# Survey and Performance Evaluation of Parallel Codes

**master thesis in computer science**

by

**Sandro Kofler**

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Dr. Radu Prodan, Institute of Computer Science

**Innsbruck, 17 August 2015**

# Certificate of authorship/originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Sandro Kofler, Innsbruck on the 17 August 2015

### Abstract

Fully utilizing the potential of parallel architectures is known to be a challenging task. In the past the software developer had to deal with this challenge using a variety of specialized programming models. The Insieme compiler, currently under development by the Distributed and Parallels Systems group at the University of Innsbruck, tries to support developers in this challenging task. The compiler automatically optimizes parallel applications for the execution on heterogenous and homogenous systems. In this master thesis the development of a testing framework used in the Insieme compiler project is shown. Input codes for integration testing and permanent monitoring of the compiler performance were collected. An integration test framework to automatize the execution of test codes was built. Each test code was classified by several metrics (e.g. scalability, memory consumption), those metrics are useful to optimize the compilation process for a high number of miscellaneous test codes.
The second part of the thesis shows a performance analysis of a product application. The analysis focuses on the scalability on parallel shared memory systems. Proposals to speed up the application are made.

# Contents

**1. Introduction** **9**
- 1.1. Motivation . . . 9
- 1.2. Related Work . . . 10
- 1.3. Insieme . . . 10
- 1.4. Integration Test Framework . . . 11
- 1.5. Input Codes . . . 12
- 1.6. Performance Analysis . . . 12

**2. The Insieme Compiler** **13**
- 2.1. INSPIRE . . . 14
  - 2.1.1. Architecture . . . 14
  - 2.1.2. Parallel Model . . . 14
- 2.2. Insieme Runtime System . . . 16
  - 2.2.1. Topology-aware Multi-Process Scheduling . . . 16
  - 2.2.2. Automatic Loop Scheduling . . . 17
  - 2.2.3. Optimizing Granularity in Task-based Parallelism . . . 17

**3. Overview of all Gathered Codes** **19**
- 3.1. Code Finding Procedure . . . 19
- 3.2. Input Codes . . . 20

**4. Test Environment** **33**
- 4.1. Hardware and Software Environment . . . 33
- 4.2. Metrics . . . 33
  - 4.2.1. Runtime and Memory . . . 34
  - 4.2.2. Lines of Code . . . 34
  - 4.2.3. Linux Perf Tool . . . 34
  - 4.2.4. Floating Point Operations . . . 35
  - 4.2.5. Memory Transfer . . . 36
  - 4.2.6. Boundness . . . 36
  - 4.2.7. Callgrind . . . 39

**5. Test Results** **41**

5.1. Shared Memory Parallelization Performance . . . . . . . . . . . . 41
5.2. Main Memory Footprint . . . . . . . . . . . . . . . . . . . . . . . 45
5.3. Boundness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**6. Detailed Analysis of Selected Codes** **51**
6.1. Amdahl's law . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.2. BOTS Health . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
6.3. OmpSCR Graphsearch . . . . . . . . . . . . . . . . . . . . . . . . 54
6.4. PARSEC Blackscholes . . . . . . . . . . . . . . . . . . . . . . . . 56
6.5. Dijkstra . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.6. NPB cg . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.7. Rodinia nn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
6.8. Rodinia bfs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

**7. Integration Test Framework** **63**
7.1. Test Database . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
7.1.1. Configuration File Options . . . . . . . . . . . . . . . . . . 64
7.2. Test Steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
7.3. Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
7.4. Output Formats . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
7.4.1. SQL Database . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.5. Command Line Arguments . . . . . . . . . . . . . . . . . . . . . . 70
7.6. Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . 71

**8. Detailed Performance Analysis at the Example of GALPROP** **79**
8.1. Testing Environment . . . . . . . . . . . . . . . . . . . . . . . . . 79
8.2. Current Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
8.2.1. Code Regions . . . . . . . . . . . . . . . . . . . . . . . . . 82
8.3. Improvements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
8.3.1. Improvements Regarding Shared Memory Systems . . . . 85
8.3.2. Serial Improvements . . . . . . . . . . . . . . . . . . . . . . 89
8.4. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
8.5. Overhead Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
8.6. Further Improvements . . . . . . . . . . . . . . . . . . . . . . . . . 95

**9. Summary** **97**

**A. Input Parameters for GALPROP** **99**
A.1. NoGamma Low Resolution input file . . . . . . . . . . . . . . . . . 99
A.2. WithGamma Low Resolution input file . . . . . . . . . . . . . . . . 107

**B. Runtime Measurements of the Optimized Version of GALPROP** **109**

**List of Figures** **111**

**List of Tables** **113**

**Bibliography** **115**

# Chapter 1.

# Introduction

This master thesis consists of two main parts.
The first part is about the search of input codes for the Insieme Compiler project. The Insieme Compiler is currently under development by the Distributed and Parallels Systems group at the University of Innsbruck. Methods used to create an integration test database for the compiler and to classify integration tests are shown. An application to execute and verify the compilation phase of all codes in the database was created. The, so called, Integration Test Framework is also capable of measuring code attributes like main memory consumption or execution time. The test framework provides an automatized way to collect performance metrics and compare results of different Insieme versions or reference compilers.
The second part shows a performance analysis of a product application from the field of astrophysics (GALPROP). The code was developed by the University of Stanford and does not scale well on shared memory parallel systems. The reasons why the application underperforms are shown and some small code improvements are presented.

## 1.1. Motivation

During a compiler development project it is important to maintain a large amount of input codes which act as integration tests. The purpose of such an input code database is to ensure that the compiler supports a high variety of language aspects and creates correct results. Since Insieme is an optimizing compiler it is important to constantly measure and compare the performance of test codes. An increase in performance indicates a progress in the Insieme development process.

## 1.2. Related Work

There exist numerous frameworks for integration testing. However most of them are designed to only execute tests without the ability of collecting performance data. On the other hand performance data frameworks are only designed to analyze a single application in detail. They are not intended to maintain a code database containing hundreds of codes and execute them in an automatized way.
A mentionable framework is PerfDMF presented in [1]. PerfDMF is a parallel performance data management framework. It deals with the problem of maintaining a database of performance evaluations of parallel systems. It uses third party profiling tools to measure performance data and handles the results. Primary objectives of PerfDMF are import/export from/to leading profiling tools and handling a large-scale profile data and a large number of experiments. The similarities to this work are automatized profiling and the possibility to maintain a database for the performance results. Both solutions do not implement the profiling itself, both use external tools to gain the results. However PerfDMF is not intended to be used with a big code database. It focuses on a detailed analysis of one application and comparing different versions of it. Whereby our solution only deals with a limited amount of metrics especially concerning parallel execution.
Most of the typical integration test frameworks only execute tests and check the results. Our framework additionally maintains a code database containing test applications. The database contains information how to compile and execute the codes. Therefore it does not make sense to compare our framework with another typical solution. We did not find any related work dealing with this type of integration testing.

## 1.3. Insieme

The main goal of the Insieme project of the University of Innsbruck is to research ways of automatically optimizing parallel programs for homogeneous and heterogeneous multi-core architectures and to provide a source-to-source compiler that offers such capabilities to the user. The core features of the Insieme Project are [2]:

- Support for multiple programming languages and paradigms such as C, Cilk, OpenMP and OpenCL (C++ and MPI support is under development),

- multi-objective optimization techniques supporting objectives such as execution time, energy consumption, resource usage efficiency and computing costs,
- the Insieme Runtime System which provides an abstract interface to the hardware infrastructure, offering online code tuning and steering, dynamic reconfiguration of hardware resources and monitoring of the application's performance,
- an input code independent Intermediate Representation (INSPIRE) for developing new compiler techniques to optimize parallel programs,
- a rich analysis and transformation toolbox which operates on INSPIRE and aims to maximize developer productivity when researching new optimizations and
- deep integration between the compiler and its associated runtime system, allowing the convenient exchange of arbitrary meta-information for novel combined optimization strategies.

## 1.4. Integration Test Framework

The Integration Test Framework was built to automatically test the Insieme Compiler. By using the framework it is easy to maintain a high amount of miscellaneous test codes. The code database provides all information to

- compile test codes using
  - a reference compiler and
  - the Insieme Compiler,
- execute the tests,
- and to compare results.

The codes act as integration tests and are executed regularly during the development process of Insieme.
To automatically maintain a database of code characteristics the integration test tool was enhanced to raise code metrics. Such metrics are for example the performance of codes using parallel shared memory systems or the lines of codes. Details about the Integration Test Framework are described in Section 7.

## 1.5. Input Codes

A big part of this thesis consists of the search of input codes for the Insieme Compiler project. Codes are listed and some characteristics of them are shown. The selection process was focused on codes using OpenMP or MPI parallel constructs. But also simple benchmark codes (e.g. the stream benchmark) or codes used in production (e.g. GALPROP, sect. 8) are included. The most important code statistics gathered are:

- run time (wall and CPU time),
- main memory footprint,
- parallel speedup/efficiency and
- no. of parallel constructs.

All these metrics are collected using a reference compiler and the Insieme Compiler. Additionally, the number of threads as well as the OpenMP scheduling variants are varied. To collect results efficiently a test tool was created to automatically execute all tests and get the results. The tool is able to parse the code suite and execute each test using a reference compiler and the Insieme Compiler. The results are either available as an SQL script to generate a database, or as a simple CSV file. By using the tool it is easy to add/remove codes, steer the execution of codes and compare different versions of Insieme.

## 1.6. Performance Analysis

The last section of this thesis shows the results of performance analysis of an application in public use. The application is the GALPROP code developed by the University of Stanford. GALPROP is a numerical code for calculating the propagation of relativistic charged particles and the diffuse emissions produced during their propagation. A deep analysis of the code was done and improvements regarding parallel shared memory systems are proposed. The results of a detailed overhead analysis are shown at the end of the performance analysis chapter (Section 8).

# Chapter 2.

# The Insieme Compiler

Fully utilizing the potential of parallel architectures, especially hybrid systems using multi-core CPUs, GPUs and distributed memory systems is known to be a challenging task. In the past the software developer had to deal with this challenge using a variety of specialized programming models. For this purpose a set of standardized APIs and language extensions was developed. Some important APIs in this context are OpenMP, Cilk, MPI and OpenCL. In recent systems combinations of these technologies became necessary and hybrid programming models such as MPI/OpenMP and OpenMP/OpenCL were established. This led to a growing effort for the developer to fully exploit all levels of parallelism provided by the underlying system.

The Insieme Compiler supports the developer in this challenging task, it tries to optimize a parallel code for a given system. Unlike conventional instruction level manipulation this kind of analyses and transformations are performed at a higher level of abstraction. Therefore the Insieme Compiler was designed as a source-to-source compiler based on a high-level intermediate representation (INSPIRE, see 2.1). The benefit of this intermediate representation is that it is able to model the parallel control flow. This is needed to optimize the source for parallel execution [3].

The Insieme Compiler supports multiple input languages and standards such as C, C++, OpenMP, Cilk, OpenCL and MPI. All input codes are transformed to INSPIRE. The output programs use the Insieme Runtime System (Insieme-RS, see 2.2) to enable and manage parallel execution and perform online code tuning and steering [4].

Figure 2.1 shows an example setup of the Insieme infrastructure for processing parallel codes. Input programs written using C/C++ based OpenMP, Cilk, OpenCL or MPI constructs are parsed using a Clang [5] based frontend and converted into INSPIRE. Using this representation analysis and optimizing is done using a growing set of re-usable tools. In the final compilation phase the optimized intermediate representation is converted again into source code. Different backend implementations, for parallel as well as sequential execution,

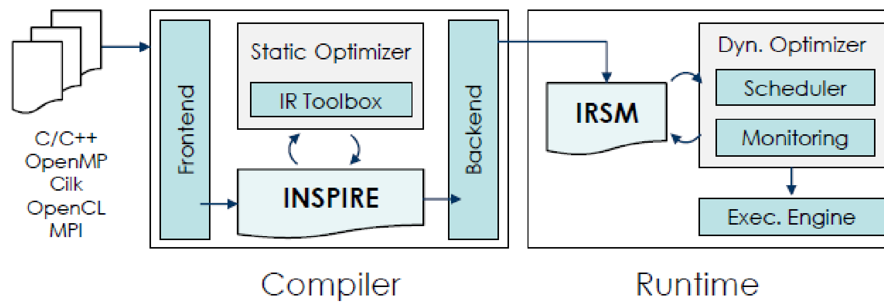


Figure 2.1.: Example setup of the Insieme infrastructure[3]

are provided for this last step. A frequent use case is based on the generation of multi-versioned code, based on the concrete scenario the optimal version is selected dynamically during run time. Beside this selection mechanism the runtime system also deals with work-load scheduling, data distribution and resource allocation issues [3]. For parallel backends the runtime system uses its own application model to handle parallel execution and synchronization.

## 2.1. INSPIRE

### 2.1.1. Architecture

This section shortly introduces into INSPIRE, the Insieme Parallel Intermediate Representation [3]. INSPIRE is a formal intermediate language which is capable of modeling heterogeneous parallel applications using a single, unified and small set of constructs. Parallel aspects of input codes are modeled at the language level, so it becomes possible for optimization utilities built on top of INSPIRE to handle parallelism natively.

### 2.1.2. Parallel Model

Below a short introduction into the parallel control flow representation of INSPIRE is given, for a deeper insight on INSPIRE see [3]. The parallel model used within INSPIRE covers the full flexibility of common parallel languages. It is based on nested thread groups. In fig 2.2 the execution of a parallel application is shown. On the top level a thread group consisting of two threads is shown. The first of these threads spawns an inner thread group consisting of three threads. The second thread creates an inner thread group which spawns

two additional groups, each consisting of a single thread. In this context the term thread is used as an arbitrary entity capable of processing a sequential control flow. The INSPIRE parallel model does not distinguish between OS-level threads, OpenMP threads, Cilk tasks, OpenCL work items, MPI processes or any other processing entities.

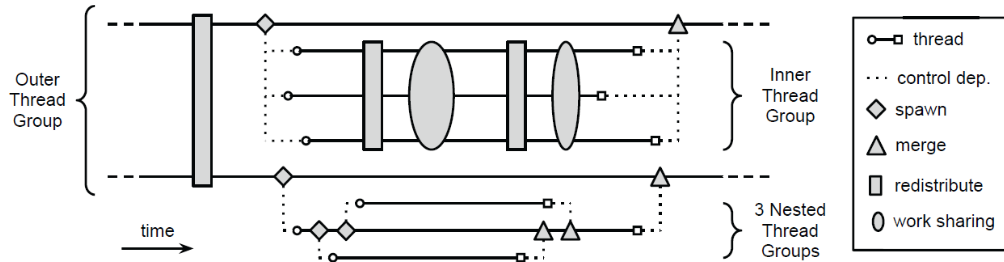


Figure 2.2.: Example execution of a parallel control flow in INSPIRE [3]

The main concepts of the parallel model of INSPIRE are:

### Jobs

Each thread group cooperatively processes a job. Each job object contains

- a set of job-local variables,
- a function to be applied to this variables
- and upper and lower boundaries for the number of threads required.

### Thread Identification

All threads within a group evaluate the call of a function, they are processing the same sequential code. Therefore all threads need to be indexed to diverge execution traces. Functions to determine the thread Identification are $getThreadID$ and $getNumThreads$.

### Spawning and Merging

It is possible to create sub-thread groups (and merge them again), appropriate functions are *spawn*, *merge* and *mergeAll*.

### Inter-Thread Communication

For inter-thread communication three primitives are offered:

1. *pfor*
   This primitive is named after its most prominent use case - the parallel for. It takes an iterator representing a set of input values as well as a function capable of processing those values. All threads within a group have to call this primitive, the work is distributed among the specified range.

2. *redistribute*
   This primitive corresponds to the scatter/gather primitives in MPI. Similar to *pfor* this is a collective operation and needs to be invoked by all threads.

3. Point-to-point communication
   For the direct communication between threads the concept of *channels* is used. Several operations like *channel.create*, *channel.send*, *channel.recv* are provided.

## 2.2. Insieme Runtime System

The following section provides a short introduction to the Insieme Runtime System (InsiemeRS) and its major optimizing techniques. The aim of InsiemeRS is to provide an environment for the execution of a parallel program specified via INSPIRE and compiled by the Insieme Compiler. To execute a program within InsiemeRS it has to be converted to INSPIRE and customized by the Insieme Compiler. One of the essential features of InsiemeRS is the close integration with the Insieme Compiler. This integration allows to forward meta information such as static analysis results from the compiler to the runtime system. Using this meta-information, dynamic knowledge only available during program execution, such as the values of program variables and input data sizes, can be combined with the results of static compiler analysis to yield better scheduling decisions. Below the major optimization techniques used in InsiemeRS are mentioned, for details see [4].

### 2.2.1. Topology-aware Multi-Process Scheduling

The number of cores in shared memory systems is currently rising sharply. New system topologies are often complex, using a hierarchy of multiple cache levels. It becomes harder and harder for developers to optimize parallel programs for

such complex systems. To overcome this problem a centralized process-level scheduling of multiple OpenMP workloads (jobs) was developed. The technique takes available topology information into account and is applicable without any changes required from the user [4].

### 2.2.2. Automatic Loop Scheduling

Loop parallelism is an important part of many OpenMP programs, therefore an optimal mapping of parallel loop iterations to threads and cores is essential. This is done by choosing the right loop scheduling model, based on several environment parameters as well as OpenMP loop characteristics.

### 2.2.3. Optimizing Granularity in Task-based Parallelism

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [4]. Like loop parallelism task parallelism is relatively easy to implement using OpenMP. But it is challenging to achieve good efficiency and scalability. The central point in this parallelization strategy is the task granularity. The granularity of tasks is defined by the length of execution time of a single task between interactions with the runtime system. Short-running tasks lead to a loss in efficiency due to runtime overhead associated with generating and launching a task. On the other hand long-running tasks minimize overhead but are hard to schedule efficiently and may therefore fail to scale on large systems. InsiemeRS implements a way to find the optimal task granularity to achieve a better performance.

# Chapter 3.

# Overview of all Gathered Codes

In this section a basic overview over all gathered codes is shown. First, the methodology used to find new input codes is described and afterwards all codes are listed and the basic code behavior is shown.

## 3.1. Code Finding Procedure

To find appropriate input codes simple methods like web search, supercomputing research group homepages or benchmark suites were used. The main focus for new input codes for the Insieme project is set on parallel codes using either OpenMP, MPI or OpenCL constructs. Additionally, big codes containing a huge amount of lines and codes using the new C++ standard are relevant. To get a good overview of the capabilities of Insieme also parallel codes which do not perform well are required. Another aspect for the codes is their use as integration tests, they are used to continually test the Insieme Compiler during its development. The results of an Insieme run are compared to the results of a reference compiler to proof that the compiler preserves semantic correctness. The last reason to maintain a large code base is to keep track of features that are not yet implemented in the Insieme Compiler. To get a good coverage of all possible code constructs it is necessary to test a large amount of well-known input codes and benchmarks. Therefore about 30 % of the gathered codes are not yet supported by Insieme, however during the development process more and more codes should work and get a better performance.

## 3.2. Input Codes

For brevity, we only focus on the more interesting and complex codes, which we selected according to the following filter criteria:

- At least 50 lines of code,
- at least 1 parallel construct and
- are contained in a given benchmark suite.

A detailed analysis would exceed the scope of this thesis, other codes are only used as integration tests for the compiler.
The tables below show a basic listing of all codes organized in corresponding code suites. In this section only a short insight is given, details of selected codes (based on their runtime, number of OpenMP pragmas and efficiency) can be found in Section 6. The shown metrics as well as the used testing environment are described in Section 4.

### Barcelona OpenMP Task Benchmark Suite (BOTS)

Traditionally, parallel applications were based on parallel loops, only a few applications used other parallelization techniques. With the release of the new OpenMP specification (3.0), task parallelism is supported. Parallel tasks allow the exploitation of irregular parallelism. As a result of the lack of benchmarks using tasks the Barcelona OpenMP Task Benchmark Suite was created. The suite contains a set of applications exploiting regular and irregular parallelism, based on tasks. All applications are available using different implementations regarding their OpenMP task models (task tiedness, cut-offs, single/multiple generators)[6].
The results in this thesis are generated by using version 1.1.2 of the task suite, Table 3.1 contains all used codes. Some codes are present in different variations:

- The *default* version only uses the basic version of the task parallelism.
- The *if_clause* version uses cutoff by if clause.
- The *manual* version uses manual cutoff.

For detailed descriptions about these OpenMP pragmas see [6].

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| alignment (for) | Protein alignment creates tasks inside of an omp for pragma | 900 | 1 | 0,247 | 0,026 |
| fft | Fast Fourier Transform | 5467 | 61 | #NA | #NA |
| fib (if_clause) | Fibonacci | 834 | 14 | #NA | 0,024 |
| fib (manual) | | | | #NA | 0,024 |
| floorplan (if_clause) | Computes the optimal placement of cells in a floorplan | 1165 | 21 | 0,610 | 0,621 |
| floorplan (manual) | | | | 0,592 | 0,760 |
| health (if_clause) | Simulates a country health system | 1221 | 9 | 0,155 | #NA |
| health (manual) | | | | 0,162 | #NA |
| nqueens (if_clause) | Solves the N queens problem | 1043 | 12 | #NA | 0,141 |
| nqueens (manual) | | | | #NA | #NA |
| sort | Uses a mixture of sorting algorithms to sort a vector | 1020 | 14 | 0,301 | 0,326 |
| sparselu (for) | Computes LU factorization of a sparse matrix creates tasks inside of an omp for pragma | 970 | 8 | #NA | #NA |
| sparselu (single) | Computes LU factorization of a sparse matrix creates tasks inside of an omp single | 967 | 8 | #NA | #NA |
| strassen (if_clause) | Computes a matrix multiply using Strassens method | 1510 | 27 | 0,224 | 0,220 |
| strassen (manual) | | | | 0,223 | 0,225 |
| uts | Computes the number of nodes in an unbalanced tree | 1094 | 5 | 0,161 | 0,203 |

Table 3.1.: Codes of the Barcelona OpenMP Task Benchmark Suite

### OpenMP Source Code Repository (OmpSCR)

The OpenMP Source Code Repository (OmpSCR) is a code suite written by the University of La Laguna. The main idea of OmpSCR is to provide the OpenMP users with an infrastructure that allows both to evaluate the performance of OpenMP codes and to compare it across different platforms and compilers. For this many small programs using a low number of OpenMP pragmas were created. All codes are available in Fortran and C, however we only used the C version since Fortran is not supported by Insieme. The repository is designed to be collaborative and incremental, it is open and the number of applications is not limited [7].
In this thesis the version 2 of the OmpSCR is used, Table 3.2 contains all used C codes.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| fft | Fast Fourier Transform using a divide and conquer algorithm. | 431 | 2 | 0,218 | #NA |
| fft6 | Fast Fourier Transform based on Bailey's 6 step FFT algorithm | 634 | 4 | #NA | #NA |
| graphsearch | Path search in a graph | 2039 | 3 | 0,116 | #NA |
| jacobi01 | Solve Helmholtz equation using Jacobi iterative method | 422 | 2 | 0,197 | #NA |
| jacobi02 | | 378 | 3 | 0,195 | #NA |
| loopsWithDepsA1 | Simple loops containing dependencies | 796 | 2 | #NA | #NA |
| loopsWithDepsB | | 792 | 3 | #NA | #NA |
| loopsWithDepsC | | 787 | 2 | #NA | #NA |
| loopsWithDepsD | | 796 | 3 | #NA | #NA |
| LUReduction | LU reduction of a 2D dense matrix | 350 | 1 | 0,173 | 0,241 |
| Mandelbrot | Estimation of the Mandelbrot Set using MonteCarlo sampling | 351 | 1 | 0,615 | #NA |
| MolecularDynamic | Simple molecular dynamics simulation | 446 | 2 | 0,736 | 0,726 |
| Pi | PI generator | 267 | 2 | 0,817 | 0765 |
| QuickSort | Sorting algorithm | 382 | 2 | 0,161 | #NA |

Table 3.2.: Codes of the OpenMP Source Code Repository (OmpSCR)

### OpenMP Validation Suite

The OpenMP Validation Suite is a collection of C and Fortran programs with OpenMP directives that were designed to validate the correctness of OpenMP implementations. The suite was developed by the High Performance Computing Center Stuttgart, the University of Houston and the TU Dresden. The contained benchmarks are designed to cover all features of version 2.5 of the OpenMP standard. Additionally, most of the 3.0 constructs are covered. Currently it is extended to the new 4.0 standard. By now the suite consists of 106 different tests, each testing different combinations of constructs [8].
Since these tests are only small code snippets including only minor amounts of computation, details are not covered in this thesis.

