PARALELIZACIÓN Y DEPURACIÓN DE APLICACIONES USANDO PROGRAMACIÓN BASADA EN PATRONES

APPLICATION PARALLELIZATION AND DEBUGGING USING PATTERN-BASED PROGRAMMING

CRISTINA VÍLCHEZ MOYA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado

Junio 2020

Directores:

Katzalin Olcoz Herrero
Luis María Costero Valero

Agradecimientos

A Katzalin, por su papel de profesora y tutora durante la carrera, por su confianza en mí, por su apoyo y compromiso con este trabajo.

A Luis, por su inmensa paciencia y dedicación.

A mis padres, por su amor incondicional.

A mis amigos, por cinco años inolvidables.

Resumen

La búsqueda de un rendimiento mejorado de nuestros sistemas informáticos es un viaje interminable. Hasta principios de los 2000, los ordenadores uni-core dominaban el marcado tecnológico; sin embargo, el problema del calor y las limitaciones del paralelismo a nivel de instrucción (ILP) condenaron esta rama de las computadoras a la obsolescencia, a favor de los procesadores multicore. Ahora, el desafío para los programadores es escribir programas que puedan aprovechar al máximo los recursos de la CPU, gracias al paralelismo. Los programadores se enfrentan a múltiples marcos de patrones paralelos y APIs (C++ Multihreads, OpenMP, Intel TBB, etc.), cada uno de los cuales tiene su propio estándar de programación. La falta de abstracciones de patrones paralelos de alto nivel y la dificultad de traducir programas entre estos modelos de patrones paralelos tan especializados aumentan la complejidad en el desarrollo de aplicaciones paralelas. **GrPPI** es una interfaz de alto nivel genérica de patrones paralelos de C++ que se presenta como una solución a este problema, ya que proporciona a los usuarios una API común para una colección de estos marcos paralelos.

En este trabajo, estudiamos el rendimiento de GrPPI usándolo para adaptar cuatro programas del conjunto de benchmark de referencia PARSEC y comparando su tiempo de ejecución con la implementación paralela original. Comparamos su rendimiento para tres back-end de ejecución (secuencial, paralelo nativo y OpenMP).

Veremos que los resultados de las pruebas testifican a favor de GrPPI, que tiene un tiempo de ejecución tan bueno como las versiones paralelas originales y más específicas de los programas, a la vez que necesita menos líneas de código y ningún conocimiento sobre los diferentes estándares de programación.

Palabras clave

API de alto nivel, patrones de paralelismo, programación paralela, programación basada en patrones, grppi, paralelismo de alto nivel, PARSEC, benchmark

Abstract

The search for improved performance of our computer systems is a never-ending journey. Up until the early 2000s, single-core processors dominated the technology market; however, the problem of heat and the limitations of instruction-level parallelism sentenced this branch of computers to obsolescence, in favour of multi-core processors. Now, the challenge for programmers is to write programs that can take full advantage of the CPU resources, thanks to parallelism. Programmers are now presented with multiple parallel pattern frameworks and APIs (C++ threads, OpenMP, Intel TBB, etc), each of which has its own programming standard. The lack of high-level parallel pattern abstractions and the difficulty to port programs between these very specialised parallel pattern models increase the complexity in developing parallel applications. **GrPPI** is a generic high-level C++ parallel pattern interface that presents itself as a solution to this problem, for it provides users with a common API for a collection of these parallel frameworks.

In this work, we study the performance of GrPPI by using it to adapt four programs from the PARSEC benchmark suite and comparing its execution time to the original parallel implementation. We compare its performance for three execution back-ends (sequential, parallel native and OpenMP).

We will see that the results of the tests testify in favour of GrPPI, which has an execution time as good as the original, more specific parallel versions of the programs while needing fewer lines of code and no knowledge about the different programming standards.

Keywords

high-level API, parallel pattern, parallel programming framework, pattern-based programming, grppi, high-level parallelization, PARSEC, benchmark

Contents

List of Figures

1

Introduction

1.1 Motivation

Parallel computing has become essential in our current society in the search for faster and more efficient programs. As the number of processors that can be built into one computer increases, so does the challenge of taking fully advantage of the hardware available to make the programs run faster. The trend for future processors is to focus on improving their performance through increasing their number of cores while only providing modest serial performance improvements; consequently, applications that require additional processing power will need to be parallel [6]. However, parallelization is, most of the time, not a trivial task; a number of problems arise when writing parallel code, such as deadlocks, data races or starvation. Furthermore, programmers are faced with different parallel pattern frameworks (FastFlow [3]) or APIs (OpenMP [1], C++ Multithreading [4]), each of which presents a different programming standard that the user must learn and comprehend. GrPPI presents itself as a possible solution to these trials.

GrPPI [7] is a generic and reusable high-level C++ parallel pattern interface that creates a layer between users and existing parallel programming frameworks to allow programmers to benefit from the improved parallel performance without requiring a deep understanding of the low-level processes. It allows programmers to take advantage of the benefits of using parallel patterns, being able to choose among different parallel programming models such as OpenMP, C++ threads and Intel TBB [9], without needing to worry about writing and correcting the synchronization code.

The questions that arise naturally, and that constitute the main motivation for this paper, are:

1. Is the generic GrPPI interface as efficient as the specific code it aims to substitute?
2. Can we measure the *effort-savings* this library provides?

1.2 Previous work

A thorough description of GrPPI's features and a study of its performance have been done previously by del Rio Astorga et al. [7]. This study, however, uses a sin-

gle video stream-processing application as evaluation benchmark, a program which is parallelized using a pipeline pattern. We would like to make a more extensive analysis of this library by studying its performance in a range of programs sufficiently representative of the actual programs of interest, that is, the applications that in-real-life computers must execute nowadays. As explained later in Section 2.2, the PARSEC benchmark [13] matches this criteria.

There are studies on the characteristics of the PARSEC benchmarks and its parallel implementation efficiency. We highlight the PhD paper by Bienia [5], where a detailed description of the purpose and implementation of each program can be found, among other interesting data related to these benchmarks.

1.3 Objectives and Work plan

The main goal of this paper is to analyse the efficiency of the GrPPI library in comparison to more specific methods and patterns of parallelization. This efficiency will be measured by comparing the execution time of the different versions, PARSEC's original parallelized version and our own GrPPI adaptation, and, ultimately, the *effort* of implementation, measured as lines of code written.

In order to do this, we will select some representative applications from the PARSEC's benchmark suite and study its implementation in order to write the equivalent code using functions from the GrPPI library. Once it is guaranteed that both programs do the same, that is their outputs are identical, we can perform measurements of their execution time for different inputs and discuss the results.

The source code can be found in the GitHUb repository `https://github.com/crivilch/GrPPI-Library`

2

Theoretical Background

2.1 Parallelization

Parallel computing is a type of computation where more than one calculation is done simultaneously. Parallelizing a problem means breaking it into smaller ones that can be processed at the same time, thus obtaining a solution at a faster speed.

There are two main types of parallelism [8]: Data Level Parallelism (DLP - data items can be operated on at the same time) and Task Level Parallelism (TLP - tasks of work that can be operated simultaneously) and two ways computer hardware can exploit application parallelism are:

1. Instruction Level Parallelism (ILP): exploits DLP through pipelining and multiple instruction issue.

2. Thread Level Parallelism (TLP): exploits both DLP and TLP via hyper-threading or multicore architecture.

Only the ILP approach can be effectively hidden from the application software. In contrast, TLP and DLP leave the problem of how to develop parallel applications that exploit these features to optimize performance to programmers [11].

Since 2003, single-processor performance improvement has dropped to less than 22% per year due to lack of usable instruction-level parallelism and low efficiency of silicon and energy consumption of its components [8]. However, the need for faster programs has only increased, exponentially, with the growing interest in cloud computing and analysis of the vast amounts of data available on the Internet. This has translated into a shift from ILP to DLP and TLP.

In order to ease the programmer's hard task, the idea of compiler support gained popularity. Ideally, the compiler for a multicore processor would take a sequential program and automatically translate it into a parallel program that makes the most optimal use of all the cores. Loops can be parallelized if each iteration is independent of the rest or if the operation behaves 'well' enough to allow reading and writing the same data in parallel. Compilers use dependence analysis to study if accesses to an array refer to the same memory cell and, if so, how many iterations apart they stand. This analysis not only helps to parallelize the program, but it is also used for other optimizations such as increasing cache locality [10].

This automatic parallelization is not always optimal, as it is often limited to loop analysis; in the end, performance of multicore processors is as good as the parallelization of the program they are executing; therefore parallel computing has become a must-know for programmers and computer engineers around the world.

There are many APIs and libraries for users to choose from when writing parallel applications; two of the most spread ones are OpenMP and POSIX threads.

OpenMP: OpenMP (Open MultiProcessing) is a specification for a set of compiler directives, library routines, and environment variables based on the fork-join model [1]. Initially a single master thread is executed until the apparition of a parallel constructor; then, the slave-threads are created as forks by the master, who then is in charge of synchronizing these threads and continuing the execution. The number of threads to be created can be set via a call to the library. When using OpenMP, the programmer can use directives to mark parallel regions of the program and then have the compiler generate efficient code for its execution [10]

POSIX threads: pthreads is the thread programming interface specified by IEEE POSIX. It consists of a set of C language programming types and procedure calls, implemented with a `pthread.h` header and a thread library. The subroutines included in this library can be classified into thread management (creating, detaching, joining), mutexes (synchronization via creation of mutually exclusive zones), condition variables (for communication between threads sharing a mutex) and synchronization (routines for reading/writing locks and barriers)[4].

OpenMP was designed to be a simpler, user-friendly approach to the use of threads, since it frees the programmer from having to create, synchronize and join them.
These two are the main back-ends we will focus on during the study of the GrPPI library's performance.

2.2 PARSEC

The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite composed of multithreaded programs. It consists of 9 applications and 3 kernels which were chosen from a wide range of application domains.

The reason why we decided to use this benchmark in the study of GrPPI's performance is its diversity and its intensive coverage of multi-threaded applications. This benchmark was conceived for the study of Shared Memory Chip-Multiprocessors, which currently are our best tool to meet the requirements of demanding applications such as video-games and data-mining; thus, the set of applications was designed to capture the characteristics of the target application space. Furthermore, the PARSEC benchmark applications have been parallelized, which provides us with a very adequate 'competitor' to time the GrPPI applications against.

Table 2.1 shows a qualitative summary of the characteristics of the benchmark programs. Every benchmark includes sets of inputs of different sizes - test, simsmall,

Table 2.1: Summary of the inherent key characteristics of PARSEC benchmarks.

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
`blackscholes`	Financial Analysis	data-parallel	coarse	small	low	low
`bodytrack`	Computer Vision	data-parallel	medium	medium	high	medium
`canneal`	Engineering	unstructured	fine	unbounded	high	high
`dedup`	Enterprise Storage	pipeline	medium	unbounded	high	high
`facesim`	Animation	data-parallel	coarse	large	low	medium
`ferret`	Similarity Search	pipeline	medium	unbounded	high	high
`fluidanimate`	Animation	data-parallel	fine	large	low	medium
`freqmine`	Data Mining	data-parallel	medium	unbounded	high	medium
`streamcluster`	Data Mining	data-parallel	medium	medium	low	medium
`swaptions`	Financial Analysis	data-parallel	coarse	medium	low	low
`vips`	Media Processing	data-parallel	coarse	medium	low	medium
`x264`	Media Processing	pipeline	coarse	medium	high	high

Source: Princeton University Technical Report TR-811-08, January 2008

simmed, simlarge and native - which pretend to be increasingly accurate approximation on real-world inputs. An input consists of all the input files required by the program and a predetermined way to invoke the binary.

