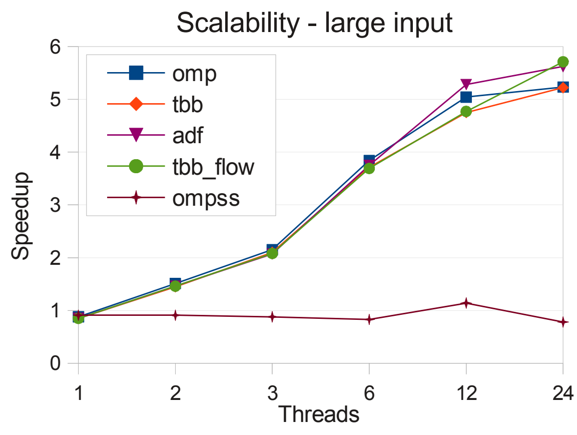


Figure 4.43: Jacobi stencil computation. Scalability with the large input.

4.3 The Properties of the DaSH Benchmark Suite

The decision to base the design of the DaSH benchmark suite on Berkeley dwarfs provides comprehensive variety of algorithms and application domains. In addition, most of the benchmarks are not trivially parallelizable, because they are either irregular or their characteristics change during the execution. Thus, DaSH is characterized with *breadth*.

For each benchmark, DaSH includes sequential, two shared memory implementations (OpenMP and TBB) and three hybrid dataflow implementations (OmpSs, ADF and TBB Flow Graph framework). Shared memory implementations are further based on work-sharing, tasking or a combination of these two. Thus, DaSH is also characterized with *depth*.

	seq (Total)	omp	tbb	adf	tbb_flow	ompss
Number of lines of code	3189	14%	13%	25%	37%	17%
Function count	242	1%	1%	9%	26%	4%
# lines of code per function	252	20%	20%	25%	24%	21%
Cyclomatic complexity	43.9	15%	12%	17%	21%	21%
Average number of tokens	1149.0	23%	26%	27%	35%	33%

Table 4.1: Code complexity of parallel implementations of DaSH benchmarks compared to sequential baseline.

Moreover, different execution aspects of DaSH benchmarks can be controlled using command line parameters, such as the size and the nature of the problem (e.g. the particle distribution in NBody methods). Also, the parameters of the parallelization can be changed, such as the number of tasks or partitions that an application will generate and the number of threads in execution. DaSH comes with an input generator that can be used to generate input files for the most of the benchmarks. Ultimately, DaSH is written in C/C++, which also makes it *portable*.

4.3.1 Code Complexity Analysis

Table 4.1 summarizes the data obtained comparing the code complexity of different implementations of the DaSH benchmarks using the *hfcca.py* tool [Yin (2012)]. Since DaSH benchmarks execute operations mostly by calling library functions, or they perform simple operations on large datasets, the number-of-lines-of code may be a misleading metric. Compared to the sequential implementation, on average we see 17-30% increase in number of lines of code of dataflow implementations, and only 14% increase in case of non-dataflow implementations. In absolute terms, this represents 50-70 additional lines of code on average. On the other hand, all parallel implementations increase the average number of tokens almost equally. This metrics is sometimes considered as a more reliable measure of the code size since it measures the number of control structures, variable names, and non-blank separators in the code.

In the extreme case, the TBB flow implementation of the *Unstructured grid* benchmark increases the code size by 268 additional lines. In this benchmark, although most of the partitions for our input mesh have up to five dependencies, there are also a few partitions with seven, eight and even nine dependencies, which complicates the implementation. Specifically, the ADF and OmpSs implementations have to provide a switch case for each number of possible input dependencies that a partition might have (the only output dependency is the partition itself). In the TBB Flow version the reason for such a high increase in the code size is that it was necessary to implement a factory construction pattern to build the nodes, as explained in Section 4.2.11.

Dwarf	DaSH benchmark	irregular structure	shared memory		dataflow		programm-ability
			barrier required	TM synch.	barrier required	TM synch.	
branch & bound	Traveling Salesman Problem	●		moderate		moderate	dataflow tasking
dense algebra	Cholesky Factorization	●	●				dataflow
dynamic programming	Knapsack problem						dataflow work-sharing
finite state machine	Pattern search			light		light	tasking
graphical models	Viterbi algorithm		●				dataflow work-sharing
map reduce	Text mining						dataflow tasking
nbody methods	Barnes-Hut	●	●	light	●	light	tasking
sparse algebra	LU Factorization	●	●				dataflow
spectral methods	3D FFT		●		●		work-sharing
structured grid	Gauss-Seidel computation		●	light		light	dataflow
unstructured grid	Jacobi iterative method	●	●	light		light	dataflow work-sharing

Figure 4.44: Summary of the programmability of the DaSH benchmarks.

Dataflow implementations also increase the number of function calls slightly more than shared memory implementations; however, in terms of the remaining metrics, all parallel implementations have similar complexity. Since the DaSH benchmarks do not rely on synchronization heavily, the cyclomatic complexity metric that measures a number of linearly independent paths through a program's source code (lower the better) is also valid for our analysis. From Table 1 we see that also in this respect the parallel implementations have the same complexity. Overall, we can conclude that dataflow implementations are characterized with similar code complexity as shared memory implementations. Our experience in developing DaSH confirms this, since once we got accustomed to the dataflow paradigm, developing dataflow implementation usually required the same amount of time as developing their shared memory counterparts.

4.3.2 Summary of the DaSH Programmability

Figure 4.44 summarizes our findings regarding the programmability of the DaSH benchmarks in different models. It further shows that no single parallel programming paradigm is suitable

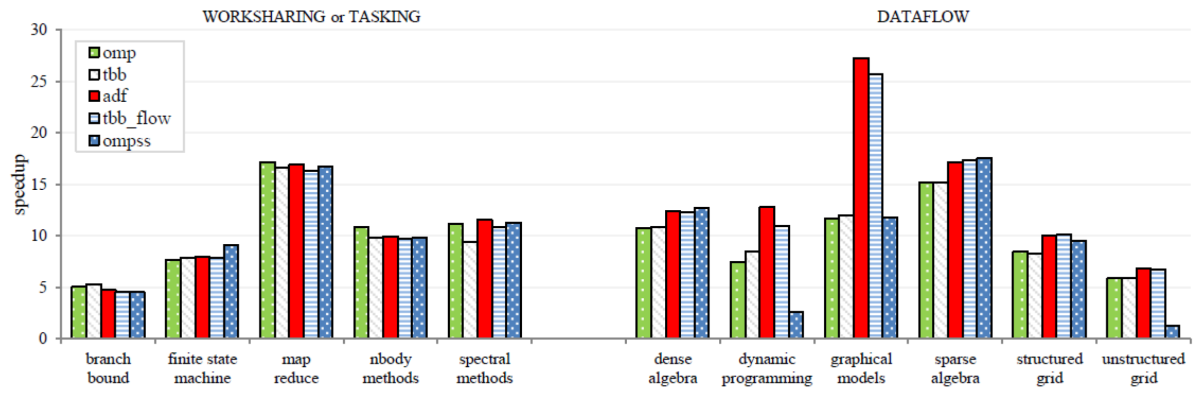


Figure 4.45: The maximum speedup of parallel DaSH benchmarks compared to the sequential baseline using the large input set. The maximum number of threads in execution is 24.

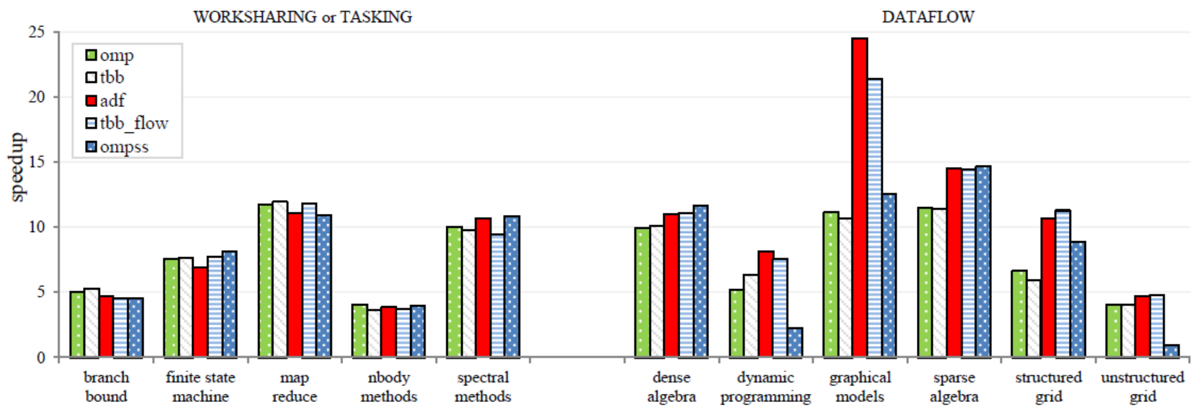


Figure 4.46: The maximum speedup of parallel DaSH benchmarks compared to the sequential baseline using the small input set. The maximum number of threads in execution is 24.

for all DaSH benchmarks. Still, dataflow is preferable for algorithms in which operations are dependent on each other in a way that prevents easy expression of the algorithm using worksharing or tasking. Moreover, since modern hybrid dataflow models are based on tasks, they can readily be used for algorithms that require tasking model for efficient implementation.

4.4 Performance Summary

Figure 4.45 and Figure 4.46 show maximum speedups achieved by parallel implementations compared to corresponding sequential baseline when large and small input data sets are used respectively. Evidently, the results support the view of the authors of Berkeley dwarfs [Asanovic *et al.* (2009)] that it is unlikely that a single programming model can support efficient implementation of the whole application spectrum. Rather, each problem should be solved in the most appropriate way.

The first group consists of DaSH benchmarks that do not show a performance improvement when dataflow is used. In *NBody methods* and *Spectral methods* dwarfs, barriers are inherent to the algorithm and using dataflow does not increase the parallelism. In these cases, using traditional tasking or work-sharing seems to be the right choice. Similarly, in *Branch and Bound* dwarf, the key is to minimize the problem space by minimizing branching, which means that dataflow implementations work in a similar way as task-based shared memory implementations. Moreover, they perform the same, which means that both approaches provide equally good support for the implementation of this dwarf.

Common to *Finite state machine* and *Map Reduce* dwarfs is that, in all parallel implementations, the problem is partitioned to a number of independent tasks. Parallelism has a simple fork-join form. Although *Map Reduce* is characterized with a straightforward dependency structure between *Map* and *Reduce* tasks, on our test machine the best results provides an implementation in which these functions are executed inside the same task, as detailed in Section 4.2.6.

