© 2015 Sweta Yamini Pothukuchi

A COMPARATIVE STUDY OF SHARED MEMORY PARALLELISM ON REGULAR AND IRREGULAR DATA STRUCTURES USING OPENMP AND GALOIS

BY

SWETA YAMINI POTHUKUCHI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor David Padua

ABSTRACT

Task and/or data parallelism can be exploited in most applications on today's multi-core processors. However, inefficient data organization, data dependencies, and hardware constraints limit scalable parallelization of these applications. In this thesis, performance and the impact of some optimizations is compared and evaluated for simple tasks using two parallel frameworks, OpenMP and Galois. Additionally, their performance on three real life applications, High Accuracy Relativistic Magnetohydrodynamics (HARM) which operates on a grid data structure; Delaunay Triangulation, which refines a triangulated mesh; and Dynamic Fracture Propagation, which operates on a triangulated mesh with adaptive refinement; is evaluated. It is found that OpenMP is a simple yet powerful tool for parallelization of most regular applications and workloads. Galois, which is specially designed for irregular data patterns, performs well for graph like structures. However, neither of them are well suited for all tasks and other frameworks must be explored to find one that is simple to use and yet powerful for all possible applications.

To my husband, for his love and support.

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my adviser Prof. David Padua for his continuous support, patience and motivation. I could not have completed this thesis without his guidance and encouragement. I would also like to express my thanks to my group mates Adam, Saeed, Carl and Chih-chieh for their ready help and cheerful discussions. I would also like to thank other graduate students in the department and at UIUC for the insightful discussions and the fun time we spent together.

I am forever grateful to my dear husband who has provided immeasurable love and support through all this. Nothing I say in words can make up for all the trouble he bore with extreme patience. He took me by the hand and brought me to school every single day.

I would also like to mention my special friends Manisha and Vijetha who have been with me for a really long time, always reaching out with their love, support, criticism and advice.

It would be foolish of me to try to put into words what I feel towards my family, who have been with me for as long as I can remember.

I am indebted to all my teachers in school, undergrad and at UIUC who have moulded and shaped me to who I am today. Last but definitely not the least, I would like to express my gratitude to the almighty power that turns the wheel of time, the eternal and the complete one that drives everything and everyone, who is the only one responsible for me completing this thesis.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 3
- 2.1 Parallel frameworks 3
- 2.2 Applications 4

CHAPTER 3 PARALLELIZATION STRATEGIES USING OPENMP AND GALOIS 6
- 3.1 Memory allocation and usage 7
- 3.2 Synchronization 9
- 3.3 Scheduling 9
- 3.4 Successive parallel loops 11

CHAPTER 4 EXPERIMENTAL METHODOLOGY 13

CHAPTER 5 PERFORMANCE EVALUATION FOR REGULAR DATA STRUCTURES 14
- 5.1 OpenMP 14
- 5.2 Galois 17
- 5.3 Comparison of OpenMP and Galois 18

CHAPTER 6 PERFORMANCE EVALUATION FOR IRREGULAR DATA STRUCTURES 21
- 6.1 OpenMP 22
- 6.2 Galois 25
- 6.3 Comparison of OpenMP and Galois 27

CHAPTER 7 PERFORMANCE EVALUATION ON REAL WORLD APPLICATIONS 29
- 7.1 High Accuracy Relativistic Magnetohydrodynamics (HARM) . 29
- 7.2 Delaunay Triangulation 32
- 7.3 Dynamic Fracture Simulation 34

CHAPTER 8 COMPARISON OF PROGRAMMING WITH OPENMP AND GALOIS 45
8.1 Parallelizing existing serial code 45
8.2 Writing parallel code from scratch 46
8.3 Qualitative comparison of OpenMP and Galois 46
8.4 Quantitative comparison of OpenMP and Galois 47

CHAPTER 9 CONCLUSION 49

REFERENCES 51

LIST OF TABLES

3.1 Set of optimizations for OpenMP 6
3.2 Set of optimizations for Galois 7

6.1 Test cases for Jacobi iterations in a mesh 21

7.1 Test cases for dynamic fracture simulation 37

8.1 Programming with OpenMP and Galois - a qualitative comparison 46
8.2 Programming with OpenMP and Galois - a quantitative comparison 48

LIST OF FIGURES

3.1 *bsc*, *mrg* and *exp* versions of parallel loops using OpenMP . . 11

5.1 Execution time of Jacobi for 1000 iterations using OpenMP . . 15
5.2 Speedup of Jacobi for 1000 iterations using OpenMP 16
5.3 Comparison of speedup across grid sizes using OpenMP 16
5.4 Comparison of speedup across iterations using OpenMP 17
5.5 Execution time of Jacobi for 1000 iterations using Galois . . . 18
5.6 Speedup of Jacobi for 1000 iterations using Galois 19
5.7 Comparison of speedup across grid sizes using Galois 19
5.8 Comparison of speedup across iterations using Galois 19
5.9 Comparison of execution time using OpenMP and Galois . . . 20

6.1 Execution time of mesh Jacobi using OpenMP 22
6.2 Speedup of mesh Jacobi using OpenMP 23
6.3 Speedup of mesh Jacobi using OpenMP 24
6.4 Evaluation of synchronization methods using OpenMP 24
6.5 Comparison of *push* and *pull* for test cases B and D using OpenMP 25
6.6 Execution time of mesh Jacobi using Galois 26
6.7 Speedup of mesh Jacobi using Galois 27
6.8 Speedup of mesh Jacobi using Galois 28
6.9 Comparison of *push* and *pull* for test cases B and D using Galois 28
6.10 Comparison of OpenMP and Galois for mesh Jacobi 28

7.1 Performance of HARM using OpenMP 31
7.2 Performance of HARM using Galois 32
7.3 Performance of HARM using OpenMP and Galois for 1024X1024 size grid on I2PC machine 32
7.4 Delaunay triangulation on Taub for 10 million points 34
7.5 Delaunay triangulation on I2PC for 10 million points 34
7.6 Initial and final triangulated meshes for the test cases 37
7.7 Stress and velocity progress at different times for the medium test case 40

7.8 Stress and velocity progress at different times for the large test case 41
7.9 Performance of Dynamic Fracture Propagation using OpenMP on medium test case 42
7.10 Performance of Dynamic Fracture Propagation using OpenMP on medium test case 42
7.11 Performance of Dynamic Fracture Propagation using Galois on medium test case 42
7.12 Allocation of nodes and triangles for 16 threads with parallel allocation 43
7.13 Allocation of nodes and triangles for 16 threads with space-filling sorted allocation 43
7.14 Performance of Dynamic Fracture Propagation using Galois . . 44
7.15 Performance of Dynamic Fracture Propagation using Galois and OpenMP 44

CHAPTER 1

INTRODUCTION

Multi-core processors offer shared memory parallelism where multiple threads operate independently on a global address space and share memory resources. This global view facilitates data sharing among the threads in an easy and efficient manner for communication. However, this makes the programmer responsible to ensure synchronization and correct access to memory. Various constructs such as locks, semaphores, barriers, etc are used for synchronization and memory access control. The major shared memory parallel frameworks are POSIX Threads (Pthreads) [1], OpenMP [2], Intel®Cilk™[3], Intel®Thread Building Blocks (TBB) [4], Galois [5], etc. This thesis is a study of parallelization using OpenMP and Galois.

OpenMP is a compiler directives driven Application Programming Interface (API) which provides an easy to use approach to parallelize applications. The main focus of OpenMP is on loops as the bulk of the computation in many programs is concentrated in loops. Galois is a framework to exploit amorphous data-parallelism in irregular programs. It provides a set of extensions and classes which extract parallelism from applications speculatively. In this thesis, an evaluation of performance using OpenMP and Galois for parallelization of applications with different data structures and algorithm patterns is presented. To this end, a few optimizations and best practices for both OpenMP and Galois are listed and their impact on performance for various tasks is evaluated. Using this knowledge, a few real applications with different data structures and algorithms are parallelized and evaluated.

Optimizations in memory allocation and utilization, synchronization and scheduling for both OpenMP and Galois are analyzed to determine the trade-offs between performance benefits and ease of use. For this analysis, emphasis is laid on regular and irregular data structures separately. Simple Jacobi iterations are used on regular 2-dimensional arrays and irregular 2-dimensional triangulated meshes.

Three real world applications are used to compare the performance of OpenMP and Galois. The first application is High Accuracy Relativistic Magnetohydrodynamics (HARM), which solves hyperbolic partial differential equations in conservative form using high-resolution shock capturing techniques [6]. This application solves the relativistic magnetohydrodynamic equations of motion on a stationary black hole. It performs multiple independent iterations over arrays to compute primitive variables and flux.

The second application is Delaunay Triangulation in two dimensions, which takes a set of points on a plane and incrementally builds a delaunay triangulation of the points. This application solely performs mesh refinement and has a single loop in the algorithm.

The third application is Dynamic Fracture Simulation, which simulates propagation of a fracture in a 2-dimensional material under constant stress. It uses finite element analysis on a triangulated mesh with adaptive refinement. This application has two distinct phases, an iterative computation phase and a mesh refinement phase.

This thesis is organized as follows: Chapter 2 presents the background; Chapter 3 describes the parallelization and optimizations for OpenMP and Galois that are considered in this thesis, Chapter 4 describes the experimental methodology, Chapters 5 and 6 evaluate the impact of optimizations on regular and irregular data, Chapter 7 evaluates the three applications and Chapter 8 compares the advantages and disadvantages of OpenMP and Galois.

CHAPTER 2

BACKGROUND

2.1 Parallel frameworks

In this thesis, parallelization of applications using two parallel frameworks OpenMP and Galois is evaluated. A brief discussion on OpenMP and Galois is presented in the following sections.

2.1.1 OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on most processor architectures and operating systems. It uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications.

OpenMP uses a set of compiler directives and library routines for parallelism [7]. It considers loops as primary parallel operations and provides easy to use API for parallelizing loops with fixed number of iterations. The framework provides a set of routines like locks, barriers, reductions, etc to enable parallelism with data dependencies.

OpenMP has minimal system requirements and most C/C++ compilers provide support for OpenMP.

2.1.2 Galois

Galois is a framework of extensions and classes which provides ready-to-use abstractions and data-structures to enable parallel computation. Galois builds its interface on Pthreads and speculatively extracts amorphous data-parallelism.

Galois is an object-oriented system where shared objects can be accessed with relative ease. Its runtime handles scheduling of the iterations and actual memory access locks using simple interfaces. This framework is particularly designed for workloads which do not have a fixed iteration space or those which operate on irregular data-structures like graphs.

Galois depends on operator formulation and Tao-analysis [8] of programs to provide opportunities to exploit parallelism by compile-time, inspector-executor, or optimistic parallelization. The main features of Galois are unordered set iterators, graph and other irregular data structures, parallel for loops and accumulators. One major advantage of Galois over OpenMP is the possibility to add additional work to the current worklist.

Galois requires Boost libraries for execution and cmake for compilation, apart from a C++ compiler.

2.2 Applications

Three applications have been used to evaluate the parallel performance. This section provides a brief introduction to these applications.

2.2.1 High Accuracy Relativistic Magnetohydrodynamics (HARM)

HARM is a conservative finite volume approach to solve hyperbolic partial differential equations (PDE). This application was developed by Grammie et al. [6] and improved by Noble et al. [9] to solve General Relativistic MagnetoHydroDynamics (GRMHD). The original application is written in C for sequential execution and is hosted by the authors [10].

2.2.2 Delaunay Triangulation

Delaunay triangulations are among the most important constructs in two dimensional computational geometry. Delaunay triangulation for a set of points is a triangulation such that no other point lies in the circumcircle of any triangle. Delaunay triangulation is implemented in OpenMP/Cilk as a

part of Problems Based Benchmarks Suite (PBBS) [11], and in Galois as a sample benchmark application in the LoneStar benchmark suite [12].

2.2.3 Dynamic Fracture Simulation

Dynamic Fracture Simulation simulates the development of damage of a material under constant stress using finite element analysis on a finely triangulated mesh with adaptive refinement. This application is based on the finite element analysis presented by Mangala et al. [13].

CHAPTER 3

PARALLELIZATION STRATEGIES USING OPENMP AND GALOIS

Naïve use of parallel frameworks does not always provide the expected performance. While the specific reason for lack of performance is application-specific, they could be generally attributed to high parallelization overhead, inefficient memory layout, load imbalance, false sharing or high synchronization overheads. There are a number of ways to handle each of these, and the right choices are again application-specific.

A set of optimizations which might benefit performance of applications are presented in the following sections. The optimizations that are evaluated in this thesis are listed in tables 3.1 and 3.2. The optimizations are categorized based on the problems they address. The table also has a brief description which will be elaborated in the following sections and a code which will be used to address these optimizations in the following chapters.

Version of an application with basic parallelization of loops without any additional optimizations is denote by the code *bsc*.

Table 3.1: Set of optimizations for OpenMP

Category	**Name**	**Description**	**Target Data**	**Code**
Basic	Basic	Basic parallelization without optimizations	All	*bsc*
Memory allocation and usage	First Touch	Initialization is done in parallel	All	*ftp*
	Space Filling	Data is sorted and then initialized in parallel	Irregular	*sfp*
Synchronization	Critical	Using critical sections	All	*crt*
	Atomic	Using atomic directives	All	*atm*
	Locks	Using OpenMP locks	All	*lck*
Successive loops	Merge	Using single parallel region and OpenMP for directives	All	*mrg*
	Explicit	Using single parallel region and explicit task distribution	All	*exp*

Table 3.2: Set of optimizations for Galois

Category	Name	Description	Target Data	Code
Basic	Basic	Basic parallelization without optimizations	All	*bsc*
Memory allocation and usage	Large Array Local	Using `LargeArray` with local allocation	Regular	*la-loc*
	Large Array Interleaved	Using `LargeArray` with interleaved allocation	Regular	*la-int*
	Parallel Allocation	Using parallel allocation	Irregular	*ftp*
	Space Filling	Using space filling parallel allocation	Irregular	*sfp*

3.1 Memory allocation and usage

Memory accesses are slow and this is specially serious in a multi-core parallel setting as memory channels are shared among several cores. Memory becomes a bottleneck for several applications and adding more cores cannot alleviate performance in these cases. Also, most modern systems have Non-Uniform Memory Access (NUMA) architectures and memory access time depends on the memory address accessed [14]. In these cases, allocating memory close to the thread using it provides considerable benefit. OpenMP and Galois allow this allocation.