2.3 GrPPI

GrPPI is a generic and reusable high-level C++ parallel pattern interface that creates a layer between users and existing parallel programming frameworks to allow programmers to benefit from the improved parallel performance without requiring a deep understanding of the low-level processes. It contains both stream and data-parallel patterns, which work as follows [7]:

- Map: computes the function $f : \alpha \to \beta$ for each element of the input x_i. The output is y_i, where $y_i = f(x_i)$.
- Reduce: computes the function $\otimes : \alpha \times \alpha \to \alpha$, which takes possibly multiple elements from the input and combines them in one output alone. The final output is one value $y = x_1 \otimes ... \otimes x_n$.
- Stencil: a generalization of the map pattern, where the function can access one element of the inputs and its neighbours.
- Divide&Conquer: this pattern solves a problem by recursively dividing it into two equivalent smaller problems, until a base case with a known answer is reached; the final output is created by merging the solutions of these subproblems.

As for stream-based parallel patterns, we highlight:

- Pipeline: processes the items from the input stream though several parallel stages. Each stage takes the output of the previous one as input and delivers the results to the next one. The functions related to each stage, f_i, must be able to be executed in parallel, so that the final output is $f_n(f_{n-1}(...(f_1(x_i))..))$.
- Farm: computes the function $f : \alpha \to \beta$ in parallel over all the items of the input stream. The operations performed by f for each element of the input should be completely independent to each other.

Every function call follows the following pattern:

```
grppi_function(ex, input, size, output, lambda)
```

The first parameter *ex* is the execution mode; this determines the back-end for the parallel execution and allows the programmer to choose among parallel native back-end (pthreads in UNIX), OpenMP, TBB and FastFlow in execution time. When creating the execution object, the user must set the number of underlying threads used by the execution implementation. Next, the GrPPI function receives pointers to the *input* sets the user wants to apply the function on and its *size*. The *output* parameter is a pointer to the structure where the result of the function will be stored. The pattern used and the function to execute determine the type of this object. The last parameter is the *lambda-function* that must be executed in parallel. Some patterns, like Divide&Conquer or Pipeline, require/can accept more than one function for different stages.

Each back-end has its own code file where all parallel patterns are implemented accordingly. The `map` function, for example, is implemented as follows:

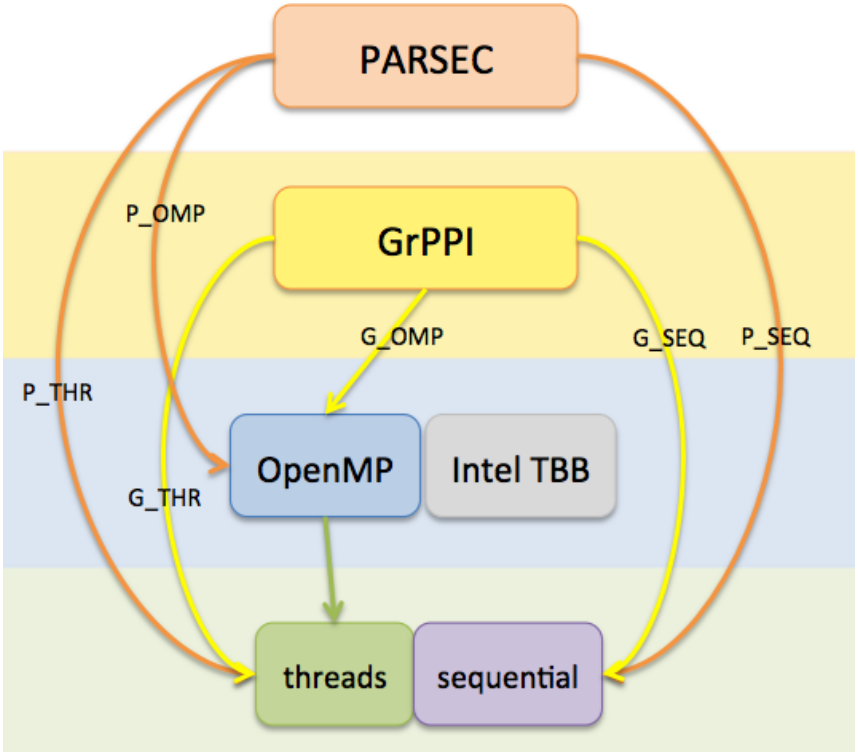
- Sequential: iterates through the input set, applying the transformer function to one element after another.
- Native (C++ threads): divides the input in as many subsets as total number of threads and creates a pool of workers (threads). Each worker takes one subset and applies the transformation to all its elements sequentially.
- OpenMP: invokes the `#pragma parallel for` directive that iterates through the input set and applies the transformation to every element.

Of course, the programmer doesn't need to know anything about the underlying implementation; it remains hidden behind the GrPPI call.

Figure 2.1 presents different possible approaches for parallelizing PARSEC's benchmark programs. More abstract APIs are presented in higher levels of the diagram and they are joined to their underlying implementation method with arrows. Once the user chooses one of the boxes as their parallelization tool, they don't need to worry about the layers under it; these will be managed by the higher-level pattern. The higher the level, the simpler the code that must be written by the final user.

Orange arrows represent PARSEC's existing implementation, while yellow arrows represent the GrPPI version of the code (that is, a version of the program where parallelization is done via calls to GrPPI functions instead of other mechanisms), which must be implemented by us. Our goal in the following sections is to study whether the use of the GrPPI library, which works as a layer between user and parallel programming frameworks, remains as efficient as the original 'hand-written' code, or if it provides better usability at the expense of performance.

Figure 2.1: Parallelization Techniques.



3

Test Environment

3.1 Arquitecture and compilation settings

In this section we present the software and hardware setup for all the tests performed in this study [2].

Microprocessor: Intel(R) Xeon(R) Gold 6138 CPU 64-bit 20-core x86 multi-socket high performance server microprocessor
Arquitecture: Skylake x86_64
Cores: 20
Sockets: 2
Frequency: 1700 MHz
Hyperthreading: Off
OS: GNU\Linux, Debian 9.11
Kernel: Linux 4.9.0-11-amd64
Cache size:

L1 - core level: 32KB for instructions, 32KB for data
L2 - core level : 1024KB Instructions + Data
L3 - processor level: 28 MB

RAM Memory: 93GB (2 NUMA nodes: 46+47GB)
Compiler: g++ (Debian 6.3.0-18+deb9u1) 6.3.0 20170516
Compilation options: -O3 -mno-sse2 –std=c++14

Generic Reusable Parallel Pattern Interface (GrPPI) 2018 Universidad Carlos III de Madrid. Version: Apr 26, 2019. Commit: 1c677f0

PARSEC 3.0-beta-20150206

3.2 Work methodology

For every studied PARSEC benchmark, we compare the performance of the GrPPI version of the code written for this paper against the original PARSEC's implementation, for both sequential and parallel (pthreads and OpenMP) execution modes.

To do this, time marks are set before and after the section of the code that has been adapted into a call to one of the functions in the GrPPI library in order to measure the time spent on its execution. This includes the time necessary for the creation and destruction of threads in the parallel modes.

Each benchmark is tested for different input sizes: small, medium and large. The same set of inputs of each size are used for each execution mode and for different number of threads in the parallel modes. Five different inputs are prepared for each input size in order to work with the mean of the results; that is, for any given input size and number of threads, the program is executed five times with the five different inputs prepared beforehand.

The number of threads is set during execution time - a maximum of 20 - and we ensure each thread doesn't use more than one core by setting the CPU affinity with `taskset`. All threads are created in the same socket to prevent data movement from one socket to the other. The CPU's clock rate is fixed to 1700MHz before each test launch.

4

Swaptions

4.1 Description

The swaptions application [5] uses the Heath-JarrowMorton (HJM) framework to price a portfolio of swaptions, employing Monte Carlo simulation to compute the prices. The program stores the portfolio in the `swaptions` array. In PARSEC's thread version, this array is then evenly partitioned into a number of blocks equal to the number of threads and one block is assigned to each thread. Each thread iterates through the swaptions in the work unit it was assigned, calling the function `HJM_Swaption_Blocking` for every entry in order to compute the price. Calculating the price of one swaption only requires information of that one swaption, and so each call to `HJM_Swaption_Blocking` is independent of the rest. Listing 4.1 shows a pseudo-code for this application's implementation.

```
1  parm swaptions[];
2  ftype pdSwaptionPrice[2];
3
4  for each element in swaptions:
5      pdSwaptionPrice =  HJM_Swaption_Blocking(swaptions[i]);
6
7      swaptions.dSimSwaptionMeanPrice = pdSwaptionPrice[0];
8
9    swaptions.dSimSwaptionStdError = pdSwaptionPrice[1];
10
11 return swaptions;
```

Listing 4.1: Swaptions Pseudocode

The input for this application is not read from a file; it is randomly generated within the application using a given seed. The list of parameters that must be specified in the executable call are: number of swaptions (ns), number of simulations for every input set (sm), number of threads (nt), random number seed (sd).

4.2 GrPPI Adaptation

Since every call to `HJM_Swaption_Blocking` for each element of the input is independent of the rest, the `for` loop is equivalent to using the `grppi::map` function

with `HJM_Swaption_Blocking` as its lambda, as shown in Listing 4.2. The original code used the index of the element in the `swaptions` vector during the call to `HJM_Swaption_Blocking` as part of randomness seed; as we don't have access to that index when writing the lambda function, an extra parameter `index` was included. `index` is a vector of integers that contains all ordered numbers from 0 to *nSwaptions*; thus `index[i]=i` and the map function will work as desired. The results of the price calculation are stored in the very same array `swaptions`, which is passed as input to the map function; therefore, the elements of this array are returned as output.

```
grppi::dynamic_execution ex = execution_mode(opt, nThreads);

map(ex, std::make_tuple(swaptions, index.begin()),size, swaptions,
[](parm swaptions, int index){

  FTYPE pdSwaptionPrice[2];
  int iSuccess =  HJM_Swaption_Blocking(pdSwaptionPrice, swaptions.
   dStrike, swaptions.dCOpenMPounding, swaptions.dMaturity,
   swaptions.dTenor, swaptions.dPaymentInterval, swaptions.iN,
   swaptions.iFactors, swaptions.dYears, swaptions.pdYield,
   swaptions.ppdFactors, swaption_seed+index, NUM_TRIALS,
   BLOCK_SIZE, 0);

  assert(iSuccess == 1);
  swaptions.dSimSwaptionMeanPrice = pdSwaptionPrice[0];
  swaptions.dSimSwaptionStdError = pdSwaptionPrice[1];

  return swaptions;}
);
```

Listing 4.2: Swaptions GrPPI call

4.3 Results

The set of inputs for this benchmark's test consists of one small-size set (-ns 32 -sm 10000), a medium-size set (-ns 64 -sm 20000) and a large set (-ns 148 -sm 1000000), with five different seeds to generate comparable outputs (-sd $\{27, 33, 100, 555, 1939\}$). The number of threads used range from 1 to 20 in the pthread and OpenMP execution modes.

The mean value of the tests is shown in Table 4.1, where the results for each execution mode using PARSEC's implementation and GrPPI's version are presented in adjacent columns for comparison. The columns corresponding to PARSEC implementation are P_OMP, P_THR and P_SEQ (for OpenMP, pthreads and sequential execution, respectively); G_OMP, G_THR and G_SEQ are the GRPPI equivalent.