The second group of DaSH benchmarks represents applications for which dataflow provides superior performance. DaSH benchmarks for *Dense algebra*, *Sparse algebra*, *Structured grid* and *Unstructured grid* dwarfs are examples that show the main value of dataflow, which is its ability to extract all the parallelism inherent in an application. While shared memory implementations rely on artificial barriers to implement a given algorithm from this group of dwarfs, which limits the parallelism and the performance, dataflow implementations support a straightforward implementation by directly mapping dependencies between algorithm operations into a task-based dataflow execution. Specifically, for these four dwarfs dataflow implementations provide on average up to 29.25% better performance than the shared memory versions.

The remaining two benchmarks from the second group, *Graphical models* and *Dynamic programming*, inherently depend on barriers to separate successive steps of the execution. Yet, in Section 4.2.5 we have shown that, using dataflow, it is possible to overlap the execution of tasks from different steps, which results in performance increase compared to shared memory implementations that depend on barriers.

Comparing dataflow implementations, we notice that for both algebra dwarfs OmpSs implementations perform the best of all parallel implementations using both small and large datasets. For other benchmarks from the second group, implementations in the ADF model and the TBB Flow Graph framework alternately achieve the best performance. In two benchmarks, Dynamic programming and Unstructured Grid, OmpSs implementations have poor performance. This is caused by inefficiencies of the Mercurium compiler. In particular, we have measured the execution time of the sequential implementation compiled with Mercurium compiler and with gcc, which revealed that the former version runs almost five times slower than the latter. We suspect that dynamic memory allocation is the cause for this suboptimal performance because the problem gets worse when we increase the problem size. However, this should not be considered as a deficiency of the OmpSs model per se, but rather as the problem in the current implementation of the Mercurium compiler.

Figure 4.45 and Figure 4.46 further show that parallel implementations of DaSH benchmarks perform consistently with different data sets. While a deeper analysis of the DaSH benchmark suite performance with extremely small or extremely large input data sizes is a part of our future work, current implementations show good stability given a significant difference in problem sizes of the small and large datasets used in this evaluation. Further, parallel implementations of the DaSH benchmarks achieve better speedup using the larger data set. In numbers, the aggregate average of maximum speedups of all parallel implementations using the large data set is 10.87, compared to 8.76x aggregate average when the small data set is used.

Finally, when we summarize the performance of all implementations in a specific model and compare it with the same value for OpenMP we find that, for the DaSH benchmarks, dataflow provides up to 23.87% performance improvement for the large data set and up to 27.36% improvement for the small data set.

Chapter 5

QuakeSquad

This chapter studies the advantages of using hybrid dataflow models for the parallelization of a video game engine. Chapter 2 has shown that this application represents a highly complex system that is hard to parallelize efficiently using a traditional shared memory programming model due to extremely complex synchronization requirements of such implementation. Yet, a game engine is the fundamental part of every video game. It abstracts the common game-related tasks, allowing a game developer to focus on the content that makes the game unique. This concept has originated in the domain of the first-person-shooter (FPS) games such as Quake, but has since become the standard in many other game genres. With each new generation of video games the complexity of game engines increases in order to present to a user the most realistic experience possible. This includes increasing the number of objects involved in the game and the level of details of all the aspects of the game, such as physics, graphics, AI and design. With the appearance of Massively Multiplayer Online Games (MMOG) the increase in complexity of game engines has been dramatic. Games like World of Warcraft [9], EVE Online [10] and QuakeLive [14] enable a vast number of users to simultaneously coexist in the game world. Evidently, computational requirements of modern game engines cannot be efficiently satisfied by running a sequential server on a single CPU.

Modern game engines are composed of multiple interacting sub-systems that are responsible for various aspects of the game. While the data and the computational characteristics of various sub-systems are essentially different, they are tightly coupled and mutually dependent. Nevertheless, a reasonable approach to the game engine parallelization is to decompose it into tasks that correspond to different sub-systems and to execute these tasks in separate threads. For example, Intel Parallel Game Engine Framework [Andrews (2009)] is designed to scale to multiple processors by executing different functional blocks in parallel. Figure 5.1 illustrates the idea. This chapter handles the parallelization of the world update part of the game en-

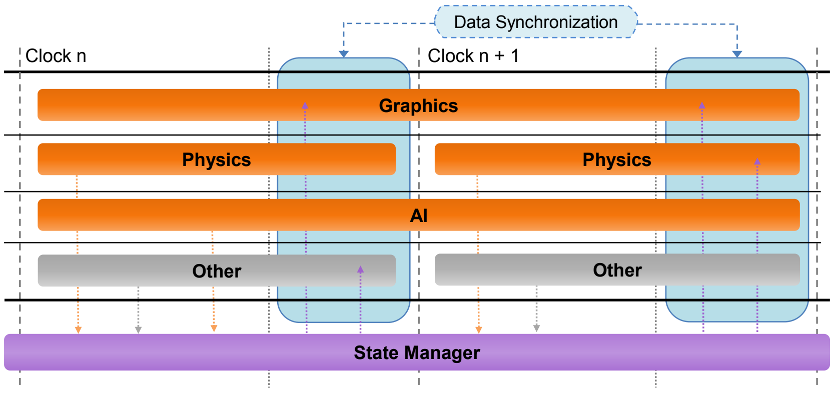


Figure 5.1: An example execution of the game engine implemented using Intel Parallel Game Engine Framework.

gine, which is considered as a core game engine sub-system. Thus, the work presented here is complementary to the sub-system based parallelization methods.

Chapter 2 has introduced the bounding box concept that represents the notion that most of the objects can affect only a small part of the world in their proximity. Application of this concept effectively decreases the complexity of the state update calculations. Furthermore, it can be used as the foundation for the lock-based parallel implementation of the game engine [Abdelkhalek & Bilas (2004)]. Namely, locking the part of the world that completely contains object's bounding box ensures the atomicity of its action. However, this locking scheme introduces a high degree of false sharing, decreases the available parallelism and has negative effects on load balancing [Abdelkhalek & Bilas (2004)]. Chapter 2 of this thesis, as well as more recent research effort [Lupei *et al.* (2010)] have shown that software transactional memory can be used for the inter-thread synchronization instead of locks, but that performance of such implementations is dominated by the extreme overhead of the STM execution and possibly high abort rate.

Overall, efficient parallel implementation of the game engine must satisfy several important conditions. First, minimizing the synchronization between threads is paramount for the performance, because overly restrictive synchronization results in excessive false sharing and limits parallelism. Second, locality-aware tasks assignment decreases the necessary synchronization and improves the cache utilization. Third, efficient parallel implementation has to properly load-balance the work, keeping all threads equally busy. This chapter shows that our novel dataflow approach for game engine parallelization – called the *BigRoom* approach – satisfies all these conditions, resulting in significant speedup compared to the sequential implementation while retaining the same code complexity.

Without the loss of generality, in this work we use the QuakeSquad game engine benchmark [Best *et al.* (2011)] to describe the problems related to the game engine parallelization that apply to many MMOG games, especially first-person shooters, role playing games and real-time strategies. We show that hybrid dataflow models decrease the complexity of the parallel game engine implementation by eliminating or restructuring the explicit synchronization that is necessary in shared memory implementations. The corresponding implementations also exhibit good scalability and better speedup than the shared memory parallel implementations, especially in the case of a highly congested game world that contains a large number of game objects.

5.1 QuakeSquad Description

Due to its complexity, the Quake game engine was substituted as a research application by its derivatives, such as the SynQuake benchmark [Lupei *et al.* (2010)] and the QuakeSquad benchmark [Best *et al.* (2011)]. These benchmarks capture the essential computational patterns and data structures found in commercial video games, such as Quake, while remaining simple enough for meaningful testing. Fundamentally, the only true difference is that the complexity of physical and mathematical equations in the real game is much higher, which could only emphasize the adventages of parallelization.

The QuakeSquad game world is set in a 2-D space that contains the following types of game entities:

- *walls* represent natural barriers in the game world.
- *bombs* explode when their timer reaches zero, reducing the health of human entities in its proximity. Bombs also project fear onto nearby humans who are in their explosion radius.
- *explosion effects* appear at the location of the bomb that has exploded and disappear gradually with time.
- *money* objects are static and their purpose is to attract humans.
- *citizens* are moving constantly through the game world trying to run away from bombs. If there are no bombs in their vicinity, the citizens move in a random direction, avoiding the areas affected by the bombs. During the gameplay the citizens may occasionally form clusters in calm areas of the world.
- *technicians* move towards the closest source of fear in an attempt to disarm the bomb.
- *bandits* move towards the money in an attempt to steal it. They also shoot bullets that may hit other humans and reduce their health. The bullets represent long-range actions, common in various real-world games.

The expired objects (exploded or disarmed bombs or killed humans) are re-spawned at random locations in the world.

Thus, QuakeSquad models many scenarios typically found in modern commercial games: a large number of dynamic entities that move towards the object of interest (quests) or away from danger, often forming clusters (hoarding) and performing various actions that affect other objects in the game world, both in their proximity (acquiring items and other forms of interactions), or in a distance (shooting, teleporting etc.).

5.2 Implementation Details and Synchronization Requirements

The most time-consuming operation involved in the update of the game world is the movement of dynamic objects (human and AI characters). This operation consists of calculating the line-of-site of the object, finding the obstacles in the direction of the move and detecting the collisions with these obstacles. In the sequential implementation of QuakeSquad, the human move calculations constitute more than 80% of the execution time. To optimize these calculations, game engines typically divide the world into a fixed grid. Each grid cell maintains lists of objects that are contained within the corresponding area. When an object crosses the cell boundary during the move, it removes itself from the current list and inserts itself into the list of the destination grid cell. Setting the proper size of grid cells can reduce the calculations of objects actions to only the current cell and its adjacent cells.

In a parallel implementation of the game, when entities are concurrently updated, two types of consistency conflicts can arise: those caused by concurrent operations on object attributes, and those caused by concurrent accesses to grid cell object lists. Therefore, some form of synchronization is needed to preserve the consistency of the game world in a parallel execution. Next, we describe all conflicting situations found in QuakeSquad that would require synchronization in a parallel implementation:

1. **Bomb processing.** In each frame, the game engine traverses the list of bombs associated with each grid cell and processes all bombs from the list. When a bomb explodes, the game engine has to decrease the health of all human entities in the explosion radius. Since the explosion can affect an area that extends the boundaries of the grid cell that contains the bomb, it is necessary to traverse the list of humans not only for the corresponding grid cell, but also for its adjacent cells. For every human entity from each of these lists, the game engine checks if it is affected by the explosion, i.e. if its location is within the explosion radius, and reduces its health accordingly. These updates have to be properly synchronized.

2. **Human move**. In QuakeSquad, all human entities are AI controlled. Thus, the engine traverses the list of humans from each grid cell and executes the move action for each

human from the list. This action consists of two components. First, the engine tries to detect bomb fear and money entice for a given human in order to establish the direction of the move. The source of the fear is determined by acquiring the location of the nearest bomb in the fear radius around the human. This operation requires synchronization because the state of the bomb can be updated by some other thread. The money entice is determined in a similar way.