Memory allocation policy adopted by the operating system plays an important role in data allocation. Actual memory is only allocated when data is first used and not when a `malloc` function is called [14]. The memory allocated is either local to the thread which used the data first, or interleaved across all possible nodes in a round-robin fashion, depending on the system's memory allocation policy.

Also, cache locality plays an important role in memory performance. Loops must be structured and parallelized to take this into consideration.

For regular data structures, this translates to parallelizing the loop that initializes the data and structuring loops traversals in a cache-aware manner. Tiling is one of the methods that improves cache locality [15]. For irregular data structures, the access pattern would also be irregular making it difficult to split the data across the threads. Given some information about the underlying data structure, sorting the data using a space-filling curve [16] in

a temporary location and then allocating the actual data is beneficial.

3.1.1 OpenMP

OpenMP does not handle memory allocation explicitly. However, by using a parallel directive in the initialization phase, different threads access different sections of the data which can affect the allocation of memory in a NUMA node which obeys first touch policy. `libnuma` provides an interface to handle memory allocation policies which can be used by programs to customize memory allocation.

Version of an application that performs initialization of dynamically allocated data in parallel is annotated by the code *ftp*. For irregular data, if the data is sorted using a space-filling curve and assigned in parallel, it is annotated by the code *sfp*.

3.1.2 Galois

Galois uses `libnuma` to allocate memory for its thread pool. Most Galois structures allocate memory using the first touch policy. Galois programs benefit by initialization in parallel, similar to OpenMP. Galois provides convenient wrappers to allocate memory for regular structures such as arrays using their data structure `Galois::LargeArray`. Irregular data structures like graphs allocate vertices and edges locally to the thread that creates and inserts them.

For regular data, a Galois application using `Galois::LargeArray` with local first touch allocation policy and parallel initialization of data is annotated by *la-loc* and interleaved allocation policy is annotated *la-int*.

For irregular data using Galois data structures, parallel allocation of data is annotated *ftp* and parallel allocation of data after it is sorted using a space-filling curve is annotated *sfp*.

3.2 Synchronization

3.2.1 OpenMP

OpenMP provides directives for barriers, critical sections and atomic operations and library routines for locks. Additionally, compilers provide a set of built-in functions for atomic memory access which can be used directly for synchronization. A choice needs to be made among these depending on the particular use. Applications using `atomic` directive, `critical` directive and lock library routines are annotated *atm*, *crt* and *lck* respectively. Only one thread can execute a critical section at a given time. This is useful when global data needs to be modified by the threads in a synchrounous fashion. Atomic sections are performed as if it were one single operation. These are useful only when the operation is simple. Locks are versatile and their usage is application and programmer dependent. Built-in functions are provided by the compilers based on the underlying hardware support. These are generally used for compare-and-swap operations and basic operations on numeric data types.

3.2.2 Galois

Galois handles synchronization in its runtime and provides synchronization wrappers for its objects. Classes that inherit the `GChecked` class can be locked when required. The basic data structures like graphs have inherent locks and can be used directly without worrying about synchronization. Galois speculatively executes iterations with logical locks implemented using compare-and-swap operations. Since Galois handles synchronization in its runtime, all Galois versions use it and alternate schemes are not considered.

3.3 Scheduling

3.3.1 OpenMP

OpenMP provides three scheduling options, static, dynamic and guided. The given iteration space is equally divided among all the threads in static

scheduling. This has very low overhead as scheduling is only done during initialization of a loop. Dynamic scheduling assigns small chunks of iterations to each thread at the onset and more work is assigned as threads complete their tasks. This has a high overhead as the scheduler is active throughout the execution of the loop. Guided scheduling combines properties of static and dynamic scheduling. It initially assigns larger chunks to each thread and reduces the chunk sizes as iterations are completed. This also has more overhead than the static version. OpenMP also provides an ordered clause which ensure that the iterations are executed in order in a deterministic fashion. Ordered execution however has a high overhead. The optimal schedule is application-specific.

These individual schedules are not evaluated in this thesis. However, static scheduling is used for loops which have similar work load per iteration and guided scheduling is used for iterations with load imbalance.

3.3.2 Galois

New iterations can be added to the iteration space dynamically in Galois unlike OpenMP. This makes scheduling of tasks more complex in Galois. Galois has several policies available for determining the order in which to execute iterations. These could be deterministic, such as queue, stack or priority queue, which have higher overhead; or non-deterministic, such as those using chunked queues or independent local queues for each thread. Galois also provides an interface to specify type traits to optimize the runtime system such as `does_not_need_parallel_push` which indicates the operator does not generate new work and push it to the worklist. The deterministic versions can be used for debugging purposes but the non-deterministic iterators maximize parallelism. We only use non-deterministic iterators in the evaluations.

Default scheduling is used for versions that allocate data serially while local queues are used for versions that allocate data in parallel. Galois implements work stealing to handle load imbalance at runtime.

3.4 Successive parallel loops

3.4.1 OpenMP

If multiple loops are successively parallelized using OpenMP, it might be beneficial to use a single parallel section to reduce the overhead of creation and desctruction of threads. One optimization and use `for` directives for inner regions. This method is annotated by the code *mrg*. Another optimization is to explicitly divide the task among the threads which can be run without OpenMP runtime interference. This method is annotated by the code *exp*. Figure 3.1 illustrates these methods. These methods are evaluated in the next sections to evaluate the benefit of these operations. Newer OpenMP implementations generally optimize creation and destruction of threads which might limit the benefits of these optimizations.

```
while (cond) {
#pragma omp parallel for
  for(int i=0; i<n1; i++) {
    ...
  }

  /* serial computation */
  ...

#pragma omp parallel for
  for(int i=0; i<n2; i++) {
    ...
  }
}
```

(a) *bsc*

```
#pragma omp parallel
while (cond) {
#pragma omp for
  for(int i=0; i<n1; i++) {
    ...
  }

#pragma omp single
  {
   /* serial computation */
    ...
  }

#pragma omp for
  for(int i=0; i<n2; i++) {
    ...
  }
}
```

(b) *mrg*

```
#pragma omp parallel
{
  int id = omp_get_thread_num
       ();
  int nth =
       omp_get_num_threads();
  int start1 = ...;
  int end1 = ...;
  int start2 = ...;
  int end2 = ...;
  while (cond) {
    for(int i=start1; i<end1;
         i++) {
      ...
    }
    // barrier if needed

    if (id == ...) {
      /* serial computation */
      ...
    }
    // barrier if needed

    for(int i=start2; i<end2;
         i++) {
      ...
    }
    // barrier if needed
  }
}
```

(c) *exp*

Figure 3.1: *bsc*, *mrg* and *exp* versions of parallel loops using OpenMP

3.4.2 Galois

Galois, on the other hand, handles the creation and scheduling of threads with its own runtime system and does not provide any user controlled parameters

in this regard.

CHAPTER 4

EXPERIMENTAL METHODOLOGY

Performance evaluation of OpenMP and Galois is presented in chapters 5, 6 and 7. All applications in the evaluation are written in C++ and compiled using GNU C++ compiler. Timing for both OpenMP and Galois is measured by using the wall clock time determined by using interfaces in their respective frameworks. Each run is repeated for three trials and the minimum execution time is considered for the evaluation. The reported time is only for the bulk of the computation. Initialization and clean-up portions of the application are not included. Speedup is computed with respect to the single thread performance of each version and not with respect to the serial version. This is to compare speedups without taking into consideration the overheads of using OpenMP or Galois. Most implementations using OpenMP use minimal object oriented features although they are written in C++. Galois is highly object oriented with basic synchronization entities as objects which adds additional overheads to Galois, both in terms of performance and number of lines of code. In the evaluation, both execution time and speedup need to be considered to give a complete picture of the performance.

Two machines are used for the performance evaluations. The first machine has two Intel®Xeon®X5650 processors with 6 cores each, operating at a frequency of 2.66 GHz. Each processor has 12MB of L3 cache and the system has 24GB RAM. It runs Scientific Linux 6.1 and the compiler used is GCC version 4.7.1. Galois version 2.2.1 is used with Boost libraries of version 1.51.0. This machine is referred to as Taub in the evaluations.

The second machine has four Intel®Xeon®E7-4860 processors with 10 cores each, operating at a frequency of 2.27 GHz. Each processor has 24MB of L3 cache and the system has 128GB RAM. It runs Scientific Linux 6.6 and the compiler used is GCC version 4.8.2. Galois version 2.2.1 is used with Boost libraries of version 1.58.0. This machine is referred to as I2PC in the evaluations.

CHAPTER 5

PERFORMANCE EVALUATION FOR REGULAR DATA STRUCTURES

The problem used for this evaluation is a basic heat transfer simulation in a 2D matrix using Jacobi iterations. The algorithm uses a 5 point stencil to compute an updated value for each of the grid points. Algorithm 1 describes the basic algorithm in detail.

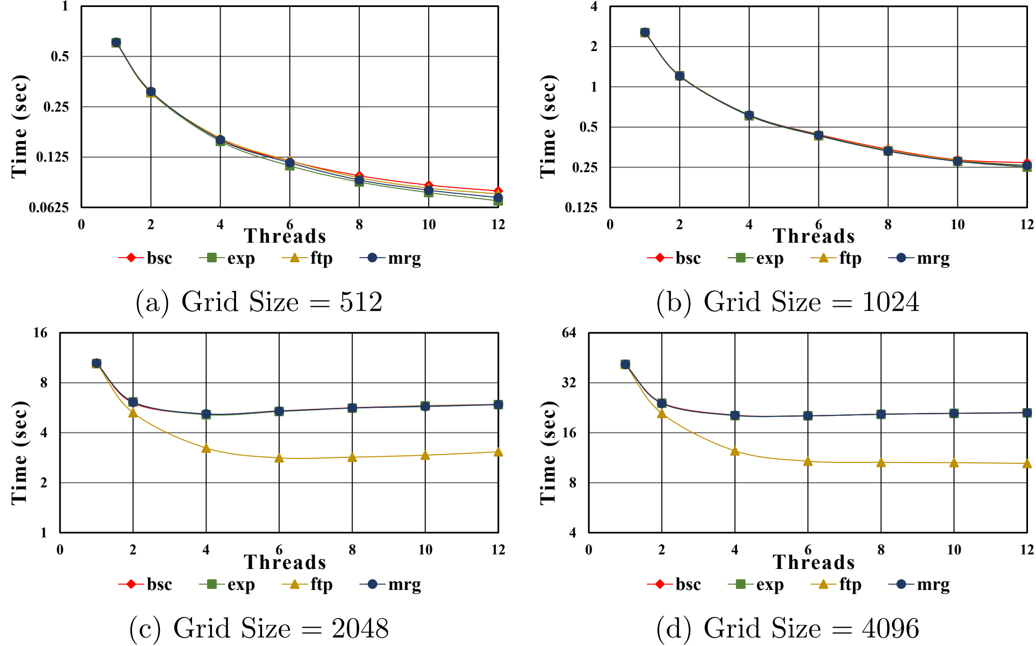
The for loops in each iteration are completely independent in this computation. They access two separate grids and do not cause any false sharing or synchronization issues. However, each iteration must be computed sequentially. We compare the different memory allocation policies for OpenMP and Galois, and the effects of merging successive loops together in OpenMP. These experiments are performed on the machine Taub.

Algorithm 1 Basic heat transfer pseudocode

```
while iteration < max_iter do
    Update boundary values
    for i in 1 to n do
        for j in 1 to n do
            Update B[i][j] using 5 point stencil of matrix A
        end for
    end for
    Swap matrix B and matrix A
    increment iteration
end while
```

5.1 OpenMP

To compare effects of parallel memory allocation and merging of successive loops, we implement four versions using OpenMP based on the optimization strategies mentioned in chapter 3. *bsc* is the basic implementation using `parallel for` directive. *ftp* has the basic implementation with data initial-



(a) Grid Size = 512

(b) Grid Size = 1024

(c) Grid Size = 2048

(d) Grid Size = 4096

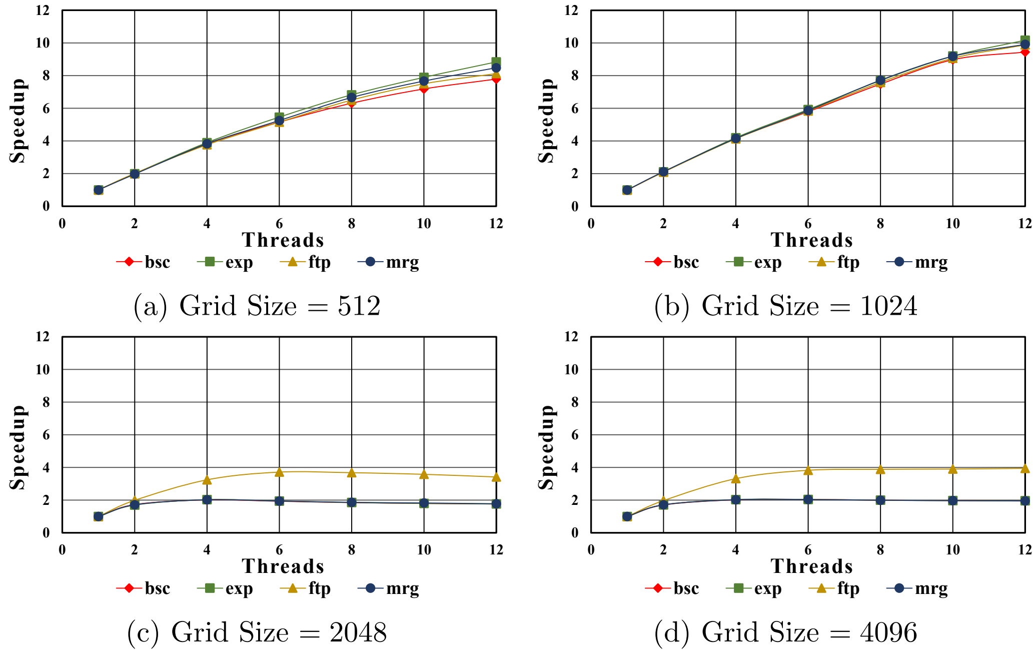
Figure 5.1: Execution time of Jacobi for 1000 iterations using OpenMP

ization performed in parallel. *mrg* has one single parallel region and `for` directives within the parallel region for loops and `single` directive for serial parts. Lastly, *exp* again has one single parallel region but with explicit task division by computing start and end indices, and barriers for synchronization.