### HPC Graph Analysis SSCA2

The HPC Graph Analysis Benchmark is based on the HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark. The benchmark was created by the graphanalysis.org compendium. It contains multiple kernels accessing a single data structure representing a weighted, directed multigraph. In addition to a kernel to construct the graph there are three additional computational kernels to operate on the graph. Each of the kernels requires irregular

access to the graph's data structure [9].
In this thesis the version 2.2 created in 2007 is used, it is the most resent version containing OpenMP parallelism. Characteristics of the benchmark:

- LOC: 7400
- #OMP: 70
- EffGCC: 0,167
- EffIns: #NA

### ASC Sequoia Benchmark Codes

The Sequoia Benchmark Code Repository is a list of 17 codes developed and maintained by the Lawrence Livermore National Laboratory. The codes are designed to run on the Sequoia system which is a 20 PFLOP/s supercomputer (currently on 3rd place of the Top500 list, June 2014). The benchmarks are available in different versions regarding parallelization (MPI, OpenMP) and the used language (Fortran, Python, C, C++). In this thesis only OpenMP and MPI versions using C and/or C++ are considered, all used benchmarks are listed in Table 3.3.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| stride | Designed to stress the memory subsystem on a node. | 600 | 0 | 0,164 | #NA |
| stride (cachec) | A variant of the stride benchmark. | 150 | 8 | 0,164 | #NA |
| stride (vecopc) | Another variant of the stride benchmark. | 288 | 12 | 0,164 | #NA |
| IRSmk | Intended to be an optimization and SIMD compiler challenge[1] | 370 | 1 | 0,164 | #NA |
| CLOMP_TM | Benchmark to measure OpenMP execution and overhead | 4800 | 20 | 0,164 | #NA |

Table 3.3.: Selected codes of the Sequoia Benchmark Suite

### SMG 2000

The SMG2000 benchmark was created by the Lawrence Livermore National Laboratory. It is a semicoarsing multigrid solver for linear systems. It is a SIMD code written in C that uses MPI and OpenMP as parallelization model. Three versions exist: MPI-only, OpenMP-only and MPI with OpenMP.

[1] This is a customized version, OpenMP parallelism was introduced

In this thesis the OpenMP-only version was used. Characteristics of the benchmark are:

- LOC: 20000
- #OMP: 18

### Application Performance Characterization Benchmarking Map (APEX-Map)

The APEX Map was designed by the Lawrence Berkeley National Laboratory. The main idea for the project was the assumption that each application or algorithm can be characterized by several major performance factors. These factors are specific to the application and independent of the computer architecture. A synthetic benchmark combines these factors to simulate the application's behavior on different test systems. Such a benchmark can be used as a realistic indicator of achievable application performance and enables the users to directly evaluate a new platform based on their own interests [10].
In this thesis the APEX-Map benchmark v1 was used. There are two versions, a sequential one and a MPI-parallelized code. The characteristics are:

MPI version

- LOC: 720
- EffGCC: 0,166

Sequential version

- LOC: 340
- Runtime GCC: 1,740
- Runtime Insieme: 1,730
- EffIns: 0,159
- EffGCC: 0,163

### ExMatEx Codes

ExMatEx means Exascale Co-Design Center for Materials in Extreme Environments, it is located at the Los Alamos National Laboratory. They created a set of so-called proxy apps. The claimed benefits of their proxy apps compared to normal benchmarks only measuring system performance are [11]:

- Well-designed proxy apps present a realistic representation of a specific requirement of a scientific workflow. E.g. a numerically intensive kernel or a kernel using a high amount of I/O operations. Real applications can spawn multiple proxy apps.

- Proxy apps consist of documentation and specification of the problem to solve, plus a reference implementation.
- Proxy apps are designed to be flexible to allow exploration of new programming models and new algorithms in order to learn how these emerging technologies will impact real applications.

In this thesis three proxy applications are reused as input codes for Insieme, their characteristics can be found in Table 3.4.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| COMD (OpenMP) | Molecular dynamics computer simulation | 3025 | 15 | 0,547 | 0,503 |
| COMD (MPI) | | 2969 | MPI | 0,164 | #NA |
| LULESH | Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics | 5992 | 34 | 0,532 | #NA |

Table 3.4.: Proxy apps used from the ExMatEx Suite

### GALPROP

GALPROP is a project of the University of Stanford. It is a numerical code for calculating the propagation of relativistic charged particles and the diffuse emissions produced during their propagation. To use GALPROP for the Insieme Compiler the most compute-intensive part of it was extracted and converted into a test case.
The characteristics of the resulting test kernel are:

- LOC: 337
- #OMP: 4
- EffGCC: 0,981
- EffIns: 0,685

A detailed analysis of parallelization aspects, scalability and improvement potential of GALPROP is shown in Section 8.

### CORAL Benchmarks

The CORAL Benchmark suite was built by the Lawrence Livermore National Laboratory (LLNL), the Argonne National Laboratory and the OakRidge National Laboratory. The purpose of several mini applications and a few larger applications is to measure the performance of their DOE production systems,

namely Sequoia, Mira and Titan. These systems are supercomputers maintained by these laboratories. For Sequoia see also Section 3.2. Mira is currently on the fifth place of the Top500 list, it reaches a theoretical peak performance of 10 PFlop/s. Titan offers even more computational power, it is a 27 PFlop/s machine and contains about 560.000 cores. It reaches rank 2 of the Top500 list [12].
Based on their parallelization aspects (only OpenMP) and their used source code language (only C and C++) some of the CORAL benchmarks are picked out and analysed, the results are shown in Table 3.5.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| AMG2013 | Algebraic Multi-Grid linear system solver for unstructured mesh physics packages | 75.000 | MPI | 0,062 | #NA |
| Graph500 | Scalable breadth-first search of a large undirected graph | 2109 | 1 | 0,363 | #NA |
| CLOMP | Measure OpenMP overheads and other performance impacts due to threading | 1169 | 52 | 0,251 | #NA |
| XSBench | Mini-app representing a key computational kernel of the Monte Carlo neutronics application OpenMC | 1.000 | 5 | 0,593 | #NA |
| HACCmk | Single core optimization and SIMD compiler challenge, compute intensity | 250 | MPI | 0,764 | 0,720 |
| AMGmk | Three compute intensive kernels from AMG | 1.800 | 6 | 0,764 | 0,720 |
| miniFE | Implements a couple of kernels representative of implicit finite-element applications | 50.000 | MPI | 1,244 | #NA |

Table 3.5.: Selected benchmarks of the CORAL benchmark suite [13]

### KineControl

The KineControl project is a cooperation between two institutes of the University of Innsbruck (Department of Geometry and CAD, Institute of Computer Science) and the Institute for Automation and Control Engineering of the Private University for Health and Life Sciences (UMIT) in Hall. It is a project with the goal to develop a new algorithm to calculate the inverse kinematics of a robot. The developed algorithm was implemented in three versions: a C# version, a C++ version and a C version. In contrary to other C++ codes presented in this thesis KineControl contains many C++11 constructs. To test the integration of C++11 in Insieme this version of KineControl was chosen.

The original implementations were already optimized regarding parallelization in [14] and [15]. Characteristics of KineControl:

- LOC: 337
- #OMP: 4
- EffGCC: 0,828

### PARSEC Benchmark Suite

PARSEC stands for Princeton Application Repository for Shared-Memory Computers. The name is already self-explanatory, it is a benchmark suite designed by the Princeton University composed of multithreaded programs. In this thesis only two of the PARSEC benchmarks were used: blackscholes and freqmine. Blackscholes calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. Freqmine employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Itemset Mining (FIMI).

blackscholes

- LOC: 513
- #OMP: 1
- EffGCC: 0,507

freqmine

- LOC: 2300
- #OMP: 4
- EffGCC: 0,308

### Velvet

Velvet is a genome sequence assembler, it uses a modification of the so-called Bruijn graph method. This is one of the benchmarks which represents a black box real world application. The test case uses two parts of the velvet project. Velveth which constructs a test data set, this can be seen as input data for the next step. Velvetg uses this data set and constructs the Bruijn graph and runs simplification and error correction over the graph. The last step is to extract the results. In 2011 velvet was parallelized using OpenMP.

velvetg

- LOC: 26575
- #OMP: 44
- EffGCC: 0,504
- EffIns: #NA

velveth

- LOC: 10242
- #OMP: 19
- EffGCC: 0,253
- EffIns: #NA

### Common OpenMP Example Codes

To not only use benchmarking codes also some common codes parallelized using OpenMP are considered. Mostly these codes are small, simple codes, not optimized to achieve a high speedup for parallel execution. A list of the examples and its characteristics can be found in Table 3.6. The developers of the codes are shown in Table 3.7.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| dijkstra | Creates a random graph and runs dijkstra algorithm | 546 | 4 | 0,232 | 0,174 |
| heated_plate | Solves the steady (time independent) heat equation in a 2D rectangular region | 293 | 10 | #NA | 0,049 |
| prime | Counts prime numbers | 211 | 1 | 0,507 | 0,443 |
| integral_estimation | Estimates the value of an integral | 210 | 1 | 0,901 | 0,829 |
| satisfy | Demonstrates an exhaustive search for solutions of the circuit satisfy problem | 320 | 1 | 0,794 | #NA |
| sgefa | Reimplements the SGEFA/SGESL linear algebra routines from LINPACK for use with OpenMP | 1500 | 1 | 0,204 | 0,146 |
| ziggurat | Uses ziggurat library in conjunction with OpenMP to compute random numbers efficiently and correctly | 1.300 | 8 | 0,204 | 0,201 |
| oddEvenSort | OddEven sorting algorithm | 70 | 2 | 0,058 | 0,031 |
| mergeSort | Merge sort algorithm | 200 | 2 | 0,283 | 0,161 |
| nBody | Naive implementation of the nBody problem | 330 | 5 | 0,236 | 0,275 |

Table 3.6.: List of rudimentary parallelized example codes

| Code | Author | Company |
|---|---|---|
| dijkstra | Norm Matloff, John Burkhardt | CS Dept., UC Davis |
| heated_plate | Michael Quinn, John Burkhardt | |
| molecular_dynamics | Bill Margo, John Burkhardt | |
| prime | John Burkhardt | |
| integralEstimation | John Burkhardt | |
| satisfy | Michael Quinn, John Burkhardt | |
| sgefa | John Burkhardt | |
| ziggurat | John Burkhardt | |
| oddEvenSort | unknown | |
| bitonicSort | unknown | |
| mergeSort | unknown | |
| nBody | Sean Ho | Department of North Carolina |

Table 3.7.: List of example codes and their authors

### Stream Memory Benchmark

STREAM (Sustainable Memory Bandwidth in High Performance Computers) is the de facto industry standard benchmark for measuring sustained memory bandwidth. It was developed by the University of Virginia. There is a sequential version as well as a parallelized version using OpenMP.

sequential

- LOC: 400
- EffGCC: 0,165
- EffIns: 0,265

parallel

- LOC: 230
- #OMP: 11
- EffGCC: 0,550
- EffIns: 0,241

### NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) are a small set of programs designed to evaluate the performance of parallel supercomputers. They are maintained by the NASA Advanced Supercomputing Division. The original suite (NPB-1) released in 1992 contained five kernels and three pseudo-applications. The paper and pencil specification came with Fortran-77 sample codes and two problem sizes (A,B) were defined. The second version (NPB-2) released in 1996 extended NPB-1 by more concrete source code implementations and new problem sizes C and D. Some of the benchmarks were parallelized using MPI. The third version

added three more benchmarks, modern programming flavors like OpenMP, Java and High Performance Fortran were introduced [16].
In this thesis the OpenMP version of NPB-3 is used, results can be found in Table 3.8.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| BT | Block Tri-diagonal solver | 2638 | 58 | 0,234 | 0,173 |
| CG | Conjuate Gradient, irregular memory access and communication | 668 | 40 | 0,168 | 0,236 |
| EP | Embarrassingly Parallel | 314 | 5 | 0,779 | 0,756 |
| FT | Discrete 3D Fourier Transform | 881 | 25 | 0,430 | 0,417 |
| IS | Integer Sort | 502 | 10 | 0,162 | 0,210 |
| LU | Lower-Upper Gauss-Seidel solver | 2694 | 46 | 0,405 | 0,361 |
| MG | Multi-Grid on a sequence of meshes | 944 | 30 | 0,351 | 0,230 |
| SP | Scalar Penta-diagonal solver | 2210 | 77 | 0,168 | #NA |

Table 3.8.: Codes of the NPB Suite

### SNU-NPB Suite

The SNU-NPB Suite is an alternative open source implementation of the NAS Parallel Benchmarks implemented in C, OpenMP and OpenCL. The suite is developed by the Center for Manycore Programming of the Seoul National University. Currently four versions of the suite are available [17]:

- NPB-SER-C: Serial C version, derived from the serial Fortran code developed by NAS.
- NPB-OMP-C: OpenMP version, derived from the OpenMP Fortran code developed by NAS.
- NPB-OCL: OpenCL version, derived from the OpenMP code developed by NAS. Runs with a single OpenCL compute device.
- NPB-OCL-MD: OpenCL version, derived from the MPI Fortran code developed by NAS. Runs with multiple OpenCL compute devices.

In this thesis only the NPB-OMP-C version was used, it contains the codes shown in Table 3.8. For detailed explanation of the benchmarks see the section about NAS Parallel Benchmarks (3.2).

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| BT | Block Tri-diagonal solver | 3940 | 36 | 0,353 | #NA |
| CG | Conjuate Gradient, irregular memory access and communication | 1000 | 19 | 0,101 | 0,121 |
| DC | Data Cube | 2750 | 1 | 0,160 | #NA |
| EP | Embarrassingly Parallel | 300 | 3 | 0,729 | #NA |
| FT | Discrete 3D Fourier Transform | 900 | 6 | 0,552 | #NA |
| IS | Integer Sort | 1000 | 9 | 0,240 | #NA |
| LU | Lower-Upper Gauss-Seidel solver | 4000 | 48 | 0,236 | #NA |
| MG | Multi-Grid on a sequence of meshes | 1300 | 15 | 0,161 | #NA |
| SP | Scalar Penta-diagonal solver | 3350 | 37 | 0,214 | #NA |
| UA | Unstructured Adaptive mesh | 8900 | 91 | 0,277 | #NA |

Table 3.9.: Codes of the SNU-NPB Suite

### Hybrid OpenMP MPI Benchmark (HOMB-MPI)

HOMB is the Hybrid OpenMP MPI Benchmark for testing hybrid codes on multicore and SMP systems. The code is an optimized Laplace Solver on a 2-D grid (Point Jacobi) with optional convergence test. It contains MPI as well as OpenMP constructs. Its characteristics are:

- LOC: 834
- #OMP: 8
- EffGCC: 0,073

### Rodinia Benchmarks

Rodinia is a benchmarking suite composed by the University of Virginia. It focuses on heterogeneous computing infrastructures with OpenMP, OpenCL and CUDA. The Rodinia suite covers a wide range of parallel communication patterns, synchronization techniques and power consumption. It also shows the growing importance of memory bandwidth limitations and the consequent importance of used data layouts [18]. In this thesis we used the OpenMP implementations of the Rodinia kernels, a list can be seen in Table 3.10.

| Name | Description | LOC | #OMP | EffGCC | EffIns |
|---|---|---|---|---|---|
| kmeans | A clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity | 1800 | 1 | 0,164 | #NA |
| backprop | A machine-learning algorithm that trains the weights of connecting nodes on a layered neural network | 600 | 2 | #NA | #NA |
| bfs | Breath First Search graph traversal algorithm | 170 | 1 | 0,166 | 0,163 |
| euler3d | An unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow (different versions) | 470 | 5 | 0,711 | 0,623 |
| euler3d_double | | 470 | 5 | 0,680 | #NA |
| pre_euler3d | | 550 | 6 | 0,647 | #NA |
| heartwall | Tracks the movement of a mouse heart over a sequence of 104 609x590 ultrasound images to record response to the stimulus | 3500 | 1 | 0,164 | 0,633 |
| hotspot | A widely used tool to estimate processor temperature based on an architectural floorplan and simulated power measurements | 250 | 2 | 0,164 | 0,169 |
| lavaMD | Calculates particle potential and relocation due to mutual forces between particles within a large 3D space | 550 | 1 | 0,165 | #NA |
| leukocyte | Detects and tracks rolling leukocytes (white blood cells) in in vivo video microscopy of blood vessels | 3500 | 3 | 0,164 | #NA |
| lud | LU Decomposition | 400 | 2 | 0,393 | 0,356 |
| myocyte | Models cardiac myocyte (heart muscle cell) and simulates its behavior | 2900 | 2 | 0,164 | #NA |
| nn | Finds the k-nearest neighbors from an unstructured data set | 200 | 2 | 0,144 | #NA |
| needle (nw) | A nonlinear global optimization method for DNA sequence alignments | 250 | 2 | 0,165 | 0,152 |
| particlefilter | Statistical estimator of the location of a target object given noisy measurements of that target's location and an idea of the object's path in a Bayesian framework | 600 | 10 | 0,164 | 0,162 |
| pathfinder | Uses dynamic programming to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights | 160 | 1 | 0,166 | 0,161 |
| srad | A diffusion method for ultrasonic and radar imaging applications based on partial differential equations | 640 | 2 | 0,178 | 0,161 |

Table 3.10.: Used Rodinia benchmark kernels

# Chapter 4.

# Test Environment

The following section describes the test environment used to execute the benchmark codes, mentioned in Section 3, and a short overview of the used profiling tools. The last part of the section shows how the classification of codes regarding their CPU and memory boundness was done.

## 4.1. Hardware and Software Environment

The used machine contains four Intel Xeon E5-4650 CPUs. Each CPU contains 8 physical cores and uses hyperthreading technology. Therefore it is possible to execute at most 32 threads without hyperthreading and 64 threads when using hyperthreading. Each core runs on a maximum clock frequency of 2.7 GHz, when using Intel Turbo Boost technology a peak of 3.3 GHz is possible. The L3 cache size of each CPU is 20 MB. Intel claims a maximum memory bandwidth of 51.2 GB per second.
The operating system is CentOS Release 6.5 which is using a Linux Kernel version 2.6.32. As reference compiler GCC version 4.6.3 was used. Threads were bound to their CPU cores using the *GOMP_CPU_AFFINITY* environment variable. The Insieme project does not maintain a versioning system. To give a clue which version of Insieme was used the output of "git describe", which represents some kind of versioning based on the last git commit, is mentioned here. The used Insieme version is v0.5.0-5571-g646d8c9-dirty.

## 4.2. Metrics

This sections shows which metrics were collected, how they are defined and which tools were used to measure them. The metrics are measured for each run of each input code.

### 4.2.1. Runtime and Memory

To measure the runtime and memory consumption of a test run the GNU time tool version 1.7 was used (/usr/bin/time). The time tool provides per-process statistics including

- user CPU time,
- system CPU time,
- walltime,
- memory consumption and
- pagefaults.

Walltime, CPU time and main memory consumption were measured. For the calculation of speedup and efficiency walltime was used.

The exact command used is

```
/usr/bin/time -f WALLTIME%e\nCPUTIME%U\nMEM%M test_run_starter
```

The results are obtained by analyzing the command output using regular expressions.

### 4.2.2. Lines of Code

To count the number of lines the CLOC (Count Lines of Code) perl script was used. It is written by Al Daniel and is published under the GNU General Public License [19]. The tool is able to filter out commented and blank lines and therefore only counts lines containing source code. CLOC automatically detects the used programming language based on file endings and/or code structure. The results are stored into a file in XML format, this file was parsed and results were retrieved.

### 4.2.3. Linux Perf Tool

Perf is a performance analyzing tool included in the Linux kernel since version 2.6.31. It supports hardware and software performance counters, tracepoints and dynamic probes. Performance counters are a set of special registers built into modern microprocessors which store the counts of hardware related activities, e.g. the number of floating point operations. They can be used for

low-level performance analysis. The dynamic tracepoint feature allows to instrument functions and even arbitrary lines of source code recompiling the program. Assuming a copy of the source code is available, tracepoints can be placed anywhere at runtime and the value of variables can be dumped each time execution passes the tracepoint. Dynamic probes is a linux debugger that can be used to insert software probes into executing code modules. When a probe is fired, a probe-handler written in an assembly-like language is executed. Perf is used with several subcommands [20]:

- `stat:` measure total event count for a single program or for the whole system for some time.
- `top:` dynamic view of the most resource consuming functions
- `record:` measure and save sampling data of a single program
- `report:` analyze files generated by record
- `annotate:` annotate source files
- `sched:` tracing/measuring of scheduler actions and latencies
- `list:` list available events

Perf was used to determine floating point operations and memory bandwidth.

### 4.2.4. Floating Point Operations

To count floating point operations (FLOPS) of a test run the linux perf tool was used. It is not trivial to determine the right performance counters to measure floating point operations for a particular machine. For this thesis PAPI[1] event counters were decoded to determine the right hardware counters for the perf tool.
PAPI uses two counters to sum up the number of FLOPS on our Xeon E5-4650 system:

- `FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE:` Number of SSE single precision FP scalar uops executed and
- `FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE:` Number of SSE double precision FP scalar uops executed

[1] PAPI: a portable interface to hardware performance counters on modern microprocessors. It is widely used to collect low level performance metrics.

These counters measure vectore instructions as well as regular floating point instructions. We used these two counters after a short plausibility test using a dummy application. The application computed 2.000.000 floating point additions, the measured perf results varied by 10 FLOPS (±0.05‰).

### 4.2.5. Memory Transfer

To measure the memory bandwidth the amount of memory transferred between the last level cache and the main memory has to be measured. This was done by counting the last level cache misses and multiplying the result by the last level cache line size.
On the used machine the last level cache misses are found in the perf metrics

- `LLC-load-misses` and
- `LLC-store-misses`.

To prove the metrics a dummy application was used. The application stored 90.000.000 integer values in main memory. Test average error of the memory transfer computed based on the measured perf values was ±1.16%.

### 4.2.6. Boundness

The term *boundness* is used in this thesis as the classification for determining whether a code is memory-bound or compute-bound. Memory-bound means that the program performance is limited by the memory bandwidth of a machine. Hence the performance may be enhanced by increasing the memory bandwidth between the last level cache and the main memory. On the other hand an application is compute-bound if its performance is restricted by the computational power of the used machine. Therefore the performance of the application increases if the computational power of the machine is increased.
One approach to classify codes regarding their boundness is the roofline model presented in [21]. This model relates processor performance to memory traffic of an application. The authors use the term "operational intensity" which represents operations per byte of DRAM traffic. This metric is shown in a 2D graph. Roofline models for AMD Opteron systems can be found in Figure 4.1. Figure 4.1a outlines the model for a 2.2 GHz AMD Opteron X2 model 2214 in a dual-socket system. The graph uses a log-log scale. The y-axis represents attainable floating-point performance. The x-axis denotes to operational intensity, varying from 0.25 FLOPS/DRAM byte-accessed to 16 FLOPS/-DRAM byte-accessed. The system has a peak floating-point performance of 17.6 GFLOPS/s and a peak memory bandwidth of 15 GB/s. The horizontal

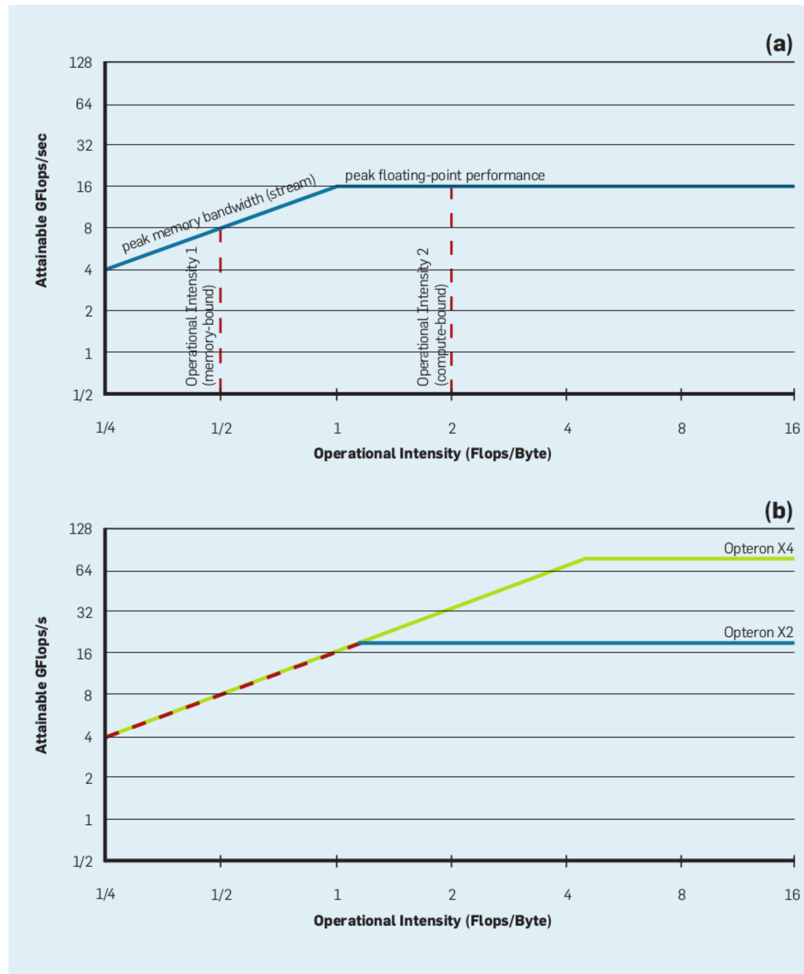


Figure 4.1.: Roofline models for AMD Opteron systems [21]

line shows the peak floating-point performance of the computer. The actual performance of an application can not be higher than this line, since this is the hardware limit. Peak memory performance (represented by GB/s) is represented by a rising line in the graph. The two lines intersect at the point of peak computational performance and peak memory bandwidth. These two lines give this model its name, roofline model. The roofline represents an upper bound of application performance on this computer.
In Figure 4.1b two rooflines are shown, one for Opteron X2 and one for Opteron X4. They share the same socket. Hence, they have the same peak memory bandwidth. Since the Opteron X4 provides more computational power the intersection point on the roofline is shifted to the right.