The second component in the move action is the actual move. Given the destination found in the previous step, the engine checks if the trajectory of the move is obstructed by some other entity. Thus, it has to traverse the lists of all the entities located in the origin grid cell, and possibly the lists of the grid cell that corresponds to the destination of the move. Again, accesses to the bomb and human lists have to be synchronized.

Finally, the engine removes the human entity from the old location and places it in a new location in the world. If it has crossed the boundary between grid cells, the engine removes the entity from the human list of the origin grid cell and inserts it into the list of the destination grid cell, which also requires synchronization.

3. **Human actions**. Some of the human entities perform additional actions beside the move. For example, technicians try to disarm the bomb by decreasing its health, while bandits try to steal the money, also by decreasing its health. Thus, per-entity synchronization is necessary to maintain the consistency of these entities.

4. **Repopulation.** At the end of each frame, the game engine repopulates the expired objects into the game world. For each expired entity, synchronization is required in two places: first, when the engine checks a chosen location and second, when it inserts the entity into the relevant grid cell list.

5. **Long-range actions.** Long-range actions represent actions that cannot be confined within the immediate neighborhood of their origin; rather they can affect an arbitrary space in the game world. In QuakeSquad, long-range actions are represented with fired bullets. Bullet processing requires synchronization only per-entity bases, in case that two bullets, processed by different threads, hit the same target.

Evidently, synchronization requirements of a parallel game engine are quite complex. Such a high degree of interleaving between concurrent operations makes fine-grained lock-based implementation impractical. Previous research efforts [Abdelkhalek & Bilas (2004), Gajinov *et al.* (2009), Zyulkyarov *et al.* (2009)] support this claim, as many months have been invested in this endeavor by the authors, with limited end results. Above all, fine-grained lock-based implementation of a highly complex system, such as a game engine, is prone to well-known problems associated with locks, such as race conditions, deadlocks, livelocks, etc. A more practical solution is to employ coarse-grained locking. For example, the approach proposed in [Best *et al.* (2011)] associates a single lock for each grid cell and processes all objects in a given grid cell by first acquiring the locks of all adjacent cells. Note that this approach requires a programmer to adopt a predetermined order of acquiring the locks to guarantee a deadlock-free execution. Using transactional memory instead of locks in theory provides fine-grained synchronization; however, in practice [Gajinov *et al.* (2009), Zyulkyarov *et al.* (2009), Lupei *et al.* (2010)], such implementations suffer from excessive TM overheads that hinder performance.

5.3 The BigRoom Dataflow Approach

This section shows that it is possible to make most of the synchronization described in the previous section implicit, thus relieving a programmer from the burden of using explicit forms of synchronization. In particular, the idea is based on a fact that most actions from a given grid cell can affect only the objects in that cell and its adjacent cells. Thus, we can almost entirely avoid synchronization if we can guarantee that while a given cell is being processed, no other grid cell that is less than two cells away in any direction is processed simultaneously. Consequently, if we divide the game world into a set of ”super-cells” (e.g. each of 3x3 ordinary cells), we can process each super cell in parallel, working on corresponding ordinary cells in each super-cell independently. Here we consider one possible approach that provides such guarantees, which we refer to as the *BigRoom* approach.

Figure 5.2 illustrates the idea. In addition to the existing world grid, we define a new coarser grid such that each cell of this grid is a 3x3 square that contains nine cells of the original grid. We refer to the cells of this new grid as *BigRooms*. Next, for each *BigRoom* grid cell we create a dedicated *BigRoom* task that processes all nine original cells row-wise, starting from the upper-left cell and moving right – Figure 5.2b. However, after a *BigRoom* task finishes processing a single small cell, it coordinates the execution with its neighbors before it starts processing the next small cell. It is fairly easy to envision the implementation in which all *BigRoom* tasks synchronize after each of the nine steps using the task barrier. Still, we further describe a dependency-based implementation that eliminates the need for this barrier. Notice that each *BigRoom* task can continue with the next step safely if and only if its adjacent tasks to the right and below are at least in the same step of execution – Figure 5.2c. Thus, there is a logical dependency between the *BigRoom* tasks as illustrated in Figure 5.2d. We employ this

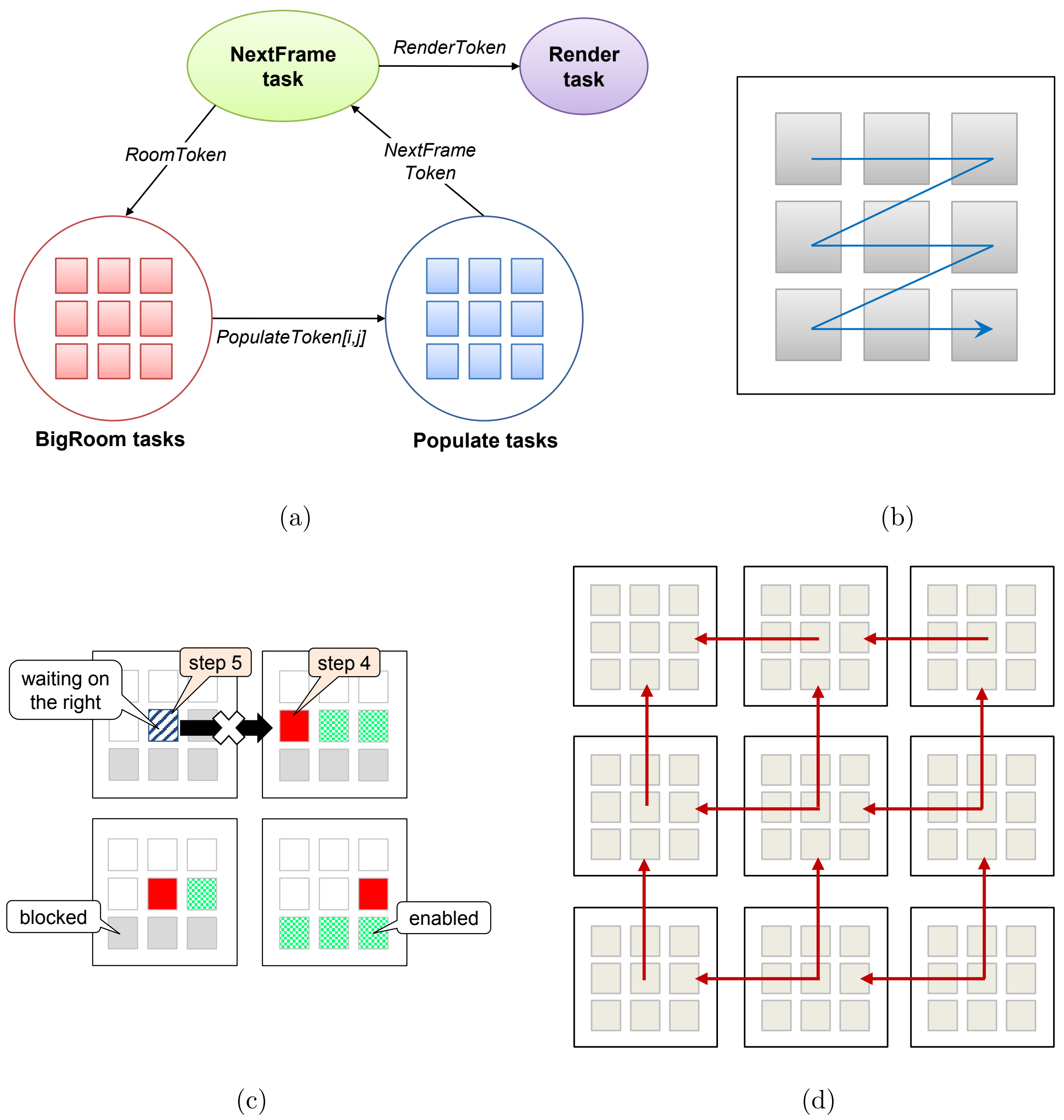


Figure 5.2: The *BigRoom* approach: a) task-dependency graph of the game world update, b) the order of processing of the original grid cells within a single *BigRoom* task, c) an example execution between adjacent *BigRoom* tasks in different steps: white squares denote already processed steps, grey squares denote blocked steps, green squares denote enabled steps, red squares denote currently processed steps and the shaded square represents a *BigRoom* task that has to wait for its adjacent task on the right to finish its current step, and d) dependency between the *BigRoom* tasks.

dependency to organize the execution of the BigRoom tasks in a dependency graph that does not rely on barriers and thus exposes more parallelism.

The execution of each small cell is almost entirely synchronization-free. The *BigRoom* task traverses the lists of bombs and humans associated with the corresponding grid cell and processes each entity from these lists. Further, each *BigRoom* is associated with a dedicated list of expired entities. When the *BigRoom* task detects that a certain entity is expired it randomly choses a *BigRoom* in which the entity will be re-populated and inserts the entity into the corresponding list of expired entites, *PopulateList[i,j]*. Since different threads can attempt to insert entities into the same populate list, these accesses have to be synchronized. However, only a small percentage of objects are repopulated in each frame, which implies that the overhead of this synchronization is negligible. Each *BigRoom* is further associated with dedicated lists for fired bullets and for explosion effects, which are used by the corresponding *Populate* task. Finally, the *BigRoom[i,j]* task produces two types of tokens: *BigRoomToken[i,j]* after each step and *PopulateToken[i,j]* after all nine steps are done.

When all *BigRoom* tasks finish all the steps, threads start executing *Populate* tasks which consume PopulateToken tokens. There is a single *Populate* task for each *BigRoom* in the world. Each *Populate* task, say *Populate[i,j]*, first traverses the corresponding effect list, *EffectList[i,j]*, and places the explosion effects from the list into the world. Next, it processes each bullet from the associated bullet list, *BulletList[i,j]*, by calculating its trajectory and detecting the nearest object that intersects this trajectory, i.e. its target. If the target is a human entity, the task has to decrease its health which requires synchronization. Again, the overhead of this synchronization is negligible because the critical section is quite short (updating a single entity attribute) and the probability that the same entity will be the target of two bullets in the same frame is very small. Next, the task traverses the list of expired entities, *PopulateList[i,j]*, and places the entities from the list into a random location within the *BigRoom[i,j]*. The last *Populate* task that finishes the execution produces the *NextFrameToken* that enables the *NextFrame* task.

The *NextFrame* task separates two successive game frames. It increases the frame counter and measures the execution time of each frame. Next, if the render benchmark parameter is set, the *NextFrame* task copies the state of the game world and generates a *RenderToken* once in every 15 milliseconds. In addition, the *NextFrame* task produces nine *BigRoomFrameToken* tokens that will be consumed by the *BigRoom* tasks in each of the nine steps of the next frame.