These versions are run for various grid sizes and number of iterations and the results are presented in figures 5.1 to 5.4. The running time and speedup for different grid sizes and 1000 iterations are shown in figures 5.1 and 5.2.

It can be observed in figures 5.1 and 5.2 that all versions perform well for smaller grid sizes with $\sim$ 10x speedup for grid size 1024 with 12 threads. However, the speedup falls drastically to $\sim$ 2x for *bsc*, *exp* and *mrg*. It falls to $\sim$ 4x for *ftp*. Jacobi iterations are memory intensive with very little computation. For smaller grid sizes that fit in cache, speedup is good. However, for larger grid sizes, memory becomes a bottleneck and performance is poor. *ftp* performs better over the other versions but is still restricted by memory. Taub has two processors of 6 cores each, and allocating memory in parallel improves locality in the NUMA machine.

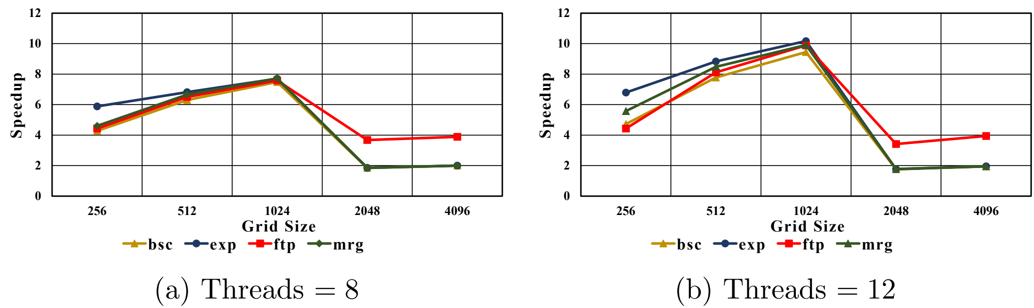
A comparison of speedups with respect to grid sizes is shown in figure 5.3 and a comparison of speedups with respect to number of iterations is shown for grid sizes of 512 and 1024 in figure 5.4. The speedups increase for larger grid sizes until 1024 which is due to increased work per thread.



(a) Grid Size = 512 (b) Grid Size = 1024

(c) Grid Size = 2048 (d) Grid Size = 4096

Figure 5.2: Speedup of Jacobi for 1000 iterations using OpenMP



(a) Threads = 8 (b) Threads = 12

Figure 5.3: Comparison of speedup across grid sizes using OpenMP

Comparing the parallelization strategies bsc, mrg and exp, it can be observed that exp performs the best, followed by mrg and then bsc. While this is as expected, it must be noted that the difference between them becomes increasingly small for larger grid sizes. The wider range at 256 narrows down by 1024 as observed in figure 5.3. With increasing work per thread, the OpenMP overheads reduce in proportion to the total time and hence give better speedups. The speedups increase with increasing iterations in general with a few outliers as observed in figure 5.4. However, the impact is not drastic.

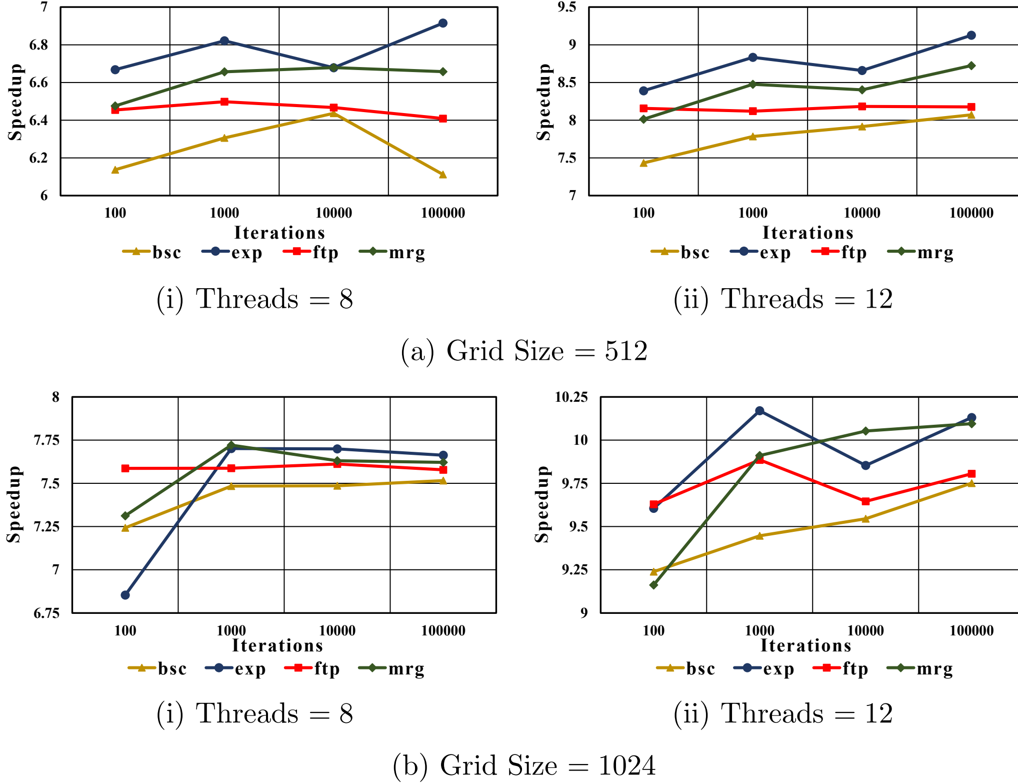


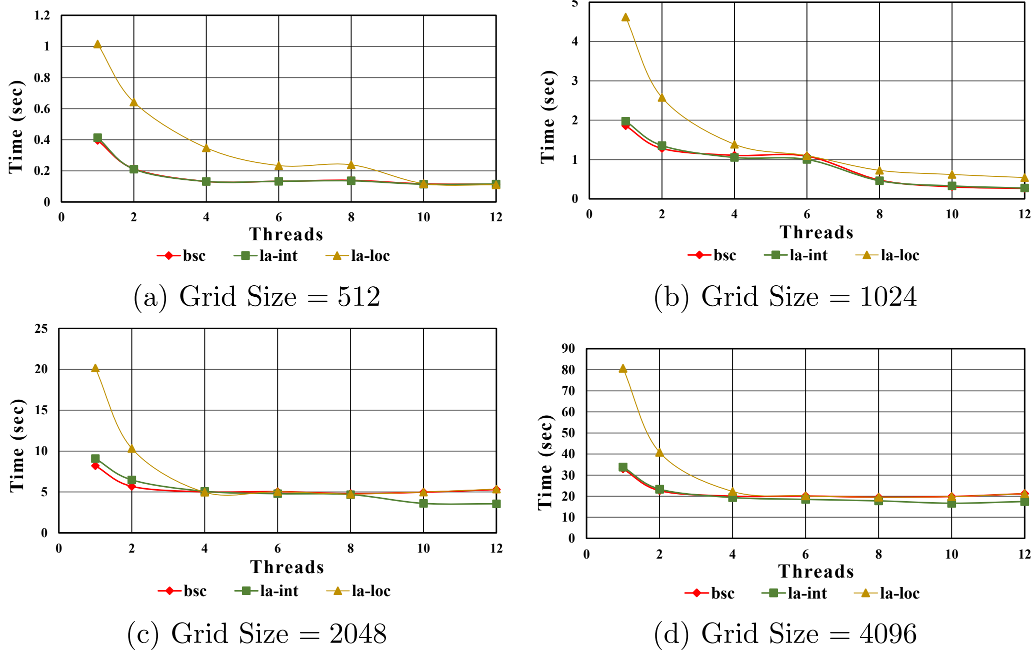
Figure 5.4: Comparison of speedup across iterations using OpenMP

5.2 Galois

Three versions of Jacobi iterations are implemented in Galois. The basic version *bsc* has dynamically allocated arrays with iterations based on loop indices. Two versions are implemented using the `Galois::LargeArray` data structure, one with interleaved allocation called *la-int* and one with local allocation based on first touch policy called *la-loc*.

A comparison of the performance of these three versions for varying grid sizes and number of iterations is shown in figures 5.5 to 5.8. Execution time and speedup for different grid sizes are shown in figures 5.5 and 5.6. It can be observed that *la-loc* has a very high overhead for serial execution but catches up with the other two versions occasionally. This gives a significant boost to its speedup but it has never outperformed the other versions. *la-int* and *bsc* have similar performance for small number of threads but diverge with *la-int* performing better with larger number of threads.

Speedups increase with increasing grid sizes for smaller grids, but fallxs down to nearly 2x as demonstrated in figure 5.7. A comparison of speedups



(a) Grid Size = 512

(b) Grid Size = 1024

(c) Grid Size = 2048

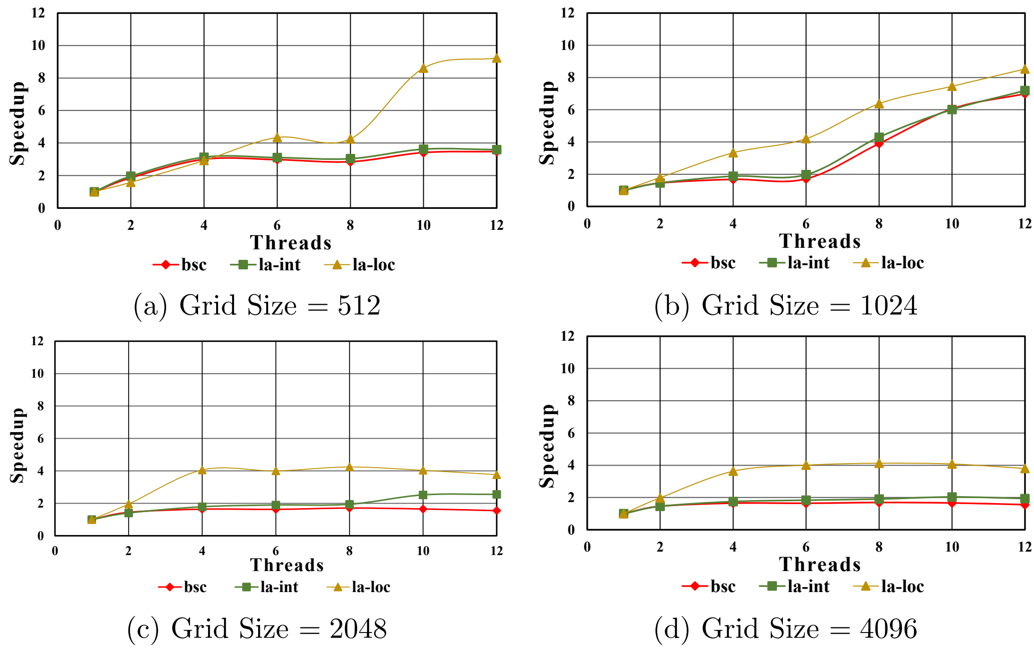
(d) Grid Size = 4096

Figure 5.5: Execution time of Jacobi for 1000 iterations using Galois

with respect to number of iterations is shown for a grid size of 1024 in figure 5.8. It can be observed that the speedup falls with increasing iterations for 8 threads but does not significantly change for 12 threads.

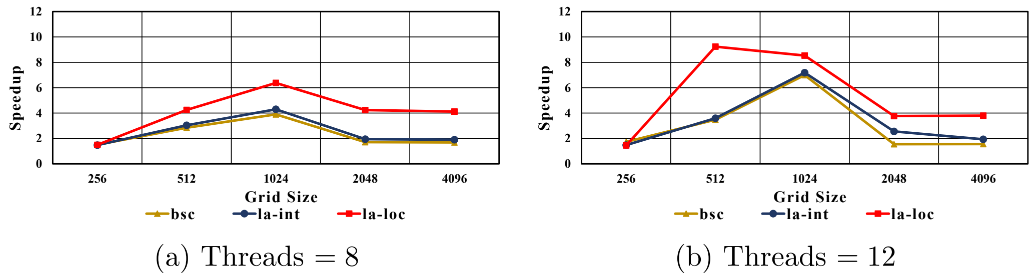
5.3 Comparison of OpenMP and Galois

A comparison of performance of Jacobi iterations for OpenMP and Galois is shown in figure 5.9. The minimum time across all implementations is considered for this comparison. It can be observed that the OpenMP overhead for 1 thread is higher than that of Galois significantly. However, OpenMP has a uniform speedup and outperforms Galois from 2 threads onwards. Galois catches up with OpenMP for a grid size of 1024 for 10 and 12 threads. However, for a grid size of 4096, *ftp* of OpenMP performs relatively better while Galois stays almost on par with the other implementations of OpenMP.