The results are displayed in three graphs in Figure 4.1, one for each input size. The curve of the graphs suggest an inversely proportional relation between the time cost and the number of threads in use. Table 4.2 shows the values of the speedup for each set of inputs in PARSEC's OpenMP execution mode, with almost perfect escalation for large input sizes even with 20 threads - for small and medium sized

Table 4.1: Swaptions test results

SIZE	NTHREADS	P_OMP	G_OMP	P_THR	G_THR	P_SEQ	G_SEQ
32	1	1.395	1.393	1.395	1.393	1.403	1.393
	2	0.699	0.707	0.699	0.699		
	5	0.306	0.306	0.306	0.349		
	10	0.176	0.175	0.175	0.219		
	15	0.132	0.132	0.132	0.176		
	20	0.089	0.089	0.095	0.570		
64	1	5.582	5.574	5.577	5.574	5.614	5.572
	2	2.797	2.792	2.790	2.794		
	5	1.136	1.135	1.138	1.396		
	10	0.613	0.611	0.612	0.873		
	15	0.438	0.438	0.437	0.700		
	20	0.351	0.351	0.351	0.614		
148	1	645.3	644.7	644.6	644.5	649.1	644.5
	2	323.2	323.0	322.3	322.9		
	5	131.0	130.8	130.7	139.6		
	10	65.53	65.47	65.44	96.05		
	15	43.74	43.64	43.68	96.07		
	20	35.00	34.92	34.93	65.48		

inputs, there is not enough work as the number of threads increases. Other execution modes show similar results.

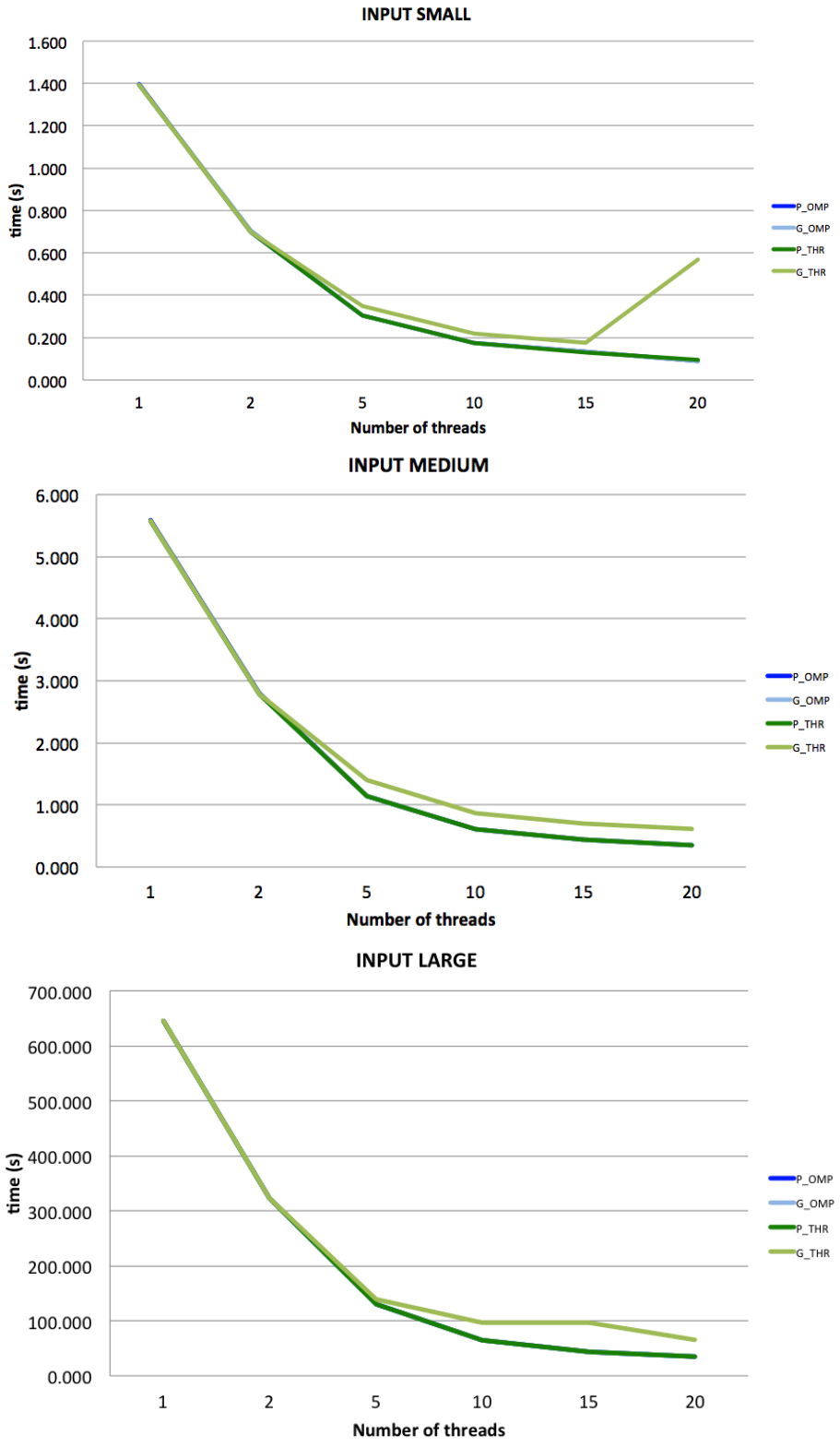
Table 4.2: Speedup for Swaptions for OpenMP

SIZE \ NTHR	2	5	10	15	20
32	2.0	4.6	7.9	10.6	15.7
64	2.0	4.9	9.1	12.7	15.9
148	2.0	4.9	9.8	14.8	18.4

Another thing to highlight is how the performance of GrPPI's version is just as good as PARSEC's original implementation if we compare the execution with OpenMP as backend. Even for small inputs, GrPPI doesn't seem to add any significant workload.

However, there is an abnormality regarding GrPPI's parallel native execution, which seems to be taking longer than the other execution modes as the number of threads increases. This poses one main question: How are the threads used by PARSEC's default thread execution mode different from the threads used by GrPPI?

Figure 4.1: Swaptions test graphs



4.4 GrPPI Threads

Considering the Swaptions benchmark results, we would like to study the reason for GrPPI's native implementation's bad performance in comparison to the rest of the execution modes.

One possible theory that came to mind was that as the number of threads increases, but not so the total number of swaptions to calculate, the amount of work for each thread diminishes; it is possible that the advantage of sharing the work doesn't make up for the time cost of creating the threads. In order to study this, we performed a test where the size of each chunk remains constant no matter the number of threads. So for each run of the test, the number of swaptions computed is $block\_size * number\_threads$. The number of simulations was fixed (-sm 20000) and the same five seeds as in the original test were used.

We see in the graphs (Figure 4.2) that the performance of GrPPI's threads is just as good as the rest of the execution modes. The time spent in execution is directly proportional to the size of the block, and it remains practically constant as the number of threads increases. In order to see this, Table 4.3 shows the time cost over block size ratio for all block sizes and all number of threads for the GrPPI thread execution mode - the results are the same for the others.

Table 4.3: Time/Size ratio for GrPPI thread

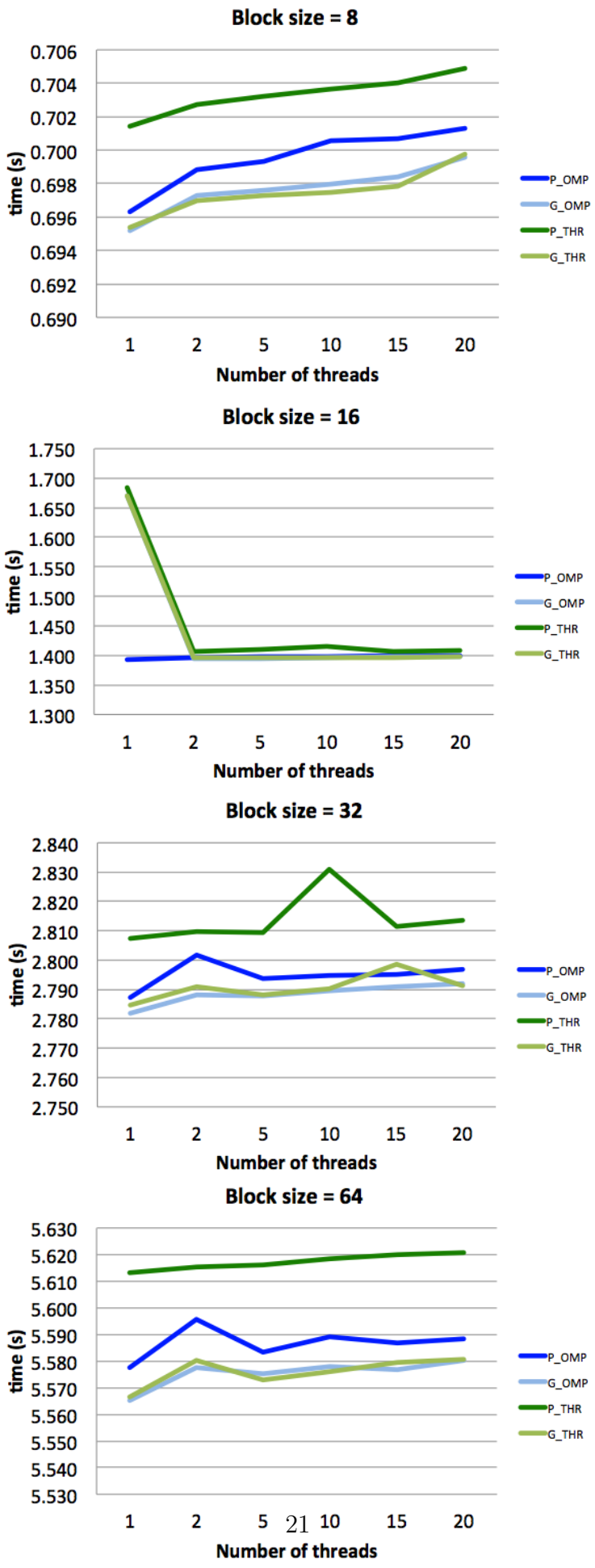
SIZE \ NTHR	1	2	5	10	15	20
8	0.087	0.087	0.087	0.088	0.088	0.088
16	0.087	0.087	0.087	0.087	0.087	0.087
32	0.087	0.088	0.087	0.087	0.087	0.087
64	0.087	0.087	0.087	0.087	0.087	0.087

The code of the map function for the parallel execution native mode divides the total amount of elements that need to be worked on in equally sized *chunks*, as presented in Listing 4.3 (extract from `parallel_execution_native::map`)

```
1  const int chunk_size = sequence_size / concurrency_degree_;
2  {
3    worker_pool workers{concurrency_degree_};
4    for (int i=0; i!=concurrency_degree_-1; ++i) {
5      const auto delta = chunk_size * i;
6      const auto chunk_firsts = iterators_next(firsts,delta);
7      const auto chunk_first_out = next(first_out, delta);
8      workers.launch(*this, process_chunk, chunk_firsts, chunk_size
     , chunk_first_out);  }
9
10   const auto delta = chunk_size * (concurrency_degree_ - 1);
11   const auto chunk_firsts = iterators_next(firsts,delta);
12   const auto chunk_first_out = next(first_out, delta);
13   process_chunk(chunk_firsts, sequence_size - delta,
    chunk_first_out);
14 }
```

Listing 4.3: Chunk size for GrPPI parallel execution

Figure 4.2: Block test

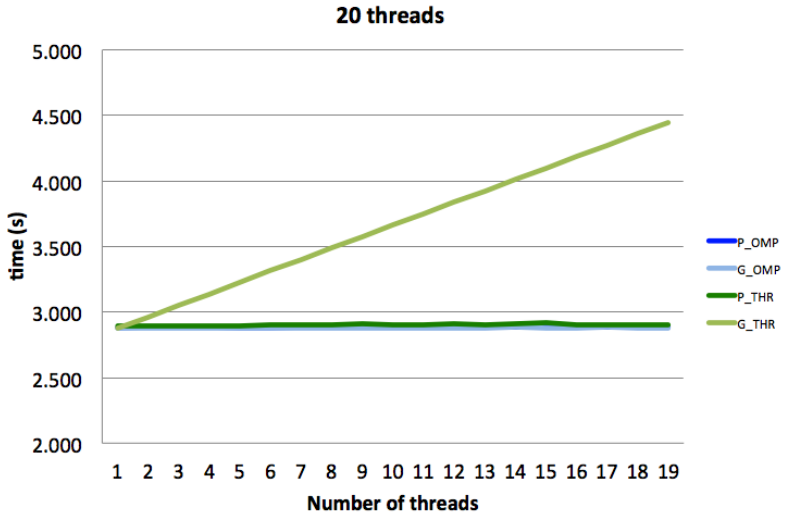


The last chunk is processed by the parent thread.