Finally, the *Render* task renders the scene and registers the callback function that is called by the OpenGL library to display the scene. Since OpenGL requires a single thread to handle the render context, this task must be bound to a specific thread.

5.3.1 Applicability of the BigRoom approach

The BigRoom approach could readily be applied to many applications which are based on collaborative virtual environments (CVEs) [Benford *et al.* (2001)]. Video games are one domain that portray virtual realities. The findings based on QuakeSquad apply to many other MMOG video games, such as role-playing games (EVE Online [CCP Games (2003)]), first-person shooters (QuakeLive [id Software (2009)] and realtime strategies (World of Warcraft [Blizzard (2004)]). Another domain represents various real-world simulations, such as Virtual Army Experience [Robertson (2009)] and Corps Battle Simulation [Mertens (1993)] systems used by the U.S. Army. Inhabited television [Benford *et al.* (2000)] is another possible domain for the *BigRoom* approach. Inhabited TV enables online audiences to participate in TV shows within shared virtual worlds. Participants are able to navigate the virtual world, interact with its contents and communicate with one another, which are characteristics common to video games. One of the most important requirements of the Inhabited TV is to establish fast-paced social interaction within a CVE and enable a large number of participants to enroll. This is analogous to the requirements of game engines.

Looking from a different perspective, the super-cell division of the game world can be seen as a stencil computation with the Moore neighborhood. Thus, the *BigRoom* approach could be applied for the solution of problems from this category, such as the Game of Life, contour tracing, number-conserving cellular automaton etc. In the previous subsection, we have described the simplest case of 3x3 super-cell size, which corresponds to the Moore neighborhood of size 1. However, other sizes are equally feasible. The only change to the approach when the super-cell size is changed is in the number of steps needed for one iteration, which is equal to the square value of the super-cell size for 2-D environments.

5.4 Parallel Game Engine Implementations

In order to compare the usefulness of hybrid dataflow models and traditional shared memory models for the game engine parallelization, we have implemented a number of versions of the QuakeSquad game engine.

The `omp` and `tbb_loop` versions use parallel loops to partition the cells of the original grid to a number of worker threads. Hence, these are work-sharing implementations and variants of the region-based locking proposed in [Abdelkhalek & Bilas (2004)]. Instead of a complex fine-grained locking scheme, we implement the coarse-grained locking of grid cells, in accordance with the lock-based implementation used in [Best *et al.* (2011)]. Specifically, we associate a dedicated lock for each grid cell and apply a predefined order of locking to avoid deadlocks: each cell task first tries to acquire its own lock, and then the locks of adjacent grid cells, starting from the upper-left neighbor and moving clockwise. If it fails to acquire a certain lock, the cell task releases already acquired locks and repeats the same procedure until it acquires all the locks. Then, it can execute the current operation without the need for any additional synchronization, since it has exclusive access to necessary entities and lists.

The `adf_room` version, realized using Atomic Dataflow Model and software transactional memory, is a straightforward dependency-based implementation, in which game engine tasks are organized into a dependency graph based on the original grid partitioning of the game world. This implementation is oblivious to the task scheduling and relies fully on the explicit synchronization described in Section 5.1. It represents the STM-based approach that corresponds to QuakeTM, Atomic Quake and QuakeSym [Lupei *et al.* (2010)] parallel game engine implementations that use STM.

5.4.1 tbb_BigRoom

Listing 5.1 shows the `tbb_BigRoom` implementation of the *BigRoom* approach developed using the Flow Graph framework of the Intel TBB library [Intel (2007-)]. We construct the flow graph before the start of the first frame – line 65. In each frame we first increment the frame counter and calculate the execution time of the previous frame – line 68. Then we execute the graph – line 70. When the graph finishes the execution, the control returns to the main loop that invokes the *Populate* tasks – line 71. These tasks are spawned using the TBB `parallel_for` template function that partitions the *Populate* tasks to available threads. There is a single *Populate* task for each *BigRoom* in the game world. Finally, if the render benchmark parameter is set, the main loop renders the scene once in 15ms – line 74. This effectively produces the frame rate of 60 frames per second, typically used in commercial games.

The flow graph is implemented using the grid of *Node* objects. Each *Node* object consists of the `multifunction_node` object, which executes the *BigRoom* task – line 4, and the `join_node` object that matches the input tokens for the `multifunction_node` object. Nodes differ in the number of input and output ports depending on the position in the *BigRoom* grid. Thus, in the NxM grid, `Node[n-1, m-1]` has no logical dependencies – line 22, while the nodes from the last row, as well as the nodes from the last column, have a single dependency – line 24. Other nodes have two logical dependencies as explained in the algorithm – line 26. The situation with output ports is reverse. Thus, `Node[0,0]` has no successors, the nodes in the first row as well as nodes in the first column have one successor, while all other nodes have two successors. In addition, all nodes consume `continue_msg` token which assures that the graph is executed exactly nine times before the control is returned to the main loop – line 54. We omit the details of the *Node* class and refer the reader to Intel TBB documentation [Intel (2007-)] for the details of the `multifunction_node`, the `join_node` and the `continue_msg` classes.

```

template <typename Input, typename Output>
void Body<Input, Output>::operator ()(const Input &in, Output &out) {
  Tasks::BigRoomTask(); // process bombs and humans
  std::get<0>(out).try_put(Tasks::token);
  std::get<1>(out).try_put(Tasks::token);
}

void Tasks::CreateGraph() {
  g = new tbb::flow::graph();
  start = new tbb::flow::broadcast_node< tbb::flow::continue_msg >(*g);
  node = new NodeBase**[bigRoomsX];

  // create a grid of BigRoom nodes
  for (int i = 0; i< bigRoomsX; i++)
    node[i] = new NodeBase*[bigRoomsY];

  // create each node
  for (int i = 0; i< bigRoomsX; i++)
    for (int j = 0; j< bigRoomsY; j++) {
      if (i==bigRoomsX-1 && j ==bigRoomsY-1)
        node[i][j] = new Node<0>(*g, i, j);
      else if (i == bigRoomsX-1 || j == bigRoomsY-1)
        node[i][j] = new Node<1>(*g, i, j);
      else
        node[i][j] = new Node<2>(*g, i, j);
    }

  // make edges from the start node and between adjacent BigRoom nodes
  for (int i = 0; i< bigRoomsX; i++)
    for (int j = 0; j< bigRoomsY; j++) {
      if (i==bigRoomsX-1 && j ==bigRoomsY-1) {
        node[i][j]->make_edges(*start, NULL, NULL);
      } else if (i == bigRoomsX-1) {
        node[i][j]->make_edges( *start, NULL,
                                node[i][j+1]->getColOutputPort());
      } else if (j == bigRoomsY-1) {
        node[i][j]->make_edges( *start,
                                node[i+1][j]->getRowOutputPort(), NULL);
      } else {
        node[i][j]->make_edges( *start, node[i+1][j]->getRowOutputPort(),
                                node[i][j+1]->getColOutputPort());
      }
    }
}

void Tasks::RunGraph() {
  for (int i = 0; i< bigRoomsX; i++)
    for (int j = 0; j< bigRoomsY; j++) {
      node[i][j]->try_put(start_token);
    }

  for (int i = 0; i < 9; ++i )
    start->try_put( continue_msg() ); // start is initial graph node

  g->wait_for_all();
}

int main(int argc, char* argv[]) {
  ...
  tbb::task_scheduler_init init( QuakeSquadParams.threads );
  World *qsWorld = World::getNewWorld();

  if ( QuakeSquadParams.render ) InitRender();
  Tasks::CreateGraph();

  while(running) {
    InitFrame(); // increment frame counter and update time

    Tasks::RunGraph();
    Tasks::Populate(qsWorld);

    if (QuakeSquadParams.render && renderTime > renderDeltaTime )
      qsWorld->render();
    if ( frameCount > qsParams.maxFrames )
      running = false;
  }
  ...
}
```

Listing 5.1: Intel TBB flow graph implementation of the BigRoom approach.

```

void Tasks::CreateTasks((World *world) {
  // NextFrame task.
  #pragma adf_task trigger_set(NextFrameToken) until(!running) relaxed {
    InitFrame(); // increment frame counter and update time
    if (qsParams.render)
      if ( renderTime > qsParams.renderDelta ) {
        world->recordState();
        RenderToken = world; // generate RenderToken token
      }

    // generate BigRoomFrameToken tokens
    for (int i = 0; i < 9; i++)
      BigRoomFrameToken = i;
    if ( ++frameCount >= qsParams.maxFrames )
      running = false;
  }

  // Render task
  #pragma adf_task trigger_set(RenderToken) pin(0) relaxed
  RenderToken->render();

  // InitFrame task - produces initial BigRoomFrameToken token
  #pragma adf_task execute_once pin(0) relaxed {
    if ( QuakeSquadParams.render ) InitRender();
    for (int i = 0; i < 9; i++)
      BigRoomFrameToken = true; // generate BigRoomFrameToken tokens
  }

  for (int i = 0; i < bigRoomsX; i++) {
    for (int j = 0; j < bigRoomsY; j++) {
      // InitBigRoom task - produces initial BigRoom tokens
      #pragma adf_task execute(1) relaxed
      BigRoomToken[i][j] = true;

      // BigRoom tasks
      #pragma adf_task trigger_set( BigRoomToken[i+1][j],
                                    BigRoomToken[i][j+1],
                                    BigRoomFrameToken) relaxed {
        /* Process current room */
        Room* room = world->getRoom(BigRoomStep[i][j]);
        room->ProcessRoom(); // process bombs and humans

        /* Update step and check if this iteration is done */
        if ( (++BigRoomStep[i][j]) == 9) {
          BigRoomStep[i][j] = 0;
          SYNCHRONIZE_BEGIN // TM or lock
          if ( (++bigRoomCount) == numBigRooms) {
            bigRoomCount = 0;
            PopulateToken = true; // generate PopulateToken token
          }
          SYNCHRONIZE_END
        }
        BigRoomToken[i][j] = true; // generate BigRoomToken token
      }

      // Populate tasks
      #pragma adf_task trigger_set(PopulateToken) relaxed {
        ProcessEffects(&EffectList[i][j]);
        ProcessBullets(&BulletList[i][j]);
        Populate(&PopulateList[i][j]);
        SYNCHRONIZE_BEGIN // TM or lock
        if ( (++repopulateCount) == numBigRooms) {
          repopulateCount = 0;
          NextFrameToken = true;
        }
        SYNCHRONIZE_END
      }
} } }

int main(int argc, char* argv[]) {
  ...
  adf_init(qsParams.threads);
  World *qsWorld = World::getNewWorld();
  Tasks::CreateTasks(qsWorld);
  #pragma adf_start
  #pragma adf_taskwait
  adf_terminate();
  ...
}
```

Listing 5.2: ADF implementation of the BigRoom approach.