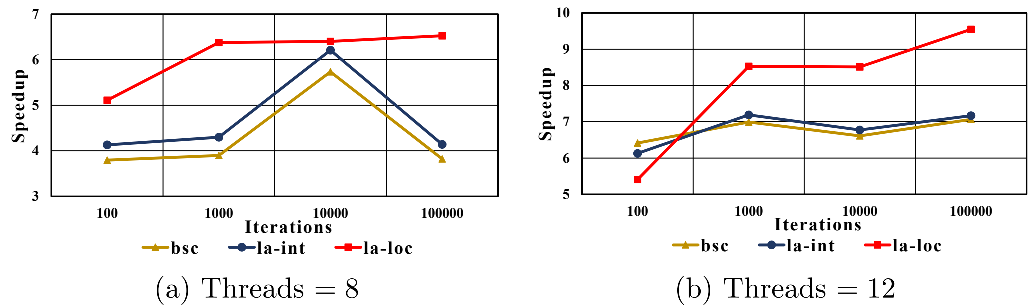
(a) Grid Size = 512
(b) Grid Size = 1024
(c) Grid Size = 2048
(d) Grid Size = 4096

Figure 5.6: Speedup of Jacobi for 1000 iterations using Galois



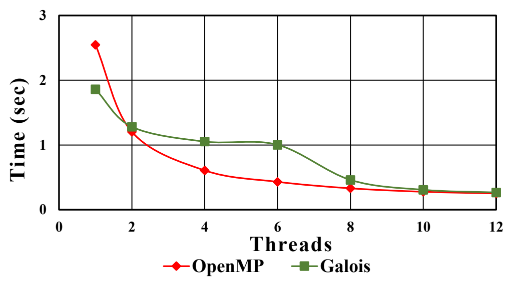
(a) Threads = 8
(b) Threads = 12

Figure 5.7: Comparison of speedup across grid sizes using Galois

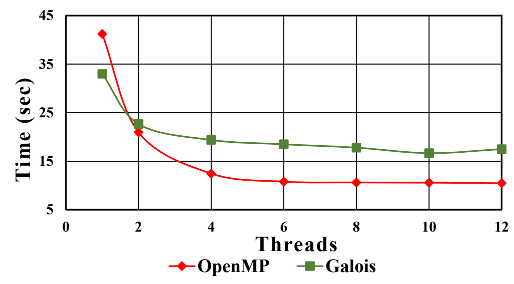


(a) Threads = 8
(b) Threads = 12

Figure 5.8: Comparison of speedup across iterations using Galois



(a) Grid Size = 1024



(b) Grid Size = 4096

Figure 5.9: Comparison of execution time using OpenMP and Galois

CHAPTER 6

PERFORMANCE EVALUATION FOR IRREGULAR DATA STRUCTURES

The problem used for evaluation of performance for irregular data structures is Jacobi iterations using a triangulated mesh. Each triangular element computes a new value based on the neighboring elements' current value. Two variants of this computation are used to perform the evaluations. In the first variant, each element reads the current value of its neighbors and computes its new value. This variant is annotated *pull* to signify that the data is pulled from the neighbors by each element. In the second variant, each element adds its current value component to its neighboring elements to compute the new value. This variant is annotated *push* to signify that the data is pushed to the neighbors by each element. *Pull* requires no explicit synchronization while *push* requires synchronization for correct computation. An element holds both the old and the new values in a struct or a class. It is very likely that these two values share the same cache-line which can cause false sharing.

The focus of this evaluation is to explore effects of memory allocation in parallel for both OpenMP and Galois. Additionally, synchronization methods for OpenMP are evaluated for the *push* variant. These experiments are performed on the I2PC machine.

The triangulated meshes used in this evaluation are described in table 6.1.

Table 6.1: Test cases for Jacobi iterations in a mesh

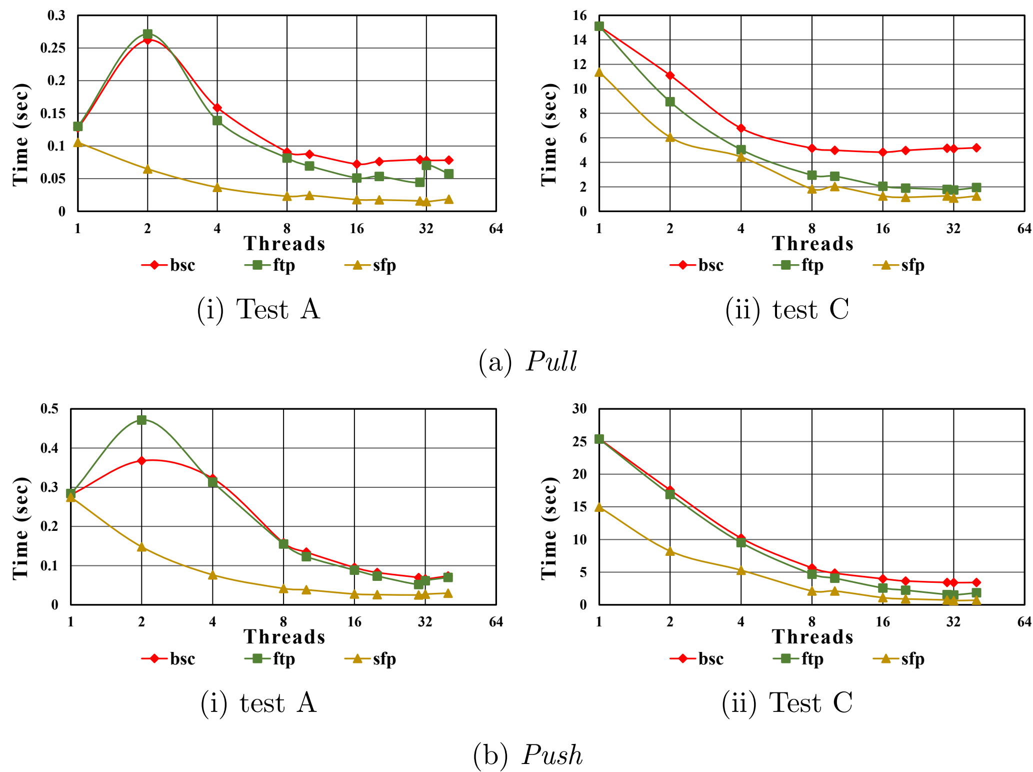
Name	Nodes	Elements
A	37,615	74,474
B	311,974	621,893
C	1,246,282	2,488,434
D	4,979,645	9,951,030

6.1 OpenMP

Memory allocation and synchronization optimizations are evaluated using OpenMP.

6.1.1 Memory Allocation

To compare the effects of memory allocation optimizations using OpenMP, three implementations are considered. The basic implementation, *bsc*, only parallelizes the computation loop. The parallel initialization version, *ftp*, reads the mesh into a temporary data-structure and initializes the final data-structure in parallel. The parallel initialization with space-filling curves, *sfp*, sorts the data in the temporary structure and then initializes in parallel. The results presented are for 100 Jacobi iterations. The `atomic` directive is used for synchronization in the *push* variant for all the three versions.



(i) Test A (ii) test C

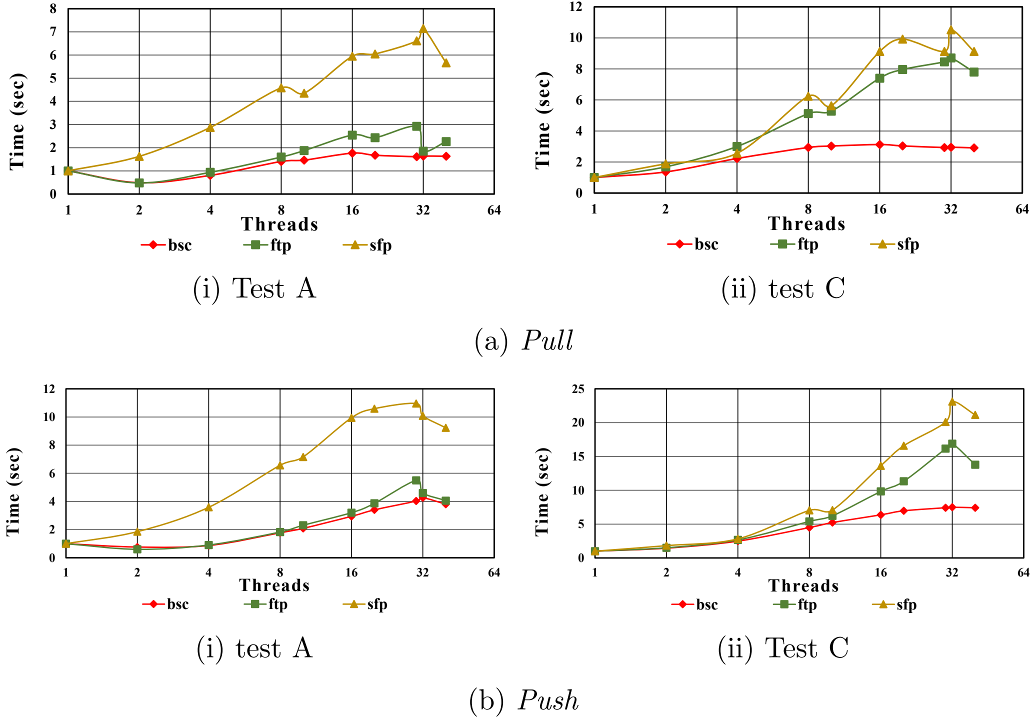
(a) *Pull*

(i) test A (ii) Test C

(b) *Push*

Figure 6.1: Execution time of mesh Jacobi using OpenMP

Figures 6.1 and 6.2 show the execution time and speedup for *push* and *pull* variants for test cases A and C.



(i) Test A

(ii) test C

(a) *Pull*

(i) test A

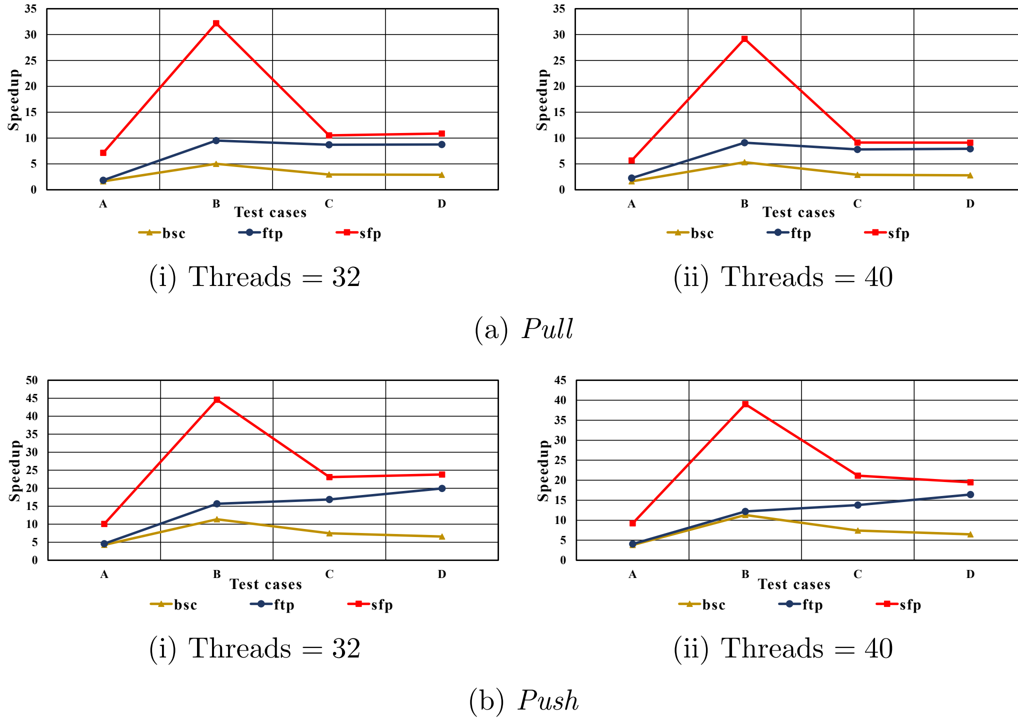
(ii) Test C

(b) *Push*

Figure 6.2: Speedup of mesh Jacobi using OpenMP

It can be observed that *sfp* performs better than the other two for both mesh sizes. This is due to improved cache locality and parallel allocation of data. It can also be observed that *ftp* performs similar to *bsc* for 1 thread as expected. It stays closer to bsc for smaller mesh size but moves closer to *sfp* for larger mesh sizes. Also, *sfp* performs better than the other two versions even for a single thread, owing to higher cache locality. Even though the execution time for a single thread is higher for *sfp*, *sfp* out-performs the other two with respect to speedup.

A comparison of speedup across mesh sizes is shown in figure 6.3. It can be seen that test case B in *sfp* performs super linearly at 32 threads. This is mainly due to the mesh fitting in cache in parallel allocation and improved locality in *sfp*. Additionally, gap between *bsc* and *ftp* widens with larger grid sizes. This is due to improved memory usage, similar to the improvement observed in the case of regular data structures. However, this improvement is limited here, due to the irregular and random access patterns.



(i) Threads = 32 (ii) Threads = 40

(a) *Pull*

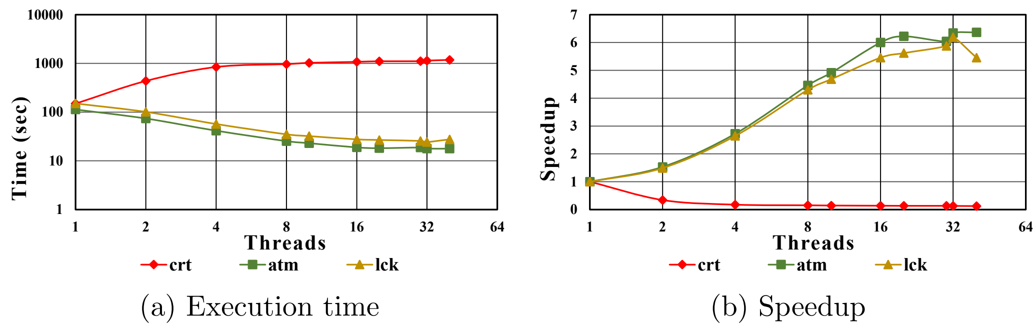
(i) Threads = 32 (ii) Threads = 40

(b) *Push*

Figure 6.3: Speedup of mesh Jacobi using OpenMP

6.1.2 Synchronization

Three synchronization methods using `atomic` directive (*atm*), `critical` directive (*crt*) and OpenMP locks (*lck*) are evaluated for the `push` variant of Jacobi iterations on a mesh.



(a) Execution time (b) Speedup

Figure 6.4: Evaluation of synchronization methods using OpenMP

A comparison of execution time and speedup is shown in figure 6.4 for the three synchronization directives on test case D. The behavior is similar for the other test cases as well. Using `critical` directive in a loop where the only operation performed is a critical operation makes the execution much

worse than serial execution. `critical` directive is useful only if each iteration spends a significantly longer time outside the critical region. Figure 6.4 shows that *atm* performs better than *lck*.