This makes us believe that the problem may arise when the number of swaptions is not multiple of the number of threads; that is, when $num\_swaptions = block\_size * num\_threads + i, i \in [1, 2, ..., num\_threads - 1]$. As it is implemented now, those i additional elements would be assigned to the parent thread. We ask ourselves: how does time scale as we increase i?

To answer this question, a new test was performed where we measured the time spent in execution for the different values of i relevant for each $block\_size$ and $num\_threads$. Figure 4.3 shows the results for $block\_size = 32$ and $num\_threads = 20$ as an example.

Figure 4.3: $num\_swaptions \nmid num\_threads$



The conclusion seems straightforward: there's a direct linear relation between the number of additional elements the parent thread has to take care of and the time consumption of the program. The simplest solution to this problem would be to share the i additional elements among all the working threads; in order to do this, we suggest the following change to the `parallel_execution_native::map` code (Listing 4.4).

```
1  const int chunk_size = sequence_size / concurrency_degree_;
2  const auto offset = sequence_size % concurrency_degree_;
3
4  {
5    worker_pool workers{concurrency_degree_};
6    for (int i=0; i!=concurrency_degree_-1; ++i) {
7
8      auto cs = chunk_size;
9      auto delta = 0;
10
11     if(i<offset){
12       cs   = chunk_size+1;
13       delta = cs * i;
14     }
```

```
15        else{
16          cs = chunk_size;
17          delta = (chunk_size + 1)*offset + (i-offset)*chunk_size;
18        }
19        const auto chunk_firsts = iterators_next(firsts,delta);
20        const auto chunk_first_out = next(first_out, delta);
21        workers.launch(*this, process_chunk, chunk_firsts, cs,
      chunk_first_out);
22      }
23
24      const auto delta = (chunk_size+1)*offset + chunk_size * (
      concurrency_degree_ - 1 - offset);
25      const auto chunk_firsts = iterators_next(firsts,delta);
26      const auto chunk_first_out = next(first_out, delta);
27      process_chunk(chunk_firsts, sequence_size - delta,
      chunk_first_out);
28    }
```

Listing 4.4: Modification of GrPPI parallel execution

Now the first $i = num\_swaptions\%num\_threads$ threads get a chunk of size $bloq\_size + 1$, while the rest receive a chunk of size $bloq\_size$ as expected.

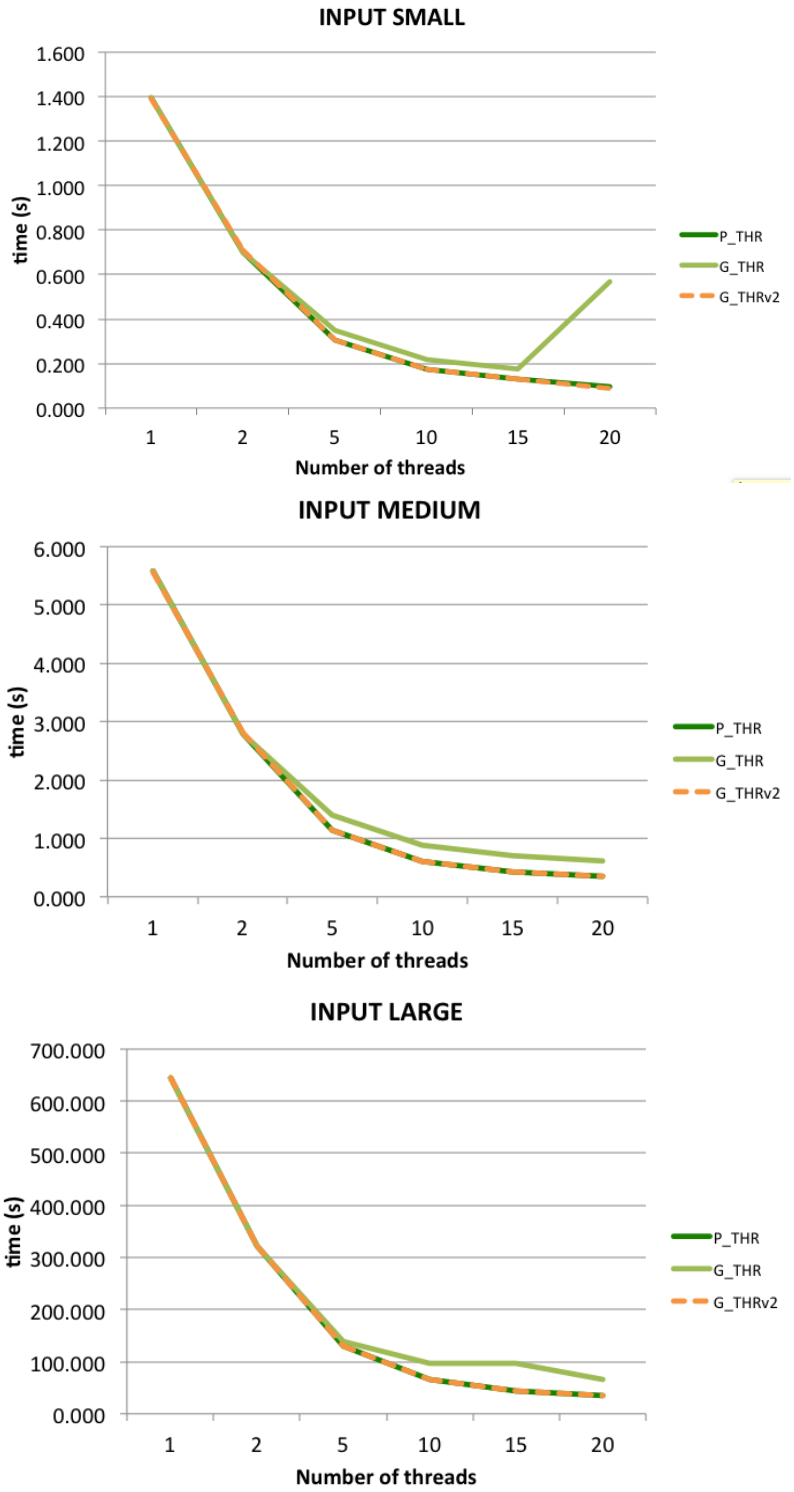
This change could mean a significant improvement in the performance of the GrPPI library's parallel native execution. To prove this, the test for the Swaptions benchmark was repeated, with the same initial conditions and parameters (Table 4.4 & Figure 4.4).

Table 4.4: Improved GrPPI threads

SIZE	NTHREADS	P_THR	G_THR	G_THRv2
32	1	1.395	1.393	1.392
	2	0.699	0.699	0.707
	5	0.306	0.349	0.305
	10	0.175	0.219	0.175
	15	0.132	0.176	0.131
	20	0.095	0.570	0.089
64	1	5.577	5.574	5.567
	2	2.790	2.794	2.799
	5	1.138	1.396	1.133
	10	0.612	0.873	0.611
	15	0.437	0.700	0.436
	20	0.351	0.614	0.350
148	1	644.6	644.5	643.5
	2	322.3	322.9	322.5
	5	130.7	139.6	130.6
	10	65.44	96.05	65.42
	15	43.68	96.07	43.60
	20	34.93	65.48	34.89

The new version of the GrPPI parallel native execution proves to be as efficient as PARSEC's pthread implementation.

Figure 4.4: Improved GrPPI threads



This chapter's final conclusions could be summarized as:

1. The Swaptions benchmark's speedup is virtually optimal, as long as the workload (input size) is big enough for the number of threads we want to use.

2. GRPPI OpenMP doesn't add any excess workload and is just as efficient as PARSEC's OpenMP and pthread versions.

3. GRPPI thread's performance was worse than other execution modes due to an inefficient workload distribution. By improving it, we have achieved optimal performance just like the other back-ends.

5

Blackscholes

5.1 Description

The Blackscholes application [5] calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent and calls function `BlkSchlsEqEuroNoDiv` for each of them to compute its price. This calculation is repeated a total of $NUM\_RUNS = 100$ times Listing 5.1 shows a pseudo-code for this application.

```
data data_vector[];
fptype prices[];

for j in range(NUM_RUNS){
    for i in data_vector:
        prices[i] =  BlkSchlsEqEuroNoDiv(data_vector[i]);
}
return prices;
```

Listing 5.1: Blackscholes pseudo-code

As indicated in the pseudo-code, each call to `BlkSchlsEqEuroNoDiv` depends on only one element of the data array, and so all calls are independent.

This benchmark's executable receives parameters *nthreads* number of threads, *inputFile* name of input file and *outputFile* name of output file.

5.2 GrPPI Adaptation

Just as in the Swaptions benchmark, this `for` loop is equivalent to a call to the `grppi::map` function with `BlkSchlsEqEuroNoDiv` as its lambda because all elements in the data vector are processed individually. There are two ways to adapt the external `for (j in range(NUM_RUNS))` loop. One option is shown in Listing 5.2: the `for` is enveloping the `map` call.

```
grppi::dynamic_execution ex = execution_mode(opt, nThreads);

```

```
for (int j=0; j<NUM_RUNS; j++){
    map(ex, std::make_tuple(sptprice.begin(), strike.begin(), rate.
    begin(),volatility.begin(), otime.begin(), otype.begin()),s,
    prices.begin(),
              [](fptype s1,fptype s2, fptype s3, fptype s4, fptype s5
    , int s6)  {
              return BlkSchlsEqEuroNoDiv(s1, s2,s3, s4, s5, s6, 0);
              });
}
```

Listing 5.2: Blackscholes GrPPI call 1

Another option is to include the `for` loop inside the lambda, as shown in Listing 5.3

```
map(ex, std::make_tuple(sptprice.begin(), strike.begin(), rate.
   begin(),volatility.begin(), otime.begin(), otype.begin()),s,
   prices.begin(), [](fptype s1,fptype s2, fptype s3, fptype s4,
   fptype s5, int s6) {
      fptype p;
      for (int j=0; j<NUM_RUNS; j++) {
        p = BlkSchlsEqEuroNoDiv(s1, s2,s3, s4, s5, s6, 0);
      }
    return p;
});
```

Listing 5.3: Blackscholes GrPPI call 2

Version 1 makes NUM_RUNS calls to the `grppi:map` function and each time the threads execute one operation, whereas Version 2 makes one single call to the GrPPI library and each of the created threads execute NUM_RUNS operations. We will compare the results of both versions in the next section.

5.3 Results

Three sets of input files were prepared, varying their size from small (1K options), to medium (100K options) and large (1M options). These input sets were created by dividing PARSEC's native input file (100M options) into files with said size. 6 tests were performed for each input size and each number of threads (in execution with OpenMP or parallel native).

The results for Version 1 are shown in Table 5.1 and Figure 5.1. The graphs for Version 2 are displayed in Figure 5.2

For the small input case, we see there's a surprising decrease of efficiency for the GrPPI threads execution, which becomes more significant as the number of threads increases. This penalisation disappears in the Version 2 execution, where the `for` was included inside the `map`. There is not such a remarkabe difference between the other input sets, where the total execution time is at least 100 times larger.