5.4.2 adf_BigRoom

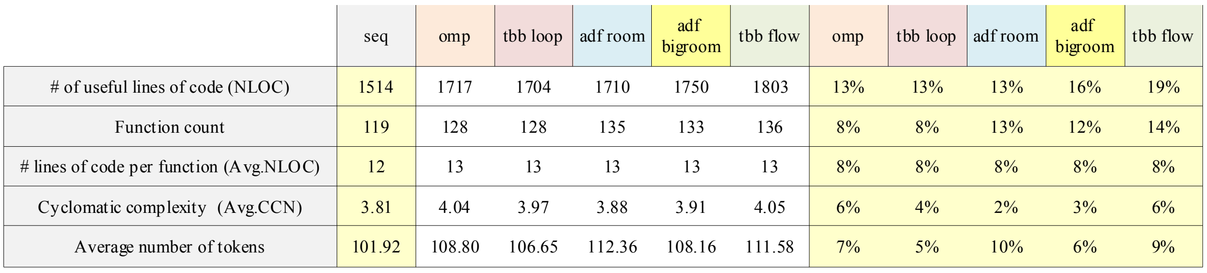
The ADF implementation in Listing 5.2 is a straightforward implementation of the *BigRoom* approach. In the main function we first create all game engine tasks – line 75. Then we start the dataflow execution and wait while the game engine executes a desired number of frames – line 77. Specific to this implementation is the addition of two types of tasks whose purpose is to produce initial tokens needed to start the ADF dataflow execution. One task – line 24 – is needed to produce the initial *BigRoomFrameToken* tokens and a number of tasks – line 33 – equal to the number of *BigRoom* grid cells is needed to produce the initial *BigRoomToken* tokens. In addition, Listing 5.2 shows the simplified code for the *BigRoom* tasks – line 37. The real implementation uses selection to check the position of the *BigRoom* task in the grid and defines the trigger set accordingly. For example, the *BigRoom* task in the lower-right corner of the grid has only the *BigRoomFrameToken* token in its trigger set, according to the *BigRoom* algorithm.

In addition, we have implemented the `adf_BigRoom_lock` version in which the synchronization is based on fine-grain locks instead of STM in order to measure the overhead that the STM introduces in the implementation of the *BigRoom* approach.

5.5 Programmability overview

Given the size of the QuakeSquad benchmark, it is hard to demonstrate the programmability advantages of one approach over the other using fragments of code. However, Listing 5.1 and Listing 5.2 almost entirely captures parallel sections of the `tbb_BigRoom` and the `adf_BigRoom` implementations of the *BigRoom* approach. We omit two details. The first is the definition of the *Node* class from the `tbb_BigRoom` implementation. The second is the synchronization, common to both implementations, that protects the access to lists of expired entities for the *Populate* tasks. The STM implementation of this synchronization is trivial. The lock-based implementation requires a single lock for each of these lists. However, there is no need to enforce locking order since threads never access two lists as a part of a single critical section.

On the other hand, the `tbb_loop` and `omp` implementations utilize `parallel_for` loops to partition the work to threads, which is simple enough. However, it shifts the complexity of parallelization towards synchronization. A programmer is thus faced with two choices: either to employ a coarse-grained locking scheme and inhibit the performance, or to apply fine-grained locking, which may increase the performance, but often requires many months to implement and ensure correct and deadlock free execution [Abdelkhalek & Bilas (2004)]. In general, both



	seq	omp	tbb loop	adf room	adf bigroom	tbb flow	omp	tbb loop	adf room	adf bigroom	tbb flow
# of useful lines of code (NLOC)	1514	1717	1704	1710	1750	1803	13%	13%	13%	16%	19%
Function count	119	128	128	135	133	136	8%	8%	13%	12%	14%
# lines of code per function (Avg.NLOC)	12	13	13	13	13	13	8%	8%	8%	8%	8%
Cyclomatic complexity (Avg.CCN)	3.81	4.04	3.97	3.88	3.91	4.05	6%	4%	2%	3%	6%
Average number of tokens	101.92	108.80	106.65	112.36	108.16	111.58	7%	5%	10%	6%	9%

Figure 5.3: Code complexity metrics of parallel QuakeSquad implementations. Cyclomatic complexity measures the number of linearly independent paths through a source code. Tokens represent identifiers, control structures etc.

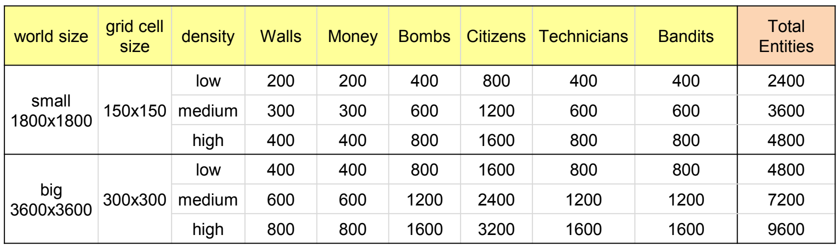
strategies require discipline in acquiring the locks, which was demonstrated by previous research [Abdelkhalek & Bilas (2004),Best *et al.* (2011)] and also in Section 5.2.

Therefore, the advantage in programmability that the implementations based on the *BigRoom* approach provide over the existing work-sharing implementations is primarily the result of decreased synchronization complexity. Specifically, most of the synchronization in the *BigRoom* approach implementations is implicit, while the remaining explicit synchronization is trivial i.e. two locks are never held simultaneously by a single thread. On the contrary, in the work-sharing implementations all synchronization is explicit and threads often have to acquire and hold two or more locks simultaneously, which requires a programmer to reason about the order of acquiring the locks that will guarantee a deadlock-free execution. We find it easier to think about data dependencies between the *BigRoom* dataflow tasks than how to implement and coordinate the locks necessary for the correct shared memory implementation. Finally, Listing 5.1 and Listing 5.2 demonstrate that the *BigRoom* approach can be efficiently implemented in different dataflow models.

Figure 5.3 summarizes various code complexity parameters of the QuakeSquad implementations that we have obtained using the hfcca.py tool [Yin (2012)]. Evidently, the *BigRoom* dataflow implementations only marginally increase the number of useful lines of code and the function count, but in general they retain the same code complexity as the shared memory implementations.

5.6 Performance Evaluation

All QuakeSquad implementations from Section 5.4, were compiled using the gcc compiler version 4.7.2, which supports transactional memory. We have used default GCC-TM runtime configuration in the evaluation of STM based implementations. In addition, we have used Intel TBB library version 4.2 for the TBB-based implementations.



world size	grid cell size	density	Walls	Money	Bombs	Citizens	Technicians	Bandits	Total Entities
small 1800x1800	150x150	low	200	200	400	800	400	400	2400
		medium	300	300	600	1200	600	600	3600
		high	400	400	800	1600	800	800	4800
big 3600x3600	300x300	low	400	400	800	1600	800	800	4800
		medium	600	600	1200	2400	1200	1200	7200
		high	800	800	1600	3200	1600	1600	9600

Figure 5.4: QuakeSquad parameters used in the performance evaluation.

,

The experiments were conducted on a Dell PowerEdge 6850 machine, with four dual-core 64-bit Intel Xeon processors running at 3.2GHz. Each core has 32KB of L1 data cache and shares 2MB of L2 cache with the adjacent core on the same socket. All cores share 16MB L3 cache memory. The machine is running Ubuntu Linux 12.10 operating system.

The experiments were conducted using two map sizes with three different densities of game objects. Figure 5.4 shows the number of objects in different settings. We have measured the time to execute 2000 game frames. We have repeated each experiment ten times and we report the average result.

5.6.1 Results

Figure 5.5 shows the maximum speedup of parallel implementations over the sequential baseline. In addition, Figure 5.6 and Figure 5.6 show the scalability results with each game world configuration.

Evidently, all implementations of the the *BigRoom* algorithm, namely `tbb_BigRoom`, `adf_BigRoom` and `adf_BigRoom_lock`, outperform the lock-based work-sharing versions, `tbb_loop` and `omp`. The difference in performance also increases in favor of these implementations as the density of the objects in the game world increases. This demonstrates the value of the *BigRoom* algorithm since game engines that implement this algorithm could simultaneously support larger number of players in the same game session, thus satisfying one of the most important requirements for the best video game experience. Comparing the best speedup obtained by the `adf_BigRoom` implementation of 4.72x with the best speedup obtained with the `tbb_loop` implementation of 3.16x directly translates to 49% increase in the maximum number of players that a single game server can support. This advantage is a result of improvements in all three aspects that are important for the game engine parallelization: i) decreased synchronization results in decreased thread blocking and/or wasted transactional memory work, ii) concurrency is increased as a direct consequence of this as well as due to a precise coordination between tasks, and iii) cache locality is increased because the *BigRoom* algorithm allows all nine grid cells that belong to a single *BigRoom* to be processed by a particular thread.

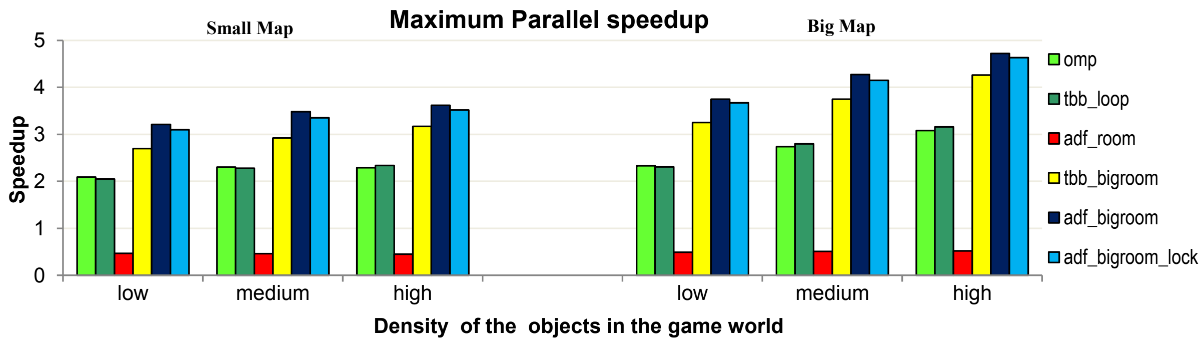


Figure 5.5: The maximum speedup of parallel implementations over the sequential baseline for two map sizes and different densities of the objects in the game world: 1) *omp* – OpenMP implementation, 2) *tbb_loop* – TBB implementation, 3) *adf_Room* – the Room algorithm [dataflow+STM], 4) *tbb_loop* – [TBB parallel_for+locks], 5) *tbb_BigRoom* – the BigRoom algorithm [TBB flow graph+locks], 6) *adf_BigRoom* – the BigRoom algorithm [ADF+STM] and 7) *adf_BigRoom_lock* – the BigRoom algorithm [ADF+locks].