For each kernel there exists a point on the x-axis representing its operational intensity. If a vertical line is drawn through that point (the vertical dashed line in Figure 4.1a) the performance of the kernel on that machine must lie somewhere along that line. If the line hits the horizontal part of the roofline the kernel is compute bound, if it hits the diagonal part it is memory bound. In Figure 4.1a, a kernel with operational intensity 2 FLOPS/Byte is compute-bound and a kernel with operational intensity 0.5 FLOPS/Byte is memory-bound.
To generate a roofline for the target machine LINPACK was used for peak computational performance and the STREAM memory benchmark for memory bandwidth.

### LINPACK

To determine the peak computational performance for the roofline model the HPC LINPACK benchmark (HPL) was used. HPL is a portable implementation of HPLinpack which is used to provide data for the Top500 list [12]. The implementation of HPL can be found in [22]. In this thesis HPL was used in conjunction with the BLAS implementation in ATLAS[2]. HPL was executed using 64 threads to fully utilize the machine.

### STREAM Memory benchmark

To get the peak memory bandwidth (between last level cache and main memory) the STREAM memory benchmark, implemented by John McCalpin, University of Virginia, was used [23]. Stream was executed using 64 threads with an array size of 10e7 wich denotes to a total array size of 762.9 megabytes. The maximum achieved memory bandwidth of the four STREAM kernels (Copy, Scale, Add and Triad) were averaged.

### Results for the target machine

The following section shows the results of STREAM and LINPACK for the used target machine. Peak memory bandwith, measured using the STREAM memory benchmark: 39.174,75 MB/s.

[2]ATLAS: Automatically Tuned Linear Algebra Software, `http://math-atlas.sourceforge.net/`

Peak floating point performace, measured by using LINPACK: 47.17 GFLOPS/s. Based on this measurements the roofline intersection point is

$$\frac{47,17\ GFLOPS/s}{39,174\ GB/s} = 1.203\ FLOPS/Byte \tag{4.1}$$

Figure 4.2 shows the roofline of the target machine.

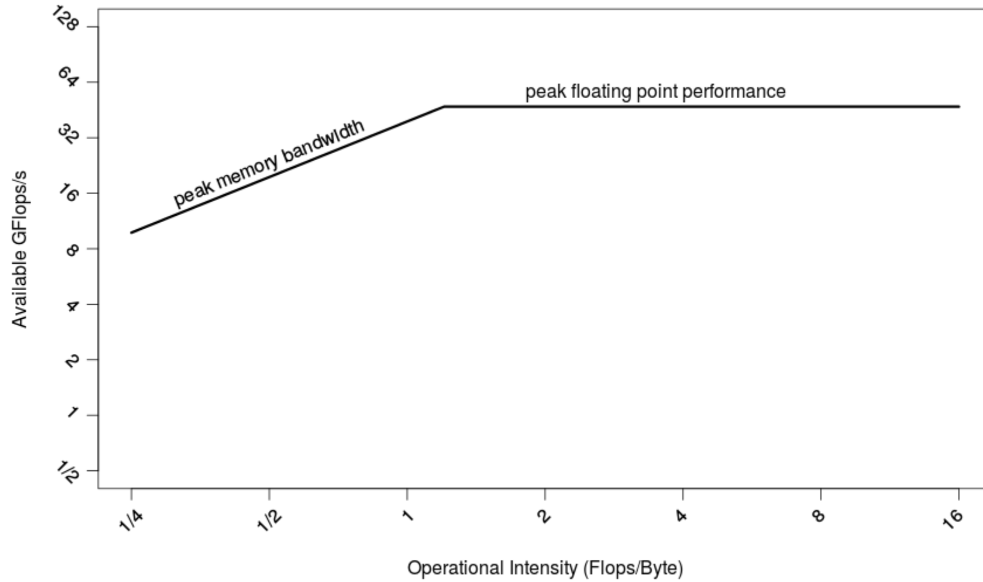


Figure 4.2.: Roofline model for the used target machine

### 4.2.7. Callgrind

For the detailed analysis of selected codes the callgrind tool included in the valgrind tool suite was used. Valgrind is a toolset for software debugging, profiling and dynamic error analysis. There are several tools integrated, in this thesis only callgrind which is based on cachegrind was used.
Cachegrind is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches and so can accurately pinpoint the sources of cache misses in the code. It identifies the number of cache misses, memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries [24].
Based on the results of cachegrind callgrind creates a callgraph. The data can be visualized using the tool KCachegrind, which gives a much better overview

of the data that callgrind collects.
In this thesis version 3.8.1 of valgrind was used, callgrind was executed using the default options.

# Chapter 5.

# Test Results

The following sections show the results of the test runs for all codes mentioned in Chapter 3. The first section shows the behaviour of input codes regarding their parallel performance using shared memory systems. The main memory consumption is shown in the second section. The last part shows how the codes are handled by the Insieme Compiler and the boundness of the codes (for details about boundness see 4.2.6).
Test runs are executed using the environment presented in Chapter 4. Each test run is executed five times. To reduce the influence of outliers the median of the results is calculated.
Each metric is measured using GCC (version 4.6.3) and the Insieme Compiler (INS). If the Insieme Compiler is not able to compile the code #NA is inserted instead of the metric results.

## 5.1. Shared Memory Parallelization Performance

In Table 5.1 runtime results of codes running with a different number of threads is shown. *Sequential runtime* refers to the execution of a version compiled with OpenMP enabled, using one thread. *Parallel runtime* refers to the same version using 64 threads. Runtime values are measured in seconds. *Efficiency* is calculated as follows:

$$
\begin{aligned}
&E_x = \frac{R_x}{R_1 * x} \\
&Efficiency = \frac{E_2 + E_4 + E_8 + E_{16} + E_{32} + E_{64}}{6} \\
&R_x \ldots \text{Runtime using x threads} \\
&E_x \ldots \text{Efficiency using x threads}
\end{aligned}
\tag{5.1}
$$

For each run, using $x$ threads, the best possible value for the efficiency $E_x$ is 1.0, the worst is 0. To reach a good overall efficiency an application has to scale good for all used variants of thread counts. This ensures that only applications scaling

well till the use of 64 threads get a good value for overall efficiency. *Insieme Factor* divides the efficiency of the Insieme version by the efficiency of the GCC version. If the factor is higher than 1 codes compiled with Insieme perform better, if the factor is lower than 1 GCC compiled codes performs better.

| Suite | Name | Sequential Runtime | | Parallel Runtime | | Efficiency | | Insieme |
|---|---|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | GCC | INS | Factor |
| BOTS | alignment | 1,090 | 1,110 | 0,582 | 1,062 | 0,247 | 0,026 | 0,105 |
| | fft | 0,002 | 0,002 | 0,020 | 0,266 | #NA | #NA | #NA |
| | fib_if | 0,000 | 0,002 | 0,020 | 0,244 | #NA | 0,024 | #NA |
| | fib_manual | 0,000 | 0,002 | 0,020 | 0,264 | #NA | 0,024 | #NA |
| | floorplan_if | 18,766 | 25,066 | 0,708 | 4,374 | 0,610 | 0,621 | 1,017 |
| | floorplan manual | 11,260 | 10,552 | 0,438 | 0,678 | 0,592 | 0,760 | 1,284 |
| | health_if | 0,520 | #NA | 2,526 | #NA | 0,155 | #NA | #NA |
| | health manual | 0,490 | #NA | 2,506 | #NA | 0,162 | #NA | #NA |
| | nqueens_if | 0,000 | 0,008 | 0,030 | 0,334 | #NA | 0,141 | #NA |
| | nqueens manual | 0,000 | 0,004 | 0,024 | 0,216 | #NA | #NA | 1,000 |
| | sort | 4,822 | 5,654 | 4,378 | 4,120 | 0,301 | 0,326 | 1,084 |
| | sparselu_for | 0,000 | 0,000 | 0,020 | #NA | #NA | #NA | #NA |
| | sparselu single | 0,000 | 0,004 | 0,010 | 0,234 | #NA | #NA | #NA |
| | strassen_if | 0,690 | 0,870 | 0,440 | 1,258 | 0,224 | 0,220 | 0,983 |
| | strassen manual | 0,690 | 0,836 | 0,440 | 1,058 | 0,223 | 0,225 | 1,009 |
| | uts | 1,392 | 1,822 | 20,740 | 3,424 | 0,161 | 0,203 | 1,260 |
| OmpSCR | fft | 0,040 | 0,000 | 0,040 | #NA | 0,218 | #NA | #NA |
| | fft6 | 0,000 | #NA | 0,010 | #NA | #NA | #NA | #NA |
| | graphsearch | 0,020 | #NA | 0,082 | #NA | 0,116 | #NA | #NA |
| | jacobi01 | 1,122 | #NA | 0,896 | #NA | 0,197 | #NA | #NA |
| | jacobi02 | 1,128 | #NA | 0,892 | #NA | 0,195 | #NA | #NA |
| | loopsA | 0,000 | #NA | 0,926 | #NA | #NA | #NA | #NA |
| | loopsB | 0,000 | #NA | 0,470 | #NA | #NA | #NA | #NA |
| | loopsC | 0,000 | #NA | 0,696 | #NA | #NA | #NA | #NA |
| | loopsD | 0,000 | #NA | 0,760 | #NA | #NA | #NA | #NA |
| | lu_reduction | 0,410 | 0,590 | 1,444 | 5,255 | 0,173 | 0,241 | 1,397 |
| | mandelbrot | 0,160 | #NA | 0,010 | #NA | 0,615 | #NA | #NA |

| Suite | Name | Sequential Runtime | | Parallel Runtime | | Efficiency | | Insieme |
|---|---|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | GCC | INS | Factor |
| | molecular | 17,654 | 49,152 | 4,738 | 11,334 | 0,736 | 0,726 | 0,986 |
| | pi | 0,820 | 1,670 | 0,040 | 0,276 | 0,817 | 0,765 | 0,936 |
| | qsort | 0,010 | #NA | 0,050 | #NA | 0,161 | #NA | #NA |
| | simple | 0,000 | 0,000 | 0,010 | 0,138 | #NA | #NA | #NA |
| | simple_pfor | 0,000 | 0,000 | 0,010 | 0,198 | #NA | #NA | #NA |
| | single | 0,000 | 0,002 | 0,010 | 0,216 | #NA | #NA | #NA |
| SSCA2 | | 0,220 | #NA | 8,356 | #NA | 0,167 | #NA | #NA |
| sequoia | stride | 82,366 | #NA | 82,556 | #NA | 0,164 | #NA | #NA |
| | cachec | 95,496 | #NA | 96,118 | #NA | 0,164 | #NA | #NA |
| | vecopc | 23,152 | #NA | 23,106 | #NA | 0,164 | #NA | #NA |
| | irs_mk | 50,452 | #NA | 2,380 | #NA | #NA | 0,164 | #NA |
| | clomp | 20,736 | #NA | 20,676 | #NA | 0,164 | #NA | #NA |
| smg2000 | | 0,080 | #NA | 7,970 | #NA | #NA | #NA | #NA |
| apex | sequential | 1,728 | 1,734 | 1,732 | 3,116 | 0,163 | 0,159 | 0,975 |
| | mpi_version | 1.201,516 | #NA | 1.201,544 | #NA | 0,166 | #NA | #NA |
| exmatex | comd | 11,892 | 12,732 | 3,680 | 6,536 | 0,547 | 0,503 | 0,921 |
| | comd_mpi | 37,740 | #NA | 37,700 | #NA | 0,164 | #NA | #NA |
| | lulesh | 172,918 | #NA | 19,238 | #NA | 0,532 | #NA | #NA |
| coral | amg2013 | 0,150 | #NA | 4,270 | #NA | 0,062 | #NA | #NA |
| | graph500 | 1,078 | #NA | 3,410 | #NA | 0,363 | #NA | #NA |
| | clomp | 39,770 | #NA | 24,646 | #NA | 0,251 | #NA | #NA |
| | xsBench | 62,248 | #NA | 4,764 | #NA | 0,593 | #NA | #NA |
| | HACCmk | 57,342 | 56,762 | 3,460 | 7,900 | 0,764 | 0,720 | 0,942 |
| | AMGmk | 0,172 | 0,300 | 0,170 | 0,920 | 0,315 | 0,332 | 1,054 |
| | miniFE | 1,700 | #NA | 1,400 | #NA | 1,244 | #NA | #NA |
| Galprop | | 164,368 | 156,216 | 13,900 | 19,984 | 0,981 | 0,685 | 0,699 |
| Common OpenMP | heatedPlate | 0,000 | 0,010 | 4,064 | 5,218 | #NA | 0,049 | #NA |
| | dijkstra | 0,864 | 0,980 | 9,400 | 21,828 | 0,232 | 0,174 | 0,752 |
| | prime | 5,442 | 5,444 | 0,374 | 1,320 | 0,507 | 0,443 | 0,873 |
| | integral-estimation | 13,996 | 14,192 | 0,386 | 0,822 | 0,901 | 0,829 | 0,920 |
| | satisfy | 0,360 | 0,000 | 0,020 | 0,158 | 0,794 | #NA | #NA |
| | sgefa | 1,010 | 1,376 | 5,198 | 7,458 | 0,204 | 0,201 | 0,984 |
| | ziggurat | 2,130 | 4,368 | 1,204 | 3,716 | 0,243 | 0,231 | 0,950 |
| | oddEvenSort | 0,100 | 0,170 | 16,920 | 20,570 | 0,058 | 0,031 | 0,527 |
| | mergeSort | 5,232 | 5,402 | 3,118 | 9,846 | 0,283 | 0,161 | 0,567 |

| Suite | Name | Sequential Runtime | | Parallel Runtime | | Efficiency | | Insieme |
|---|---|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | GCC | INS | Factor |
| | nBody | 0,030 | 0,060 | 0,636 | 0,738 | 0,236 | 0,275 | 1,164 |
| KineControl | | 48,874 | #NA | 1,368 | #NA | 0,828 | #NA | #NA |
| Parsec | blackscholes | 0,492 | #NA | 0,536 | #NA | 0,507 | #NA | #NA |
| | freqmine | 0,412 | #NA | 0,284 | #NA | 0,308 | #NA | #NA |
| Velvet | velvetg | 3,382 | #NA | 1,116 | 0,000 | 0,504 | #NA | #NA |
| | velveth | 10,700 | #NA | 4,032 | 0,000 | 0,253 | #NA | #NA |
| Stream | stream_c | 0,784 | #NA | 0,458 | #NA | 0,385 | #NA | #NA |
| | stream_d | 0,434 | 0,444 | 0,436 | 0,362 | 0,165 | 0,265 | 1,602 |
| | stream_omp | 0,154 | 0,174 | 0,220 | 1,096 | 0,550 | 0,241 | 0,439 |
| NPB | bt | 0,426 | 0,516 | 7,506 | 12,634 | 0,234 | 0,173 | 0,739 |
| | cg | 0,180 | 0,310 | 7,780 | 7,015 | 0,168 | 0,236 | 1,402 |
| | ep | 3,142 | 3,938 | 0,202 | 0,428 | 0,779 | 0,756 | 0,971 |
| | ft | 0,590 | 0,682 | 0,240 | 0,692 | 0,430 | 0,417 | 0,969 |
| | is | 0,020 | 0,030 | 0,086 | 0,330 | 0,162 | 0,210 | 1,298 |
| | lu | 0,260 | 0,426 | 1,210 | 3,288 | 0,405 | 0,361 | 0,890 |
| | mg | 0,040 | 0,052 | 0,910 | 1,122 | 0,351 | 0,230 | 0,656 |
| | sp | 0,380 | #NA | 14,850 | #NA | 0,168 | #NA | #NA |
| SNU-NPB | bt | 0,090 | #NA | 1,994 | #NA | 0,353 | #NA | #NA |
| | cg | 0,060 | 0,070 | 4,382 | 4,712 | 0,101 | 0,121 | 1,201 |
| | dc | 0,070 | #NA | 0,114 | #NA | 0,160 | #NA | #NA |
| | ep | 2,260 | #NA | 0,156 | #NA | 0,729 | #NA | #NA |
| | ft | 0,170 | #NA | 0,206 | #NA | 0,552 | #NA | #NA |
| | is | 0,020 | #NA | 0,140 | #NA | 0,240 | #NA | #NA |
| | lu | 0,040 | #NA | 1,094 | #NA | 0,236 | #NA | #NA |
| | mg | 0,010 | #NA | 1,190 | #NA | 0,161 | #NA | #NA |
| | sp | 0,040 | #NA | 5,172 | #NA | 0,214 | #NA | #NA |
| | ua | 0,810 | #NA | 2,090 | #NA | 0,277 | #NA | #NA |
| homb-mpi | | 5,900 | #NA | 10,610 | #NA | 0,073 | #NA | #NA |
| rodinia | kmeans | 6,500 | #NA | 6,482 | #NA | 0,164 | #NA | #NA |
| | backprop | 0,000 | #NA | 0,000 | #NA | #NA | #NA | #NA |
| | bfs | 1,602 | 1,722 | 1,586 | 3,472 | 0,166 | 0,164 | 0,987 |
| | euler3d | 170,436 | 218,774 | 9,486 | 20,206 | 0,711 | 0,623 | 0,877 |
| | euler3d double | 230,064 | #NA | 14,864 | #NA | 0,680 | #NA | #NA |
| | pre_euler3d | 177,260 | #NA | 13,798 | #NA | 0,647 | #NA | #NA |
| | heartwall | 1,230 | 5,090 | 1,230 | 0,776 | 0,164 | 0,633 | 3,861 |

| Suite | Name | Sequential Runtime | | Parallel Runtime | | Efficiency | | Insieme |
|---|---|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | GCC | INS | Factor |
| | hotspot | 0,200 | 0,230 | 0,200 | 0,894 | 0,164 | 0,169 | 1,032 |
| | lavaMD | 9,138 | #NA | 9,092 | #NA | 0,165 | #NA | #NA |
| | leukocyte | 3,678 | #NA | 3,688 | #NA | 0,164 | #NA | #NA |
| | lud | 7,764 | 10,400 | 6,818 | 16,152 | 0,393 | 0,356 | 0,906 |
| | myocyte | 9,416 | #NA | 9,422 | #NA | 0,164 | #NA | #NA |
| | nn | 0,066 | #NA | 19,430 | #NA | 0,144 | #NA | #NA |
| | needle | 5,854 | 6,380 | 5,810 | 9,012 | 0,165 | 0,152 | 0,920 |
| | particle_filter | 6,402 | 6,552 | 6,410 | 10,328 | 0,164 | 0,162 | 0,989 |
| | pahtfinder | 1,110 | 1,316 | 1,546 | 4,670 | 0,166 | 0,161 | 0,970 |
| | srad | 2,142 | 2,130 | 1,970 | 3,478 | 0,178 | 0,161 | 0,904 |

Table 5.1.: Shared memory performance of all input codes

## 5.2. Main Memory Footprint

Table 5.2 shows the main memory consumption of all measured codes. In column *Sequential Memory* memory consumption of a version compiled with OpenMP enabled, using one thread is shown. The *Parallel Memory* column refers to the same version executed using 64 threads. Memory consumption was measured in KBytes. *Insieme Factor* divides the memory consumption of the Insieme version by the memory consumption of the GCC version. If the factor is lower than 1 codes compiled with Insieme use less memory, if the factor is higher than 1 codes compiled with GCC use less memory.