The `grppi:map` function for C++ threads creates and destroys the set of threads at the beginning and at the end of the call; that means that Version 1 of our Blackscholes adaptation creates and destroys *nthr* threads NUM_RUNS times, while in Version 2 we only create them once. The time penalty of creating and destroying the

threads is only significant for small inputs and becomes meaningless for larger inputs.

Table 5.1: Blackscholes test results

SIZE	NTHREADS	P_OMP	G_OMP	P_THR	G_THR	G_THRv2	P_SEQ	G_SEQ
1K	1	0.0171	0.0173	0.0171	0.0168	0.0167	0.0170	0.0167
	2	0.0089	0.0090	0.0087	0.0113	0.0113		
	5	0.0039	0.0039	0.0036	0.0101	0.0106		
	10	0.0025	0.0025	0.0021	0.0249	0.0249		
	15	0.0022	0.0022	0.0016	0.0368	0.0373		
	20	0.0023	0.0022	0.0015	0.0507	0.0508		
100K	1	1.702	1.711	1.698	1.667	1.666	1.690	1.665
	2	0.851	0.861	0.849	0.843	0.843		
	5	0.344	0.344	0.340	0.342	0.344		
	10	0.171	0.172	0.170	0.189	0.189		
	15	0.115	0.116	0.121	0.147	0.148		
	20	0.087	0.087	0.094	0.135	0.134		
1M	1	17.03	17.11	16.98	16.88	16.65	16.90	16.64
	2	8.563	8.554	8.490	8.392	8.400		
	5	3.404	3.422	3.404	3.365	3.366		
	10	1.705	1.725	1.714	1.701	1.721		
	15	1.140	1.169	1.142	1.161	1.165		
	20	0.860	0.859	0.858	0.894	0.894		

Observing the application's speedup, we can say that it's proportional to the number of threads (Table 5.2), much like the Swaptions benchmark.

Table 5.2: Speedup for Blackscholes

SIZE \ NTHR	2	5	10	15	20
1K	1.9	4.4	7.0	7.8	7.6
100K	2.0	4.9	9.9	14.7	19.6
1M	2.0	5.0	10.0	14.9	19.8

Speedup for each set of inputs in PARSEC's OpenMP execution mode - other execution modes show similar results.

Finally, let's compare the original GrPPI's performance with our improved version from Section 4.4 (G_THRv2). We see that for all input sets, both implementations are just as efficient. This is due to the fact that the time cost of calculating a single option is minimal, about 20 times smaller than the ones measured for Swaptions; this, together with the difference in the number of elements to be processed in every test (32 VS 1K for the small input case) explains the lack of difference between the old and new GrPPI thread mode even when the number of elements isn't divisible by the number of threads.

Figure 5.1: Blackscholes test 1 graphs

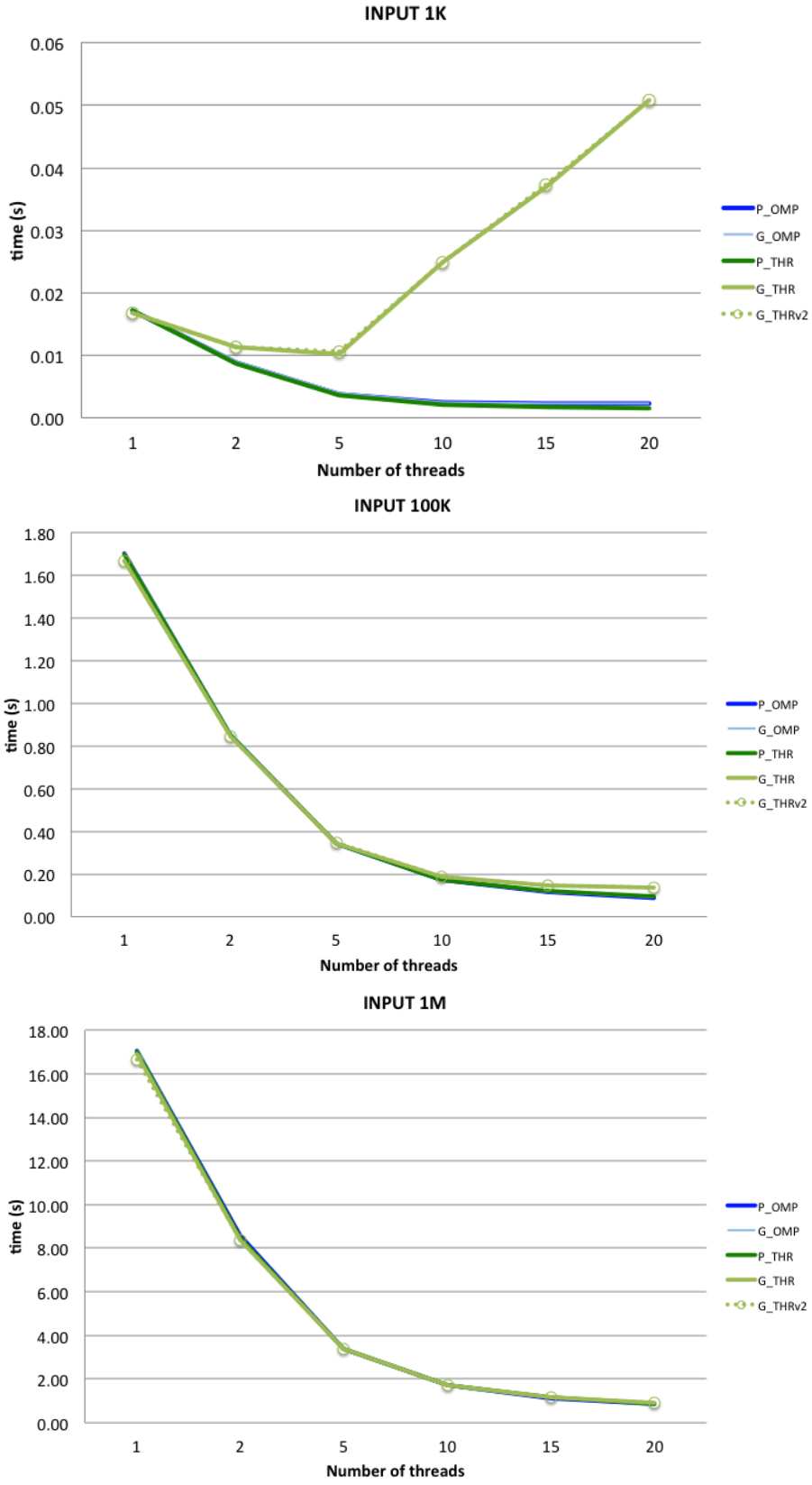
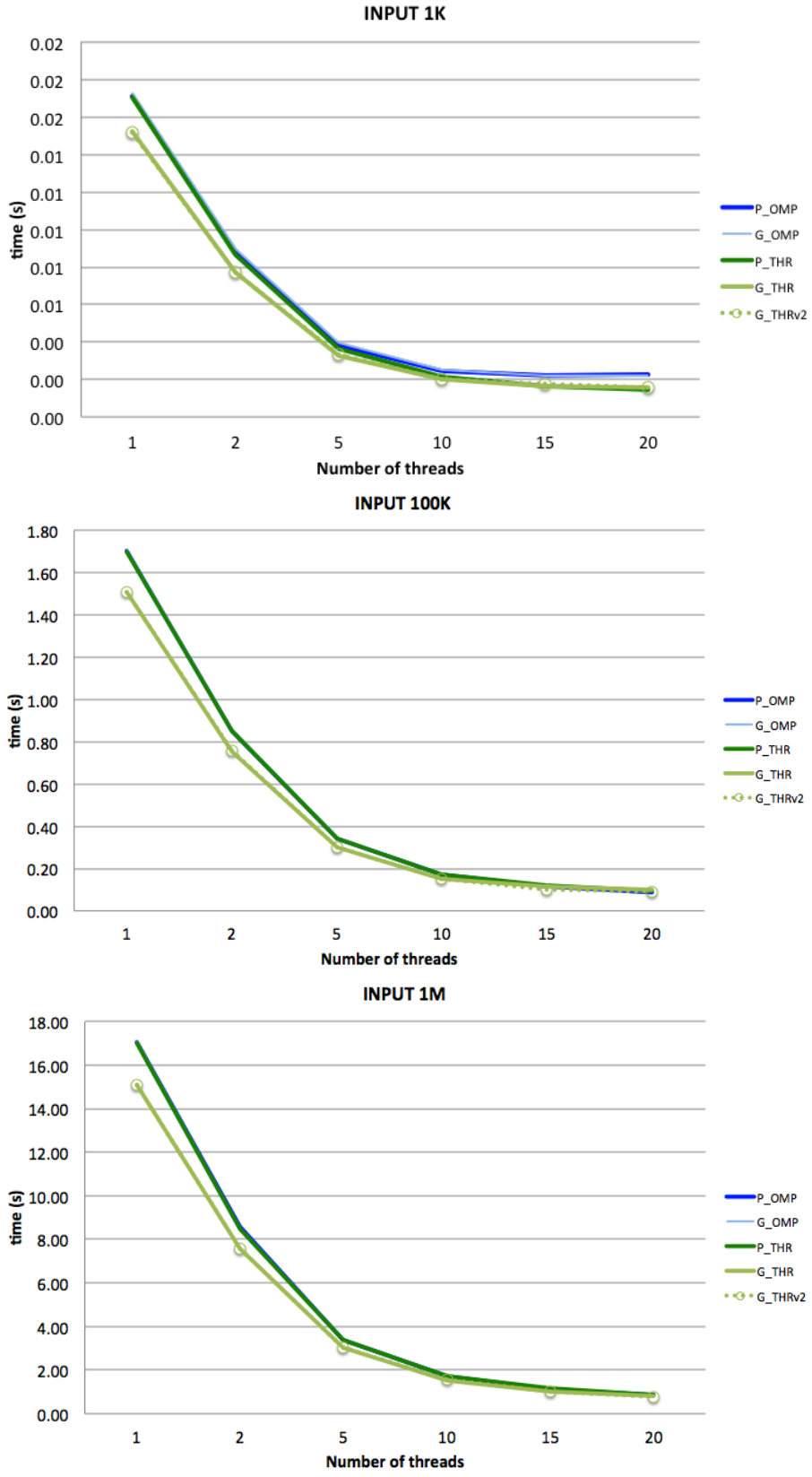


Figure 5.2: Blackscholes test 2 graphs



The main conclusions for this chapter are:

1. The Blackscholes benchmark's speedup is virtually optimal, as long as the workload (number of options) is big enough for the number of threads we want to use.

2. Excess creation of threads adds little penalization to the program's execution time; it is only perceivable for small inputs.

3. For applications where the time-consumption of the lambda operation is small, GrPPI's native parallel back-end's original workload distribution is as efficient as the improved version.

From now on we will only show the results for GrPPI's modified version.

6

Streamcluster

6.1 Description

The streamcluster kernel [5] solves the clustering problem: given a number of points, it finds a predetermined number of medians so that each point is assigned to its nearest center.

As input, it receives numbers $k1, k2$ - minimum and maximum number of centers allowed; d dimension of the points; n number of data points; *chunksize* number of points to handle per step; *clustersize* maxinum number of intermediate centers; *nproc* the number of threads to use; and a seed s to generate the input points randomly.

This benchmark only has a native parallel version implemented and it works as follows:
The only parallel function is the one where the program spends most of the time, `pkmedian`. This function calculates the cost of opening a new center, and it divides the work among all the threads created by assigning one block of points to each thread.