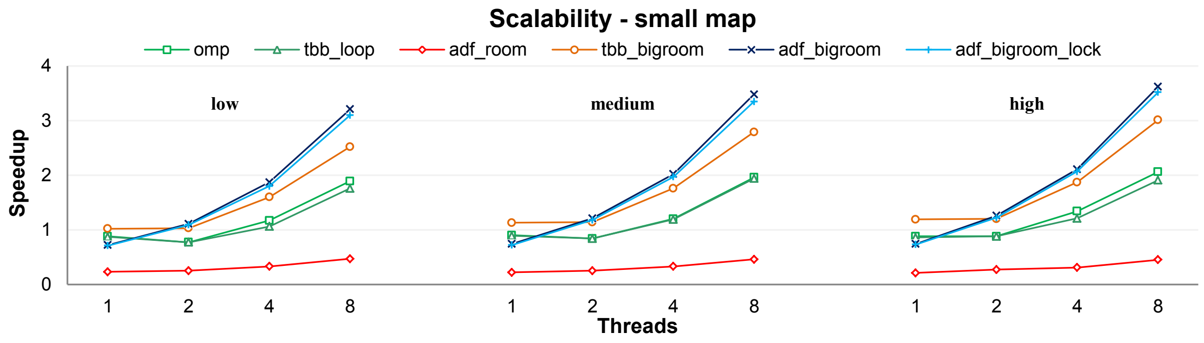


Figure 5.6: The scalability of parallel implementations for the small map configuration.

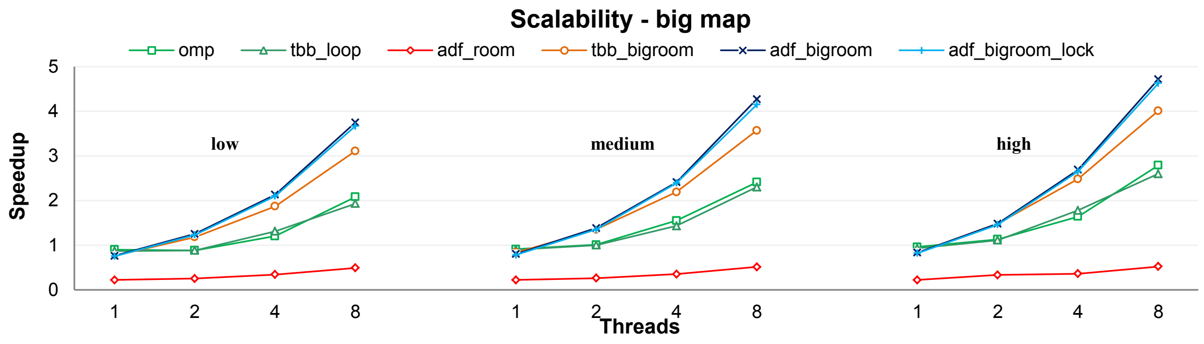


Figure 5.7: The scalability of parallel implementations for the big map configuration.

Comparing different implementations of the *BigRoom* algorithm, we see that the ADF implementations outperform the `tbb_BigRoom` version, especially when the density of objects in the game world increases. Also, both ADF implementations perform almost identically, which means that the STM overhead of explicit synchronization required by this algorithm is negligible. This further supports the value of the *BigRoom* algorithm since TM provides much better programmability than locks, as discussed in Chapter 2.

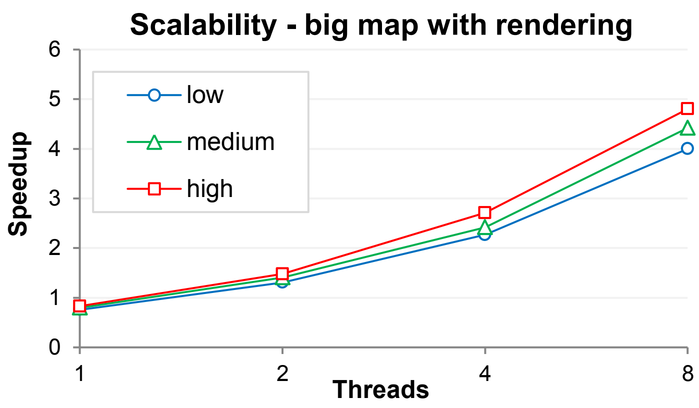


Figure 5.8: The scalability of the adf_BigRoom implementation compared to sequential baseline when the rendering is enabled.

	total in tasks	InitFrame	InitBigRoom	BigRoom	Populate	NextFrame	Render	wasted
sequential	95.58	1.46	0.01	81.94	0.66	2.02	9.49	4.42
adf_BigRoom	82.22	1.15	0.45	70.83	0.84	2.48	6.46	17.78

Figure 5.9: The percentage of time spent in tasks – big map with rendering.

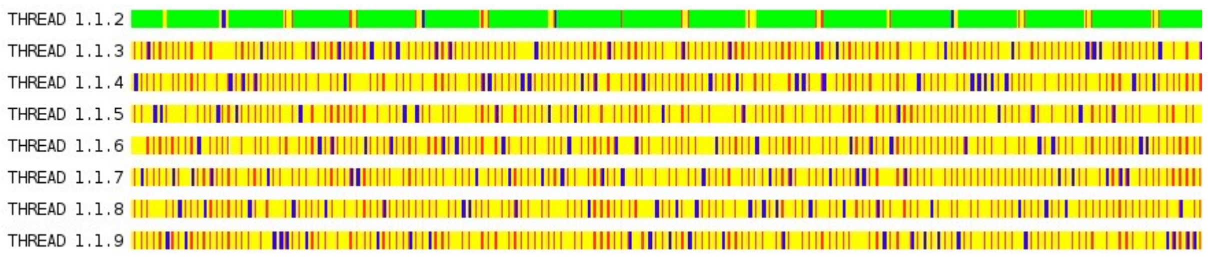


Figure 5.10: Trace segment of the adf_BigRoom execution with eight threads. Most of the execution time of thread 0 (THREAD 1.1.2) consists of running the render task. Colour legend: yellow – *BigRoom* task, red – *Populate* task, blue – *NextFrame* task and green – *Render* task

Finally, the adf_room implementation confirms the previous findings from QuakeTM, Atomic Quake and SynQuake [Lupei *et al.* (2010)] that the STM overhead hinders the performance of the corresponding implementation. Although the adf_room implementation scales and achieves 2.2x speedup on average, this is insufficient to alleviate the 5x slowdown due to the STM overhead of the single-threaded execution. Note that the size of the critical sections in this algorithm is much larger than in the *BigRoom* algorithm.

5.6.2 Results With Rendering

Next, we compare the adf_BigRoom implementation with the sequential baseline when the rendering is enabled. Figure 5.8 shows the results of this comparison. Since we do not parallelize the rendering task, the speedup of this execution is somewhat lower than the speedup obtained in the execution without rendering. Figure 5.10 shows the execution trace of the adf_BigRoom implementation with eight threads. Although the scene is rendered each 15 milliseconds, the *Render* task takes most of the execution time of thread 0 to which this task is bound. This is not surprising given that the *Render* task has to render 9600 game objects (high density). Thus, most of the dataflow execution of the game world update is carried out by the remaining seven threads. Looking at Figure 5.7 and Figure 5.8, we see that the speedup of the `adf_BigRoom` implementation without rendering for seven threads is similar to the speedup obtained with eight threads when the rendering is enabled.

Finally, Figure 5.9 presents the execution breakdown of the sequential implementation and the `adf_BigRoom` implementation running with eight threads with rendering. For the parallel implementation, we sum the time that each thread spends in a given task and divide that value with the total execution time of all threads. The wasted time comprises of the time spent in all operations that are not part of any task. Thus, in a parallel implementation this includes token passing – 1.61%, scheduling – 1.19% and the thread idle time – 14.92% (threads are idle when they are unable to find a ready task in any of the per-thread work-stealing task queues). Further inspection reveals that thread 0 is idle only 4.67% of the time (due to a long execution time of the *Render* task), while for the rest of the threads the idle time varies between 13.41% and 18.36%. This confirms good load-balancing properties of the *BigRoom* algorithm. Moreover, a random fragment of the execution trace in Figure 5.10 further shows that threads are equally busy and spend most of the time executing tasks.

5.7 Related work

This section outlines existing methods for the game engine parallelization.

5.7.1 Distributed game servers

Although the implementation details of the commercial games are vague, it is safe to assume that game servers that drive these games are distributed. Colyseus [Bharambe *et al.* (2006)] is an example from the research community that presents a distributed architecture for interactive multiplayer games as a way to solve the scalability problems of centralized game servers. Colyseus partitions the game state by enforcing the single-copy consistency model in which each game object resides on exactly one node in the system. The owner node provides a serialization order to object updates, while other nodes keep replicas of the objects required for the local computation in order to decrease the latency of update operations. Distributing the game server is an important problem, however orthogonal to the problem of game world update parallelization that is addressed in this chapter.

5.7.2 Inter-modular parallelization

A reasonable approach to the game engine parallelization is to execute the tasks that correspond to different engine sub-systems in separate threads. One of the first attempts in this direction was the proposal for constructing the dependence graph of different tasks involved in the sequential game loop based on Bernsteins Conditions for identifying potentially parallel tasks [El Rhalibi *et al.* (2006)]. Similarly, Intel Parallel Game Engine Framework [Andrews (2009)] is designed to scale to multiple processors by executing different functional blocks in parallel.

The work presented in this chapter is complementary to these efforts, since we are concerned with the parallelization of the world update component of the game engine. Moreover, the dataflow approach that we employ could be easily extended to the entire game engine by connecting dataflow graphs that represent each of the components into a single super-graph. Particularly, component coordination has been already implemented in Cascade [Best *et al.* (2009)], the parallel programming environment targeted specifically at video game engines. In addition, Cascade handles data parallelism in Physics and AI sub-systems by dividing the work into batches executed in parallel as task instances. The performance gain comes from the pipelined execution of producer and consumer tasks, which in a work-sharing implementation would be separated by the implicit barrier, commonly found in parallel-for constructs. However, Cascade does not address the complexities of the world update synchronization, but only the parts of the Physics and AI sub-systems that can be executed independently and without side-effects. In this work, we discuss difficulties involved in the parallelization of the world update component, describe related synchronization requirements and provide a dependency-based algorithm that minimizes the necessary explicit synchronization, and thus improves the programmability. Finally, the authors report the 2x speedup of this approach over the sequential execution on an 8-core machine [Best *et al.* (2009)], whereas our approach results in a much better speedup of up to 4.72x on eight cores.