6.1.3 Other

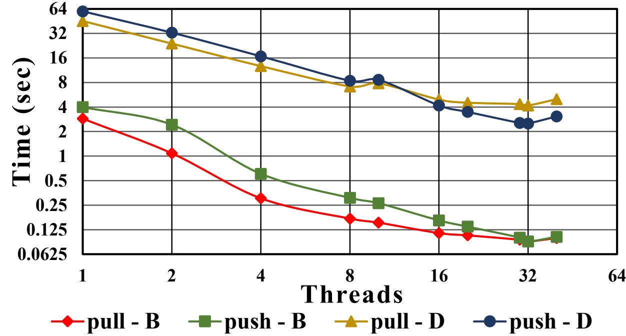
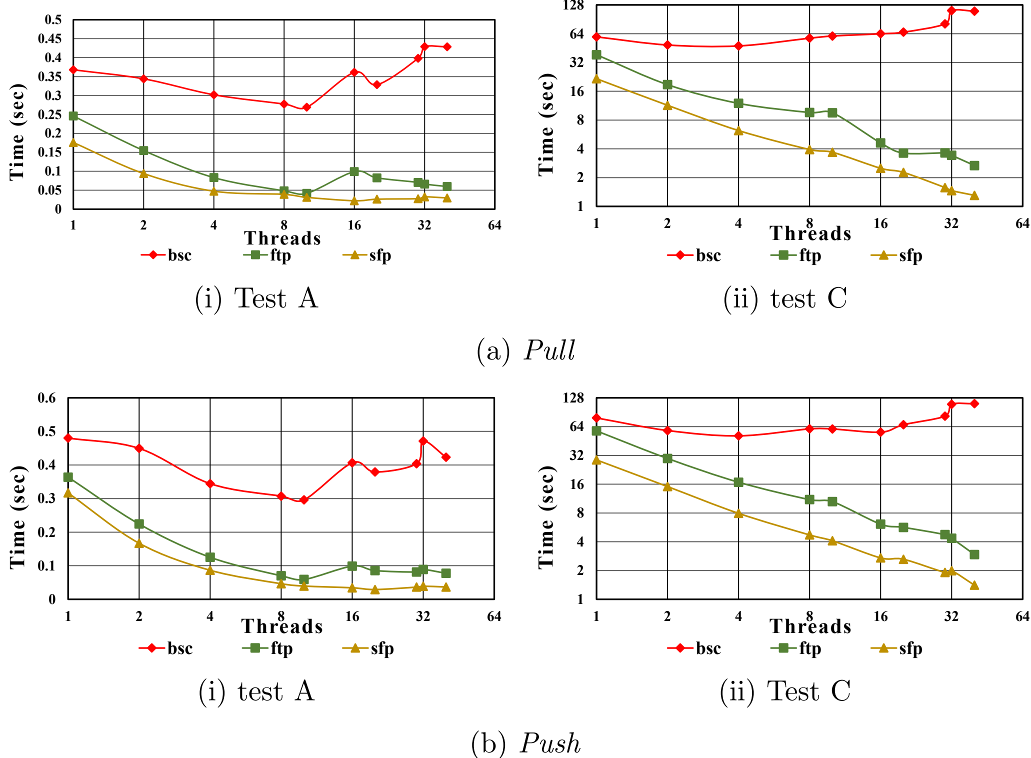


Figure 6.5: Comparison of *push* and *pull* for test cases B and D using OpenMP

The two variants *push* and *pull* are compared for test cases B and D using space-filling parallel allocation *sfp* in figure 6.5. As expected, *pull* takes lesser time than *push* generally. However, it is interesting to note that this trend reverses for the larger mesh, D, beyond 10 threads. This is the observed behavior for test case C as well, although it has not been shown in the figure. The I2PC machine used for running these tests has 10 cores per processor which indicates that in a NUMA setting, pull is more expensive than push. One major reason could be false sharing as the location read from and the location written to are very close to each other. Hence they may fall on the same cache line. This causes the observed performance degradation.

6.2 Galois

Memory allocation optimizations are evaluated for Galois considering three implementations, *bsc* where the data structure is built serially, *ftp* where the data structure is built in parallel, and *sfp* where data is sorted using a space-filling curve and then the data structure is built in parallel. The same two variants *push* and *pull* are implemented using Galois.



(i) Test A

(ii) test C

(a) *Pull*

(i) test A

(ii) Test C

(b) *Push*

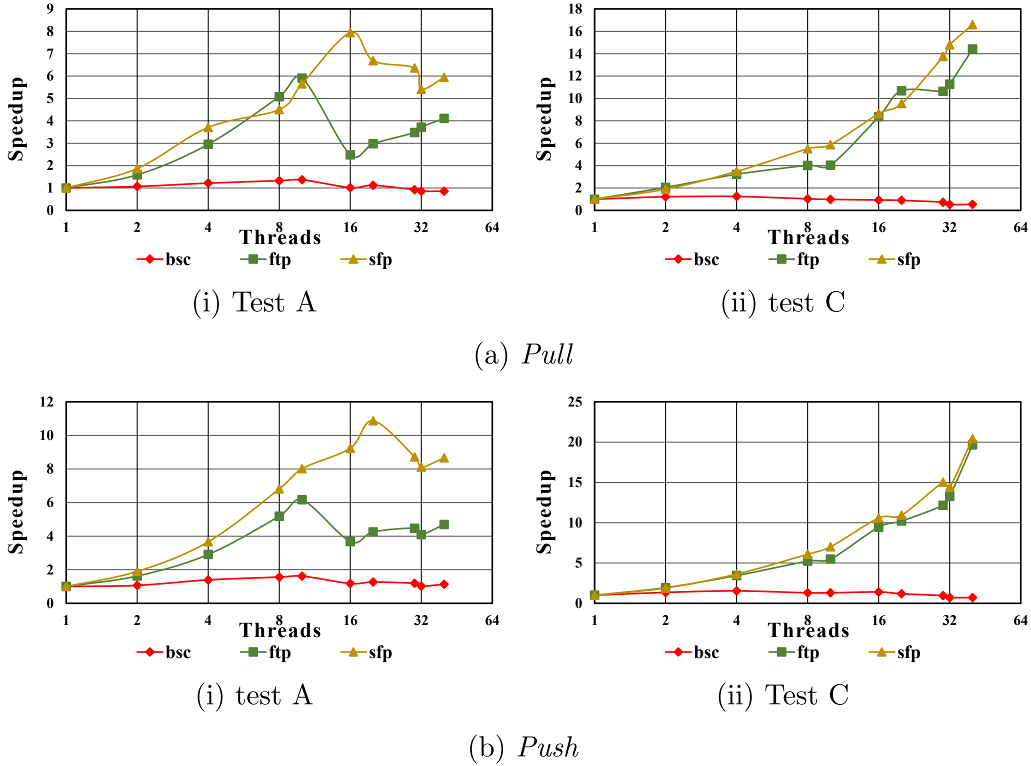
Figure 6.6: Execution time of mesh Jacobi using Galois

Figures 6.6 and 6.7 show the execution time and speedup for *push* and *pull* variants for test cases A and C.

From figures 6.6 and 6.7, *sfp* performs better than the other two for all grid sizes. This is due to improved cache locality and parallel allocation of data. In addition, *ftp* and *sfp* do not have much speedup beyond 10 threads for both *push* and *pull* for the smaller test case A but this is not the case for the larger test case C.

A comparison of speedup across mesh sizes is shown in figure 6.8. It shows the abnormally high speedup for test case B is present for Galois too. We assume this must be due to the same reasons as applicable for OpenMP. It can also be noted that the speedup of *bsc* is significantly low.

A comparison of the two variants *push* and *pull* for test cases B and D using space-filling parallel allocation *sfp* is shown in figure 6.9. *Pull* performs better than *push* throughout except for a little overlap for high number of threads in test case B.



(i) Test A

(ii) test C

(a) *Pull*

(i) test A

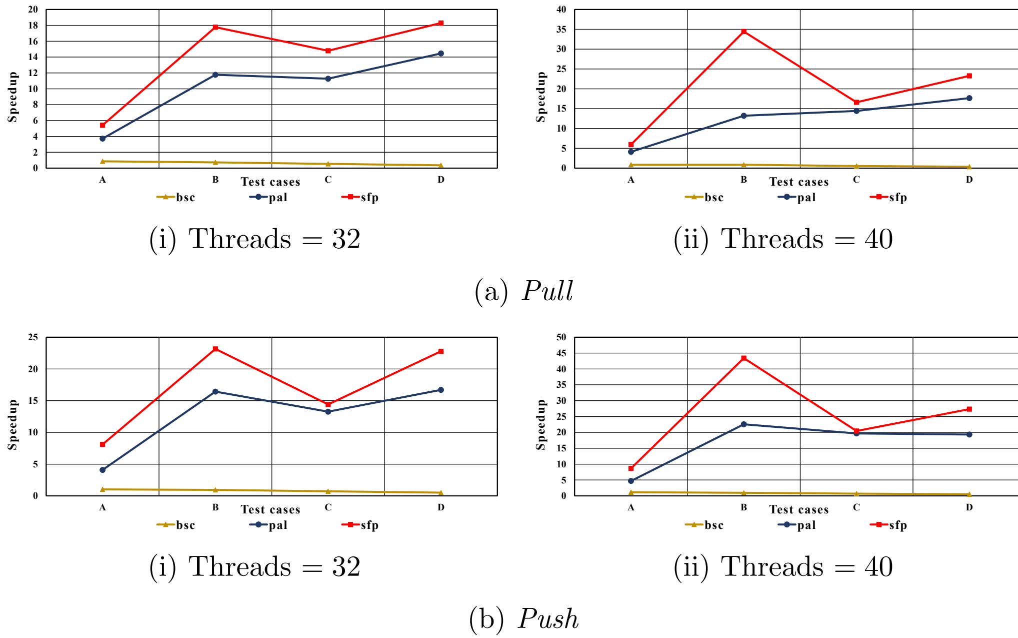
(ii) Test C

(b) *Push*

Figure 6.7: Speedup of mesh Jacobi using Galois

6.3 Comparison of OpenMP and Galois

Figure 6.10 shows the execution time of Galois and OpenMP for two test cases, B and D using *sfp*. It is clear that OpenMP has lesser serial overhead in both test cases due to its simpler code structure. However, Galois performs more steadily for the larger test case D where the gap between OpenMP and Galois is less while the gap increases significantly for the smaller test case B.



(i) Threads = 32 (ii) Threads = 40

(a) *Pull*

(i) Threads = 32 (ii) Threads = 40

(b) *Push*

Figure 6.8: Speedup of mesh Jacobi using Galois

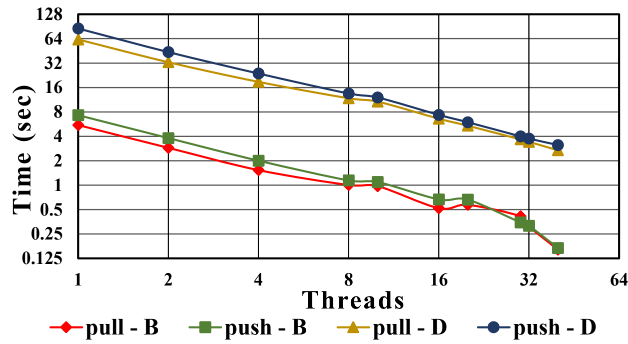
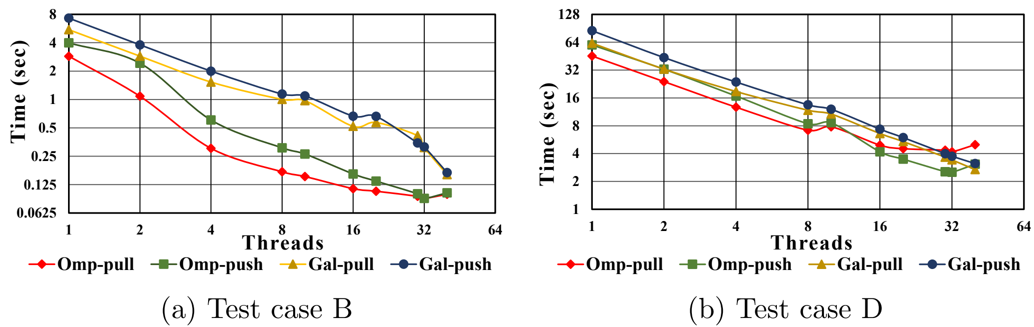


Figure 6.9: Comparison of *push* and *pull* for test cases B and D using Galois



(a) Test case B (b) Test case D

Figure 6.10: Comparison of OpenMP and Galois for mesh Jacobi

CHAPTER 7

PERFORMANCE EVALUATION ON REAL WORLD APPLICATIONS

Three applications with different data layouts and algorithms are selected for performance evaluation of parallelization using OpenMP and Galois. High Accuracy Relativistic Magnetohydrodynamics (HARM) operates on a set of arrays, Delaunay Triangulation operates on a set of points to construct a triangulated mesh, and Dynamic Fracture Propagation operates on a triangulated mesh.

7.1 High Accuracy Relativistic Magnetohydrodynamics (HARM)

7.1.1 Description

HARM is a conservative finite volume approach to solve hyperbolic partial differential equations (PDE). This application was developed by Grammie et al. [6] and extended by Noble et al. [9] to solve general relativistic magnetohydrodynamics (GRMHD).

7.1.2 Algorithm

The main data structures used are grids for primitive variables **P**, conserved variables **U** and fluxes **F**. The algorithm uses a conservative approach and updates a combination of a set of "conserved" variables at each timestep. A vector of conserved variables **U** for each grid point is updated using fluxes **F**. Then, multidimensional Newton-Raphson routine uses the updated **U** and **P** from the previous timestep to update **P**.

The algorithm uses half steps to advance in each iteration. An overview of the steps are described in Algorithm 2

Algorithm 2 HARM pseudocode

```
function MAIN()
    Initialize data
    while time < max_time do
        Advance half step
        Fix primitive variables at half step
        Advance half step
        Fix primitive variables after full step
        Update time
        Update timestep
    end while
end function

function ADVANCE()
    Update flux in both directions
    Fix fluxes
    for all grid points do
        Compute conserved variables using updated flux
        Solve to obtain updated primitive variables
    end for
    Compute maximum safe timestep
end function

function UPDATE FLUX()
    for all grid points do
        Evaluate slope of primitive variables
    end for
    for all grid points do
        Compute a slope-limited extrapolation of primitive variables
        Compute wave speeds
        Update flux
    end for
end function
```

Each of these steps perform Jacobi-like iterations that read one matrix and write to another matrix with a 9 point stencil. For each of these loops, the iterations are independent. However, a number of such loops need to be run in each iteration.