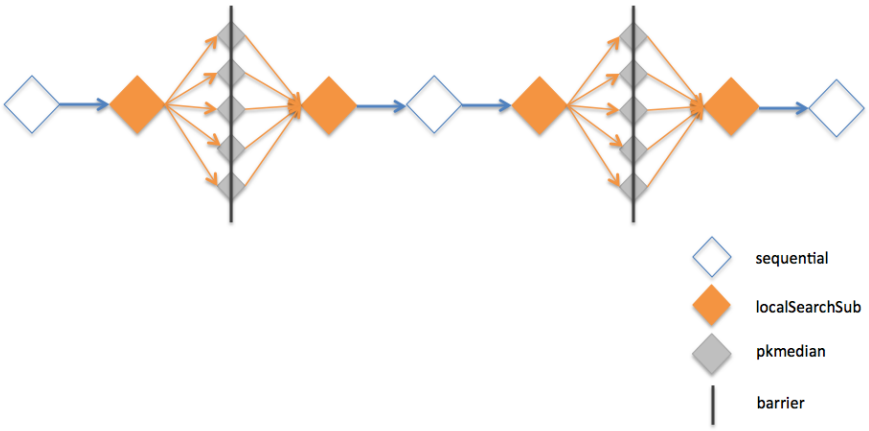
There are, however, some sections of the code that shouldn't be executed by more than one thread, such as writing global variables or shuffling the set of points. For this, PARSEC uses a thread barrier to make sure all the threads have finished all the operations before and then checks for the thread with id 0 to do the sequential work. Listing 6.1 shows an example of this situation.

```
#ifdef ENABLE_THREADS
  pthread_barrier_wait(barrier);
#endif

  if( pid == 0 ) {
    // sequential code
  }
```

Listing 6.1: Example barrier call

Figure 6.1 is an abstract representation of this benchmark's parallel execution. It begins with a first sequential section, followed by a call to the `localSeachSub` function. This function creates *nproc* threads, each of which will execute `pkmedian`. The vertical black line represents the barrier that will affect all these threads by

Figure 6.1: Streamcluster diagram



making them wait to be synchronized at more than one point throughout the execution. After that first call to `localSeachSub` there is another sequential section, after which there is a new call to `localSeachSub`.

The need for a synchronization barrier is clearly the bottleneck of this program's performance.

6.2 GrPPI Adaptation

To adapt it into a GrPPI version, two approaches were implemented. The first one is quite straightforward: where PARSEC creates *nproc* threads calling the `pkmedian` function, one call to `grppi::map` that creates *nproc* threads with this function as its lambda works equivalently.

```
1  grppi::map(ex, arg.begin(), nproc, arg.begin(),
2        [](pkmedian_arg_t & r){
3                pkmedian((void*)&r);
4                return r;
5                });
```

Listing 6.2: Streamcluster Version 1

The `arg` vector is a vector of size *nproc*, and it contains the id for each thread and the barrier that will synchronize the threads just like in PARSEC's implementation.

Another approach was to study the code of the `pkmedian` function and make the calls to the GrPPI library inside it. In this case, we would be able to protect the sequential parts of the code by keeping them outside these calls; that is: where we used to have a call to the barrier followed by a piece of sequential code, now we have the same code outside any GrPPI calls, and thus executing sequentially. Listing 6.3 shows an extract of `pkmedian` where the parallel section has been adapted into a call to GrPPI patterns, and the section of the code that should be executed

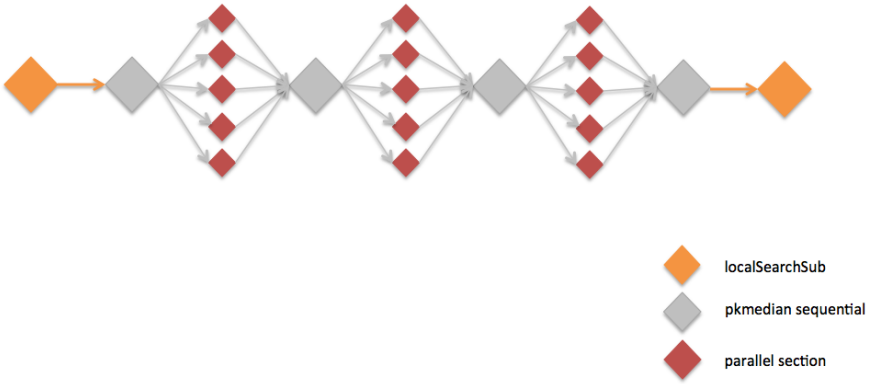
by only one thread sequentially (protected by `if (pid == 0)` before) is left outside:

```
grppi::map(ex, points->p, points->num, points->p,[p0, dim](Point
 punto)
          {float distance = dist(punto, p0, dim);
           punto.cost = distance * punto.weight;
           punto.assign=0; return punto;});


for(int i = 1; i < points->num; i++ )  {
      //... sequential code
}

Point out = grppi::reduce(ex, points->p, points->num, p, [](const
  Point & x, const Point & y){ Point w; w.cost = x.cost + y.cost;
  return w;});
```

Listing 6.3: Streamcluster Version 2

Figure 6.2 shows a graphic representation of this execution. Instead of having multiple threads executing `pkmedian`, we have identified the parallel sections and isolated the threads to those sections.

Figure 6.2: Streamcluster Version 2 diagram



Where there was a barrier in Version 1, there's a gray square (sequential) now

6.3 Results

The parameters for each input size were:

- small size - $n = 4096, d = 32, chunksize = 4096$
- medium size - $n = 8192, d = 64, chunksize = 8192$
- large size - $n = 16384, d = 128, chunksize = 16384$

with values $k1 = 10, k2 = 20, clustersize = 1000$ and seeds $27, 33, 100, 555, 1939$ for every test set. The results are shown in Table 6.1 and Figure 6.3. G_THR and G_SEQ are the execution times for the high-level parallelism adaptation (Version 1). G_THR_LLP and G_SEQ_LLP are the times for Version 2.

Table 6.1: Streamcluster test results

SIZE	NTHREADS	THR	G_THR	G_THR_LLP	SEQ	G_SEQ	G_SEQ_LLP
4K	1	0.388	0.383	0.385	0.380	0.376	0.377
	2	0.241	0.234	0.263			
	5	0.214	0.228	0.208			
	10	0.339	0.356	0.345			
	15	0.589	0.582	0.559			
	20	0.872	0.841	0.881			
8K	1	1.805	1.797	1.812	1.795	1.787	1.800
	2	0.920	0.965	0.971			
	5	0.483	0.532	0.507			
	10	0.543	0.571	0.514			
	15	0.757	0.746	0.736			
	20	1.095	1.057	1.042			
16K	1	8.4	8.4	8.4	8.4	8.4	8.4
	2	4.2	4.3	4.3			
	5	1.9	1.9	1.9			
	10	1.21	1.24	1.22			
	15	1.20	1.24	1.25			
	20	1.39	1.50	1.39			

There are two observations worth highlighting:

– The performance of both GrPPI implementations is the same and just as good as the original PARSEC pthread mode.

– This benchmark shows terrible speedup as the number of executing threads increases.

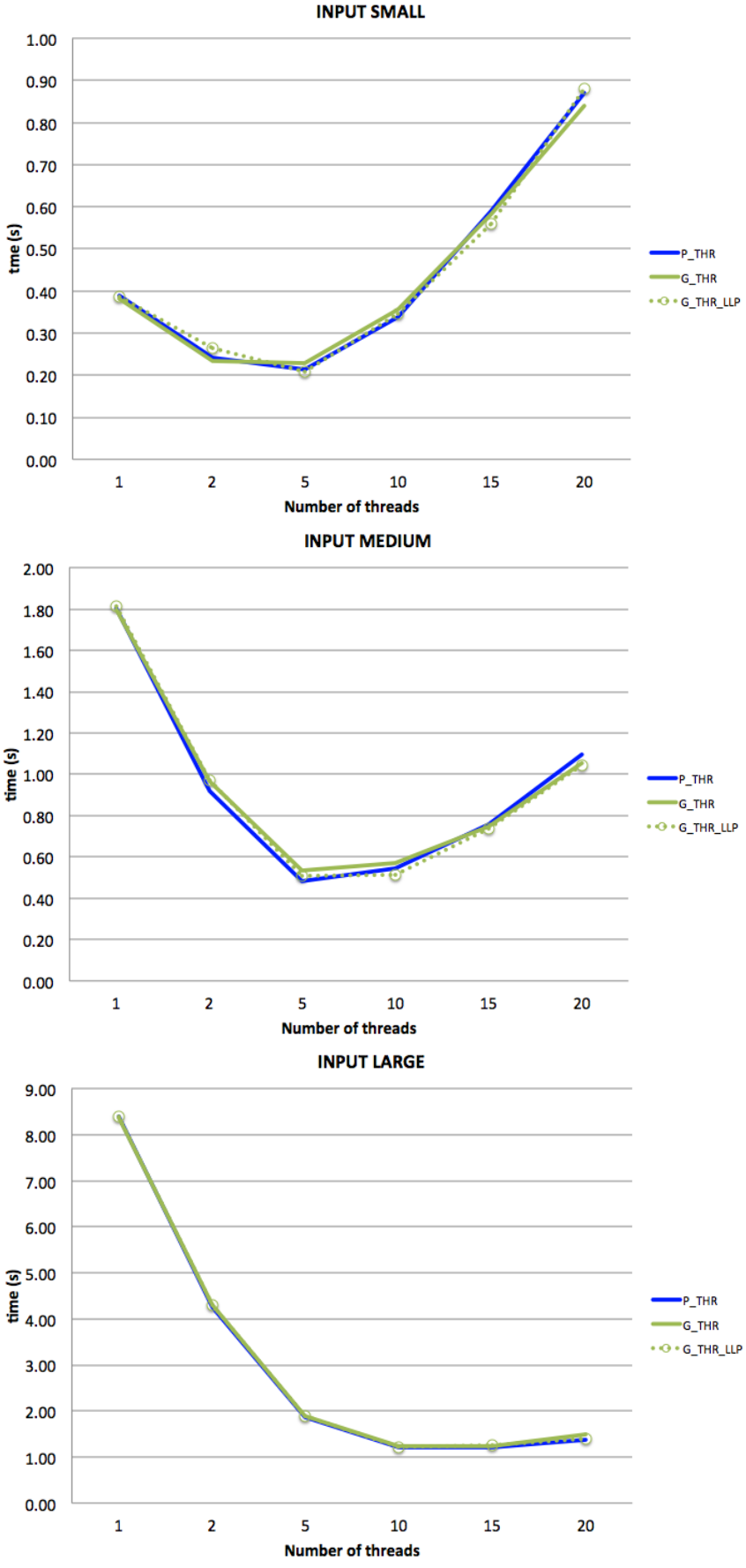
One may have thought that Version 1 of our GrPPI implementation (one unique call to the GrPPI library from a high-level function) would have obtained better results, since it allows the program to divide the work from an early stage and involves only one call to the GrPPI library, thus the threads are only created once. The results show, however, that the low-level calls are just as efficient, despite the threads being created and destroyed with each call. This makes us think that the sequential section of the program could be affecting the efficacy of the parallelization - which brings us to the second point.

Table 6.2 presents the speedup for the different input sizes. Unlike previous benchmarks, where the speedup was nearly optimal for all number of threads, we see a decrease in the execution speed for all execution modes as the number of threads increases, the most obvious one being the small-sized input case, where running the program with 20 threads is 100% slower than running with 1 thread. The maximum speedup is found at 5 threads, except for the large input where 15 threads gain a speedup of only 7.