5.7.3 Parallelization of the game world update

The first attempt to parallelize the game world state update was the work done by Abdelkhalek et al [Abdelkhalek & Bilas (2004)]. Using the Quake game engine, the authors apply region-based locking of the game world to synchronize simultaneous state updates by different threads. However, this locking scheme introduces a high degree of false sharing, limits the available parallelism and has negative effects on load balancing [Abdelkhalek & Bilas (2004)].

In Atomic Quake [Zyulkyarov *et al.* (2009)], we have studied difficulties involved in converting already parallelized, lock-based game engine [Abdelkhalek & Bilas (2004)] into its transactional memory counterpart. In contrast, as described in Chapter 2 of this thesis, in QuakeTM we have used STM to parallelize the Quake game engine starting from the sequential version. Finally, the authors of [Lupei *et al.* (2010)] have further studied the effects of using STM for the parallelization of the SynQuake game benchmark. They have found that STM substantially reduces the problem of false sharing and that locality-aware task assignment policy that allocates the processing of nearby entities in the game to the same thread reduces the true sharing between threads, resulting in increased scalability (the BigRoom approach provides similar policy by grouping nearby grid cells into super-cells). However, the overall performance of these implementations was severely influenced by substantial overheads of the STM execution. Compared to the sequential baseline, parallel implementations based on STM perform worse in all cases.

Chapter 6

Conclusions

This thesis describes Atomic Dataflow model, a new hybrid parallel programming model that integrates dataflow abstractions into the shared memory model. The ADF model provides language constructs that allow a programmer to delineate a program into a set of tasks and to explicitly define data dependencies for each task. The task dependency information is conveyed to the ADF runtime system that constructs a dataflow task graph that governs the execution of a program. Additionally, the ADF model allows tasks to share data. The key idea is that computation is triggered by dataflow between tasks but that, within a task, execution occurs by making atomic updates to common mutable state. To that end, the ADF model employs transactional memory, which guarantees atomicity of shared memory updates.

The ADF model, and similar hybrid dataflow models in general, provide a flexible support for expressing different forms of parallelism on a multicore architecture by exposing the expressiveness of dataflow directly to a programmer. For example, a computation can be carried out in a pure dataflow fashion, in a traditional task-based style of shared memory programming, or in a combination of these two methods. Moreover, such combination can help to overcome some of the deficiencies specific to each of these models. In particular, in a hybrid dataflow model, complex data can be treated as a mutable state and updated in-place, instead of creating a new copy of data on each update that is required in a pure dataflow model. Alternatively, synchronization related problems specific to shared memory programming may be entirely eliminated or hidden from a programmer by reorganizing program tasks according to their data dependencies. As a result, the performance of transactional memory applications can be improved by eliminating unnecessary conflicts, which was demonstrated in this thesis using the parallelization of the game engine as an example.

While each hybrid dataflow model comes with a set of a few applications that prove the concept and show the potential of the model, a more extensive application bundle is necessary in order to fully evaluate and compare these models. In this thesis, I have presented DaSH the first comprehensive benchmark suite for hybrid dataflow models. DaSH covers a wide range of application domains and provides two input data sets and the tool for generating new input data that should facilitate comprehensive evaluation of the benchmarks that comprise the suite. Many aspects of benchmarks behavior and their parallelization can be further controlled using command line parameters. Therefore, DaSH is characterized with three important properties: breadth, depth and portability.

Using DaSH for the evaluation of the ADF, OmpSs and TBB Flow Graph models, we have identified two main benefits that a hybrid dataflow model can offer. First, it can provide a straightforward implementation of certain types of irregular algorithms, which follows only logical dependencies present in the algorithm, and does not restrict the concurrency by inserting unnecessary barriers. And second, we have identified applications for which dataflow provides barrier-free implementation even when the algorithm inherently depends on barriers.

Overall, hybrid dataflow models, and the ADF model in particular, are a promising alternative to traditional shared memory programming models. However, it is important for one such model to provide good support for work-sharing. In addition, we have shown that current API support, provided by the three hybrid dataflow models that we evaluate in this thesis, is not sufficient for an elegant implementation of an application characterized by tasks that can have an arbitrary number of dependencies. We have suggested that an API extension that would enable supplying an arbitrary number of dependencies to a task using C++ vectors could solve this problem. Finally, we have shown that dataflow models based on data-dependency graphs provide more flexibility compared to OmpSs that builds a task dependency graph, but also that OmpSs provides better programmability and performance for algebra problems. In addition, our evaluation has shown the importance of the cache-aware scheduler for the dataflow programming, which is supported in the ADF model with the *pin* clause of the *adf_task* directive.

Finally, we have studied the applicability of hybrid dataflow models and shared memory models for the game engine parallelization. We have described a novel dataflow approach, *BigRoom*, that enables easy and efficient parallelization of core game engine sub-systems. This approach utilizes inter-task dependencies to coordinate the execution of game world update tasks and effectively hide most of the explicit synchronization from a programmer. Moreover, the remaining synchronization can be efficiently handled using software transactional memory. Both factors improve the programmability of the *BigRoom* approach compared to the existing work-sharing parallelization methods. Furthermore, we have shown that the dataflow implementations based on the *BigRoom* approach provide good scalability and excellent speedup compared to the sequential baseline and shared memory implementations of the game engine, especially in the case of a highly congested game. Increased parallel performance is the result of simplified synchronization requirements of the *BigRoom* implementations. This factor, along with the fine-grained dataflow task scheduling that avoids global synchronization, provide better concurrency than the shared memory implementations based on work-sharing. Consequently, the *BigRoom* dataflow approach allows larger number of players to simultaneously take part in a single game session, thus satisfying one of the most important aspects of multiplayer video games.

We plan to continue this research by looking into other real world irregular applications that could benefit from the hybrid dataflow model. In addition, we are interested in a research of optimal scheduling alternatives and language constructs that could improve the expressiveness of these models. Given the recent publication of the latest OpenMP 4.0 standard that introduces task dependency into its tasking model makes DaSH appealing for users who wish to adopt this new paradigm, which is why we have released DaSH as an open-source benchmark suite that should facilitate wider adoption of dataflow for parallel programming on current and future multicore systems based on shared memory architecture.

Chapter 7

Publications on the topic

- Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Mateo Valero, *Nebelung: Execution Environment for Transactional OpenMP*, International Journal of Parallel Programming. Vol 36, number 3 - May 2008.
- Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, Mateo Valero, *Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server*, 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) – February 2009.
- Vladimir Gajinov, Ferad Zyulkyarov, Adrián Cristal, Osman S. Unsal, Eduard Ayguadé, Tim Harris, Mateo Valero, *QuakeTM: Parallelizing a Complex Serial Application Using Transactional Memory*, 23rd ACM/SIGARCH International Conference on Supercomputing (ICS'09) – New York, US, Jun 2009.
- Vladimir Gajinov, Miloš Milovanović, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Mateo Valero, *Integrating Dataflow Abstractions into Transactional Memory*, Systems for Future Multi-Core Architectures (SFMA'11), EuroSys 2011 Workshop – Salzburg, Austria, April 2011.
- Vladimir Gajinov, Srdjan Stipić, Osman S. Unsal, Tim Harris, Eduard Ayguadé, Adrián Cristal, *Supporting Stateful Tasks in a Dataflow Graph*, Poster session, The 21st International Conference on Parallel Architectures and Compilation Techniques (PACT-2012) – Minneapolis, US. September 2012.
- Vladimir Gajinov, Srdjan Stipić, Osman S. Unsal, Tim Harris, Eduard Ayguadé, Adrián Cristal, *Integrating Dataflow Abstractions into the Shared Memory Model*, 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2012) – New York, US, October 2012.

- Vladimir Gajinov, Igor Erić, Srdjan Stipić, Osman S. Unsal, Eduard Ayguadé, Adrián Cristal, *DaSH: A Benchmark for Hybrid Dataflow and Shared Memory Programming Model - with Comparative Evaluation of Three Hybrid Dataflow Models*, In Proceedings of 11th conference on ACM Computing Frontiers – Cagliary, Italy, May 2014.
- Vladimir Gajinov, Igor Erić, Sasa Stojanović, Veljko Milutinović, Osman Unsal, Eduard Ayguadé and Adrián Cristal, *A Case Study of Hybrid Dataflow and Shared-memory Programming Models: Dependency-based Parallel Game Engine.* In Proceedings of 26th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2014) – Paris, France, October 2014.

References

Abdelkhalek, A. & Bilas, A. (2004). Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 72, IEEE. 17, 18, 27, 31, 110, 113, 117, 121, 122, 127, 128

Abdelkhalek, A., Bilas, A. & Moshovos, A. (2003). Behavior and performance of interactive multi-player game servers. *Cluster Computing*, **6**, 355–366. 17

Adl-Tabatabai, A., Lewis, B., Menon, V., Murphy, B., Saha, B. & Shpeisman, T. (2006). Compiler and runtime support for efficient software transactional memory. In *ACM SIGPLAN Notices*, vol. 41, 26–37, ACM. 27

Amer, A., Maruyama, N., Pericàs, M., Taura, K., Yokota, R. & Matsuoka, S. (2013). Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM. In *Proceedings of International Supercomputing Conference*, ISC'13, 255–266, IEEE. 50, 54

Andrews, J. (2009). Designing the framework of a parallel game engine. *Articles on Intel Software Network*. 109, 127

Arvind & Culler, D.E. (1986). Dataflow architectures. *Annual review of computer science*, **1**, 225–253. 42

Arvind & Nikhil, R.S. (1987). Executing a program on the MIT tagged-token dataflow architecture. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, 1–29, Springer-Verlag, London, UK. 10, 11

Arvind, Nikhil, R.S. & Pingali, K. (1987). I-structures: Data structures for parallel computing. In *Proceedings of the Workshop on Graph Reduction*, 336–369, Springer-Verlag, London, UK. 11, 36

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D. & Yelick, K. (2009). A view of the parallel computing landscape. *Communications of the ACM*, **52**, 56–67. 49, 106

Barnes, J. & Hut, P. (1986). A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 446–449. 79

Barth, P.S., Nikhil, R.S. & Arvind (1991). M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, 538–568, Springer-Verlag, London, UK. 37

Bellens, P., Palaniappan, K., Badia, R.M., Seetharaman, G. & Labarta, J. (2011). Parallel implementation of the integral histogram. In *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*, 586–598, Springer. 50

Benford, S., Greenhalgh, C., Craven, M., Walker, G., Regan, T., Morphett, J. & Wyver, J. (2000). Inhabited television: Broadcasting interaction from within collaborative virtual environments. *ACM Transactions on Computer-Human Interaction (TOCHI) - Special issue on human-computer interaction and collaborative virtual environments*, **7**, 510–547. 117