| Suite | Name | Sequential Memory (KB) | | Parallel Memory (KB) | | Insieme |
|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | Factor |
| BOTS | alignment | 19.449 | #NA | 19.449 | #NA | #NA |
| | fft | 5.043 | 13.062 | 5.849 | 590.252 | 103 |
| | fib_if | 5.043 | 11.993 | 6.758 | 1.573.084 | 235 |
| | fib_manual | 5.040 | 13.587 | 11.673 | 1.638.614 | 143 |
| | floorplan_if | 5.043 | 18.057 | 97.078 | 21.890.880 | 229 |
| | floorplan manual | 5.043 | 14.672 | 48.636 | 1.646.080 | 36 |
| | health_if | 41.270 | #NA | 44.659 | #NA | #NA |
| | health manual | 41.270 | #NA | 47.900 | #NA | #NA |
| | nqueens_if | 5.043 | 15.369 | 6.640 | 4.324.448 | 654 |

| Suite | Name | Sequential Memory (KB) | | Parallel Memory (KB) | | Insieme |
|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | Factor |
| | nqueens manual | 5.046 | 13.590 | 9.862 | 1.486.188 | 153 |
| | sort | 2.100.614 | 2.110.508 | 2.105.091 | 31.713.484 | 16 |
| | sparselu_for | 5.056 | #NA | 5.676 | #NA | #NA |
| | sparselu single | 5.040 | 11.369 | 8.985 | 560.339 | 64 |
| | strassen_if | 237.859 | 522.508 | 676.774 | 1.101.462 | 3 |
| | strassen manual | 237.881 | 373.552 | 664.009 | 764.220 | 2 |
| | uts | 8.508 | 21.180 | 46.697 | 57.361.768 | 1.230 |
| OmpSCR | fft | 10.269 | #NA | 12.547 | #NA | #NA |
| | fft6 | 5.040 | #NA | 13.702 | #NA | #NA |
| | graphsearch | 10.899 | #NA | 13.517 | #NA | #NA |
| | jacobi01 | 2.347.011 | #NA | 2.349.242 | #NA | #NA |
| | jacobi02 | 2.347.018 | #NA | 2.349.178 | #NA | #NA |
| | loopsA | 5.046 | #NA | 10.723 | #NA | #NA |
| | loopsB | 5.043 | #NA | 7.014 | #NA | #NA |
| | loopsC | 5.037 | #NA | 7.072 | #NA | #NA |
| | loopsD | 5.043 | #NA | 7.021 | #NA | #NA |
| | lu_reduction | 65.190 | 71.344 | 70.669 | 9.421.888 | 134 |
| | mandelbrot | 5.046 | #NA | 5.338 | #NA | #NA |
| | molecular | 6.349 | #NA | 11.840 | #NA | #NA |
| | pi | 5.040 | 6.531 | 5.040 | 610.275 | 122 |
| | qsort | 5.050 | #NA | 5.424 | #NA | #NA |
| | simple | 5.040 | 7.987 | 5.043 | 608.506 | 122 |
| | simple_pfor | 5.037 | 12.794 | 8.269 | 610.240 | 76 |
| | single | 5.050 | 8.179 | 5.030 | 613.552 | 124 |
| SSCA2 | | 0,220 | #NA | 8,356 | #NA | #NA |
| sequoia | stride | 19.456 | #NA | 19.446 | #NA | #NA |
| | cachec | 19.450 | #NA | 19.450 | #NA | #NA |
| | irs_mk | 140.768 | 142.048 | 146.208 | 743.120 | 6 |
| | vecopc | 19.382 | #NA | 19.386 | #NA | #NA |
| | clomp | 65.427 | #NA | 59.139 | #NA | #NA |
| smg2000 | | 16.419 | #NA | 18.608 | #NA | #NA |
| apex | sequential | 2.100.487 | 2.101.946 | 2.100.474 | 2.189.978 | 2 |
| | mpi_version | 293.731 | #NA | 293.731 | #NA | #NA |

| Suite | Name | Sequential Memory (KB) | | Parallel Memory (KB) | | Insieme |
|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | Factor |
| exmatex | comd | 68.579 | 89.069 | 74.122 | 14.860.467 | 202 |
| | comd_mpi | 120.723 | #NA | 121.440 | #NA | #NA |
| | lulesh | 87.354 | #NA | 114.509 | #NA | #NA |
| coral | amg2013 | 67.888 | #NA | 87.344 | #NA | #NA |
| | graph500 | 39.210 | #NA | 49.821 | #NA | #NA |
| | clomp | 5.046 | #NA | 8.474 | #NA | #NA |
| | xsBench | 23.256.764 | #NA | 23.256.768 | #NA | #NA |
| | HACCmk | 7.562 | 27.392 | 10.634 | 18.028.838 | 1.699 |
| | AMGmk | 60.726 | 75.059 | 66.221 | 1.388.848 | 22 |
| | miniFE | 37.776 | #NA | 40.000 | #NA | #NA |
| Galprop | | 335.466 | 354.342 | 337.715 | 42.882.608 | 128 |
| Common OpenMP | heatedPlate | 0,000 | 0,010 | 4,064 | 5,218 | #NA |
| | dijkstra | 1.565.373 | 1.632.768 | 1.567.568 | 2.201.187 | 2 |
| | prime | 6.349 | 9.536 | 11.840 | 1.396.451 | 119 |
| | integral-estimation | 13,996 | 14,192 | 0,386 | 0,822 | 0,167 |
| | satisfy | 6.349 | 6.518 | 11.840 | 606.915 | 52 |
| | sgefa | 6.349 | 32.403 | 11.840 | 11.054.349 | 939 |
| | ziggurat | 6.349 | 10.118 | 11.840 | 787.834 | 68 |
| | oddEvenSort | 5.040 | 14.048 | 5.008 | 27.021.184 | 5.398 |
| | mergeSort | 1.877.565 | 1.885.635 | 1.883.050 | 1.965.517 | 2 |
| | nBody | 5.056 | 30.650 | 11.542 | 630.666 | 61 |
| KineControl | | 12.294 | #NA | 59.952 | #NA | #NA |
| Parsec | blackscholes | 5.053 | #NA | 5.059 | #NA | #NA |
| | freqmine | 83.235 | #NA | 732.531 | #NA | #NA |
| Velvet | velvetg | 453.114 | #NA | 446.944 | #NA | #NA |
| | velveth | 1.301.210 | #NA | 1.319.754 | #NA | #NA |
| Stream | stream_c | 940.493 | #NA | 942.720 | #NA | #NA |
| | stream_d | 0,434 | 0,444 | 0,436 | 0,362 | 0,997 |
| | stream_omp | 0,154 | 0,174 | 0,220 | 1,096 | 0,444 |
| NPB | bt | 12.544 | 85.232 | 18.035 | 705.517 | 46 |
| | cg | 13.600 | 85.536 | 15.824 | 865.744 | 61 |
| | ep | 7.322 | 12.227 | 442.534 | 865.574 | 4 |
| | ft | 57.568 | 63.725 | 68.854 | 885.283 | 14 |
| | is | 5.059 | 10.381 | 22.221 | 853.421 | 40 |
| | lu | 5.405 | 78.192 | 10.880 | 695.379 | 78 |

| Suite | Name | Sequential Memory (KB) | | Parallel Memory (KB) | | Insieme |
|---|---|---|---|---|---|---|
| | | GCC | INS | GCC | INS | Factor |
| | mg | 7.696 | 31.136 | 25.459 | 1.438.790 | 61 |
| | sp | 6.173 | #NA | 8.400 | #NA | #NA |
| SNU-NPB | bt | 5.059 | #NA | 20.771 | #NA | #NA |
| | cg | 11.965 | 21.491 | 14.186 | 2.310.618 | 165 |
| | dc | 138.365 | #NA | 138.365 | #NA | #NA |
| | ep | 7.418 | #NA | 420.102 | #NA | #NA |
| | ft | 45.082 | #NA | 78.845 | #NA | #NA |
| | is | 5.053 | #NA | 8.733 | #NA | #NA |
| | lu | 5.472 | #NA | 7.680 | #NA | #NA |
| | mg | 7.274 | #NA | 9.818 | #NA | #NA |
| | sp | 5.066 | #NA | 23.667 | #NA | #NA |
| | ua | 19.997 | #NA | 24.224 | #NA | #NA |
| homb-mpi | | 2.135.408 | #NA | 2.137.616 | #NA | #NA |
| rodinia | kmeans | 911.798 | #NA | 911.795 | #NA | #NA |
| | backprop | 5.027 | #NA | 5.037 | #NA | #NA |
| | bfs | 157.366 | 160.349 | 160.634 | 817.926 | 6 |
| | euler3d | 55.811 | 85.286 | 61.264 | 2.457.258 | 42 |
| | euler3d double | 100.074 | #NA | 102.250 | #NA | #NA |
| | pre_euler3d | 177 | #NA | 76.566 | #NA | #NA |
| | heartwall | 138.186 | 139.645 | 138.186 | 702.394 | 6 |
| | hotspot | 27.699 | 34.323 | 27.709 | 597.590 | 23 |
| | lavaMD | 34.013 | #NA | 34.016 | #NA | #NA |
| | leukocyte | 157.386 | #NA | 157.389 | #NA | #NA |
| | lud | 72.714 | 76.013 | 72.712 | 24.977.364 | 345 |
| | myocyte | 2.289.389 | #NA | 2.289.392 | #NA | #NA |
| | nn | 5.242 | #NA | 7.488 | #NA | #NA |
| | needle | 5,854 | 6,380 | 5,810 | 9,012 | 0,998 |
| | particle_filter | 6,402 | 6,552 | 6,410 | 10,328 | 1,000 |
| | pahtfinder | 164.051 | 167.318 | 172.797 | 3.279.495 | 20 |
| | srad | 2,142 | 2,130 | 1,970 | 3,478 | 0,920 |

Table 5.2.: Main memory footprint of all input codes

## 5.3. Boundness

Table 5.3 shows if test codes are memory or compute bound. The table only lists codes Insieme is capable to successfully compile and run. *Boundness Factor* denotes the operational intensity of an application compiled with GCC running on 64 threads. Operational intensity was calculated using the values presented in Section 4.2.6. The last column *Boundness* shows if the code is memory or compute bound.

| Suite | Name | Boundness Factor (Flops/Byte) | Boundness |
|---|---|---:|---:|
| BOTS | fft | 0,0367 | Memory |
| | fib_if | 0,0306 | Memory |
| | fib_manual | 0,0205 | Memory |
| | floorplan_if | 0,0122 | Memory |
| | floorplan_manual | 0,0114 | Memory |
| | nqueens_if | 0,0213 | Memory |
| | nqueens_manual | 0,0174 | Memory |
| | sort | 1118 | Memory |
| | sparselu_single | 0,0488 | Memory |
| | strassen_if | 46,7518 | Compute |
| | strassen_manual | 47,4742 | Compute |
| | uts | 0,013 | Memory |
| OmpSCR | lu_reduction | 9,4698 | Compute |
| | pi | 1633,0098 | Compute |
| | simple | 0,0208 | Memory |
| | simple_pfor | 0,0203 | Memory |
| | single | 0,0213 | Memory |
| | molecular | 136,6499 | Compute |
| sequoia | irs_mk | 24,152 | Compute |
| smg2000 | | 0,0292 | Memory |
| apex | sequential | 11,3281 | Compute |
| exmatex | comd | 157,9175 | Compute |
| coral | amg2013 | 0,0777 | Memory |
| | HACC_OC | 2686,4509 | Compute |
| | AMGmk | 4,8274 | Compute |
| | miniFE | 2,4114 | Compute |
| Galprop | | 4,3301 | Compute |

| Suite | Name | Boundness Factor (Flops/Byte) | Boundness |
|---|---|---|---|
| Common | heatedPlate | 0,0993 | Memory |
| OpenMP | dijkstra | 0,0103 | Memory |
| | prime | 2623,4363 | Compute |
| | integral-estimation | 6221,6965 | Compute |
| | satisfy | 0,0255 | Memory |
| | sgefa | 0,3036 | Memory |
| | ziggurat | 91,7248 | Compute |
| | oddEvenSort | 0,0547 | Memory |
| | mergeSort | 1,9486 | Compute |
| | nBody | 4,1396 | Compute |
| Stream | stream_d | 0,0203 | Memory |
| | stream_omp | 2,0727 | Compute |
| NPB | bt | 5,8642 | Compute |
| | cg | 1,3199 | Compute |
| | ep | 95,0317 | Compute |
| | ft | 9,45 | Compute |
| | is | 6,4843 | Compute |
| | lu | 12,7946 | Compute |
| | mg | 2,106 | Compute |
| SNU-NPB | cg | 2,4175 | Compute |
| homb-mpi | | 8,996 | Compute |
| rodinia | bfs | 0,0012 | Memory |
| | euler3d | 61,0544 | Compute |
| | heartwall | 1697,5879 | Compute |
| | hotspot | 9,9488 | Compute |
| | leukocyte | 543,8547 | Compute |
| | lud | 0,6074 | Memory |
| | needle | 0 | Memory |
| | particle_filter | 3877,8655 | Compute |
| | pahtfinder | 0,017 | Memory |
| | srad | 821,2534 | Compute |

Table 5.3.: Boundness of selected input codes

# Chapter 6.

# Detailed Analysis of Selected Codes

Some codes were selected based on

- their sequential runtime (below 60s),
- parallelization model (OpenMP) and
- their efficiency (smaller than 0,2 or higher than 0,8).

These criteria where chosen to ensure that the codes are interesting for the Insieme compiler to optimize. The goal was to analyze various small codes. For big codes the compilation process gets more and more complex, therefore they are not that useful for the Insieme development process. Another criteria was to find applications which scale bad to retain the potential for Insieme to optimize the code.
All analysis was done on the architecture presented in Section 4. To show how important it is to parallelize even small parts of an application Amdahl's law is useful. Therefore a short introduction in Amdahl's law is shown in the first section.

## 6.1. Amdahl's law

$$\begin{aligned}
&S(N) = \frac{1}{(1-P) + \frac{P}{N}} \\
&S(N) \ldots \text{maximal achievable speedup by using N processors} \\
&S_x \ldots \text{proportion of a program that runs fully parallel} \\
&1 - P \ldots \text{serial part}
\end{aligned} \tag{6.1}$$

If N approaches infinity the maximum value for the speedup only depends on the serial part. As an example, if P is 90%, that means 10% of the program is executed in serial, the highest achievable speedup is 10, no matter how many cores are used [25].
This calculation shows that optimization of this part is needed, even if it produces only about 15% of the runtime.

## 6.2. BOTS Health

The health benchmark (see also Section 3.2) included in the Barcelona OpenMP Task Suite (BOTS) simulates a country health system. The code scales bad for an increasing number of used threads. The sequential runtime on the used test system is 0.52 seconds, when increasing the number of threads until 64 the runtime increases up to 2.56 seconds. Details of the measured runtimes can be found in Figure 6.1.

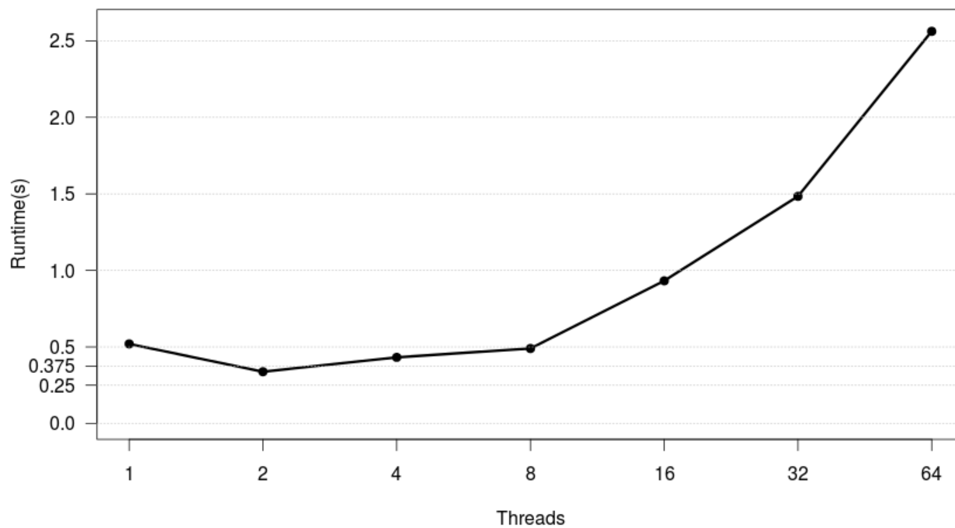


Figure 6.1.: Runtime in seconds of the health benchmark

It can be seen that the runtime decreases if the number of threads is increased to 2. If the thread count is further increased the performance gets worse. It is necessary to analyze the code and check why the code does not scale for a higher number of threads. At first glance the following parallelization issues are possible:

- A high amount of overhead for thread creation/suspension and synchronization,
- a high amount of non parallelized code or
- the problem size is too small to split it up for a higher number of threads.

### Parallelization Details

The function doing the main calculation in this application is *sim_village_par*. This method is basically running in parallel. The method recursively calls itself, for each recursive call an OpenMP task is created. At the end an *OpenMP taskwait* pragma waits for the end of all tasks.
Figure 6.2 shows how runtime is shared among all functions used in the application. The analysis is done using callgrind from the valgrind tool suite (see 4.2.7). On the left side the analysis without parallelization, on the right side a run using 32 threads is shown.

| Incl. | Self | Called | Function |
|---|---|---|---|
| 50.96 | 50.96 | 47 529 522 | my_rand |
| 70.96 | 20.05 | 2 253 875 | check_patients_population |
| 98.49 | 7.31 | 6 570 | <cycle 1> |
| 12.37 | 6.02 | 2 253 511 | GOMP_task |
| 8.66 | 4.11 | 2 253 510 | GOMP_taskwait'2 <cycle 1> |
| 3.75 | 3.75 | 2 253 515 | _int_free |
| 3.61 | 3.61 | 2 390 896 | _int_malloc |

| Incl. | Self | Called | Function |
|---|---|---|---|
| 99.27 | 37.15 | 64 | gomp_team_barrier_wait_end |
| 23.90 | 23.90 | 47 529 522 | my_rand |
| 9.56 | 9.56 | 3 317 | gomp_sem_wait_slow |
| 33.29 | 9.41 | 2 253 875 | check_patients_population |
| 6.00 | 6.00 | 22 305 | gomp_mutex_lock_slow |
| 10.01 | 2.83 | 2 253 511 | GOMP_task |
| 35.55 | 2.37 | 667 160 | <cycle 1> |
| 1.76 | 1.76 | 2 253 514 | _int_free |
| 1.70 | 1.70 | 2 390 928 | _int_malloc |

Figure 6.2.: Callgrind analysis of the health benchmark

It is visible that the method *gomp_team_barrier_wait_end* ist the most time consuming method in the parallel version. That indicates a high amount of waiting time at the barrier at the end of the application. This waiting time (approximately 47% of the total runtime) is lost, no calculation is done here. This is the main reason why the application does not scale for a higher number of threads.
When no threads are used the methods *my_rand* and *check_patients_population* are the most time consuming methods. These methods take about 70% of the total runtime, in the parallel execution only 35% of the total runtime are used for the calculation. The rest of the runtime is used for thread management overhead.
It would have been possible that the proportion of thread management overhead decreases if a larger problem size is used. Another run of the benchmark using a higher problem size showed that this is not the case. Runtime using one thread is 11 seconds, for 16 threads 116 seconds and for 32 threads 140 seconds.
Additionally an analysis using Intel vTune showed that most of the calculation is running in parallel, the amount of non parallelized code is below 10%. Altough this parallelization was not done in a good way.
All in all the overhead for thread creation and synchronization is too high to

reach a good speedup. This overhead increases for a higher number of threads to manage.

## 6.3. OmpSCR Graphsearch

Graphsearch included in the OpenMP Source Code Repository (OmpSCR, see also 3.2) searches a path between two nodes in a given graph. A short analysis of the code shows that the benchmark does not scale well. The runtime increases if the number of threads is increased, shown in Figure 6.3. If the same application is executed using 32 threads instead of 1 thread the runtime is multiplied by a factor of 4. This leads to a bad efficiency value of 0.116. An interesting value is the runtime using 16 threads, details why the runtime is decreasing between 8 and 16 threads can be found in the section below. The short runtime indicates that the problem size is rather small, this would imply that a good scaling is not possible.

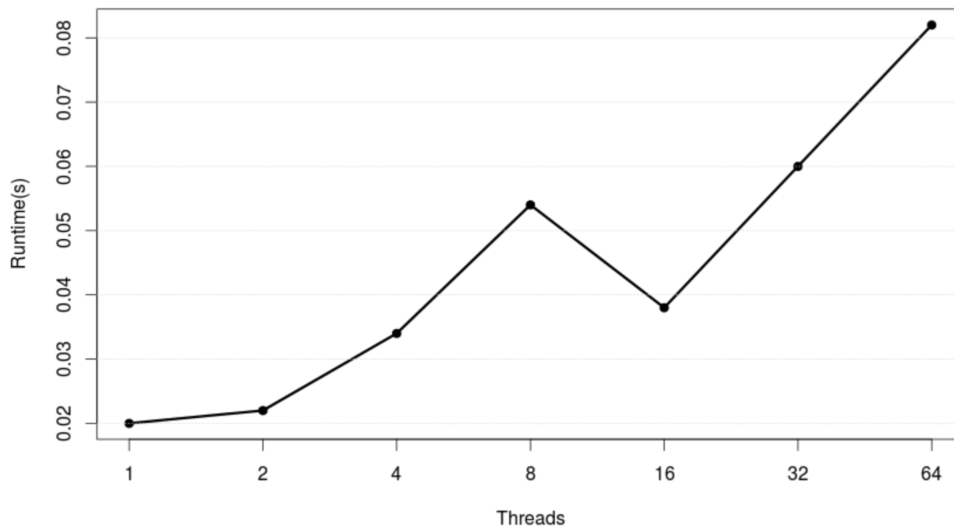


Figure 6.3.: Runtime in seconds of the graphsearch benchmark

### Parallelization Details

A short analysis using callgrind shows why the code does not scale well. An excerpt of the results of callgrind are shown in Figure 6.4. It compares the

results using 1 thread on the left with the results using 32 threads on the right. The two most time consuming methods of the code:

- *tg_read*, which reads out a file and creates a data structure representing a graph. This method does not contain any parallel computation.
- *testPath*, which does the main computation. Here a parallel for loop is implemented.

It can be seen that the most time is used to read out the file and to create the data structure, 95% if running sequentially. The main computation only consumes 4.5% of the runtime if 1 thread is used.

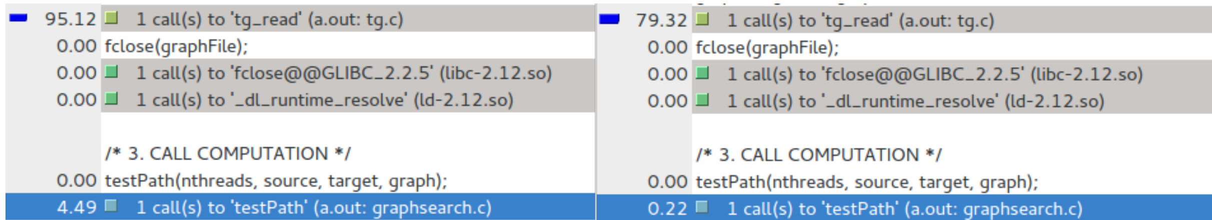


Figure 6.4.: Callgrind analysis of the graphsearch benchmark, 1 thread vs 32 threads

If we look at the results of a run using 32 threads it can be seen that the computation method only takes 0.22% of the runtime, the file operations take 79%. The remaining 20% (not visible in the callgrind analysis above) are used for creation and suspending of the OpenMP threads.
Based on these results the absolute runtime values are shown in Table 6.1. It can be seen that the parallel part of the application reaches a speedup of 6 by using 32 threads. The decrease of runtime between 8 and 16 threads is also visible in the table. While the amount of OpenMP overhead stays the same the calculation time is decreasing. Therefore the total execution time decreases.
The main problem here is the high amount of sequential code, as a result of that it is not possible for the whole application to scale well. Since these sequential file operations cannot be reduced it is difficult to optimize the application.

| | 1 thread | 8 threads | 16 threads | 32 threads | Speedup |
|---|---|---|---|---|---|
| Calculation | 0.0008 s | 0.0135 s | 0.00836 s | 0.000132 s | 6.06 |
| File Operations | 0.019 s | 0.0505 s | 0.034 s | 0.047 s | 0.40 |
| OpenMP Overhead | 0 | 0.0030 s | 0.0032 s | 0.009 s | #NA |

Table 6.1.: Absolute runtime values of the graphsearch benchmark

## 6.4. PARSEC Blackscholes

The blackscholes benchmark contained in the PARSEC benchmark suite (see 3.2) solves the black-scholes differential equation. The equation represents a mathematical model of a financial market containing certain derivative investment instruments. Details of the black-scholes model can be found in [26].
If we look at the runtime measurements shown in Figure 6.5 we can see that the runtime is decreasing as expected if the number of threads is increased. A bad outlier is the runtime value reached by using 64 threads, the runtime increases sharply from a value of 0.06 seconds by using 32 threads to a value of 0.536 seconds. The reason for this is that the test machine only runs on 32 physical cores. Therefore the processor is only able to execute 32 floating point operations in parallel. An analysis of the boundness of the application (see also Section 4.2.6) showed that the code is compute bound. That means that the application is constrained by the computational power of the machine. This fact and the additional overhead to create and maintain threads lead to this bad result.
Beside this behavior the scaling works, the runtime decreases nearly by a factor of two if the number of threads is doubled. Therefore a more detailed analysis of the parallelisation was not done.

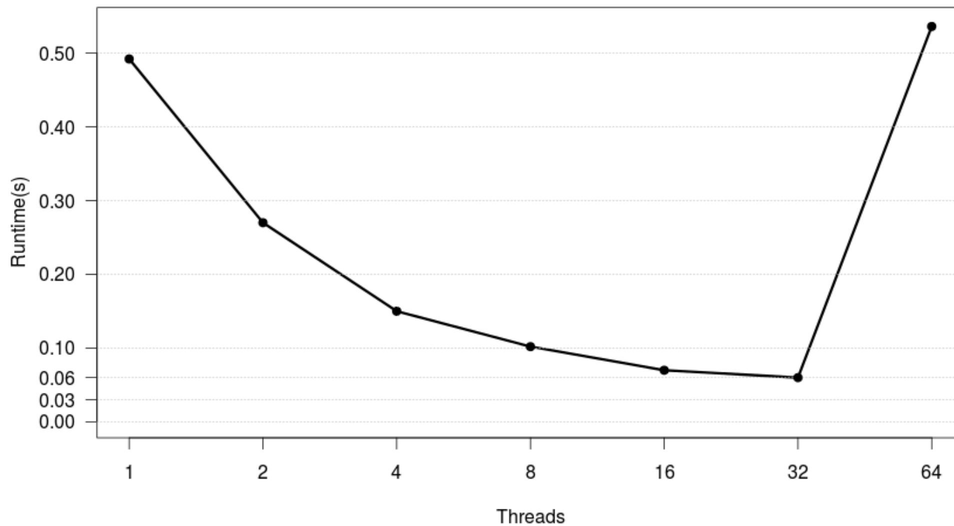


Figure 6.5.: Runtime in seconds of the blackscholes benchmark

## 6.5. Dijkstra

The implementation of Dijkstra's algorithm[1] presented in 3.2 solves the problem to find the shortest path between two nodes in a graph. The implementation was parallelized using a basic OpenMP parallel region. The array containing the distances between nodes is shared among all threads. Each thread calculates the distances of its assigned part of the array. In Figure 6.6 runtime in a parallel execution of the algorithm is shown. On the first sight it can be seen that the application scales well till 4 threads. If more threads are used the runtime increases. To find out why the application shows bad scaling a further analysis of the parallelization was done.