7.1.3 Implementation

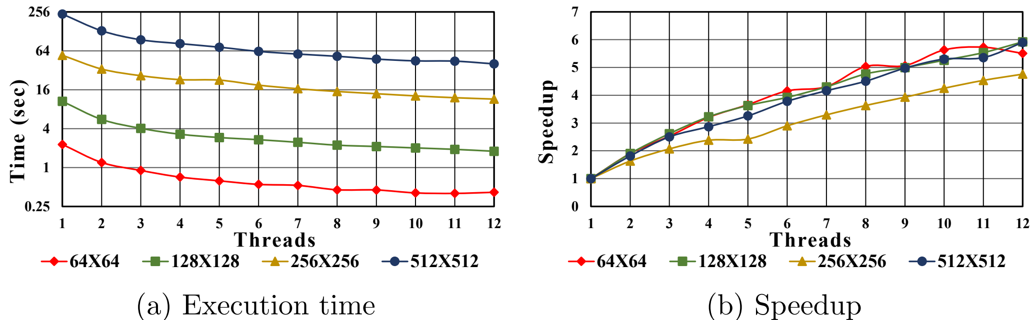
The sequential version of the code is obtained from the astrophysical code library hosted by Grammie et. al [6], [9]. It is implemented in C using a set of static 5 dimensional arrays as the basic data structures.

As a primer, the code is converted from C to C++ by hand as Galois only works for C++. Also, functions with global variables that are modified in each iteration were changed to receive them as parameters instead.

Using both OpenMP and Galois, the same set of loops were parallelized using their corresponding parallelization constructs.

7.1.4 Evaluation

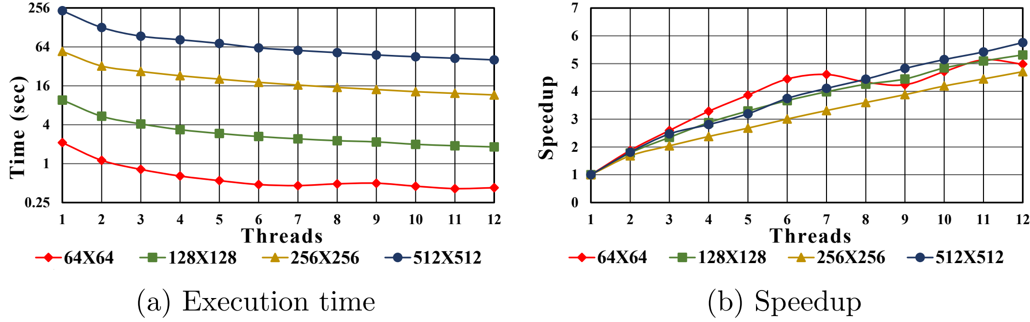
Figures 7.1 and 7.2 show execution time and speedups of HARM on the machine Taub using OpenMP and Galois, respectively. The results shown are for 100 iterations with different grid sizes. Both OpenMP and Galois perform almost similar in this case subject to experimental errors. This could be because they are both run for a moderate number of iterations and perform significant computation in each step. The memory is allocated on the stack as all the arrays are static.



(a) Execution time (b) Speedup

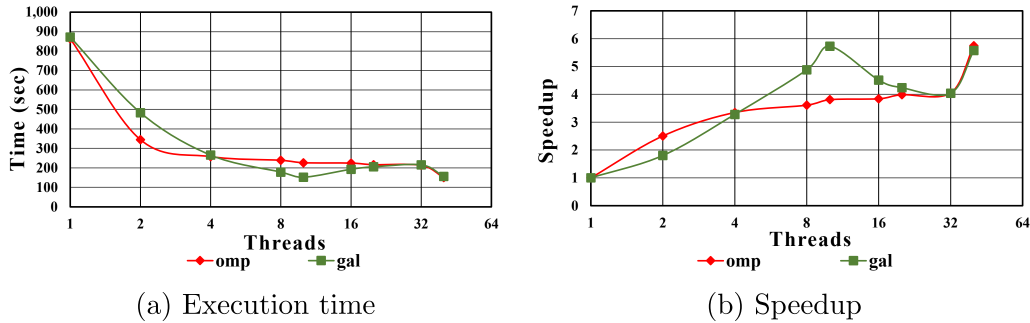
Figure 7.1: Performance of HARM using OpenMP

Comparison of OpenMP and Galois are shown in figure 7.3.



(a) Execution time (b) Speedup

Figure 7.2: Performance of HARM using Galois



(a) Execution time (b) Speedup

Figure 7.3: Performance of HARM using OpenMP and Galois for 1024X1024 size grid on I2PC machine

7.2 Delaunay Triangulation

7.2.1 Description

Delaunay triangulation is a triangulation of a given set of points $\mathcal{P}$ such that no point in $\mathcal{P}$ lies in the circumcircle of any triangle in the triangulation. Delaunay triangulations are often used to build meshes for finite element methods, for modelling surfaces and terrains, etc. There are several methods for computing Delaunay triangulation and the following implementations use an incremental approach [17].

7.2.2 Algorithm

Delaunay triangulation implementation uses an incremental construction. The main steps of the algorithm are given in algorithm 3.

The main data structures used are a graph to represent the triangulation and a quad tree of all the inserted points to obtain a close triangle.

Algorithm 3 Delaunay triangulation pseudocode

```
Initialize data
Compute bounding triangle containing all the given points
for all Points do
    Locate triangle containing the point to be inserted
    Compute the cavity - region that is affected by insertion
    Update cavity
end for
```

7.2.3 Implementation

OpenMP

Problems Based Benchmarks Suite (PBBS) has an implementation of Delaunay triangulation using OpenMP/Cilk [11]. Although it was primarily designed to be used for Cilk, the OpenMP version could be setup with minimal changes.

This implementation picks a few set of points in each iteration and tries to insert these points to the triangulation. Any points that failed due to conflicts are added to the remaining worklist.

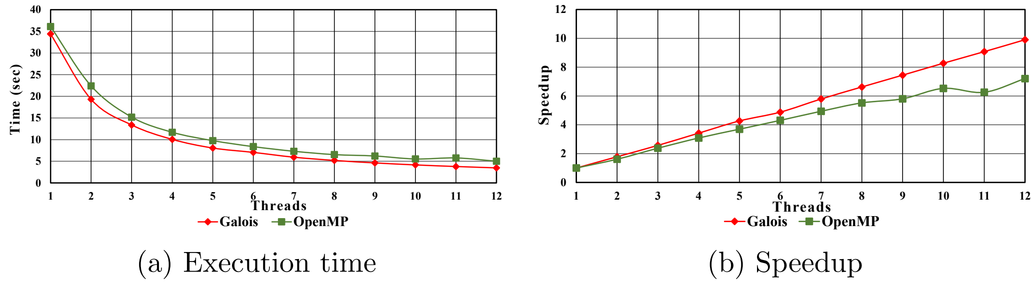
It is done in two phases. The first phase is the cavity building phase, where all the selected points are located and the cavities are computed. This phase does not modify the graph. The second phase modifies the graph where each thread reserves its cavity and updates it. Any point whose cavity could not be reserved is added back to the worklist as a new cavity must be computed.

Galois

Galois provides an implementation of Delaunay triangulation as a sample application. This implementation uses a `WorkSet` for points to be inserted, a `Galois::FirstGraph` for the triangulation and a `QuadTree` for point location. The algorithm uses Galois specified for loop where cavity for each node where the containing triangle is located, cavity is built and updated. This does not have two different phases as in the OpenMP implementation, and reserves the cavity as it builds.

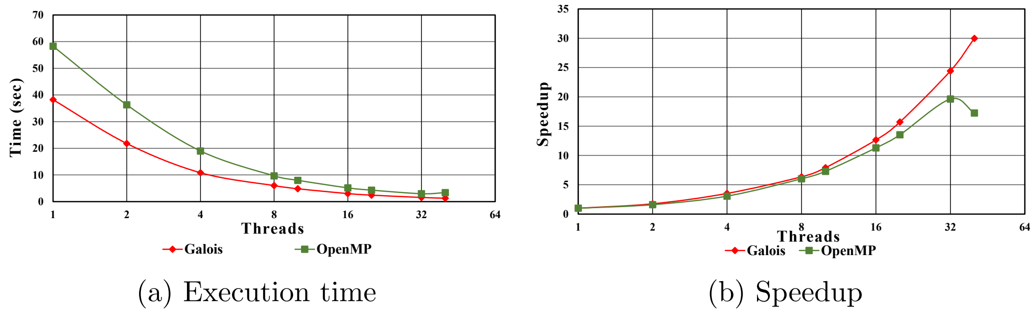
7.2.4 Evaluation

Figures 7.4 and 7.5 show the execution time and speedup for a test case of 10 million points on Taub and I2PC respectively. Galois has a speedup of 10x and 30x while OpenMP has 7x and 20x on Taub and I2PC respectively. Galois performs particularly well on these kind of applications as it is tuned to handle such data. Also, there is a single Galois loop in this application which makes the overheads minimum. OpenMP on the other hand, is not well suited for these kinds of applications and consequently, performance suffers.



(a) Execution time (b) Speedup

Figure 7.4: Delaunay triangulation on Taub for 10 million points



(a) Execution time (b) Speedup

Figure 7.5: Delaunay triangulation on I2PC for 10 million points

7.3 Dynamic Fracture Simulation

7.3.1 Description

Dynamic fracture simulation solves dynamic structural problems using second order explicit central difference finite element method in a plane with 3 node triangular elements.

7.3.2 Algorithm

The main data structures in this problem are a list of nodes and a graph of the triangulated mesh.

The structure of the code in high level is described in Algorithm 4.

Algorithm 4 Dynamic fracture simulation pseudocode

```
Initialize data
while time < max_time do
    Update time
    for all nodes do
        Update displacement
    end for
    for all triangles do
        Compute stress and strain tensor
        Smoothen stress using viscous stress
        Compute damage parameter
        Add element's contribution to nodal internal force vectors
    end for
    for all nodes do
        Update velocity and acceleration
    end for
    if Mesh refinement needs to be performed then
        for all triangles do
            Mark if refinement is needed
        end for
        Refine mesh
        Recompute lumped mass
        Transfer old solution to new by interpolation
        Adapt the timestep
    end if
    for all Boundary nodes do
        Apply boundary conditions
    end for
end while
```

Majority of the execution time is spent in the the computation. However, refinement increases the problem size gradually which affects the total execution time. A key data dependency in the computation phase is the computation of internal nodal force vector by looping over the triangles and updating its incident nodes.

7.3.3 Implementation

OpenMP

The sequential code is implemented in C++ with minimal object-oriented programming. The nodes and triangles are stored in a `vector` class of the C++ standard template library. Indices are used as the primary point of reference and each triangle stores indices of its incident nodes and neighboring triangles.

The update to nodes while looping over triangles is performed using the OpenMP atomic operation as stress contribution is only added to the existing stress. For mesh refinement, each triangle is marked to be either refined into 4 triangles or bisected into two. The triangles to be refined are first processed and then the necessary triangles are bisected. A map is kept to check if an edge has already been bisected and neighboring triangles are reserved for refinement. Updates in this section are using built-in atomic operations.

Two implementations of OpenMP, one using an array of structures for nodes and elements, and another using a structure of arrays are implemented. They are annotated *struct* and *no-struct* respectively.

Galois

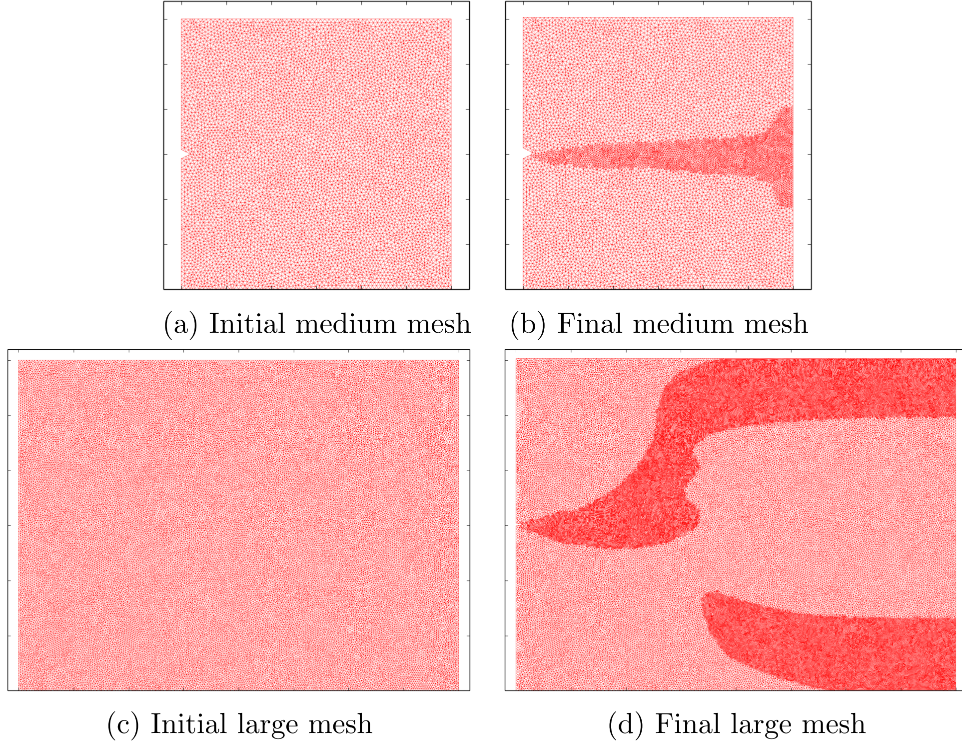
The Galois code is implemented in C++ with Galois objects. The nodes extend `Galois::GChecked` and are stored in a `Galois::InsertBag`. The triangles are stored in a `Galois::FirstGraph`, where each vertex is a triangle and the edges indicate adjacency. Unlike OpenMP, pointers are used instead of indices to point to the nodes. The edges are maintained by the graph data structure and no extra pointers are used.

The nodes are updated using Galois native conflict detection. For triangulation, a similar phase of marking the triangles to be refined or bisected is used. However, a worklist of these triangles is built and processed instead of iteration over all the triangles. Conflicts for insertion and deletion of triangles in the mesh during refinement is also handled by the Galois data structures.

Three versions of dynamic fracture propagation have been implemented in Galois using different memory allocation schemes, serial allocation (*bsc*), parallel allocation (*ftp*) and parallel allocation with data sorted in a space

filling manner (*sfp*).