Benford, S., Greenhalgh, C., Rodden, T. & Pycock, J. (2001). Collaborative virtual environments. *Communications of the ACM*, **44**, 79–85. 116

Best, M., Fedorova, A., Dickie, R., Tagliasacchi, A., Couture-Beil, A., Mustard, C., Mottishaw, S., Brown, A., Huang, Z., Xu, X. *et al.* (2009). Searching for concurrent design patterns in video games. *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*, 912–923. 127

Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A. & Brownsword, A. (2011). Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, 640–652, ACM, New York, NY, USA. 110, 111, 114, 117, 122

Bharambe, A., Pang, J. & Seshan, S. (2006). Colyseus: a distributed architecture for online multiplayer games. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation*, vol. 3, 12–12. 126

Blizzard (2004). World of warcraft. http://us.blizzard.com/en-us/games/wow/, accessed February 2014. 117

Bohm, W., Najjar, W., Shankar, B. & Roh, L. (1993). An evaluation of coarse grain dataflow code generation strategies. In *Proceedings of Conference on Programming Models for Massively Parallel Computers*, 63–71, IEEE. 13

BSC (2004-). Mercurium compiler. https://pm.bsc.es/projects/mcxx, accessed: February 2014. 54

Carrier, J., Greengard, L. & Rokhlin, V. (1988). A fast adaptive multipole algorithm for particle simulations. *SIAM Journal on Scientific and Statistical Computing*, **9**, 669–686. 79

CCP Games (2003). Eve online. http://www.eveonline.com/, accessed: February 2014. 117

Chase, D. & Lev, Y. (2005). Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 21–28, ACM. 43

Cooley, J.W. & Tukey, J.W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, **19**, 297–301. 88

Culler, D., Sah, A., Schauser, K., Von Eicken, T. & Wawrzynek, J. (1991). *Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine*, vol. 19. ACM. 13

Culler, D.E., Goldstein, S.C., Schauser, K.E. & von Eicken, T. (1993). TAM – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, **18**, 347–370. 14

Dean, J. & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, 10–10, USENIX Association, Berkeley, CA, USA. 76

Dennis, J.B. & Misunas, D.P. (1974). A preliminary architecture for a basic data-flow processor. *SIGARCH Computer Architecture News*, **3**, 126–132. 9, 10

Donnelly, K. & Fluet, M. (2006). Transactional events. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, 124–135, ACM, New York, NY, USA. 37

Dooley, I., Mei, C., Lifflander, J. & Kale, L.V. (2010). A study of memory-aware scheduling in message driven parallel programs. In *Proceedings of International Conference on High Performance Computing (HiPC 2010)*, 1–10, IEEE. 53

Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X. & Planas, J. (2011). OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, **21**, 173–193. 49, 51

El Rhalibi, A., Merabti, M. & Shen, Y. (2006). Improving game processing in multithreading and multiprocessor architecture. *Technologies for E-Learning and Digital Entertainment*, 669–679. 127

Foina, A.G., Planas, J., Badia, R.M. & Ramirez-Fernandez, F.J. (2011). P-means, a parallel clustering algorithm for a heterogeneous multi-processor environment. In *Proceedings of International Conference on High Performance Computing and Simulation (HPCS)*, 239–248, IEEE. 50

Fuchs, H., Abram, G. & Grant, E. (1983). Near real-time shaded display of rigid objects. In *ACM SIGGRAPH Computer Graphics*, vol. 17, 65–72, ACM. 17

Gajinov, V., Zyulkyarov, F., Unsal, O.S., Cristal, A., Ayguadé, E., Harris, T. & Valero, M. (2009). QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international conference on Supercomputing*, 126–135, ACM. 113, 114

Gilmore, P.C. & Gomory, R.E. (1961). A linear programming approach to the cutting-stock problem. *Operations research*, **9**, 849–859. 64

Goodman, D., Khan, S., Seaton, C., Guskov, Y., Khan, B., Luján, M. & Watson, I. (2012). DFScala: High level dataflow support for Scala. In *Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM), 2012*, 18–26, IEEE. 53

Gurd, J.R., Kirkham, C.C. & Watson, I. (1985). The Manchester prototype dataflow computer. *Communications of the ACM - Special section on computer architecture*, **28**, 34–52. 11

Harris, L.J., T. & Rajwar, R. (2010). *Transactional Memory (Second Edition)*. Morgan & Claypool Publishers. 5

Harris, T., Marlow, S. & Peyton-Jones, S. (2005). Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, 49–61, ACM. 37

Herlihy, M. & Moss, J.E.B. (1993). Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, 289–300, ACM, New York, NY, USA. 5, 7

Iannucci, R.A. (1988). Toward a dataflow/von Neumann hybrid architecture. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, 131–140, IEEE Computer Society Press, Los Alamitos, CA, USA. 13, 41

id Software (2009). Quakelive. `http://www.quakelive.com`, accessed: February 2014. 117

Intel (2007-). Intel threading building blocks. `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm`, accessed: January 2014. 50, 51, 118

Jagannathan, R. (1995). Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, 113–129, IEEE Computer Society Press. 13

Johnston, W.M., Hanna, J.R.P. & Millar, R.J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys*, **36**, 1–34. 4

Kale, L.V. & Krishnan, S. (1993). Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, 91–108, ACM, New York, NY, USA. 53

Karypis, G. & Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, **20**, 359–392. 98

Kavi, K.M., Hurson, A.R., Patadia, P., Abraham, E. & Shanmugam, P. (1995). Design of cache memories for multi-threaded dataflow architecture. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, 253–264, ACM, New York, NY, USA. 44

Knight, T. (1986). An architecture for mostly functional languages. *Proceedings of the 1986 ACM conference on LISP and functional programming*, 105–112. 7

Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K. & Chew, L.P. (2007). Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 211–222, ACM, New York, NY, USA. 38

Kurzak, J., Ltaief, H., Dongarra, J. & Badia, R.M. (2010). Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, **22**, 15–44. 50

Lesani, M. & Palsberg, J. (2011). Communicating memory transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 157–168, ACM, New York, NY, USA. 37

Luchangco, V. & Marathe, V.J. (2011). Transaction communicators: Enabling cooperation among concurrent transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 169–178, ACM, New York, NY, USA. 37

Lupei, D., Simion, B., Pinto, D., Misler, M., Burcea, M., Krick, W. & Amza, C. (2010). Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European conference on Computer systems - EUROSYS*, 41–54, ACM. 110, 111, 114, 118, 125, 128

McGuire, M. (2000). Quake 2 bsp file format. `http://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml`, accessed 2009. 17

Mertens, S. (1993). The corps battle simulation for military training. In *Proceedings of the 25th Conference on Winter Simulation*, WSC '93, 1053–1056, ACM, New York, NY, USA. 117

Microsoft (2007-). Tpl dataflow library. `http://msdn.microsoft.com/en-us/library/hh228603.aspx`, accessed: January 2014. 53

Montanuy, O. (1996). Unofficial quake-c specification. `http://www.gamers.org/dEngine/quake/spec/quake-spec34/index1.htm`, accessed 2009. 20

Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A. & Tian, X. (2008). Design and implementation of transactional constructs for c/c++. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, 195–212, ACM, New York, NY, USA. 27

Nikhil, R.S. (1989). Can dataflow subsume von neumann computing? In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, 262–272, ACM, New York, NY, USA. 13, 14

Nikhil, R.S., Papadopoulos, G.M. & Arvind (1992). T: a multithreaded massively parallel architecture. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, 156–167, ACM, New York, NY, USA. 14

Nikolopoulos, D., Ayguade, E. & Polychronopoulos, C. (2001). Scaling irregular parallel codes with minimal programming effort. In *ACM/IEEE Conference on Supercomputing*, 5–5, IEEE. 38

Orozco, D., Garcia, E., Pavel, R., Khan, R. & Gao, G. (2011). Tideflow: The time iterated dependency flow execution model. In *First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, 1–9, IEEE. 53

Papadopoulos, G.M. & Culler, D.E. (1990). Monsoon: an explicit token-store architecture. *SIGARCH Computer Architecture News*, **18**, 82–91. 10, 12, 13

Papadopoulos, G.M. & Traub, K.R. (1991). Multithreading: a revisionist view of dataflow architectures. *SIGARCH Computer Architecture News*, **19**, 342–351. 13

Perez, J.M., Badia, R.M. & Labarta, J. (2008). A dependency-aware task-based programming environment for multi-core architectures. In *IEEE International Conference on Cluster Computing*, 142–151, IEEE. 49

Peyton Jones, S., Gordon, A. & Finne, S. (1996). Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, 295–308, ACM, New York, NY, USA. 37

Ramadan, H.E., Rossbach, C.J. & Witchel, E. (2008). Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, 246–257, IEEE Computer Society, Washington, DC, USA. 37

Robertson, A. (2009). The army rolls through Indianapolis: Fieldwork at the virtual army experience. *Transformative Works and Cultures*. 117

Seaton, C., Goodman, D., Lujan, M. & Watson, I. (2012). Applying dataflow and transactions to lee routing. In *Workshop on Programmability Issues for Heterogeneous Multicores*. 50, 54

Shavit, N. & Touitou, D. (1995). Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, 204–213, ACM, New York, NY, USA. 5, 7

Silc, J., Robic, B. & Ungerer, T. (1998). Asynchrony in parallel computing: From dataflow to multithreading. *Journal of Parallel and Distributed Computing Practices*, **1**, 1–33. 4, 41

Stanford, C. (2013). 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`, accessed: February 2014. 98

Stavrou, K., Kyriacou, C., Evripidou, P. & Trancoso, P. (2007). Chip multiprocessor based on data-driven multithreading model. *International Journal of High Performance Systems Architecture*, **1**, 34–43. 53

Tseng, H.W. & Tullsen, D.M. (2012). Software data-triggered threads. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, vol. 47, 703–716, ACM. 53

Wang, C., Chen, W., Wu, Y., Saha, B. & Adl-Tabatabai, A. (2007). Code generation and optimization for transactional memory constructs in an unmanaged language. *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07), San Jose, CA, USA*. 28

Yin, T. (2012). hfcca.py tool. `https://github.com/terryyin/hfcca`, accessed: January 2014. 104, 122

Yoo, R., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A. & Lee, H. (2008). Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 265–274, ACM. 26

Zyulkyarov, F., Gajinov, V., Unsal, O.S., Cristal, A., Ayguadé, E., Harris, T. & Valero, M. (2009). Atomic quake: using transactional memory in an interactive multiplayer game server. In *ACM Sigplan Notices*, vol. 44, 25–34, ACM. 19, 31, 113, 114, 128

Zyulkyarov, F., Harris, T., Unsal, O.S., Cristal, A. & Valero, M. (2010). Debugging programs that use atomic blocks and transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, 57–66, ACM, New York, NY, USA. 27