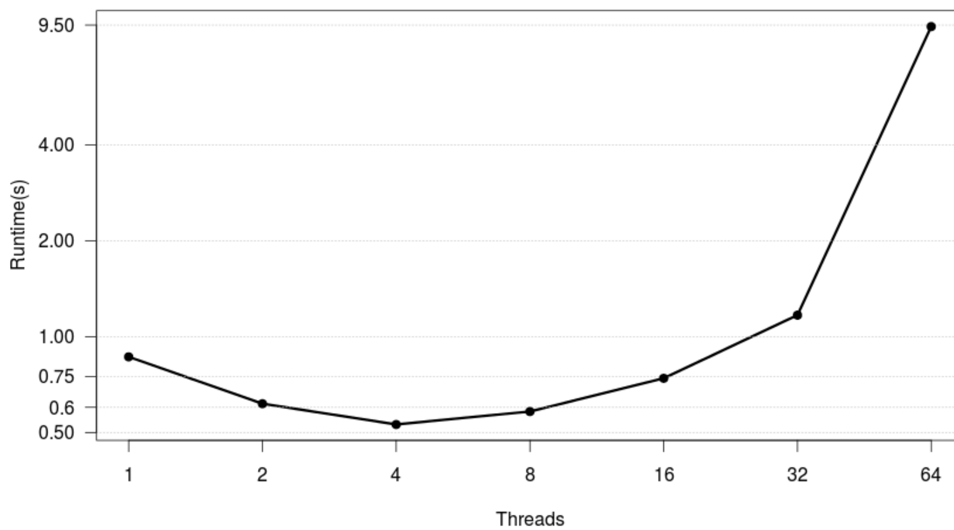


Figure 6.6.: Runtime in seconds of the dijkstra algorithm implementation

### Parallelization Details

The implementation uses a big integer matrix representing the distances between all nodes, $a(i, j)$ contains the distance of nodes $i$ and $j$. If two nodes are not directly connected the matrix cell contains a negative value. Each thread works on a part of the matrix and calculates the node distances in its tile. At the end of the algorithm the results of all threads are gathered and the best path

[1]Dijkstra's algorithm: `https://en.wikipedia.org/wiki/Dijkstra's_algorithm`

is selected. The main problem in this algorithm is that the calculation time of each thread differs. Therefore the higher the number of threads, the higher the waiting time at the end to gather all results, this is the typical problem of a bad load balancing. An analysis of the waiting time (per thread) raised using Intel vTune is presented in Figure 6.7. The results show that the waiting time per thread increases heavily if more threads are used. This could be avoided if the work distribution is improved and each thread takes approximately the same amount of time for calculation.

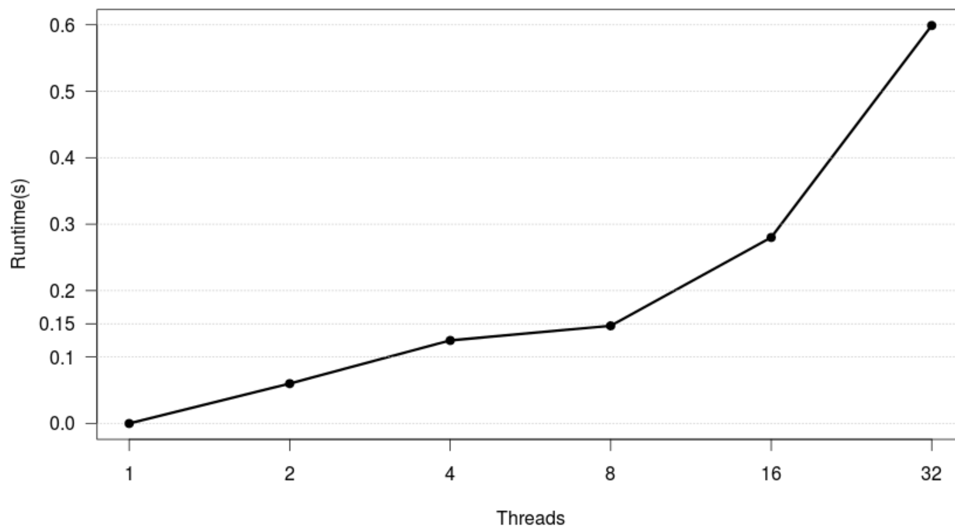


Figure 6.7.: Waiting time of the dijkstra code (per thread)

## 6.6. NPB cg

The runtime of the parallel cg implementation of the benchmark suite NAS Parallel Benchmarks (see 3.2) stays nearly the same regardless how many threads are used. This indicates that either there is no parallelization at all or the gain of parallel execution is equal to the overhead created by thread management and synchronization. The upper bound of this behavior is 32 threads. Like for the Blackscholes benchmark (Section 6.4) the runtime measured using 64 threads increases because of hyperthreading. Runtime measurements are found in figure 6.8.

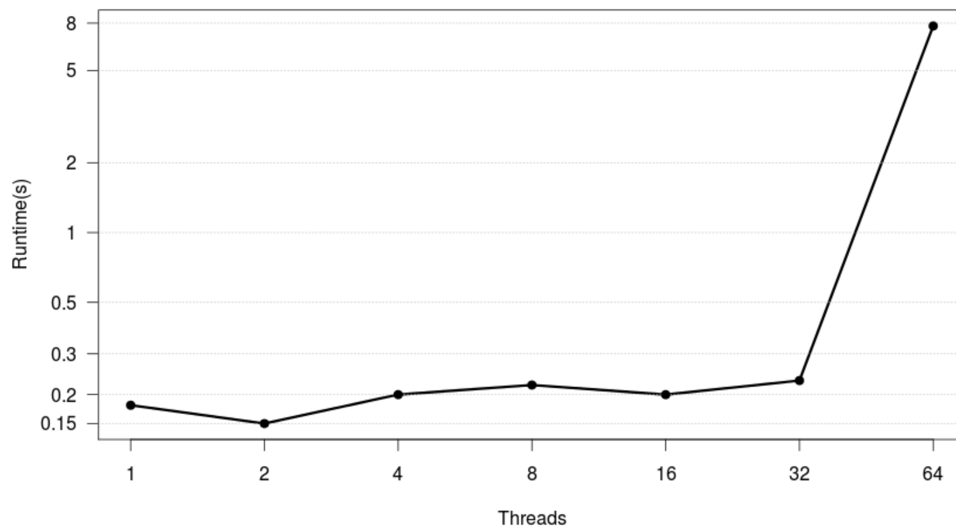


Figure 6.8.: Runtime in seconds of the NPB/cg implementation

### Parallelization Details

The main calculation method of the application is *conj_grad*, it contains 13 for loops. Each loop was parallelized on its own using an *omp parallel for* pragma. That means after each loop an implicit barrier is introduced and all threads have to wait. This leads to a high amount of waiting time. For the used problem size S the waiting time and the speed improvement gained by parallel execution balance each other, see Figure 6.9. It can be seen that the waiting time increases by increasing the number of threads but the computation time decreases. The runtime stays nearly the same. To solve this problem it is necessary to reduce the number of OpenMP for loops in order to reduce the number of implicit barriers.
If the problem size is increased the amount of waiting time stays nearly the same but the amount of computation time increases. As a result a better speedup is achievable. Efficiency measurements of problem sizes A,B,C,W and S are shown in Figure 6.10.

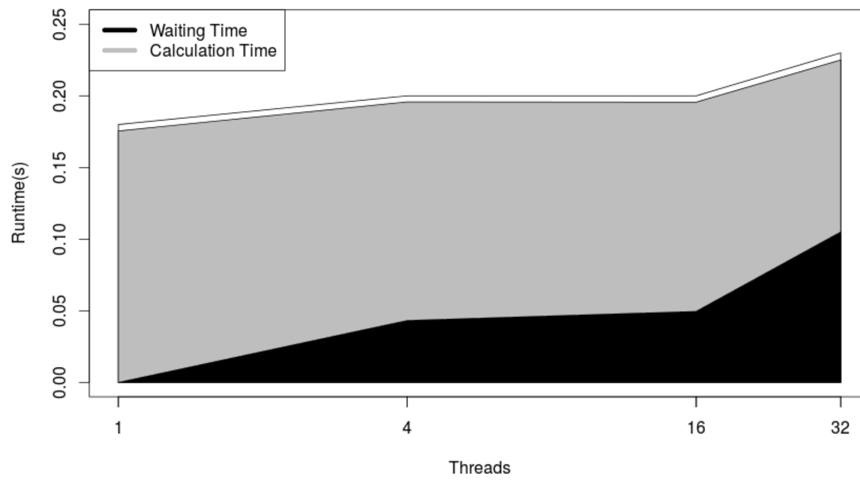


Figure 6.9.: Waiting/computation time of the CG benchmark

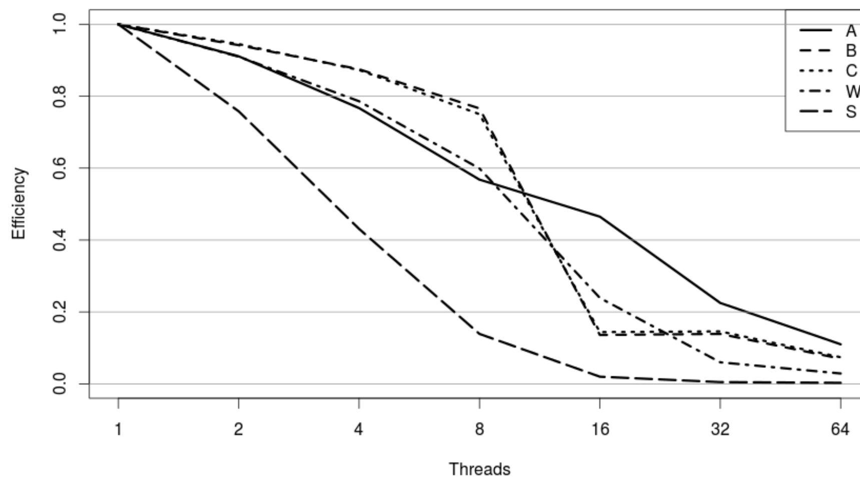


Figure 6.10.: Efficiency of different problem sizes of the CG benchmark

## 6.7. Rodinia nn

Figure 6.11 shows runtime measurements of the NN benchmark included in the Rodinia suite. The benchmark finds the k-nearest neighbors from an un-

structured data set, see also 3.2. The results show that runtime increases if number of threads is increased.

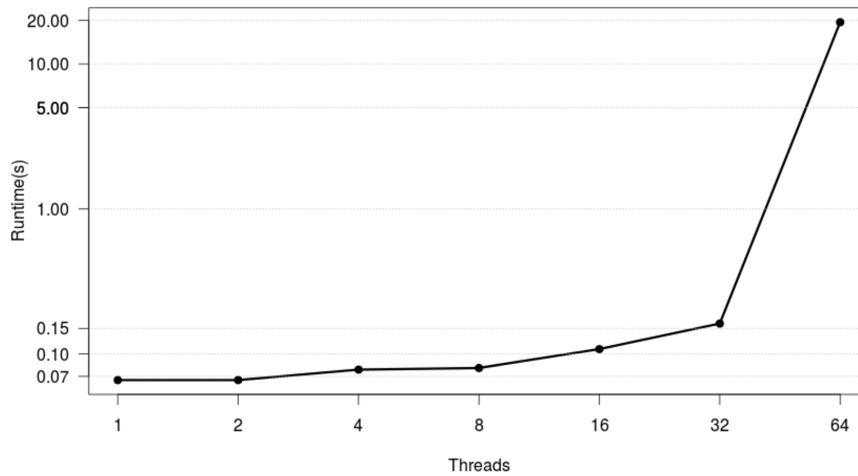


Figure 6.11.: Runtime in seconds of the NN implementation in Rodinia

### Parallelization Details

The code contains of two main loops:

1. An outer loop which is executed 7491 times for the default input problem
2. and an inner loop executed 10 times for each outer loop execution.

The problem here is that only the inner loop is parallelized using a *pragma omp parallel for*. This structure inherits some problems:

- For each outer loop iteration an implicit barrier is inserted after the inner loop.
- The maximum number of concurrently running threads is 10, since the parallel inner loop only contains 10 iterations. If more threads are used some of them have no work to do. This value depends on how many input databases are used, therefore the scalability of the application is bound to the input problem size.
- All the calculation of the outer loop is done sequentially.

- At the begin of a parallel region overhead is generated by copying shared variables, spawning threads or assigning loop iterations to thread workers. This overhead is executed for each outer loop iteration, therefore 7491 times.

The results of an analysis using callgrind and Intel vTune confirm these problems. A high amount of time is spent in OpenMP routines *GOMP_parallel_start* and *GOMP_parallel_end*, those are called by a *#pragma omp parallel for* in the main method.

## 6.8. Rodinia bfs

The bfs bechmark contained in the rodinia benchmark suite (see 3.2) implements a Breath First Search graph traversal algorithm[2]. The benchmark uses a high amount of runtime for reading and writing of files. These input output operations take longer than the actual computational part. Therefore a good scaling for parallel execution of the whole application is not easy achievable. The amount of time used for input file reading and computing are shown in fig 6.12. It can be seen that the computing time decreases, but the high effort used for input/output overwhelms the gained performance improvement.

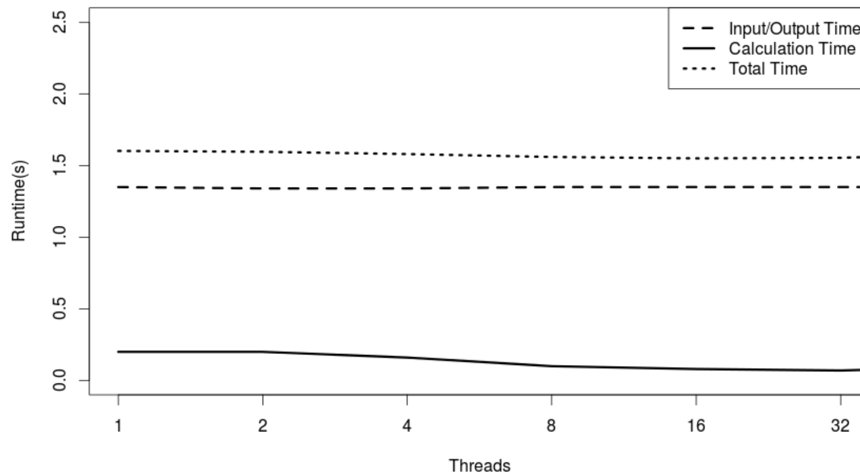


Figure 6.12.: Runtime in seconds of the BFS implementation in Rodinia

[2]Breadth First Search algorithm: `https://en.wikipedia.org/wiki/Breadth-first_search`

# Chapter 7.

# Integration Test Framework

The Integration Test Framework was developed to automatically test the Insieme Compiler regarding correctness and performance. By using the framework it is easy to maintain a high amount of miscellaneous test codes. The test database is organized as a simple directory hierarchy extended with configuration files. For example, the source files of a C++ test compiled with OpenMP support has to be present in the directory cpp/openmp. Each execution of a test code is split into several test steps. Each test step represents either compilation, execution or the checking of the output of an execution step. To gain more details of the test codes the test framework was extended to collect several characteristics (from now on called metrics) of the test execution. Examples of such metrics would be the *lines of code* or the *speedup using 2 threads*. To easily parse the large amount of information gained by these metrics it is possible to save the results in a SQL file or in a comma separated values (CSV) file.

## 7.1. Test Database

The test database is organized in a directory hierarchy, it is possible to create as many subfolders as necessary. The leaves of the hierarchy represent test cases. Each folder may contain two configuration files,

- the **test configuration file** which steers how a test case is executed and which test steps are used and
- the **test.cfg** file which contains all subfolders which have to be parsed to get further test cases.

The directory tree is parsed in a top-down manner. Each configuration file of a directory inherits the configurations of the parent directory. For the top of the tree there exists a main configuration file which defines options for all test cases, such a globally valid option would be the path of the Insieme compiler.

### 7.1.1. Configuration File Options

In Table 7.1 all possible options to steer the execution of a test case are listed. Each option applies to either one test step or, if no step is specified, to all steps. The syntax of an option has to adhere to the following rules:

- `configuration_option[test_step]=value`, an option for a specific test step
- `configuration_option=value`, an option applied to all test steps
- `value` is one of
  - an absolute value, e.g. the path to a compiler binary
  - a boolean value, e.g. enable openmp support
  - a list of strings, e.g. the list of input files

| Name | Description | Default value |
|---|---|---|
| `compiler` | Path to the used compiler binary | |
| `use_libmath` | Specify if `-lm` is used for compilation | 1 |
| `use_libpthread` | Specify if `-lpthread` is used for compilation | 1 |
| `includes` | List of includes needed for compilation | |
| `files` | List of compilation input files | Name of test case |
| `definitions` | List of definitions needed for compilation | |
| `compFlags` | Additional compiler flags | |
| `libPaths` | List of external library directories | |
| `libNames` | List of external libaries needed for linking | |
| `executionFlags` | Runtime arguments | |
| `excludeSteps` | List of test steps excluded for execution | |
| `use_omp` | Use OpenMP for compilation | 0 |
| `use_o3` | Use `-O3` optimization flag | 1 |
| `use_c` | Use the C compiler | 1 for C test steps<br>0 for C++ steps |
| `use_gnu90` | Use C standard gnu90 for compilation | 0 |
| `use_gnu99` | Use C standard gnu99 for compilation | 0 |
| `use_cpp` | Use the C++ compiler | 0 for C test steps<br>1 for C++ steps |
| `use_cpp11` | Use C++ standard 11 (`-std=c++11`) | 0 |
| `outputAwk` | Awk regular expression to compare output of test execution | |
| `boost_include` | Path to the boost headers | Path used in Insieme |

| Name | Description | Default value |
|---|---|---|
| boost_lib | Path to the boost library | Path used in Insieme |
| mpfr_home | Path to the mpfr library | Path used in Insieme |
| gmp_home | Path to the gmp library | Path used in Insieme |
| third_part_libs_home | Default path for the third party libraries | Path used in Insieme |
| sortdiff | Path to the sortdiff binary | test/sortdiff |
| time_executable | Path to the time command | /usr/bin/time |

Table 7.1.: Test framework configuration file options

## 7.2. Test Steps

Table 7.2 contains a list of all implemented test steps. Per default all steps are executed in this order. Some steps depend on the successful execution of previous steps. Each step is available in a C and a C++ version, although the table below only contains the C++ steps.

| Name | Description | depends on |
|---|---|---|
| insiemecc_c++_compile | Invokes the Insieme compiler first. Afterwards a third-party compiler is used to generate a binary out of the source code produced by Insieme. | |
| insiemecc_c++_execute | Execute binary generated by insiemecc_c++_compile | insiemecc_c++_compile |
| ref_c++_compile | Compile input code using a reference compiler | |
| ref_c++_execute | Execute binary generated by ref_c++_compile | ref_c++_compile |
| insiemecc_c++_check | Compare output of insiemecc_c++_execute and ref_c++_execute | ref_c++_execute insiemecc_c++_execute |
| main_c++_sema | Dump the generated intermediate language and run semantic checks | |
| main_run_c++_convert | Run the Insieme compiler using runtime backend and dump produced source code | |
| main_run_c++_compile | Compiles the produced source code of main_run_c++_convert using a third-party compiler | main_run_c++_convert |
| main_run_c++_execute | Execute the binary generated in main_run_c++_compile | main_run_c++_compile |
| main_run_c++_check | Compare output of main_run_c++_execute and ref_c++_execute | main_run_c++_execute ref_c++_execute |
| main_seq_c++_convert | Run the Insieme compiler using sequential backend and dump produced source code | |
| main_seq_c++_compile | Compiles the produced source code of main_seq_c++_convert using a third-party compiler | main_seq_c++_convert |

| Name | Description | depends on |
| --- | --- | --- |
| main_seq_c++_execute | Execute the binary generated in main_seq_c++_compile | main_seq_c++_compile |
| main_seq_c++_check | Compare output of main_seq_c++_execute and ref_c++_execute | main_seq_c++_execute ref_c++_execute |

Table 7.2.: List of all test steps

In _check test steps the output of two depending steps are compared using a previously configured awk script. Steps may differ depending on input parameters e.g. an execution step can occur twice using a different number of OpenMP threads. An overview of test steps including their dependencies can be found in Figure 7.1.
An example minimal sequence of test steps would be

- insiemecc_c_compile
- insiemecc_c_execute
- ref_c_compile
- ref_c_execute
- insiemecc_c_check

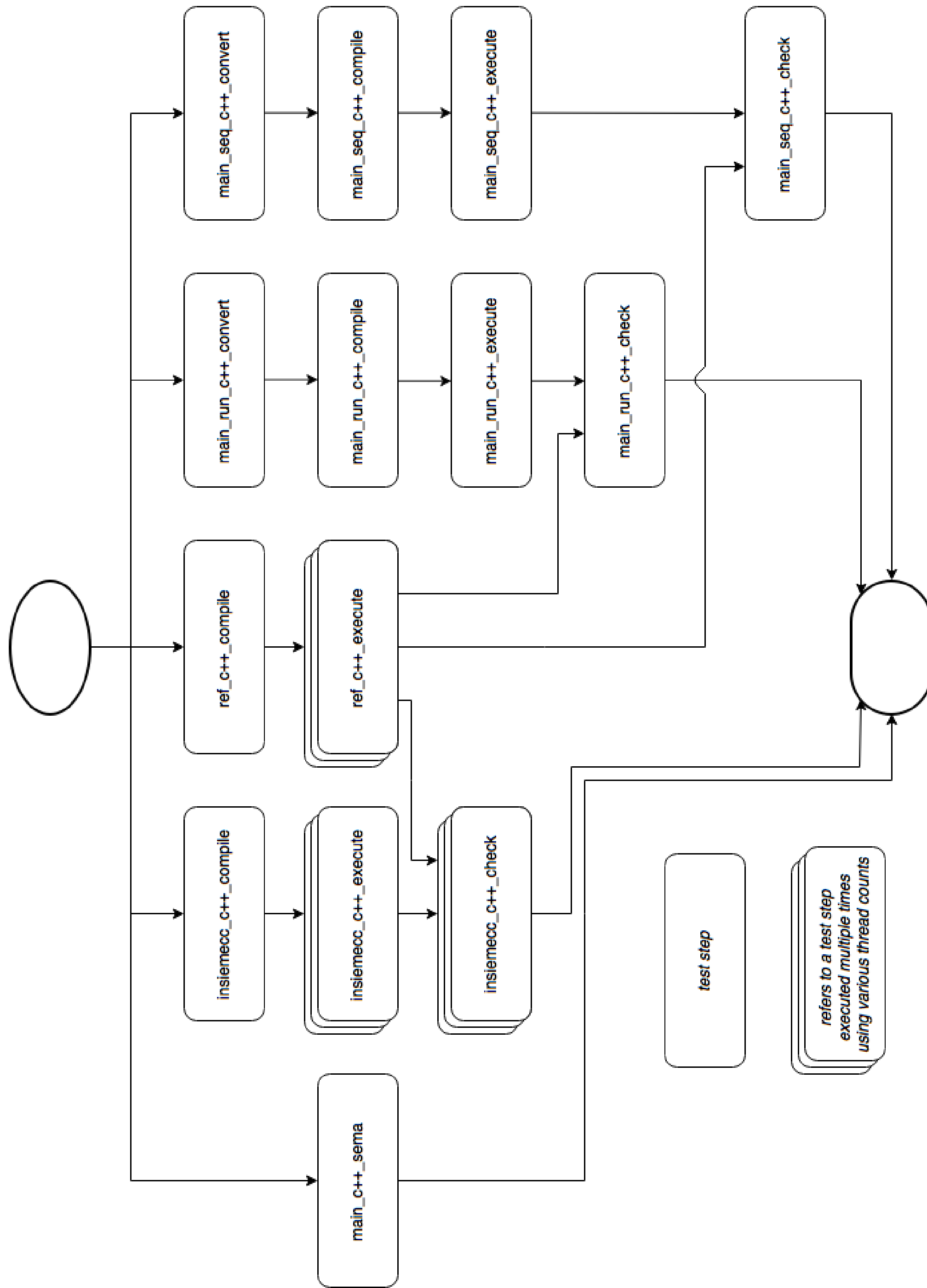


Figure 7.1.: Overview of the integration test steps

## 7.3. Metrics

In Table 7.3 all metrics collected by the integration test tool are shown. The metrics are collected in each step and saved in different output formats (shown in 7.4).

| Name | Description | collected by |
| --- | --- | --- |
| runtime | Walltime in seconds, represents the actual runtime of the application. | /usr/bin/time |
| cputime | Cputime in seconds, represents the amount of time which the processor is actively working. | /usr/bin/time |
| size | Size of input code in | /usr/bin/du |
| parType | Used parallelization technique for compiling, one of<br>- OpenMP<br>- OpenCL<br>- sequential | |
| numOmpPragmas | Count of openMP pragmas in source code | |
| memory | Main memory consumption | /usr/bin/time |
| loc | Lines of code of the input program | CLOC (see 4.2.2) |
| flops | Floating point operations | gnu perf tool |
| llc-load-misses | Last Level Cache Load misses | gnu perf tool |
| llc-store-misses | Last Level Cache Store misses | gnu perf tool |
| any | Any gnu-perf metrics defined by parameters (see 7.5) | gnu perf tool |

Table 7.3.: List of all collectable metrics

## 7.4. Output Formats

The integration test tool supports two output formats to export the collected metrics:

- **SQL**: SQL script to create a MySQL database and save all collected results, details of the generated database can be found in Section 7.4.1.
- **CSV**: A comma separated value file containing all test runs and results.