Table 6.2: Speedup for Swaptions

SIZE \ NTHR	2	5	10	15	20
4K	1.6	1.7	1.1	0.7	0.5
6K	2.0	3.7	3.3	2.4	1.6
16K	2.0	4.5	6.9	7.0	6.0

Figure 6.3: Streamcluster test graphs



The fact that there seems to be a limit to the degree of parallelization of this program when there wasn't one for the previous benchmarks seems to point to the barrier as the main cause of this limitation. The cost of synchronizing all the threads in order to protect the execution in certain sections of the code surpasses the benefits of distributing the workload. Other studies of this benchmark have reached the same conclusion [12] and the best solution is to improve the implementation of the barrier in order to make it more efficient. That work is, however, beyond the scope of this paper.

The conclusions that arise from this Chapter are:

1. Both versions of the GrPPI adaptation show the same efficiency.
2. This benchmark's performance worsens as the number of threads increases due to the synchronization barrier.

7

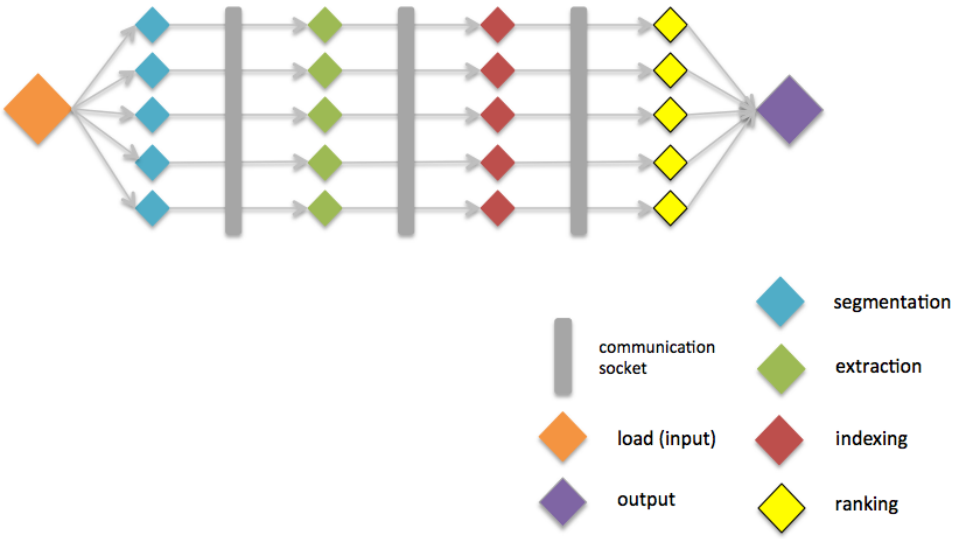
Ferret

7.1 Description

The ferret application [5] is used for image similarity search; that is, given a set of image queries, its output is a list of the k images from an image database that are most similar to each of the queries. As input, this program must receive: *database* path to image database, *table* name of `.lsh` file, *query dir* path to folder of image queries, numbers k, d, nt (search for the k most similar images, with queue size between stages d and nt threads for the middle stages, and *out* name of output file).

The benchmark is parallelized using a pipeline model with six stages. The first and last stages are for input and output. The middle four stages are for image segmentation, feature extraction, indexing of candidates sets and ranking, and are completely parallelizable, since they take one element (one image) from the previous stage and process it independently. To allow communication between stages, 5 queues are created at the beginning of the program; each stage takes elements from their own queue and sends its output onto the next stage's queue. Figure 7.1 shows a diagram of this program execution pattern.

Figure 7.1: Ferret diagram



7.2 GRPPI Adaptation

This benchmark seems like the perfect match for the `grppi::pipeline` function.

```
grppi::pipeline(ex,
  [&pd]()->std::experimental::optional<load_data *>{return t_load()
   ;},

  grppi::farm(NTHREAD_SEG,
  [](load_data * v_in){return t_seg(v_in);}),
  grppi::farm(NTHREAD_EXTRACT,
  [](seg_data * v_in){return t_extract(v_in);}),
  grppi::farm(NTHREAD_VEC,
  [](extract_data * v_in){return t_vec(v_in);}),
  grppi::farm(NTHREAD_RANK,
  [](vec_query_data* v_in){return t_rank(v_in);}),

  [](rank_data * v_in){return t_out(v_in);}
  );
```

Listing 7.1: Ferret GrPPI call

The first and last stages must be executed sequentially, since they read from and write to files. That is not the case for the other four stages, that can distribute the work among various threads without synchronization problems. In order to achieve different number of threads for the different stages, the `farm` parallel pattern is called with the number of threads as its first parameter. As with the other patterns, the pipeline's parallel implementation is controlled via the execution model parameter, that can be set to operate in sequential or in parallel through the different supported frameworks.

7.3 Results

For this test, PARSEC's input image databases were used as input. Small-size test had 64 query images; medium-size, 256; and large-size input had 1000 images. For all tests parameters k and d had value 10. The number $nthreads$ shown in the table is the number of threads assigned for each middle stage, so for each execution the total number of threads is $nthreads * 4 + 2$.

One of the first things to appreciate from the collected data in Table 6.1 is how the pipeline execution, both PARSEC's original and our GrPPI version, are at least twice as fast as PARSEC's serial version, even when executed with only one thread per stage. This speaks about the efficiency of the pipeline as a parallelization pattern, where one stage doesn't have to wait for the previous one to finish processing the whole input, but can start working on the elements that have already been outputted.

The speedup gained by increasing the number of threads comparing with the serial execution is shown in Table 7.2. The speedup for larger input sizes is closer to the number of threads for the middle stages, $nthreads$, than to the total number of threads $4 * nthreads + 2$. Even not taking into account the first and last stages - sequential - this speedup is far from optimal; we are comparing the execution with

Table 7.1: Ferret test results

SIZE	NTHREADS	THR	G_THR	G_OMP	P_SEQ
64	1	0.385	0.389	0.387	0.644
	2	0.215	0.217	0.216	
	3	0.164	0.165	0.165	
	4	0.148	0.149	0.148	
	5	0.145	0.207	0.256	
	6	0.141	0.192	0.245	
256	1	7.51	7.56	7.52	13.17
	2	3.78	3.80	3.78	
	3	2.55	2.58	2.56	
	4	1.95	1.97	1.96	
	5	1.66	2.06	2.17	
	6	1.40	1.68	2.09	
1000	1	34.5	34.8	34.6	71.8
	2	17.3	17.5	17.4	
	3	11.6	11.7	11.7	
	4	8.8	8.9	8.8	
	5	7.7	8.2	11.7	
	6	6.0	7.0	7.6	

*Total number of threads = 4*NTHREADS + 2*

one single thread to the execution with $4 * nthreads$ threads, but the increase of speed is much smaller.

Table 7.2: Speedup for ferret - 1

SIZE \ NTHR	1	2	3	4	5	6
64	1.7	3.0	3.9	4.3	3.1	3.3
256	1.7	3.5	5.1	6.7	6.4	7.8
1000	2.1	4.1	6.1	8.1	8.7	10.2

Speedup for each set of inputs in GrPPI's native parallel execution mode against PARSEC's sequential execution.

Table 7.3 shows the speedup comparing to the $nthreads = 1$ execution.

Table 7.3: Speedup for ferret - 2

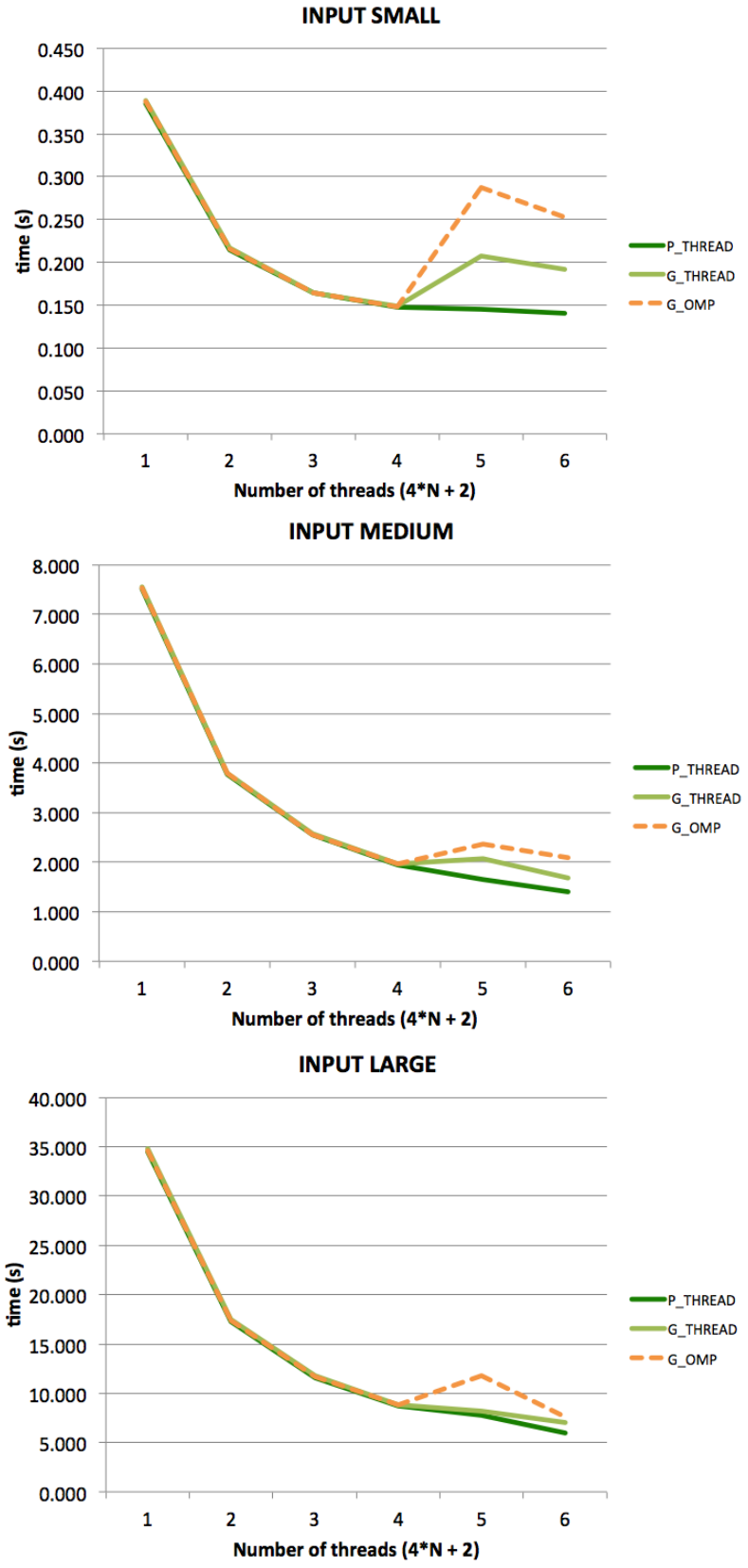
SIZE \ NTHR	2	3	4	5	6
64	1.8	2.3	2.6	2.6	2.7
256	2.0	2.9	3.8	3.7	4.5
1000	2.0	3.0	3.9	4.8	5.5

Speedup for each set of inputs in GrPPI's native parallel execution mode against native execution with 1 thread.

From this point of view, the speedup *nthreads* is optimal, since we are adding one thread to each stage. It is important to consider that for $nthreads > 4$, the number of threads is greater than the total number of computer cores, thus gener-

ating oversubscription, so those results aren't completely reliable.