7.3.4 Evaluation



(a) Initial medium mesh (b) Final medium mesh

(c) Initial large mesh (d) Final large mesh

Figure 7.6: Initial and final triangulated meshes for the test cases

Table 7.1: Test cases for dynamic fracture simulation

Test	Initial		Final		Iterations
	Nodes	Elements	Nodes	Elements	
Medium	7074	13852	9376	18429	80000
Large	37615	74474	72497	144001	31000

Two test cases are used for this evaluation. The first is a medium sized square material with a side of $0.03m$. It has an initial crack length of $0.001m$. The simulation runs for a duration of $0.25ms$. A constant vertical velocity of $0.8m/s$ is imposed on the top boundary with an initial ramp-up phase for 10% of the time. The bottom boundary is held stationary while the size boundaries are fixed in the x direction but free to move in y. The crack is traction free. The second is a larger material which is $0.06m$ long and

$0.08m$ wide with a $0.001m$ long initial crack. The simulation runs for $0.1ms$ with a constant velocity of $3m/s$ imposed on the top border. The remaining conditions are similar to the first test case including the initial ramp-up phase.

Table 7.1 gives the numerical size of the test cases with initial and final mesh sizes and number of iterations. Figure 7.6 shows the initial and final meshes for the two test cases.

The stress and velocity distribution for both the test cases at different times of simulation are shown in figures 7.7 and 7.8. It can be seen that the cut is along the center for the medium test case which has a lower imposed velocity for a longer time duration where as the damage for the large test case is not very uniform. This is due to a higher velocity imposed for a short duration of time.

OpenMP

The execution time and speedup for the medium test case are shown in figure 7.9. Clearly, flattening out the array of structures into a series of arrays did not gain any benefit for this application. One major reason is that although the initializations can be vectorized, the bulk of the computation accesses data randomly and cannot be vectorized.

The execution time and speedup for both the test cases without structures are shown in figure 7.10. It can be seen that with fewer iterations and much larger mesh size, large test case has a speedup of $23x$ compared to the $13x$ speedup of medium test case.

Galois

Figure 7.11 shows execution time and speedup using Galois on the medium test case. It can be observed that a significant slowdown occurs when moving from 10 to 16 threads as the application is run on a I2PC machine which has 10 cores per processor. The application does not efficiently handle the transition from execution on a single processor to execution on multi-processors with a NUMA memory layout. The improvement from serial allocation to parallel allocation is significant, however the performance of sorted *sft* and unsorted *ftp* parallel allocation is not significant.

The allocation of the mesh and nodes using 16 threads for the medium test case is shown in figures 7.12 and 7.13. Although the initial distribution maximizes locality, the mesh loses this upon refinement. Refinement increases the number of triangles significantly and Galois redistributes the additional triangles uniformly for load balancing. However, as the refinement is progressive, this allocation becomes random and increases memory accesses across NUMA domains.

The execution time and speedup for both the test cases is shown in figure 7.14 using the space filling sorted version *sfp*. It can be observed that although the speedup does not increase proportionally, speedup is significantly improved in the large test case compared to the medium test case. The size of the mesh is about 5 times larger, increasing the work per thread per iteration and reducing the overheads. Also, it has been observed that Galois performance degrades for higher number of iterations, which is the case with $80,000$ iterations in the medium case where as only $31,000$ iterations in the large test case. A step-up in execution time from 10 threads to 16 as observed in the medium test case is absent in the large test case. However, performance flattens out in this region and improves again for higher number of threads. This behavior is similar to the observed behavior in the analysis of irregular data structures in chapter 6, where alois handles larger data structures better.

Comparison of OpenMP and Galois

A comparison of performance using OpenMP and Galois is shown in figure 7.15. It can be observed that OpenMP performs better than Galois through out for the large test case, where as it has similar performance for the medium test case up to 4 threads but diverges later.

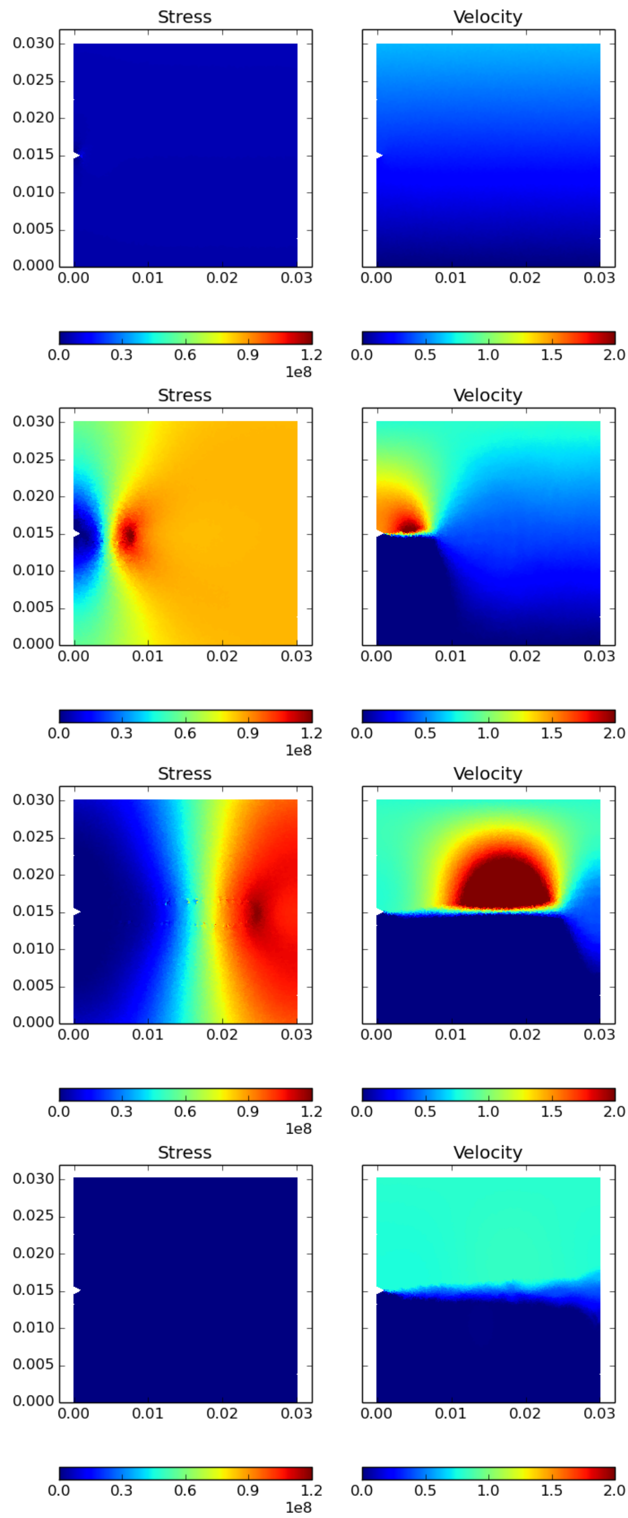


Figure 7.7: Stress and velocity progress at different times for the medium test case

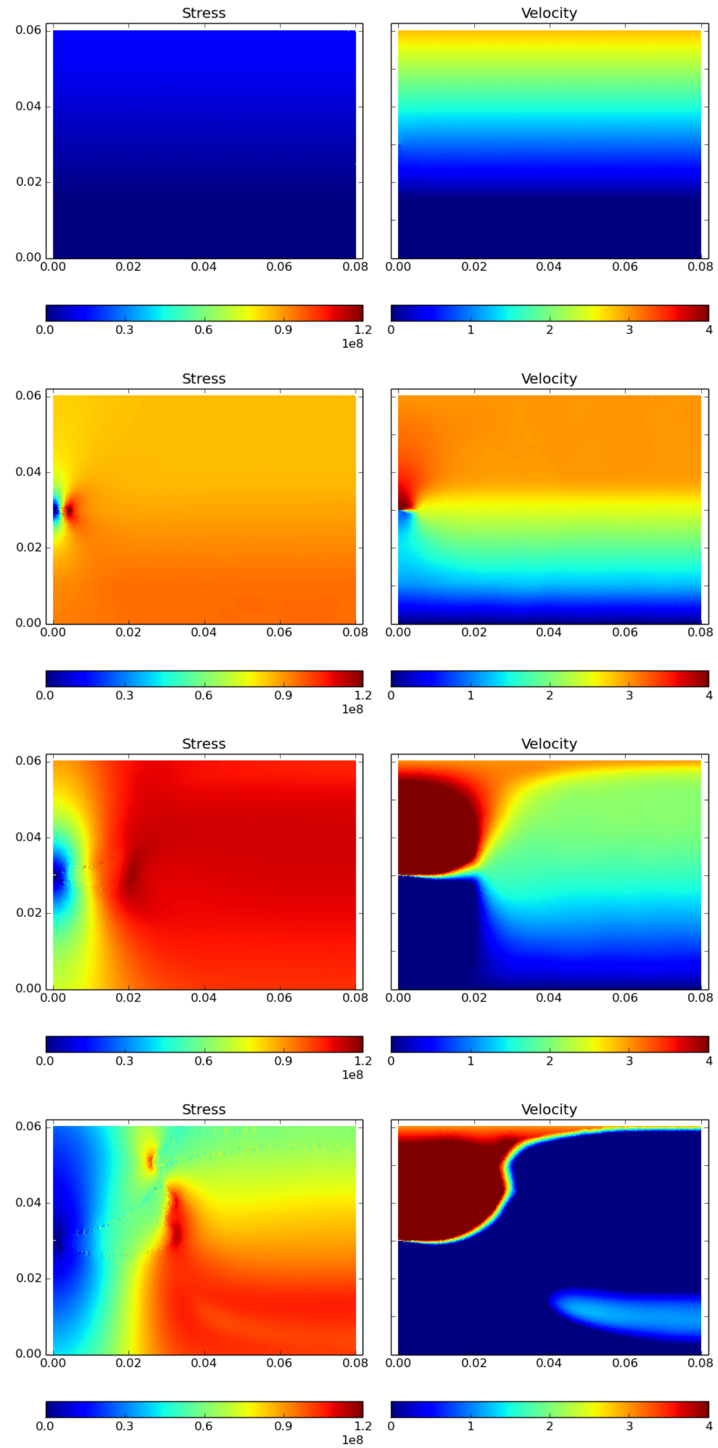
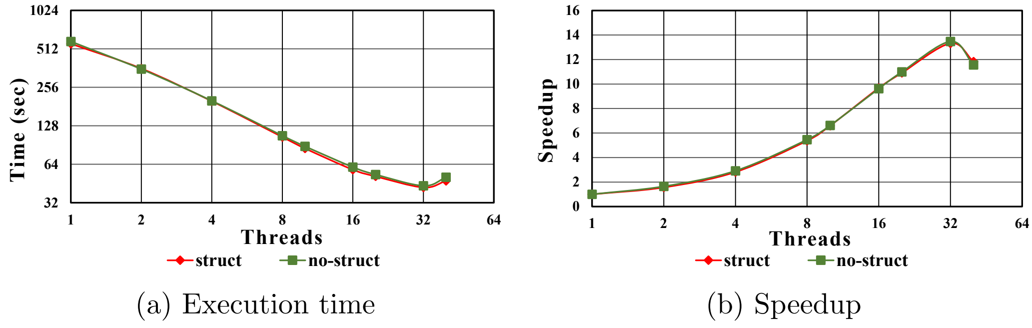
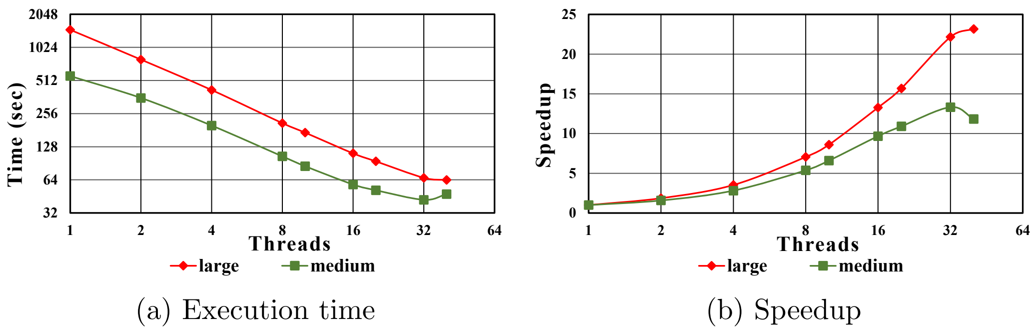


Figure 7.8: Stress and velocity progress at different times for the large test case



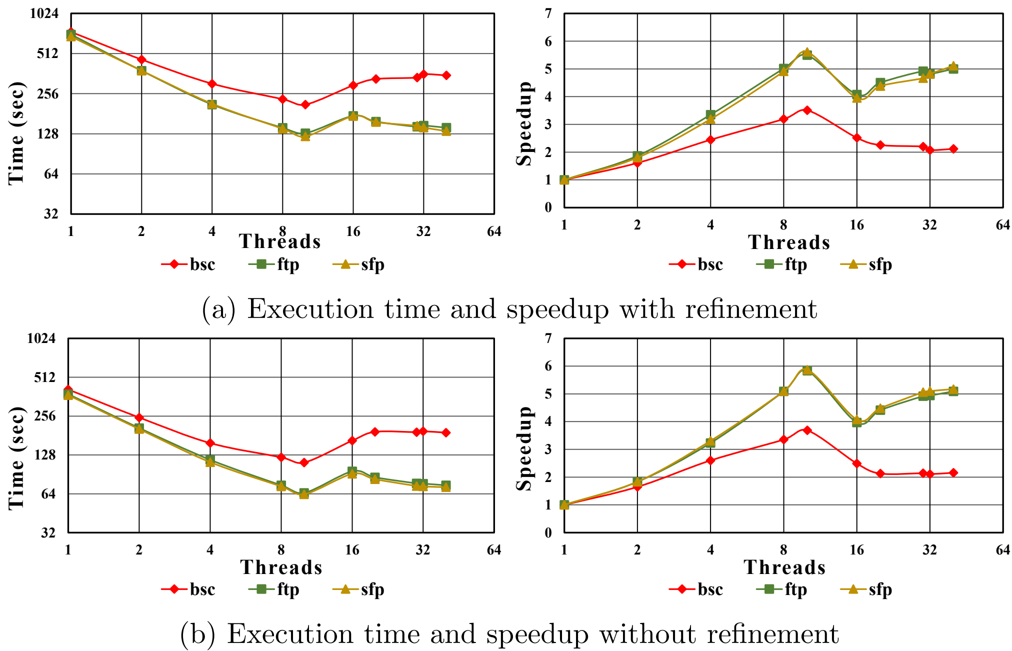
(a) Execution time (b) Speedup

Figure 7.9: Performance of Dynamic Fracture Propagation using OpenMP on medium test case