### 7.4.1. SQL Database

The Entity-Relationship diagram of the generated database is shown in Figure 7.2.

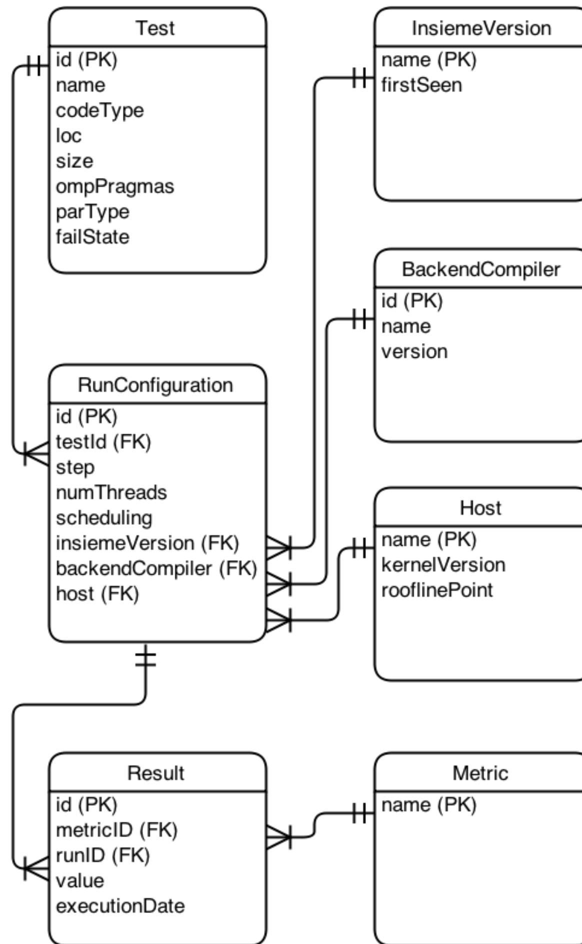


Figure 7.2.: ER Diagram of the generated database

Descriptions of generated tables:

- **Test**: Represents a test case and its static attributes.
- **RunConfiguration**: Represents the environment for an executed test step.
  - *Step*: The executed test step.
  - *Scheduling*: The used scheduling variant for openmp parallel for constructs (e.g. STATIC).

- **InsiemeVersion**: Represents a version of Insieme including a description and a timestamp.
- **BackendCompiler**: A reference compiler and its version (e.g. GCC 4.8.2)
- **Host**: A machine on which tests are executed including its roofline point (see Section 4.2.6) and kernel version.
- **Metric**: Description of a measured metric (e.g. runtime).
- **Result**: The most important object in the database, it contains the result of a measured metric of a test execution. The result contains of:
  - a run configuration,
  - an execution date,
  - a metric description and
  - the actual value.

## 7.5. Command Line Arguments

To reduce the amount of command line arguments two end user tools were made, the first one only executes and checks the integration tests, the second one additionally collects metrics. Some arguments are set as default in the metrics tool. In Table 7.4 all available command line arguments including their default values for the metrics and the integration tool are shown.

| Name | Description | default value | |
|---|---|---|---|
| | | integration | metrics |
| `-h [--help]` | Display help message | NO | NO |
| `-c [--config]` | Print the configuration of the test cases | NO | NO |
| `-m [--mock]` | Make a mock run just printing the commands | NO | NO |
| `-p [--panic]` | Stop on first test step not suceeding | NO | NO |
| `-l [--list]` | List all test cases, do not execute them | NO | NO |
| `-w [--worker]` | The number of parallel workers executing cases | 1 | #NA |
| `--cases` | The test cases to be executed | all cases | all cases |
| `-s [--step]` | The test steps to be executed | all steps | all steps |
| `-r [--repeat]` | The number of times the tests should be executed | 1 | 1 |
| `--clean` | Remove all output files | NO | YES |
| `--nocolor` | No highlighting of output | NO | NO |
| `--use-median` | Use median instead of average to calculate metrics | #NA | NO |

| Name | Description | default value | |
|---|---|---|---|
| | | integration | metrics |
| `--no-perf` | Disable perf metrics | #NA | NO |
| `-S` | Enable runs on all scheduling variants for openmp parallel for pragmas (static, dynamic, guided) | #NA | NO |
| `--no-overwrite` | Do not overwrite existing SQL database | #NA | NO |
| `-t [--threads]` | Number of threads used to execute each test step e.g. `-t 4` executes each test step using 1,2 and 4 threads | #NA | numCores |
| `--load-miss` | Perf code to determine LLC-load-misses | #NA | |
| `--store-miss` | Perf code to determine LLC-store-misses | #NA | |
| `--flops` | Perf code to determine FLOPS | #NA | |
| `-P [--perf-metric]` | A perf code to be measured additionally | #NA | |
| `-o [--output]` | Output formats (SQL or CSV or both) | #NA | no one |
| `-f [--force]` | Force to execute all tests, even commented ones | NO | NO |

Table 7.4.: List of all command line arguments

## 7.6. Implementation Details

This section lists all modules of the integration test framework and shortly introduces in the functionality implemented in each module. For the two test tools *integration test framework* and *metric collection* two different main methods exist. The other modules are shared for both binaries. The following section shortly describes all modules, a class diagram of the application can be found at the end of this section.

### Main (Integration Testframework)

The main method of the integration test runner is implemented in the file *integration_test.cxx*. The following aspects are implemented here:

- Parses the given command line arguments.
- Based on the arguments creates a list of cases by using the method *loadCases* implemented in testframework.
- Prints out an overview of all test cases.
- Creates a list of test cases using the method *getTestSteps* in testframework.
- Executes each test case, if required in parallel using OpenMP.

- Collects information about the execution results.
- Prints the results of the execution (runtime and memory consumption).
- Prints a summary of the integration test run.

### Main (Metric collection)

The main method of the metric collection binary (implemented in *metrics.cxx*) contains the same functionality as the corresponding integration testframework main method. Additional features are:

- Ensures that either C or C++ version for a test case is executed, not both of them.
- Disables the functionality to run tests in parallel to ensure accurate measurements.
- Inserts additional test steps to execute them using a different count of OpenMP threads and different pragma openmp for scheduling variants.
- Implements a functionality to backup test results during execution and restore them if the framework crashes.
- Write all required output formats to files using the *TestOutput* class.

### TestFramework

The testframework module implemented in *test_framework.h* contains universal methods and structures used in several parts of the application:

- Structure **Options**
  contains options to control the application flow. Most of them are set using input parameters or are created in the main method.
  Sample options are *num_threads, clean_files, test_cases, test_steps, perf_metrics.*
- Structure **Colorize**
  defines options for colorized output (used in main methods).
- Method **getGitVersion**
  uses the command *git describe* to return a version of the source code.
- Method **loadCases**
  parses the test framework directories defined as command line parameters and creates a structure containing the appropriate test cases.

- Method **getTestSteps**
  creates a list of test steps based on the given options and command line parameters.

### Integrationtestcase

The class *IntegrationTestCase* contains all information needed to describe and execute a test case. Implementation of this class can be found in the file *tests.h*. The class contains:

- name,
- directory,
- input files,
- include directories needed for compilation,
- external libraries (directories and names),
- interceptedNameSpaces, a list of namespaces intercepted by the Insieme compiler frontend,
- interceptedHeaderFileDirectories, a list of directories containing header files to be intercepted,
- enableOpenmp, a flag which indicates if OpenMP should be used,
- enableOpenCL, a flag indicating if OpenCL should be used,
- enableCXX11, a flag to enable the C++11 standard
- and a structure *properties* containing other test case options.

**Method getCompilerArguments**
This method parses the properties of a test case and creates, based on a specific test step, a list of arguments which have to be passed to the compiler. An example of such a argument:

- Test case property *use_libmath* is set to true,
- test step is *ref_c_compile*,
- used compiler is *gcc*
- the resulted compiler flag is **-lm**.

In file *tests.cpp* methods to manipulate test cases are implemented:

- Method **loadAllCases** takes the path of a directory and loads all contained test cases into a collection. For this it recursively calls itself until the leaf of the directory tree is reached, then the method
- **loadTestCase** creates a test case object. Loads all information of the test configuration file into it and uses
- **loadProperties** to parse all properties of the test configuration files into the test case object.
- Methods **getAllCases** which returns all test cases of a default path and **getAllCasesAt** which parses all cases of a specified path use the three methods above to return a collection of all requested test cases.

### Properties

The properties module implemented in *properties.h* contains all properties of a test case specified in its configuration file. Methods to parse the input file and create collections for several options are implemented here.

### TestStep

The file *test_step.h* contains all information needed for a specific test step:

- Structure **TestSetup** contains information needed to execute a test step, for example number of threads or execution directory.
- **StepType**, defines if the step is one of
  - compile step,
  - run step,
  - output check step or
  - a static metric step.
- **TestCase**, a concrete test case which this test step belongs to.
- **TestRunner**, a class which is able to execute the test case.
- String **name**
- Vector **dependencies**, defines which test steps need to be executed before this test step. For Example the test step *ref_c_execute* depends on the successful execution of *ref_c_compile*.

- Method **run**, executes the test case using the specified test runner.

The file *test_step.cpp* contains methods to create and execute test steps for a given test case. It is used by the main methods to execute test cases. The most important components here are

- Method **scheduleSteps**, returns a list of test steps based on test case, dependencies and command line arguments. This method uses
- method **getStepByname** which returns a test step object based on its name. Uses
- method **createFullStepList**. This method creates all test step objects. If statistic binary is used additional test steps for different thread counts and scheduling variants are created.
- A method **filterSteps** filters specific test steps based on a given command line argument.
- Several methods to create specific test steps are also defined here, e.g. a *RefCheckStep* or an *InsiemeCompileStep*.
- A test runner **executeWithTimeout** is also implemented here. It uses the method *runCommand* to create the execution command (including environment variables and metric measurement commands) executes the test step and returns an object of *TestResult*. After a specified timeout the execution of the compiled binary is aborted to limit the runtime of the test tool.

### TestResult

The TestResult module contains all results of a test step, it is created after the execution of the test. The main components are:

- String name,
- integer return value,
- success flag,
- a list of metric results as well as a deviation value if multiple runs are done,
- string output of the execution,
- string standard error output,

- a string representing the executed command,
- a list of produced files,
- an integer value of the used thread count,
- the used scheduling policy,
- a map containing static metrics (e.g. lines of code) and
- a flag if the execution was aborted.

There are also three methods implemented:

- **Clean**, deletes all produced files,
- **returnAvg**, returns average values of multiple test runs and
- **returnMedian**, returns the median of multiple test runs.

### TestOutput

The test output class implements methods to write test results onto different outputs. Currently only two output methods are implemented:

1. CSV, write all results into a comma separated value file
2. SQL, create a SQL script to generate a database and fill in the results.

### Class Diagram

Figure 7.3 shows a class diagram of the main components of the integration test framework.

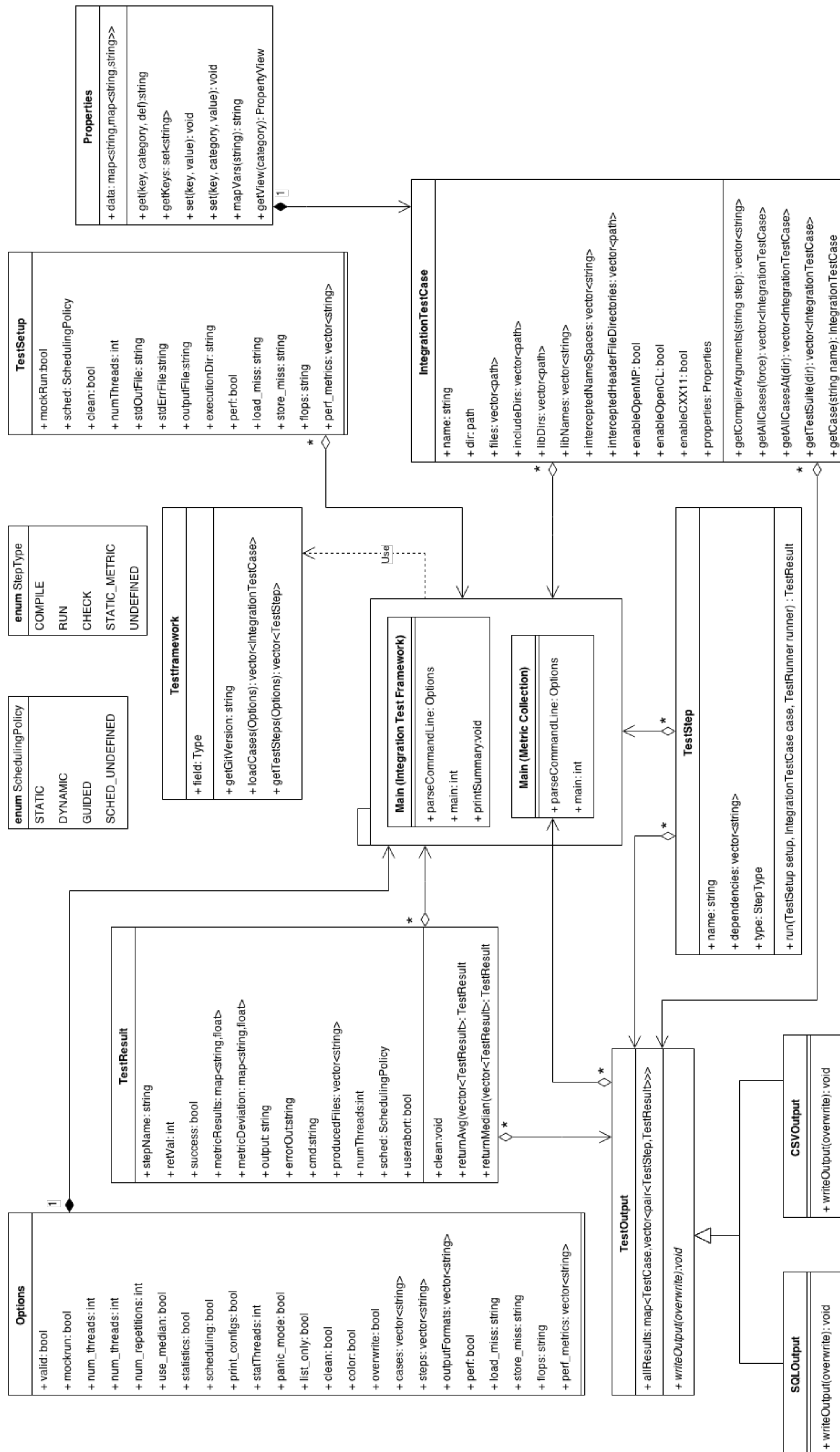


Figure 7.3.: Class diagram of the integration test framework

# Chapter 8.

# Detailed Performance Analysis at the Example of GALPROP

*This work was supported by the Austrian Ministry of Science BMWF as part of the Konjunkturpaket II of the Focal Point Scientific Computing at the University of Innsbruck.*

The following chapter shows a detailed performance analysis of the GALPROP code. GALPROP is a numerical code for calculating the propagation of relativistic charged particles and the diffuse emissions produced during their propagation. It was developed by the University of Stanford, the latest public version is 54.1.98 released in July 2011. For more information about the code see [27].
In the following sections a deep analysis of the application is done and improvements regarding shared memory parallelism are shown.

## 8.1. Testing Environment

All analysis and improvements were done on the MACH [28] supercomputer. MACH is a high performance computing system of the Austrian Center for Scientific Computing and is a collaborative effort of the University of Innsbruck and the University of Linz. MACH comprises one large shared memory system with 2048 cores composed by 256 8-core Intel Xeon E78837 processors. MACH is equipped with an overall of 16 TB of main memory.
The used compilers are intel icpc and ifortran in version 12.0.4.
As profiling tools intel VTune amplifier 2014 and callgrind from the Valgrind (version 3.5.0) tool suite were used.
GALPROP comes with several sample input files, two of them are chosen for testing:

- NoGamma
  Provides a run lasting approximately one hour using one core. Characteristics are:
  - 3 dimensions are used,
  - only primary hydrogen and electrons and
  - secondary electrons are computed.
  - Gamma ray sky maps are not calculated.
  - This job only solves the diffusion equation, hence it might be suitable to probe the performance of the solver.
- WithGamma
  Same as NoGamma but gamma ray skymaps are calculated.

Each input file comes with different input sizes, they differ in the accuracy of the results delivered by GALPROP. For the analysis two input sizes were chosen Lr (LowResolution) and Mr (MediumResolution). For the exact input file parameters see appendix A.

## 8.2. Current Behavior

The original version of GALPROP includes some parallel shared memory aspects using OpenMP. However the parallelization was done in a very rudimentary way, therefore the performance results on shared memory systems with more than eight cores were not satisfactory. The speedup[1] and efficiency[2] results of a basic black box measurement are shown in Figures 8.1 and 8.2. It can be seen that the application only gets three to six times faster when using 15 cores. By increasing the number of cores the speedup even reaches a value lower than 1, which means the application gets slower by using more threads.

[1] Speedup: $S_p = \frac{T_1}{T_p}$, where $T_1$ is the runtime using one thread $T_p$ the runtime using $p$ processors. Ideally $p$ is equal to $S_p$.

[2] Efficiency: $E_p = \frac{S_p}{p}$, where $S_p$ is the speedup using $p$ threads. Ideally the efficiency is 1, which indicates a perfect speedup.

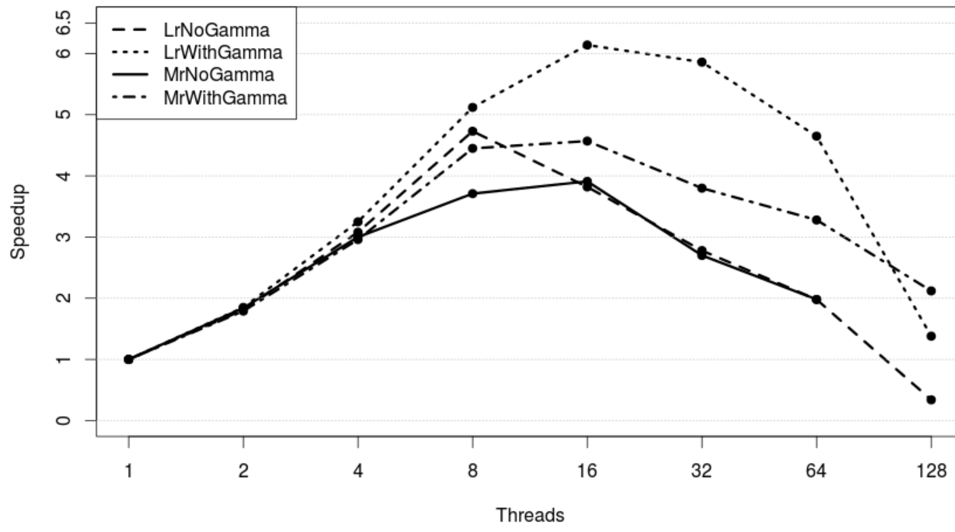


Figure 8.1.: Speedup of the original GALPROP version

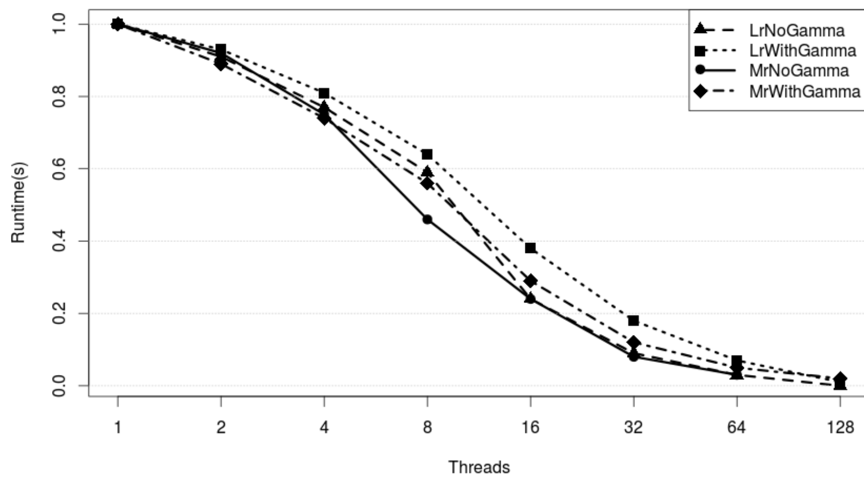


Figure 8.2.: Efficiency of the original GALPROP version

### 8.2.1. Code Regions

Based on their parallelization strategy, the GALPROP code was divided into seven code regions:

- NoGamma,
- gen_IC_emiss,
- Serial,
- gen_pi0,
- gen_bremss,
- gen_synch and
- gen_IC_skymap.

For the WithGamma input all code regions are executed, for the NoGamma input file only the NoGamma part is used.
The results of a shared memory analysis in intel vTune are shown in Figure 8.3. Each row represents the execution of one OpenMP thread. For each thread the CPU load is shown in brown. The light green shows idle times and therefore lost performance. The small red parts show OpenMP overhead and spin time, e.g. thread management, work distribution, scheduling, synchronization. The yellow lines show synchronization points, e.g. barriers or critical regions.

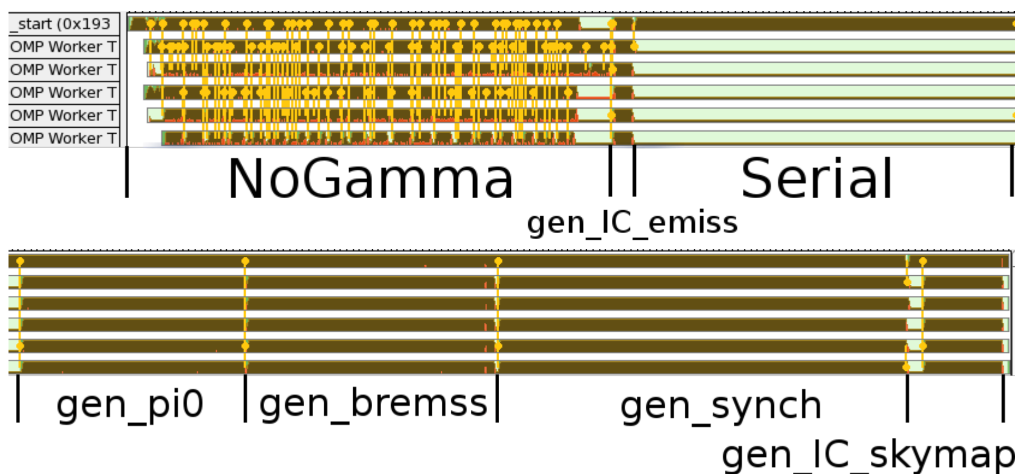


Figure 8.3.: Shared memory analysis of GALPROP running with six threads using intel vTune

This analysis already exposes the major problems of the parallelization strategy. The NoGamma part contains a high number of synchronization points. How the efficiency of the code regions behaves when increasing the number of threads is shown in Figures 8.4 and 8.5. It can be seen that in general all code regions scale. Exceptions here are NoGamma, gen_synch and Serial.

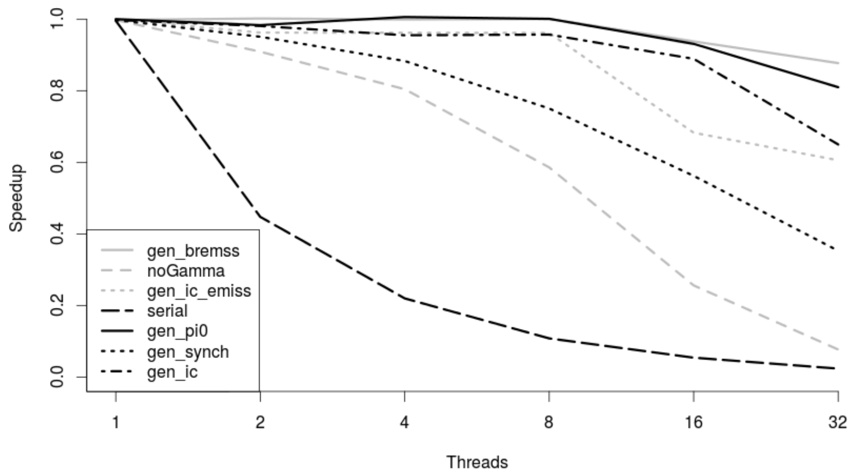


Figure 8.4.: Efficiency of code regions

The following part shows a deeper insight into the code regions.