Figure 7.2: Ferret test graphs



Stage-wise, the program is parallelizable and behaves as it should. However, the application as a whole doesn't show most optimal speedup. What is the cause for this? Section 7.4 studies this question

7.4 Core-distribution analysis

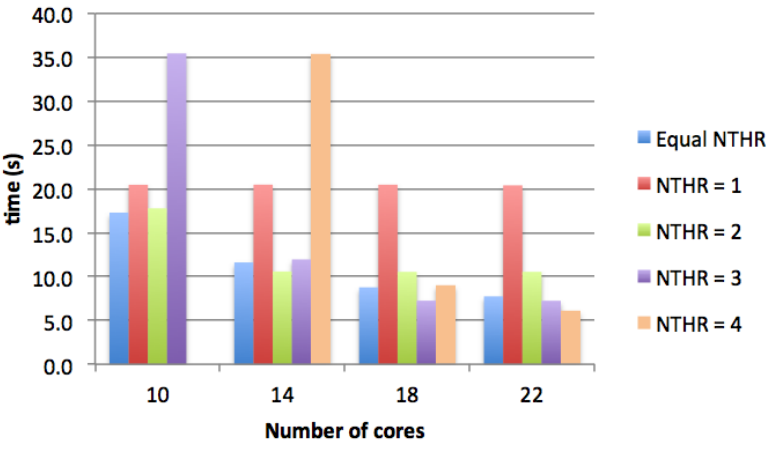
A first idea that could explain the non-optimal speedup for the Ferret benchmark is that maybe not all stages are equally time-consuming, and so the number of cores assigned to each one of them should be different.

We measured the time spent in each stage every time one element is processed, for the large input set with $k = 10$ and, although the values measured varied for each image, the upper values for each of the middle stages are:

t_seg	t_ext	t_ind	t_rank
0.006	0.001	0.003	0.030

We see that there is one stage about 10 times slower than the rest: the ranking stage. Will the application's performance improve if we assign more cores to this one stage instead of sharing them evenly among all stages? Figure 7.3 shows the times measured for different core-distribution strategies on the left graph. The table on the right shows how many threads were assigned to each stage in each execution. Note that the first and last stage must be executed sequentially and therefore only have 1 thread.

Figure 7.3: Ferret core-distribution



CORES	S1	S2	S3	S4	S5	S6
10	1	2	2	2	2	1
	1	1	1	1	5	1
	1	2	2	2	2	1
	1	3	3	3	1	1
14	1	3	3	3	3	1
	1	1	1	1	10	1
	1	2	2	2	7	1
	1	3	3	3	4	1
	1	4	4	4	1	1
18	1	4	4	4	4	1
	1	1	1	1	15	1
	1	2	2	2	12	1
	1	3	3	3	9	1
	1	4	4	4	4	1
22	1	5	5	5	5	1
	1	1	1	1	17	1
	1	2	2	2	14	1
	1	3	3	3	11	1
	1	4	4	4	8	1

For a given number of total cores in use, four different distributions are presented:

- *EqualNTHR*: all middle stages have the same number of threads.
- *NTHR=1*: all middle stages have 1 thread except for the Ranking stage, which has $total\_cores - 3 * 1 - 2$ threads.
- *NTHR=2*: all middle stages have 2 threads except for the Ranking stage, which has $total\_cores - 3 * 2 - 2$ threads.

– *NTHR=3*: all middle stages have 3 threads except for the Ranking stage, which has $total\_cores - 3 * 3 - 2$ threads.

– *NTHR=4*: all middle stages have 4 threads except for the Ranking stage, which has $total\_cores - 3 * 4 - 2$ threads.

We see that executions where the Ranking stage is executed in only 1 thread show a much worse performance than other executions, even if the number of threads for the other stages is greater (i.e. compare *EqualNTHR* and *NTHR=3* for a total of 10 cores). However, assigning all threads to the Ranking stage isn't the best solution either. Observing the *NTHR=1* columns, we see how the three stages with only 1 thread are preventing the application from running faster in spite of the Ranking stage running on a big number of threads.

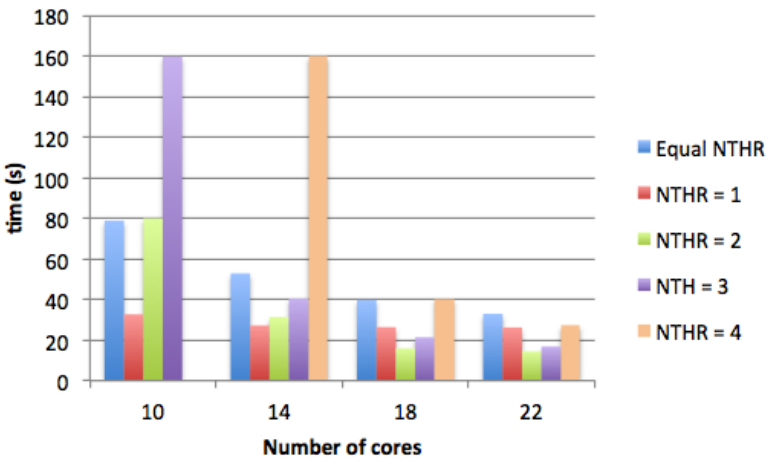
Best performance is found in executions with a $\frac{ranking-threads}{other-stages}$ ratio of about 2; for greater ratios, the other stages become the application's bottleneck (i.e. See NTHR=3 for 18 and 22 cores, where there is barely any improvement).

Increasing the number of image queries doesn't affect the time spent in each stage; it only increases the number of times they will be called. Increasing the number of similar images we want to find, the k parameter, does affect the time spent in one stage:

t_seg	t_ext	t_ind	t_rank
0.006	0.001	0.003	0.300

The Ranking stage continues to be the most expensive stage, but now the time difference is even greater. We performed the same core-distribution experiments, presented in Figure 7.4

Figure 7.4: Ferret core-distribution - k = 50



CORES	S1	S2	S3	S4	S5	S6
10	1	2	2	2	2	1
	1	1	1	1	5	1
	1	2	2	2	2	1
	1	3	3	3	1	1
14	1	3	3	3	3	1
	1	1	1	1	10	1
	1	2	2	2	7	1
	1	3	3	3	4	1
	1	4	4	4	1	1
18	1	4	4	4	4	1
	1	1	1	1	15	1
	1	2	2	2	12	1
	1	3	3	3	9	1
	1	4	4	4	4	1
22	1	5	5	5	5	1
	1	1	1	1	17	1
	1	2	2	2	14	1
	1	3	3	3	11	1
	1	4	4	4	8	1

For this test, we see that the *NTHR=1* is behaving much better than the *EqualNTHR*. The time-difference between the Ranking stage and the other three is so big that assigning the maximum number of threads to this one stage is the best strategy when using a small number of cores. Note how the *NTHR=1* strategy with 10 cores is faster than *EqualNTHR* with 18 cores. For greater number of cores available, the

three fast stages may become the bottleneck, and we can obtain a somewhat better execution time with a $\frac{ranking-threads}{other-stages}$ ratio of 6 (See *NTHR=2* for 18 cores).

This Chapter's conclusions are listed below:

1. GrPPI's pipeline works just as well as a manually implemented pipeline pattern like PARSEC's.

2. When the pipeline stages are unbalanced, time-consumption wise, finding the optimal ratio between the stages' assigned number of threads yields the best performance.

8

Conclusions

Throughout the course of this work we have worked with and studied the efficiency of the GrPPI library as an alternative for more standard parallelization APIs. This section presents a recap of the conclusions obtained thus far.

8.1 Tests Analysis

1. The Swaptions and Blackscholes benchmarks' speedup is virtually optimal, as long as the workload (input size, number of options) is big enough for the number of threads in use.

2. GrPPI OpenMP doesn't add any excess workload and is just as efficient as PARSEC's OpenMP and pthread versions.

3. GrPPI thread's performance was worse than other execution modes due to an inefficient workload distribution. By improving it, we have achieved optimal performance just like the other back-ends.

4. This optimization is only relevant for applications where the main function's *execution time per element / number of elements* ratio is big enough.

5. Multiple calls to the GrPPI library imply a hidden work of creating and destroying threads; however, this has insignificant influence in the program's total execution time.

6. The need of thread synchronization will affect the application's performance, decreasing its speedup for larger number of threads

7. GrPPI's pipeline works just as well as a manually implemented pipeline pattern like PARSEC's.

8. When the pipeline stages' are unbalanced, time-consumption wise, finding the optimal ratio between the stages' assigned number of threads yields the best performance.

8.2 Usability

In the previous sections of this work we have studied the performance of GrPPI in comparison to more specific parallel implementations in terms of execution time. Now we want to compare the programming effort required when using GrPPI. Although this may seem like quite a subjective criterion, we will measure it through an objective variable: lines of code [7].

Table 8.1 shows the number of lines of code added to the sequential version of each benchmark for the different parallel implementations.

Figure 8.1: Lines of code per implementation

	pthreads	OpenMP	GRPPI	
blackscholes	15	5	2	87%
swaptions	18	3	2	89%
stream	111	-	37	67%
ferret	64	-	4	94%

We see that GrPPI requires on average 84% fewer lines of code than its pthreads counterpart. This difference is mainly due to the requirement to create the threads by invoking the `pthread_create(...)` function and then synchronizing with `pthread_join(...)`. All this work is done automatically by the GrPPI call when using a `parallel_execution_native` backend, and thus translates into lines of code saved.

For the Streamcluster benchmark, the low level parallelization version was considered. The number of lines of code is higher than the other applications because, since some functions required major modifications in order to be adapted for GrPPI use, we decided to compare the length of these functions as a whole.

8.3 Future Work

GrPPI proves to be an efficient, user-friendly parallel pattern interface which provides programmers with the benefits of existing parallel frameworks without the hardship of understanding their implementation. GrPPI promises to help make more readable, versatile parallel programs with no additional overheads.

As future work, we would like to further look into the compatibility of the GrPPI patterns among themselves. Studying the behaviour of GrPPI's other back-ends would also be interesting, specially the Intel TBB execution mode, since the TBB runtime library manages threads differently than OpenMP.

We also want to study the GrPPI adaptation of more applications from the PARSEC benchmark, which might give us the chance to use other parallel patterns from the GrPPI library. In particular the Dedup application, although tagged as a pipeline pattern due to its five stages, doesn't have a direct translation into the `grppi::pipeline` function like Ferret did. The challenge this benchmark presents

is that some stages fragment their input, producing more than one output element. This doesn't sit well with the pipeline pattern, where the number of inputs and outputs for each stage must be the same. Finding the best solution to this problem is our first priority.

Bibliography

[1] OpenMP - Enabling HPC since 1997. https://www.openmp.org, 2007-2019. Accessed 2020/06/10.

[2] Xeon Gold 6138 - Intel. https://en.wikichip.org/wiki/intel/xeon_gold/6138, 2019. Accessed 2020/05/25.

[3] Danelutto M. Kilpatrick P. Aldinucci, M. and M. Torquati. Fastflow: High-Level and Efficient Streaming on Multicore. In Programming multi-core and many-core computing systems. https://doi.org/10.1002/9781119332015.ch13, 2017.

[4] Blaise Barney. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/#Pthread, 2020. Accessed 2020/06/10.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] Jaswinder Pal Singh Christian Bienia, Sanjeev Kumar and Kai Li. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. PhD thesis, Princeton University, January 2008.

[7] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24), 2017. doi: 10.1002/cpe.4175. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4175.

[8] John L. Henessy and David A. Patterson. *Computer Architecture: A Quantitative Approach. Fifth Edition*. Elsevier, 225 Wyman Street, Waltham, MA 02451, USA, 2012.

[9] Intel. Intel® Threading Building Blocks. https://github.com/oneapi-src/oneTBB, 2020.

[10] Samuel P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan Claypool, Purdue University, 2012.

[11] Luis Miguel Sanchez, Javier Fernández, Rafael Sotomayor, Soledad Escolar, and J. Daniel Garcia. A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. *New Generation Computing*, 2013.

[12] G. Southern and J. Renau. Analysis of parsec workload scalability. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[13] Princeton University. PARSEC. `https://parsec.cs.princeton.edu/`, 2007-2009. Accessed 2020/06/10.