(a) Execution time (b) Speedup

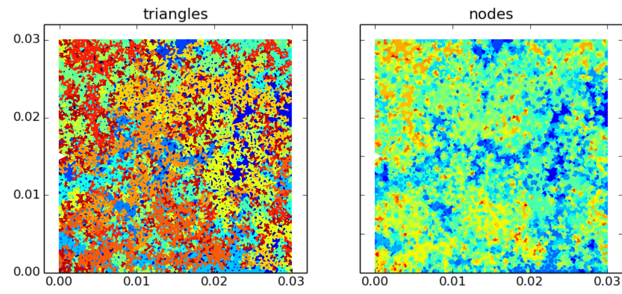
Figure 7.10: Performance of Dynamic Fracture Propagation using OpenMP on medium test case



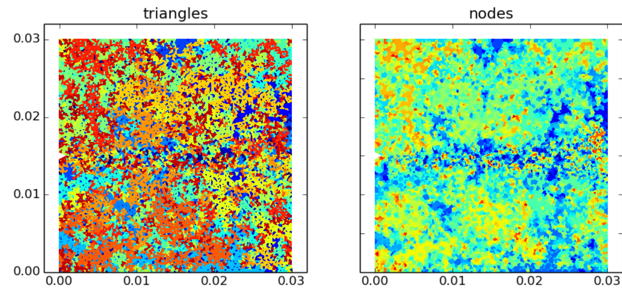
(a) Execution time and speedup with refinement

(b) Execution time and speedup without refinement

Figure 7.11: Performance of Dynamic Fracture Propagation using Galois on medium test case

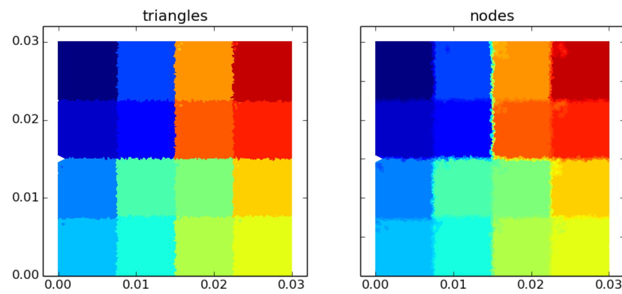


(a) Initial allocation

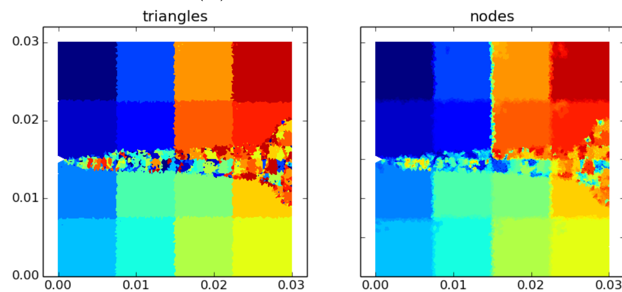


(b) Final allocation

Figure 7.12: Allocation of nodes and triangles for 16 threads with parallel allocation

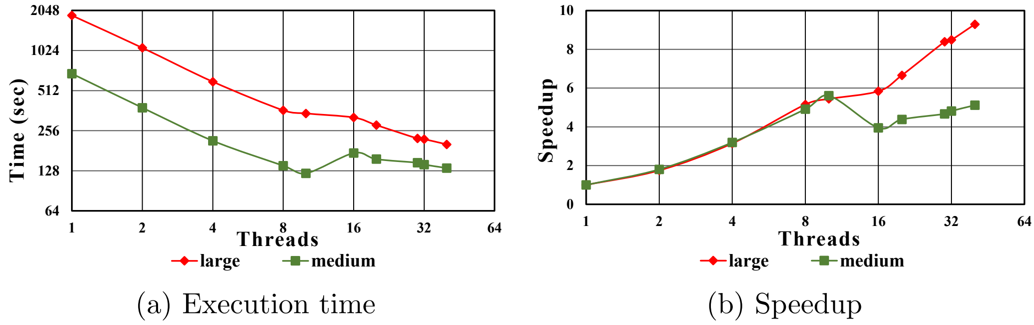


(a) Initial allocation



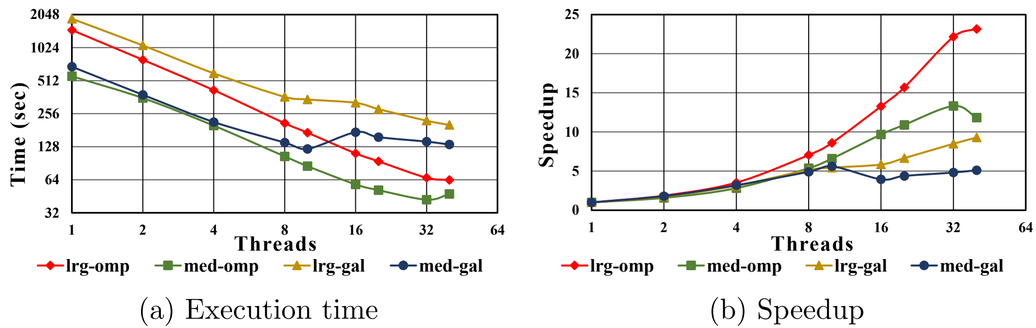
(b) Final allocation

Figure 7.13: Allocation of nodes and triangles for 16 threads with space-filling sorted allocation



(a) Execution time (b) Speedup

Figure 7.14: Performance of Dynamic Fracture Propagation using Galois



(a) Execution time (b) Speedup

Figure 7.15: Performance of Dynamic Fracture Propagation using Galois and OpenMP

CHAPTER 8

COMPARISON OF PROGRAMMING WITH OPENMP AND GALOIS

Parallel frameworks need to be easy to use and powerful for them to be easily adopted by the programming community. An evaluation of ease of programming with OpenMP and Galois is discussed with respect to two aspects, parallelizing an existing serial code and writing parallel code from scratch.

Table 8.1 qualitatively compares programming with OpenMP and Galois and table 8.2 demonstrates a quantitative comparison based on the dynamic fracture simulation application.

8.1 Parallelizing existing serial code

Existing serial code is generally not written with parallelization in consideration. This, in itself, creates a few difficulties irrespective of the parallelization framework adopted. Excessive use of global variables that are modified frequently may cause incorrect parallel executions, leading to bugs. Identification of data dependencies and isolation of race conditions is a tedious task for a large code base. Some algorithms are inherently suitable only for serial execution and perform poorly in parallel. However, rewriting a huge code base is not generally an option.

OpenMP provides non-intrusive compiler directives which do not call for any modification of existing data structures or major code refactoring. Time-critical sections of the code can be identified and parallelized with a few simple directives, assuming the data dependencies allow for parallelization. Galois, on the other hand, works on functors or function pointers and code needs to be refactored to obtain parallelism. Galois works best with its built-in data structures and modifying these in a huge code base is often cumbersome.

8.2 Writing parallel code from scratch

It is beneficial to write code with a parallel perspective from the outset. This allows for cleaner data structure layouts and algorithm choices. However, programming with OpenMP does not have significant difference from sequential programming apart from the compiler directives and runtime calls. Galois provides an extensive library of parallel classes and routines which can be readily used. Synchronization and scheduling are handled by the Galois runtime and the programmer only needs to work on the functionality on the program.

8.3 Qualitative comparison of OpenMP and Galois

Table 8.1: Programming with OpenMP and Galois - a qualitative comparison

Criteria	OpenMP	Galois
Regular data structures	Easy	Difficult
Irregular data structures	Difficult	Easy
Major development time	Synchronization	Writing suitable functors
Code Readability	Less	More
Debugging serial code	Similar	Similar
Debugging parallel errors	Difficult	Easy
Learning curve	Gentle	Steep
Resources for learning	Plenty	Limited

Table 8.1 shows a few criteria to compare and contrast OpenMP and Galois. Parallelization of regular data structures is easy with OpenMP as this data is typically accessed in a loop and OpenMP has a rich feature set to handle general data dependencies in these cases. Galois, on the other hand, requires more complex parallelization methods irrespective of using their `LargeArray`.

Parallelization of irregular data structures is easier to handle in Galois as Galois provides a set of standard irregular data structures which are easy to use and it handles synchronization in its runtime. OpenMP on the other hand needs explicit locks or constructs to avoid race conditions, some of which are not apparent in an obvious manner.

Major programming time using OpenMP is spent in synchronization strategies and choosing the data structure layout, specially for irregular applications. Major programming time using Galois is spent in writing the functors. Although the functionality is same in OpenMP and Galois, Galois classes must be properly encapsulated for best results and this requires more programmer time. Although increased readability is a bonus.

OpenMP code with the numerous synchronization constructs is not easily readable whereas the abstractions in Galois provide an easy to read code structure. However, OpenMP code is easier to read for regular data structures as loops are explicitly declared, clearly depicting the behavior of the code. Galois, using functors, makes it difficult to get the big picture.

Debugging serial code or execution of code with a single thread running has nearly equal difficulty using both frameworks. Also, it is no worse than debugging without any parallel framework. Debugging parallel execution is more tricky due to non-deterministic behavior. Galois supports deterministic parallel execution at the cost of performance which can be used to identify reproducible bugs. In OpenMP programs, synchronization errors are most common as the programmer is responsible for correct race-free memory accesses.

OpenMP has simple interfaces and is easy to achieve modest speedups with relatively little effort. Galois on the hand has a steep learning curve to familiarize oneself with the data structures and functionalities of the different constructs before achieving desirable performance gains.

OpenMP is extensively used and has a wide range of online resources for both beginners and advanced users. Galois has limited examples and provides only auto generated documentation online. However, it must be noted that Galois users group mailing list is active and queries get prompt and helpful responses.

8.4 Quantitative comparison of OpenMP and Galois

A quantitative comparison of OpenMP and Galois is presented in table 8.2 using the Dynamic Fracture Simulation application. The total lines include blank spaces and comments. C++ code, C++ header code and total code are obtained using `cloc`. The compiler directives in OpenMP include all

Table 8.2: Programming with OpenMP and Galois - a quantitative comparison

Criteria	OpenMP	Galois
Total lines	1582	2139
C++ code	1179	756
C++ header code	57	828
Total code	1236	1584
Compiler directives	29	N/A
Get calls	N/A	61
Executable size	0.4 MB	6.7 MB

`pragma` calls. These do not include other OpenMP library routine calls such as `set_omp_num_threads`. The Get calls field for Galois lists the number of calls made to the runtime system to obtain an object. These include all the calls, irrespective of whether contention is checked or not.

Galois has more lines of code and also uses an elaborate Galois code base and Boost libraries. The executable generated is 16 times the size of the OpenMP executable. Quantitatively, OpenMP provides a more concise solution, both in terms of code volume and deliverable size.

CHAPTER 9

CONCLUSION

This thesis presented an evaluation of two parallel frameworks, OpenMP and Galois. A detailed study of optimization in memory allocation and usage have been performed for both OpenMP and Galois along with a few other optimization analyses for OpenMP. It has been found that allocating memory in parallel benefits both OpenMP and Galois. Sorting data using a space-filling curve before allocation in parallel provides the best performance for irregular data structures in both OpenMP and Galois. Also, naïve parallelization of loops using OpenMP definitely has poorer performance, but explicitly parallelizing loops does not yield a significant benefit. Memory is a bottleneck for parallelization of simple tasks dealing with large data sets. Galois performs better for applications with small number of iterations on large data rather than those that have large number of iterations over moderate data size.

It has also been found that performance for parallelizing existing serial code HARM using both OpenMP and Galois have similar benefits. Galois provided the best speedup for Delaunay triangulation with about 30x speedup for 40 threads as opposed to about 20x for OpenMP. Galois has performed well for small number of iterations over a large amount of data, and it has been true with Delaunay triangulation as well. OpenMP, on the other hand, performed better in Dynamic Fracture Propagation with a speedup of 23x as compared to 9x using Galois. Dynamic fracture propagation has a large number of iterations with moderately sized data which could be the reason for Galois not performing on par with OpenMP.

In conclusion, OpenMP is a simple and effective parallel framework that can achieve reasonable performance with relative ease. However, OpenMP is more suited for regular data structures or iterations over irregular data structures but not for mesh refinements and dynamically updated workloads. Galois provides a reasonably easy to use framework to parallelize applications with irregular data and access patterns. Galois is not inherently suitable for

regular structures. Both the frameworks have their own advantages and disadvantages and handle almost disjoint set of work loads. Other parallel frameworks need to be explored to find one that is easy to use and powerful to handle a majority of workloads.

REFERENCES

[1] F. Mueller, "A library implementation of POSIX threads under UNIX," in *In Proceedings of the USENIX Conference*, 1993, pp. 29–41.

[2] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216.

[4] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[5] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 211–222.

[6] C. F. Gammie, J. C. McKinney, and G. Toth, "HARM: A Numerical scheme for general relativistic magnetohydrodynamics," *Astrophys. J.*, vol. 589, pp. 444–457, 2003.

[7] OpenMP Architecture Review Board, "OpenMP application program interface version 3.1," 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[8] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 12–25.

[9] S. C. Noble, C. F. Gammie, J. C. McKinney, and L. Del Zanna, "Primitive variable solvers for conservative general relativistic magnetohydrodynamics," *Astrophys. J.*, vol. 641, pp. 626–637, 2006.

[10] Astrophysical Fluid Dynamics Group, University of Illinois, Urbana-Champaign, "Astrophysical code library." [Online]. Available: http://rainman.astro.illinois.edu/codelib

[11] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 68–70.

[12] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[13] S. Mangala, T. Wilmarth, S. Chakravorty, N. Choudhury, L. Kal, and P. Geubelle, "Parallel adaptive simulations of dynamic fracture events," *Engineering with Computers*, vol. 24, no. 4, pp. 341–358, 2008.

[14] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, pp. 40:40–40:51, July 2013.

[15] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: ACM, 1999, pp. 215–228.

[16] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*, ser. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2012.

[17] L. Guibas, D. Knuth, and M. Sharir, "Randomized incremental construction of delaunay and voronoi diagrams," *Algorithmica*, vol. 7, no. 1-6, pp. 381–413, 1992.