### gen_IC

As seen in the Figures 8.3 and 8.4 gen_IC scales very well, the runtime decreases nearly linearly for an increasing number of threads. Therefore this region is not considered in detail.

### gen_synch

As gen_IC gen_synch's runtime is decreasing continuously by increasing the number of threads. This part is the longest part when using one thread but for a higher number of threads it gets more and more irrelevant. Nevertheless a small part of this code region is not parallelized yet, that is the reason why the efficiency values are not optimal.

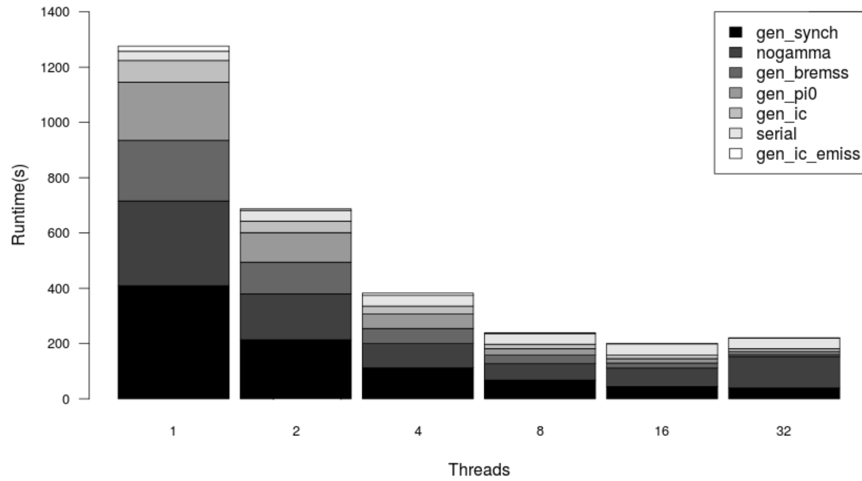


Figure 8.5.: Absolute runtime of each region

### NoGamma

This region is responsible for a big part of the runtime and does not scale well (shown in Figure 8.4). This part is the main problem for optimization, if this part gets slower for a high number of threads the whole calculation does not scale well. The two main procedures in this area are *propagate_particles* and *Galprop::propel*. Propel is the troublemaker regarding runtime, propagate_particles only calls propel several times.
The *propel* method offers two solution ways whereby solution_method=2 is faster but numerical less stable than solution_method=1. In solution_method=1 a basic parallelization is already done. The problems in this solution are loop carried dependencies in several nested loop constructs. Therefore it is necessary to insert a high number of barriers (as seen in Figure 8.3). Barriers guarantee that all threads are synchronized. If the workload is not evenly distributed faster threads have to wait for slower threads. Waiting time is generated. If now the number of threads is increased the waiting time grows and at a certain point the overhead is higher than the improvement gained by load splitting. This is one of the main problems a developer has to deal with when parallelizing applications called load inbalance.

### Serial

The Serial code region is not parallelized at all, the whole calculation is done on one thread. As already shown in Section 6.1 it is very important to paralellize even small parts of the calculation. Therefore this part is an essential issue to increase the scalability of the whole application.

### gen_bremss and gen_pi0

These two regions scale well, therefore no detailed analysis is needed.

### gen_IC_emiss

This part consumes nearly no runtime, a basic parallelization is already done. So this part is neglectable.

# 8.3. Improvements

## 8.3.1. Improvements Regarding Shared Memory Systems

This section shows the improvements done regarding parallelization to run the application using multiple threads. Optimizations were done in the code regions NoGamma, Serial and gen_synch.

### NoGamma

In the *propagate_particles* method a for loop is executed, which executes the propel method in each iteration. This for loop does not contain any loop carried dependencies, therefore each iteration can be executed in parallel on different cores. Unfortunately the number of iterations is not very high, for all tested problems it is four. The parallelization of this loop would only scale up to four threads. As a result of this, nested parallelism was introduced. Nested parallelism means that each thread splits the load onto multiple sub-threads. Nested parallelism allows to use a high number of available cores even if the number of the outer loop iterations is low.

An example listing of nested parallelism (optimal for a system using six cores) is shown in listing 8.1.

Listing 8.1: Nested parallelism example

```
omp_set_nested(1);  //enable nested parallelism
#pragma omp parallel for num_threads(3) //parallelize outer
    loop
for 1 to 3 do
serial_execution()
...
#pragma omp parallel for num_threads(2) //inner loop
  for 1 to 10 do
    serial_execution()
    ...
  end for
  ...
end for
```

If six cores are available the parallelization of one of the two for loops is not sufficient for load splitting in an optimal way. If the outer loop is parallelized only three cores can be utilized. If the inner loop is chosen some parts of the calculation remain serial. In this case the optimal way is to use three threads for the outer loop and each thread creates again two threads on the inner loop. In the NoGamma region the outermost loop is located in propagate_particles, the inner loops are present in the Galprop::propel method.
To efficiently use this concept an optimal number of outer and inner threads has to be calculated. This is done by using the following algorithm:

$$n_species \text{ ... number of loop iterations of outer loop}$$
$$maxThreads \text{ ... number of available cores}$$
$$numInnerThreads = min(8, max(1, floor(\frac{maxThreads}{n_{species}})))$$
$$numOuterThreads = min(maxThreads, n_species)$$

The best number of inner threads is the number of available cores divided by the number of iterations of the outer loop. If this value is smaller than 1 it is set to 1. If this value is bigger than 8 it is set to 8 because in this particular case the inner loop iterations of the propel method only scale till a thread count of 8 (see also NoGamma in Figure 8.4). If we allow setting the inner thread count to a value higher than 8 the inner loop calculation would get slower and the runtime of the whole part would increase. This is a result of the high amount of loop carried dependencies, details in section 8.2.1.
The best number of outer threads is either the number of cores available or the number of outer loop iterations.

Sometimes it is needed to tune the thread counts, this is done by the algorithm shown in listing 8.2.

Listing 8.2: Algorithm to tune number of threads in nested parallelism

```
// While not all cores used, increase number of inner threads
while(numOuterThreads*numInnerThreads!=maxThreads){
  numInnerThreads++;

  //If the number of threads exceeds number of cores reduce number of outer threads
  if(numOuterThreads*numInnerThreads>maxThreads)
    numOuterThreads--;

  //If we run out of bounds (1 for outer, 8 for inner) no better values are possible -> return to original values (before tuning)
  if(numOuterThreads=1 || numInnerThreads=8)
    set thread values back to original
    break;
}
```

The results of this improvement are shown in Figure 8.6. The dashed line refers to the old version, the solid line represents the improved version.

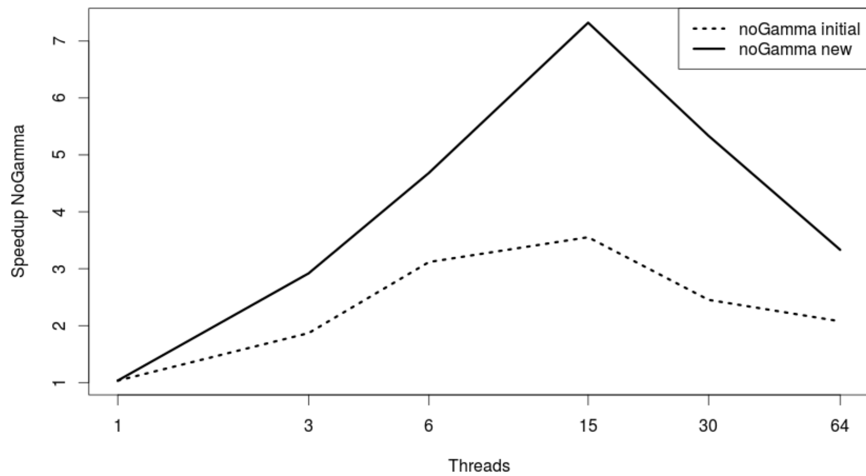


Figure 8.6.: Speedup improvement results of the NoGamma region

It can be seen that the NoGamma part scales better than before, but as already mentioned the high number of barriers in the Galprop::propel method slows down the whole application.
At a certain point the synchronization overhead is too high. This point is not avoided it is only moved to a higher number of threads. Now the NoGamma part scales well until 15 threads, before the optimization this point was at eight threads.

### gen_synch

The *gen_synch_emiss* method in this region contains a large for loop with no loop carried dependencies. This loop was parallelized using a simple *pragma omp parallel for* pragma. Parallelization of this loop led to a small performance improvement of the gen_synch region, shown in Figure 8.7.

### Code Region Serial

In this region two big loops are executed, both of them not containing any loop carried dependencies. Basic parallelization was done, results can be found in Figure 8.7.

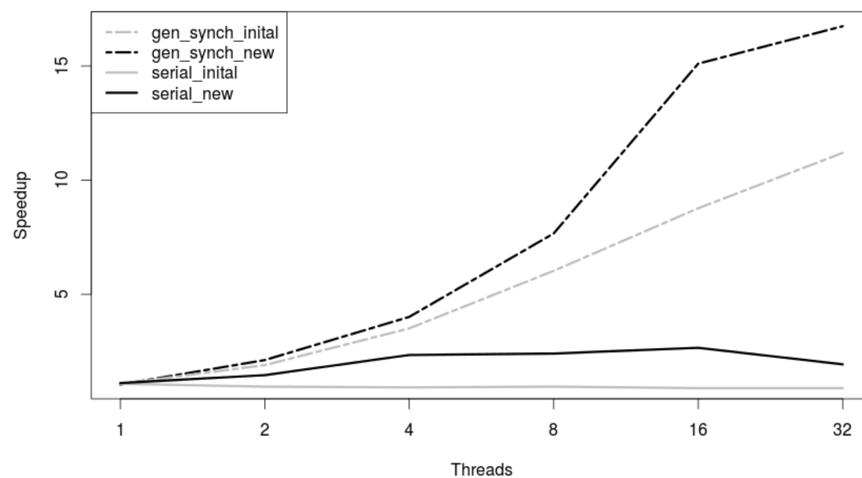


Figure 8.7.: Performance improvement results of the Serial and the gen_synch region

### 8.3.2. Serial Improvements

In addition to the optimization regarding shared memory systems some improvements of the serial code were done. This includes:

- Usage of slow data structures,
- string comparisons,
- dead code,
- duplicate code and
- many other bad code smells.

However, the code is still highly unstructured and contains the usage of many bad structures.

#### Code Region Serial

The method *readGasMaps* contains unnecessary redundant string comparisons, they were replaced by boolean variables. Additionally, some if conditionals were removed and replaced by if-else constructs. Listings 8.3 and 8.4 below show an excerpt of the changes:

Listing 8.3: initial readGasMaps

```
readGasMaps (String method){
  ...
  if(method == "HIR")
  //occurs 20 times
    ...
  if(method == "COR")
  //occurs 20 times
    ...
}
```

Listing 8.4: optimized readGasMaps

```
readGasMaps(String method){
  ...
  bool cor;
  //NEW: boolean
  if(method=="HIR")
    cor=false;
  else
    cor=true;
  ...
  if(cor)
    ...
  else
  //NEW: else
    ...
}
```

The optimization removed 40 string comparisons, which are rather slow, and replaces them by 20 if-else constructs which use the cached boolean result for comparing.

The benefits of the new version are rather small, the optimization is more a refactoring step to reach better code readability than a performance improvement. Measurement results for two input parameters are listed in Table 8.1.

| input problem | initial | optimized |
|---|---|---|
| HIR | 17.6s | 17.4s |
| COR | 18.3s | 17.1s |

Table 8.1.: Performance optimization results of readGasMaps

#### NoGamma - Method B_field_3D_model

This method contained a high number of string comparisons, done by a row of if constructs. These comparisons can be reduced by using if-else constructs instead. Listings 8.5 and 8.6 show a short excerpt of the optimization:

Listing 8.5: initial B_field_3D_model

```
...
if(name=="test"){
  ...
}
if(name=="circular"){
  ...
}
if(name=="circular2"){
  ...
}
  //25 more
...
```

Listing 8.6: new B_field_3D_model

```
...
if(name=="test"){
  ...
}
else if(name=="circular"){
  //NEW: else if
  ...
}
else if(name=="circular2"){
  ...
}
  //25 more
...
```

The benefit of this version is that not all 25 comparisons have to be done for each call of the method. The results of this optimization are shown in Table 8.2. Since the runtime of the B_field_3D_model method is highly dependent

| initial | optimized |
|---|---|
| 19.2s | 17.1s |

Table 8.2.: Performance optimization results of B_field_3D_model

on input values the table shows the total runtime of the method (sum of all calls) during a run of GALPROP using the input problem WithGamma.

## 8.4. Results

This section lists all results of the performance improvements. The absolute values used for the charts can be found in appendix B.
In Figure 8.8 the achieved increase of speedup is shown. Dashed lines refer to the old version, solid ones to the new improved version. It can be seen that, especially for a large number of threads, the speedup is better than before. But it still does not scale for a large number of threads.

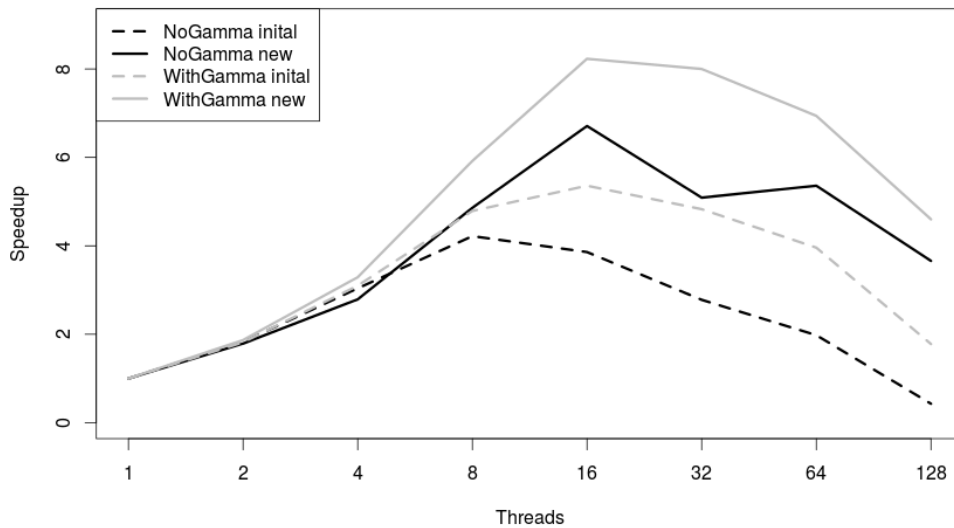


Figure 8.8.: Speedup improvement results

In Figure 8.9 the absolute runtime values are compared. An interesting fact is that the runtime of the initial version is growing very high for a number of threads bigger than 64. This is reduced in the new version.

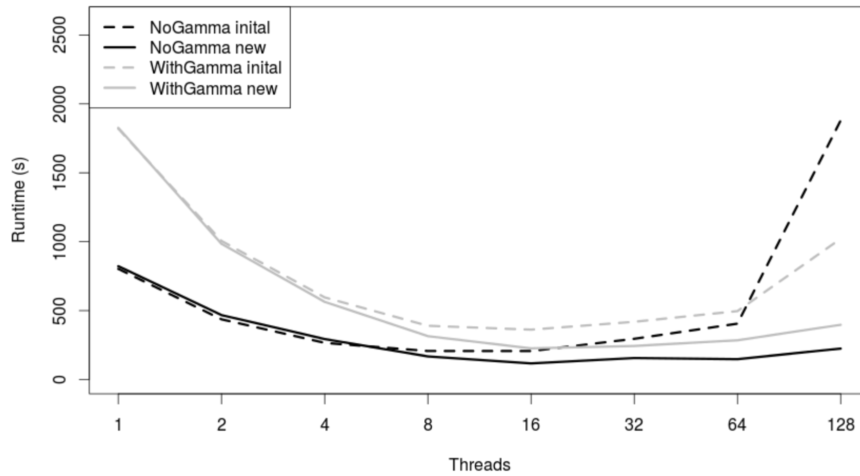


Figure 8.9.: Runtime improvement results

## 8.5. Overhead Analysis

The following section shows an analysis of the overheads occurring in GALPROP. This is helpful to get a detailed understanding of the remaining problems.

In Figure 8.10 the overhead of LrWithGamma based on the number of threads is shown. It can be seen that the cpu time stays nearly the same for any number of threads. Synchronization overhead and wait time increase rapidly by increasing the number of threads. This is the consequence of the high number of synchronization pragmas. OpenMP overhead is negligible.

To get a better understanding which region generates this overhead an overhead analysis per region was done. Results of a LrWithGamma run using 16 threads are shown in Figure 8.11. The chart shows the waiting time (black) and synchronization overhead (white). It is visible that especially NoGamma and Serial produce a high amount of waiting time. In NoGamma this is a result of the high number of synchronizations. The reason for the high waiting time in Serial is the missing parallelization. In this particular case 15 threads have to wait until the first thread finishes the execution. Gen_pi0 and gen_synch produce nearly no waiting time, they are parallelized well.

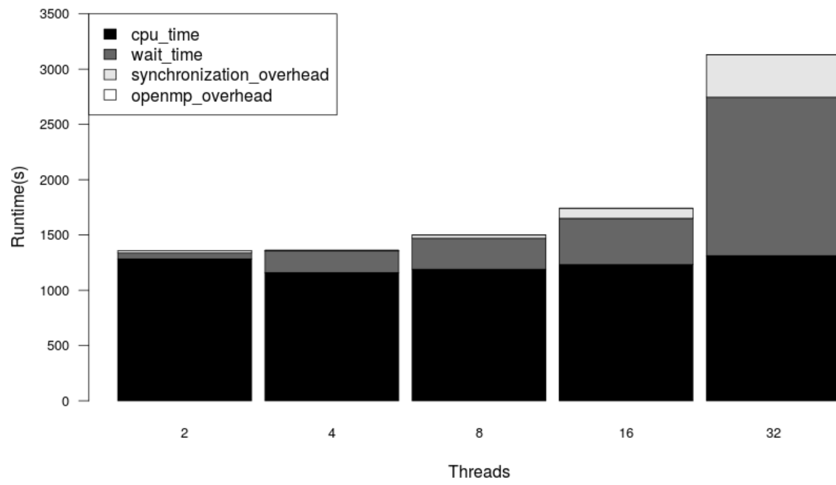


Figure 8.10.: Overhead analysis of GALPROP running with input LrWithGamma

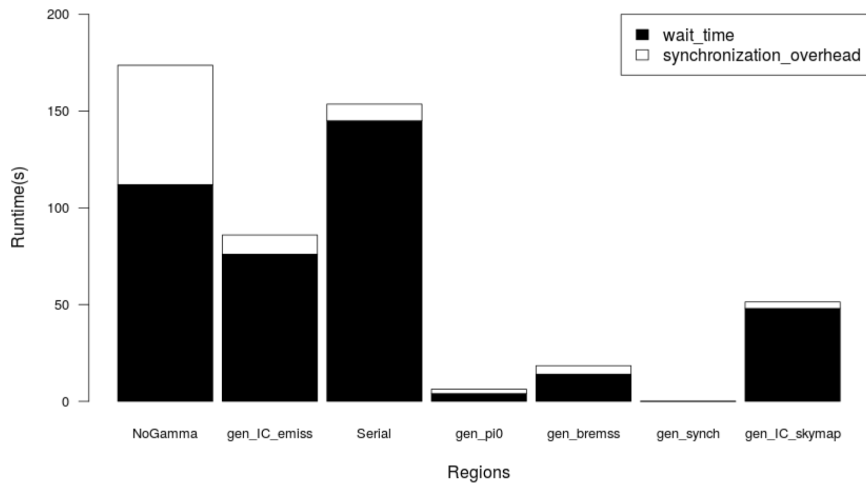


Figure 8.11.: Overhead analysis based on code regions, input LrWithGamma, number of threads is 16

Another interesting chart is the normalized waiting time shown in Figure 8.12.

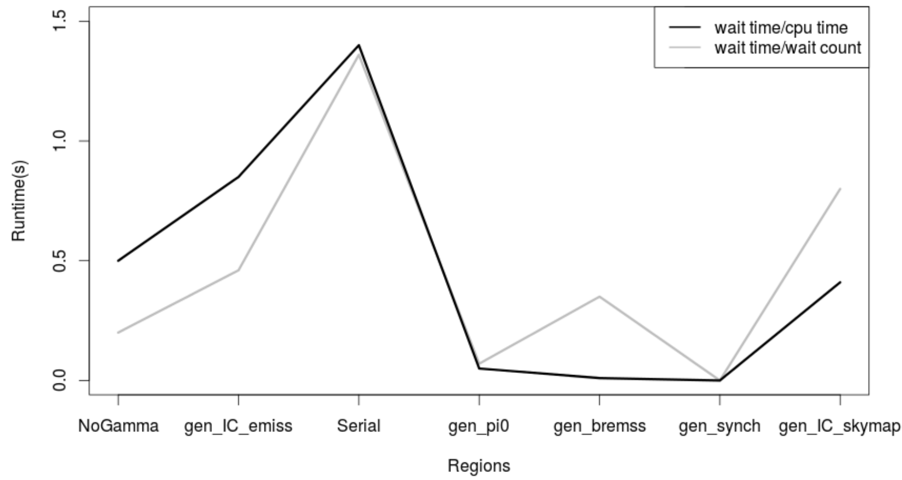


Figure 8.12.: Normalized waiting time of GALPROP, input LrWithGamma, number of threads is 16

This analysis exposes the fact that NoGamma is not the worst code region, gen_IC_emiss and Serial behave even worse. NoGamma consumes a high amount of absolute waiting time but also a high amount of cpu time. Additionally, the wait time per wait is low. Gen_IC_emiss and Serial consume for each second of cpu time over one second of waiting time and each wait lasts for 0.8 to 1.3 seconds. The other code regions, as already seen in other performance statistics, perform well.

## 8.6. Further Improvements

#### Propagate_particles solution_method=1

As already mentionend in 8.2.1 this method is not easy to parallelize. An approach to get rid of the nested loops and dependend matrix operations would be to change the whole mathematical algorithm. This would require a rewriting of the whole method.

#### Improve Load Balancing in Gamma Part

Load balancing in the Gamma part is done very rudimentary, maybe this could be improved by changing the whole algorithm.

#### Prove if Methods in Gamma Part are Independent

The numerous methods called in all code regions (except NoGamma) may run in parallel, if they do not depend on each other. Most of the methods use results calculated before but maybe some of them depend on the same results. To find such parallelization opportunities it is necessary to create a dependency graph of all methods and search for parallel paths.

#### Serial Code Improvements

The whole code is full of slow structures, e.g. string comparisons, cache unfriendly array accesses. The change of the whole code to the usage of performant collections, especially to new C++11 features would decrease the runtime. Since some parts of the code are not easy to read, this would be a big and time consuming task.

# Chapter 9.

# Summary

This master thesis presented the search of input codes for the Insieme compiler project. The compiler, currently under development by the DPS group, assists developers in optimizing applications for parallel execution. The compiler supports C and C++ codes using parallelization paradigms OpenMP, OpenCL and MPI. A runtime system to execute and further optimize Insieme-compiled applications is provided.
Input codes for the Insieme project are collected and several metrics to categorize the codes were measured. The most important metrics are speedup, efficiency, memory consumption and boundness (memory/compute). All these metrics are raised by executing the code using a variable count of OpenMP threads (1, 2, 4, 8, 16, 32, 64). The results of the codes compiled with GCC are compared to the results gained using the Insieme compiler. The metrics can be used to optimize the compilation process for a variety of input codes.
The collected test codes additionally act as integration tests for Insieme. To provide continuous testing during the development process an integration test framework was built. The test framework is able to compile test codes using a reference compiler and the Insieme compiler. After the compilation phase test codes are executed and execution results of the different versions are compared for a basic proof of validity. Additionally the test framework is able to gain all mentioned metrics and write them into a CSV and/or into a SQL database.
The second part of the thesis deals with the optimization of a product application in the field of astrophysics (GALPROP). The code was developed by the University of Stanford. Parallelization of the code is present but it was done very rudimentarily. Reasons why the code does not scale well are shown and proposals to improve the parallel execution are made. For the work on GALPROP, the MACH supercomputer of the Austrian Center for Scientific Computing was used.

# Appendix A.

# Input Parameters for GALPROP

## A.1. NoGamma Low Resolution input file

| | |
|---|---|
| **n_spatial_dimensions** | 3 |
| **r_min** | 0.0 |
| **r_max** | 20.00 |
| **dr** | 0.15 |
| **z_min** | −4.0 |
| **z_max** | +4.0 |
| **dz** | 0.25 |
| **x_min** | −15.0 |
| **x_max** | +15.0 |
| **dx** | 1 |
| **y_min** | −15.0 |
| **y_max** | +15.0 |
| **dy** | 1 |
| **Ekin_min** | $1.0e2$ |
| **Ekin_max** | $1.0e6$ |
| **Ekin_factor** | 1.5 |
| **p_Ekin_grid** | *Ekin* |
| **E_gamma_min** | 100. |
| **E_gamma_max** | 100000. |
| **E_gamma_factor** | 10 |
| **integration_mode** | 0 |
| **nu_synch_min** | $1.0e6$ |
| **nu_synch_max** | $1.0e10$ |
| **nu_synch_factor** | 2.0 |
| **long_min** | 0.25 |
| **long_max** | 359.75 |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | | | |
|---|---|---|---|
| **lat_min** | $-89.75$ | | |
| **lat_max** | $+89.75$ | | |
| **d_long** | 0.5 | | |
| **d_lat** | 0.5 | | |
| **healpix_order** | 7 | | |
| **lat_substep_number** | 1 | | |
| **LoS_step** | 0.01 | | |
| **LoS_substep_number** | 1 | | |
| **D0_xx** | $5.80e28$ | | |
| **D_rigid_br** | $4.0e3$ | | |
| **D_g_1** | 0.33 | | |
| **D_g_2** | 0.33 | | |
| **diff_reacc** | 1 | | |
| **v_Alfven** | 30. | | |
| **damping_p0** | $1.e6$ | | |
| **damping_const_G** | 0.02 | | |
| **damping_max_path_L** | $3.e21$ | | |
| **convection** | 0 | | |
| **v0_conv** | 0. | | |
| **dvdz_conv** | 10. | | |
| **nuc_rigid_br** | $9.0e3$ | | |
| **nuc_g_1** | 1.98 | | |
| **nuc_g_2** | 2.42 | | |
| **inj_spectrum_type** | *rigidity* | | |
| **electron_g_0** | 1.60 | | |
| **electron_rigid_br0** | $4.0e3$ | | |
| **electron_g_1** | 2.42 | 2.3 | 2.54 |
| **electron_rigid_br** | $1.0e9$ | | |
| **electron_g_2** | 5.0 | | |
| **He_H_ratio** | 0.11 | | |
| **n_X_CO** | 10 | | |
| **X_CO** | $1.9E20$ | | |
| **propagation_X_CO** | 2 | | |
| **nHI_model** | 1 | | |
| **nH2_model** | 1 | | |
| **nHII_model** | 1 | | |
| **B_field_model** | 050100020 | *bbbrrrzzz* | |
| **B_field_name** | *galprop_original* | | |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | | | | |
|---|---|---|---|---|
| **n_B_field_parameters** | 10 | | | |
| **B_field_parameters** | $7.0e-6$ | 50.0 | 2.00 | 00.00 |
| | $0.00e-6$ | 0.0 | 0.0 | 0.0 |
| | 0.0 | 0.0 | | |
| **fragmentation** | 1 | | | |
| **momentum_losses** | 1 | | | |
| **radioactive_decay** | 1 | | | |
| **K_capture** | 1 | | | |
| **ionization_rate** | 0 | | | |
| **solution_method** | 1 | | | |
| **start_timestep** | $1.0e8$ | | | |
| **end_timestep** | $1.0e2$ | | | |
| **timestep_factor** | 0.5 | | | |
| **timestep_repeat** | 20 | | | |
| **timestep_repeat2** | 0 | | | |
| **timestep_print** | 10000 | | | |
| **timestep_diagnostics** | 10000 | | | |
| **control_diagnostics** | 0 | | | |
| **network_iterations** | 1 | | | |
| **prop_r** | 1 | | | |
| **prop_x** | 1 | | | |
| **prop_y** | 1 | | | |
| **prop_z** | 1 | | | |
| **prop_p** | 1 | | | |
| **use_symmetry** | 0 | | | |
| **vectorized** | 0 | | | |
| **source_specification** | 0 | | | |
| **source_model** | 1 | | | |
| **source_parameters_1** | 0.475063 | 1.25 | 2.35 | |
| **source_parameters_2** | 2.16570 | 3.56 | 5.56283 | |
| **source_parameters_3** | 15.0 | | | |
| **source_parameters_4** | 10.0 | | | |
| **n_cr_sources** | 0 | | | |
| **cr_source_x_01** | 10.0 | | | |
| **cr_source_y_01** | 10.0 | | | |
| **cr_source_z_01** | 0.1 | | | |
| **cr_source_w_01** | 0.1 | | | |
| **cr_source_L_01** | 1.0 | | | |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | | | |
|---|---|---|---|
| **cr_source_x_02** | 3.0 | | |
| **cr_source_y_02** | 4.0 | | |
| **cr_source_z_02** | 0.2 | | |
| **cr_source_w_02** | 2.4 | | |
| **cr_source_L_02** | 2.0 | | |
| **SNR_events** | 0 | | |
| **SNR_interval** | $1.0e4$ | | |
| **SNR_livetime** | $1.0e4$ | | |
| **SNR_electron_sdg** | 0.00 | | |
| **SNR_nuc_sdg** | 0.00 | | |
| **SNR_electron_dgpivot** | $5.0e3$ | | |
| **SNR_nuc_dgpivot** | $5.0e3$ | | |
| **proton_norm_Ekin** | $1.00e+5$ | | |
| **proton_norm_flux** | $5.75e-9$ | $6.75e-9$ | $5.75e-9$ |
| **electron_norm_Ekin** | $3.45e4$ | | |
| **electron_norm_flux** | $0.32e-9$ | | |
| **max_Z** | 1 | | |
| **use_Z_1** | 1 | | |
| **use_Z_2** | 1 | | |
| **use_Z_3** | 1 | | |
| **use_Z_4** | 1 | | |
| **use_Z_5** | 1 | | |
| **use_Z_6** | 1 | | |
| **use_Z_7** | 1 | | |
| **use_Z_8** | 1 | | |
| **use_Z_9** | 1 | | |
| **use_Z_10** | 1 | | |
| **use_Z_11** | 1 | | |
| **use_Z_12** | 1 | | |
| **use_Z_13** | 1 | | |
| **use_Z_14** | 1 | | |
| **use_Z_15** | 1 | | |
| **use_Z_16** | 1 | | |
| **use_Z_17** | 1 | | |
| **use_Z_18** | 1 | | |
| **use_Z_19** | 1 | | |
| **use_Z_20** | 1 | | |
| **use_Z_21** | 1 | | |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | |
|---|---|
| **use_Z_22** | 1 |
| **use_Z_23** | 1 |
| **use_Z_24** | 1 |
| **use_Z_25** | 1 |
| **use_Z_26** | 1 |
| **use_Z_27** | 1 |
| **use_Z_28** | 1 |
| **use_Z_29** | 0 |
| **use_Z_30** | 0 |
| **iso_abundance_01_001** | $1.06e{+}06$ |
| **iso_abundance_01_002** | 0. |
| **iso_abundance_02_003** | 9.033 |
| **iso_abundance_02_004** | $7.199e{+}04$ |
| **iso_abundance_03_006** | 0 |
| **iso_abundance_03_007** | 0 |
| **iso_abundance_04_009** | 0 |
| **iso_abundance_05_010** | 0 |
| **iso_abundance_05_011** | 0 |
| **iso_abundance_06_012** | 2819 |
| **iso_abundance_06_013** | $5.268e{-}07$ |
| **iso_abundance_07_014** | 182.8 |
| **iso_abundance_07_015** | $5.961e{-}05$ |
| **iso_abundance_08_016** | 3822 |
| **iso_abundance_08_017** | $6.713e{-}07$ |
| **iso_abundance_08_018** | 1.286 |
| **iso_abundance_09_019** | $2.664e{-}08$ |
| **iso_abundance_10_020** | 312.5 |
| **iso_abundance_10_021** | 0.003556 |
| **iso_abundance_10_022** | 100.1 |
| **iso_abundance_11_023** | 22.84 |
| **iso_abundance_12_024** | 658.1 |
| **iso_abundance_12_025** | 82.5 |
| **iso_abundance_12_026** | 104.7 |
| **iso_abundance_13_027** | 76.42 |
| **iso_abundance_14_028** | 725.7 |
| **iso_abundance_14_029** | 35.02 |
| **iso_abundance_14_030** | 24.68 |
| **iso_abundance_15_031** | 4.242 |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | |
|---|---|
| **iso_abundance_16_032** | 89.12 |
| **iso_abundance_16_033** | 0.3056 |
| **iso_abundance_16_034** | 3.417 |
| **iso_abundance_16_036** | 0.0004281 |
| **iso_abundance_17_035** | 0.7044 |
| **iso_abundance_17_037** | 0.001167 |
| **iso_abundance_18_036** | 9.829 |
| **iso_abundance_18_038** | 0.6357 |
| **iso_abundance_18_040** | 0.001744 |
| **iso_abundance_19_039** | 1.389 |
| **iso_abundance_19_040** | 3.022 |
| **iso_abundance_19_041** | 0.0003339 |
| **iso_abundance_20_040** | 51.13 |
| **iso_abundance_20_041** | 1.974 |
| **iso_abundance_20_042** | $1.134e{-06}$ |
| **iso_abundance_20_043** | $2.117e{-06}$ |
| **iso_abundance_20_044** | $9.928e{-05}$ |
| **iso_abundance_20_048** | 0.1099 |
| **iso_abundance_21_045** | 1.635 |
| **iso_abundance_22_046** | 5.558 |
| **iso_abundance_22_047** | $8.947e{-06}$ |
| **iso_abundance_22_048** | $6.05e{-07}$ |
| **iso_abundance_22_049** | $5.854e{-09}$ |
| **iso_abundance_22_050** | $6.083e{-07}$ |
| **iso_abundance_23_050** | $1.818e{-05}$ |
| **iso_abundance_23_051** | $5.987e{-09}$ |
| **iso_abundance_24_050** | 2.873 |
| **iso_abundance_24_052** | 8.065 |
| **iso_abundance_24_053** | 0.003014 |
| **iso_abundance_24_054** | 0.4173 |
| **iso_abundance_25_053** | 6.499 |
| **iso_abundance_25_055** | 1.273 |
| **iso_abundance_26_054** | 49.08 |
| **iso_abundance_26_056** | 697.7 |
| **iso_abundance_26_057** | 21.67 |
| **iso_abundance_26_058** | 3.335 |
| **iso_abundance_27_059** | 2.214 |
| **iso_abundance_28_058** | 28.88 |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | | |
|---|---|---|
| **iso_abundance_28_060** | 11.9 | |
| **iso_abundance_28_061** | 0.5992 | |
| **iso_abundance_28_062** | 1.426 | |
| **iso_abundance_28_064** | 0.3039 | |
| **total_cross_section** | 2 | |
| **cross_section_option** | 012 | |
| **t_half_limit** | $1.0e4$ | |
| **primary_electrons** | 1 | |
| **secondary_positrons** | 0 | |
| **secondary_electrons** | 1 | |
| **knock_on_electrons** | 0 | |
| **secondary_antiproton** | 0 | |
| **tertiary_antiproton** | 0 | |
| **secondary_protons** | 0 | |
| **gamma_rays** | 0 | |
| **pi0_decay** | 0 | |
| **IC_isotropic** | 0 | |
| **IC_anisotropic** | 0 | |
| **synchrotron** | 0 | |
| **bremss** | 0 | |
| **globalLuminosities** | 0 | 1 |
| **DM_positrons** | 0 | |
| **DM_electrons** | 0 | |
| **DM_antiprotons** | 0 | |
| **DM_gammas** | 0 | |
| **DM_double0** | 2.8 | |
| **DM_double1** | 0.43 | |
| **DM_double2** | 80. | |
| **DM_double3** | 40. | |
| **DM_double4** | 40. | |
| **DM_double5** | 40. | |
| **DM_double6** | 30. | |
| **DM_double7** | 50. | |
| **DM_double8** | 40. | |
| **DM_double9** | $3.e-25$ | |
| **DM_int0** | 1 | |
| **DM_int1** | 1 | |
| **DM_int2** | 1 | |

Table A.1.: Input parameters of the NoGamma low resolution testset

| | |
|---|---|
| **DM_int3** | 1 |
| **DM_int4** | 1 |
| **DM_int5** | 1 |
| **DM_int6** | 1 |
| **DM_int7** | 1 |
| **DM_int7** | 1 |
| **DM_int8** | 1 |
| **DM_int9** | 1 |
| **skymap_format** | 0 |
| **output_gcr_full** | 1 |
| **warm_start** | 0 |

Table A.1.: Input parameters of the NoGamma low resolution testset

## A.2. WithGamma Low Resolution input file

The WithGamma Low Resolution input file is almost identical to the NoGamma input file (shown in Table A.1), the differences are shown in Table A.2.

| | |
|---|---|
| **gamma_rays** | 1 |
| **pi0_decay** | 3 |
| **IC_isotropic** | 1 |
| **synchrotron** | 2 |
| **bremss** | 1 |

Table A.2.: Differences of the WithGamma Low Resolution and NoGamma Low Resolution input files

# Appendix B.

# Runtime Measurements of the Optimized Version of GALPROP

| Threads | Speedup | | Efficiency | | Runtime | |
|---|---|---|---|---|---|---|
| | initial | optimized | initial | optimized | initial | optimized |
| 1 | 1 | 1 | 1 | 1 | 803 | 822.5 |
| 2 | 1.83 | 1.79 | 0.91 | 0.89 | 437 | 467.5 |
| 4 | 3.04 | 2.79 | 0.76 | 0.7 | 266 | 293.5 |
| 8 | 4.22 | 4.86 | 0.53 | 0.61 | 207 | 167.5 |
| 16 | 3.86 | 6.71 | 0.24 | 0.42 | 206.5 | 117 |
| 32 | 2.78 | 5.09 | 0.09 | 0.16 | 295.5 | 156 |
| 64 | 1.98 | 5.36 | 0.03 | 0.08 | 405.5 | 147.5 |
| 128 | 0.43 | 3.66 | 0 | 0.03 | 1880 | 224.5 |

Table B.1.: Runtime comparison of GALPROP initial and optimized version. Input problem **NoGamma**.

| Threads | Speedup | | Efficiency | | Runtime | |
|---|---|---|---|---|---|---|
| | initial | optimized | initial | optimized | initial | optimized |
| 1 | 1 | 1 | 1 | 1 | 1820 | 1826.5 |
| 2 | 1.82 | 1.87 | 0.91 | 0.94 | 1006 | 984.5 |
| 4 | 3.1 | 3.29 | 0.78 | 0.82 | 596 | 563.5 |
| 8 | 4.79 | 5.92 | 0.6 | 0.74 | 390 | 314.5 |
| 16 | 5.36 | 8.23 | 0.33 | 0.51 | 362 | 226 |
| 32 | 4.83 | 8 | 0.15 | 0.25 | 419 | 243.5 |
| 64 | 3.96 | 6.94 | 0.06 | 0.11 | 496.5 | 284.5 |
| 128 | 1.78 | 4.6 | 0.01 | 0.04 | 1021.5 | 397 |

Table B.2.: Runtime comparison of GALPROP initial and optimized version. Input problem **WithGamma**.

# List of Figures

2.1. Example setup of the Insieme infrastructure . . . . . . . . . . . . 14
2.2. Example execution of a parallel control flow in INSPIRE . . . . . 15

4.1. Roofline models for AMD Opteron systems [21] . . . . . . . . . . 37
4.2. Roofline model for the used target machine . . . . . . . . . . . . 39

6.1. Runtime in seconds of the health benchmark . . . . . . . . . . . 52
6.2. Callgrind analysis of the health benchmark . . . . . . . . . . . . 53
6.3. Runtime in seconds of the graphsearch benchmark . . . . . . . . 54
6.4. Callgrind analysis of the graphsearch benchmark, 1 thread vs 32 threads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.5. Runtime in seconds of the blackscholes benchmark . . . . . . . . 56
6.6. Runtime in seconds of the dijkstra algorithm implementation . . 57
6.7. Waiting time of the dijkstra code (per thread) . . . . . . . . . . . 58
6.8. Runtime in seconds of the NPB/cg implementation . . . . . . . . 59
6.9. Waiting/computation time of the CG benchmark . . . . . . . . . 60
6.10. Efficiency of different problem sizes of the CG benchmark . . . . 60
6.11. Runtime in seconds of the NN implementation in Rodinia . . . . 61
6.12. Runtime in seconds of the BFS implementation in Rodinia . . . . 62

7.1. Overview of the integration test steps . . . . . . . . . . . . . . . 67
7.2. ER Diagram of the generated database . . . . . . . . . . . . . . . 69
7.3. Class diagram of the integration test framework . . . . . . . . . . 77

8.1. Speedup of the original GALPROP version . . . . . . . . . . . . 81
8.2. Efficiency of the original GALPROP version . . . . . . . . . . . . 81
8.3. Shared memory analysis of GALPROP running with six threads using intel vTune . . . . . . . . . . . . . . . . . . . . . . . . . . 82
8.4. Efficiency of code regions . . . . . . . . . . . . . . . . . . . . . . 83
8.5. Absolute runtime of each region . . . . . . . . . . . . . . . . . . . 84
8.6. Speedup improvement results of the NoGamma region . . . . . . 87
8.7. Performance improvement results of the Serial and the gen_synch region . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

8.8. Speedup improvement results . . . . . . . . . . . . . . . . . . . . . . 91
8.9. Runtime improvement results . . . . . . . . . . . . . . . . . . . . . . 92
8.10. Overhead analysis of GALPROP running with input LrWithGamma . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
8.11. Overhead analysis based on code regions, input LrWithGamma, number of threads is 16 . . . . . . . . . . . . . . . . . . . . . . . . . 93
8.12. Normalized waiting time of GALPROP, input LrWithGamma, number of threads is 16 . . . . . . . . . . . . . . . . . . . . . . . . . 94

# List of Tables

3.1. Codes of the Barcelona OpenMP Task Benchmark Suite . . . . . 21
3.2. Codes of the OpenMP Source Code Repository (OmpSCR) . . . 22
3.3. Selected codes of the Sequoia Benchmark Suite . . . . . . . . . . 23
3.4. Proxy apps used from the ExMatEx Suite . . . . . . . . . . . . . 25
3.5. Selected benchmarks of the CORAL benchmark suite [13] . . . . 26
3.6. List of rudimentary parallelized example codes . . . . . . . . . . 28
3.7. List of example codes and their authors . . . . . . . . . . . . . . 29
3.8. Codes of the NPB Suite . . . . . . . . . . . . . . . . . . . . . . . . 30
3.9. Codes of the SNU-NPB Suite . . . . . . . . . . . . . . . . . . . . 31
3.10. Used Rodinia benchmark kernels . . . . . . . . . . . . . . . . . . 32

5.1. Shared memory performance of all input codes . . . . . . . . . . 45
5.2. Main memory footprint of all input codes . . . . . . . . . . . . . 48
5.3. Boundness of selected input codes . . . . . . . . . . . . . . . . . 50

6.1. Absolute runtime values of the graphsearch benchmark . . . . . . 55

7.1. Test framework configuration file options . . . . . . . . . . . . . . 65
7.2. List of all test steps . . . . . . . . . . . . . . . . . . . . . . . . . . 66
7.3. List of all collectable metrics . . . . . . . . . . . . . . . . . . . . . 68
7.4. List of all command line arguments . . . . . . . . . . . . . . . . . 71

8.1. Performance optimization results of readGasMaps . . . . . . . . 90
8.2. Performance optimization results of B_field_3D_model . . . . . . 90

A.1. Input parameters of the NoGamma low resolution testset . . . . 99
A.1. Input parameters of the NoGamma low resolution testset . . . . 100
A.1. Input parameters of the NoGamma low resolution testset . . . . 101
A.1. Input parameters of the NoGamma low resolution testset . . . . 102
A.1. Input parameters of the NoGamma low resolution testset . . . . 103
A.1. Input parameters of the NoGamma low resolution testset . . . . 104
A.1. Input parameters of the NoGamma low resolution testset . . . . 105
A.1. Input parameters of the NoGamma low resolution testset . . . . 106

A.2. Differences of the WithGamma Low Resolution and NoGamma Low Resolution input files . . . . . . . . . . . . . . . . . . . . . . . 107

B.1. Runtime comparison of GALPROP initial and optimized version. Input problem **NoGamma**. . . . . . . . . . . . . . . . . . . . . . . 109
B.2. Runtime comparison of GALPROP initial and optimized version. Input problem **WithGamma**. . . . . . . . . . . . . . . . . . . . . 109

# Bibliography


[1] K.A. Huck, A.D. Malony, R. Bell, and A. Morris. Design and implementation of a parallel performance data management framework. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 473–482, June 2005.

[2] Insieme Compiler Project. Distributed and parallel systems (dps) on the university of innsbruck. http://www.insieme-compiler.org/mission.html, November 2014.

[3] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. Inspire: The insieme parallel intermediate representation. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 7–17, Sept 2013.

[4] Peter Thoman. *Insieme-RS, A Compiler-supported Parallel Runtime System.* Dissertation, Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck, 2013.

[5] clang: a c language family frontend for llvm. http://clang.llvm.org/, July 2014.

[6] A Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 124–131, Sept 2009.

[7] AJ. Dorta, C. Rodriguez, and F. de Sande. The openmp source code repository. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244–250, Feb 2005.

[8] High performance computing center stuttgart. Openmp validation suite. http://www.hlrs.de/research/current-projects/openmp-validation-suite/, July 2014.

[9] GraphAnalysis.org compendium. Hpc graph analysis. http://www.graphanalysis.org/benchmark/, July 2014.

[10] Lawrence Berkeley National Laboratory. Apex map benchmark. http://crd.lbl.gov/groups-depts/ftg/projects/previous-projects/apex/, July 2014.

[11] ExMatEx Center at Los Alamos National Laboratory. Exmatex proxy apps. http://www.exmatex.org/proxy-over.html, July 2014.

[12] Top 500 list of supercomputers. http://top500.org/lists/2014/06/, July 2014.

[13] Argonne CORAL collaboration, Oak Ridge and Livermore National Laboratory. Coral benchmark suite. https://asc.llnl.gov/CORAL-benchmarks, July 2014.

[14] Kofler Sandro. Kinecontrol: Parallelization for shared memory parallel computing systems. Bachelor thesis, Institute of Computer Science, University of Innsbruck, 2012.

[15] Moosbrugger Stefan. Kinecontrol: Conversion from c# to c++ and sequential optimizations. Bachelor thesis, Institute of Computer Science, University of Innsbruck, 2012.

[16] NASA Advanced Supercomputing Division. Npb suite. http://www.nas.nasa.gov/publications/npb.html, July 2014.

[17] Center for Manycore Programming. Snu npb suite. http://aces.snu.ac.kr/SNU_NPB_Suite.html, July 2014.

[18] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[19] Al Daniel. Cloc: Count lines of code. http://cloc.sourceforge.net/, 2014.

[20] the free encyclopedia Wikipedia. perf (linux). http://en.wikipedia.org/wiki/Perf_%28Linux%29, 2014.

[21] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[22] Hpc linpack benchmark. http://khmel.org/?p=527, October 2014.

[23] Stream memory benchmark. http://www.cs.virginia.edu/stream/ref.html, October 2014.

[24] tool suite Valgrind. http://valgrind.org/info/tools.html, June 2015.

[25] the free encyclopedia Wikipedia. Amdahl's law. http://en.wikipedia.org/wiki/Amdahl%27s_law#Parallelization, July 2014.

[26] the free encyclopedia Wikipedia. Black-scholes model. http://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model, April 2015.

[27] The GALPROP development team. The galprop code for cosmic-ray transport and diffuse emission production. http://galprop.stanford.edu/, 2014.

[28] Zentraler Informatikdienst der Universität Innsbruck. Mach: collaborative system of the universities innsbruck and linz. http://www.uibk.ac.at/zid/systeme/hpc-systeme/mach/, July 2014.