A comparative analysis between parallel models in C/C++ and C#/Java

A quantitative comparison between different programming models on how they implement parallism

GÖRAN ANGELO KALDÉREN
and
ANTON FROM

KTH Information and Communication Technology

Bachelor of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:157

KTH Information and Communication Technology

A comparative analysis between parallel models in C/C++ and C#/Java

A quantitative comparison between different programming models on how they implement parallelism

GÖRAN ANGELO KALDERÉN, ANTON FROM

Bachelor's Thesis at ICT
Supervisor: Artur Podobas
Examiner: Mats Brorsson

TRITA-ICT-EX-2013:157

Abstract

Parallel programming is becoming more common in software development with the popularity of multi core processors on the rise. Today there are many people and institutes that develop parallel programming APIs for already existing programming languages. It is difficult for a new programmer to choose which programming language to parallel program in when each programming language has supporting APIs for parallel implementation. Comparisons between four popular programming languages with their respective most common parallel programming APIs were done in this study. The four programming languages were C with OpenMP, C++ with TBB, C# with TPL and Java with fork/join. The comparisons include overall speedup, execution time and granularity tests.

The comparison is done by translating a chosen benchmark to other programming languages as similar as possible. The benchmark is then run and timed based on how long the execution time of the parallel regions. The number of threads are then increased and the execution time of the benchmark is observed. A second test is running the benchmark on different granularity sizes with the constant number of available threads, testing the behavior on how large or fine grained tasks each language can handle.

Results show that the programming language C with OpenMP gave the fastest execution time while C++ gave the best overall speedup in relation to its sequential execution time. Java with fork/join was on par with C and C++ with a slight decay of overall speedup when the number of threads was increased and the granularity became too fine grained. Java could handle the granularity test better than C where it could handle very fine granularity without losing the overall speedup. C# with TPL performed the worst in all scenarios not excelling in any tests.

Referat

En komparativ analys mellan parallella modeller i C/C++ och Java/C#

Med den ökande populariteten av flerkärniga lösningar har parallellprogrammering börjat bli ett mer vanligt tillvägagångssätt att programmera i mjukvaruutveckling. Idag är det många personer och institutioner som utvecklar parallellprogrammerings APIer för redan existerande programmeringsspråk. Det är svårt för en ny programmerare att välja ett programmeringsspråk att parallellprogrammera i när varje programmeringsspråk har stöttande APIer för parallel implementering. I denna studie har fyra populära programmeringsspråk jämförts med deras respektive mest vanliga parallellprogrammerings APIer. De fyra programmeringsspråken var C med OpenMP, C++ med TBB, C# med TPL och Java med fork/join. Jämförelserna innefattar den generella uppsnabbningen, exekveringstiden och kornigheten.

Jämförelsen görs genom att översätta ett utvalt prestandatest till andra programmeringsspråk så lika varandra som möjligt. Prestandatestet körs sedan och tidtagning sker baserat på hur lång exekveringstiden av de parallella regionerna är. Sedan ökas antalet trådar och exekveringstiden av prestandatestet observeras. Ett andra test kör prestandatestet med olika storlek på kornigheten med ett konstant antal möjliga trådar och testar beteendet på hur stor eller liten kornighet på uppgifterna varje språk kan hantera.

Resultaten visar att programmeringsspråket C med OpenMP hade den snabbaste exekveringstiden, medan C++ hade bäst generell uppsnabbning. Java med fork/join höll jämna steg med C och C++ med en lätt tillbakahållning av den generella uppsnabbningen när antalet trådar ökade och kornigheten minskade. Java hanterade kornigheten bättre än C där den kunde hantera väldigt liten kornighet utan att förlora den generella uppsnabbningen. C# med TPL hade sämst resultat i alla scenarion och framstod inte i något utav testerna.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel programming is becoming the leading way of programming in future software development [5]. With the limitations on how fast a single processor can calculate, the development of multi core processors are on the rise. The reason for the sudden stop in single core processors is the amount of power needed to increase processor frequency speeds [12]. This was neither efficient nor possible leaving multi core solutions more attractive. Multi core processors in personal computers today need little to no parallelism for each program executed.

Today's personal computers only have two to eight cores to utilize meaning that if you had two cores, opening a browser would run on the first core and opening a music payer will run on the second core, without parallelizing these programs running them simultaneously utilizes all cores. Although if Moore's law were to continue without the capability of increasing the speed of one processor but increasing the number of cores, later on this will not be enough. To fully take advantage of multi core processors parallelism must be implemented for the program to run efficiently on these processors. Not only is parallelism useful on multi core processors, taking advantage of a wide network of computers to calculate heavy equations also needs a parallelism to run these algorithms correctly. Several years of research has brought a variety of parallel software development tools giving current developers simple yet efficient APIs for existing programming languages.

Today's software developers have little experience in parallel programming since sequential programs worked fine before multi core processors became popular in the market. The problem remains, which of these programming languages provide the best implementation of parallelism. Which is the most execution time efficient programming language for developers to use for existing and coming multi core processors.

1.1 Background

Developing applications can be done in several different programming languages but implementing parallelism with the help of their respective API is quite new. These

API tools were developed for current software developers on existing programming languages so that upgrading existing programs and systems for more concurrent functionality would be easy but this was not necessary until recently.

The task is to perform a comparative analysis of how parallel models are implemented in different programming languages with their respective APIs. This report will focus on C/C++ and Java/C# and compare which of these languages give the best performance. These programming languages according to TIOBE [3] are currently the most popular and widely used programming languages by software developers and are appropriate to compare. The comparison will be done by benchmarking different typical parallel programs and calculating their performance. The benchmarking programs need to be translated from C/C++ code to Java/C# code as similar as possible to make it possible to compare these languages.

Parallel programming brings up different terms such as Tasks, Threads and Task-Centric Parallelism. Tasks in this research are much like threads in computer programming; they are pieces of code that will run in parallel or concurrently. Instead of handling each task as a thread, the tasks will use an existing thread that is already made yet idle and use it as a container to be executed. Compared to creating new threads, creating new tasks does not take as much resources, as tasks doesn't need to create overheads and alike compared to threads. Task-Centric Parallelism focuses on handling tasks instead of threads where threads function as workers which are already running as idle threads. These workers then get a task to execute and when finished takes another task or idles, instead of killing each thread then starting another thread up for each new task.

Benchmarking programs for parallel programming purposes are ranged from simple splitting of tasks to complex algorithms still capable of concurrency. Some examples of simple parallel benchmarking programs are a simulation of aligning sequences of proteins where the alignment of different proteins can be split into parallel regions and sparse LU factorization. An example of a more complex parallel benchmarking program would be computing a Fast Fourier Transformation. Classic parallel programs may also be used for benchmarking purposes such as the N Queens Problem and Vector Sorting using a mixture of algorithms. These benchmarking programs are the same benchmarking programs used by the Barcelona OpenMP Suite Project (BOTS) [14].

1.2 Problem Statement

Research on comparing different programming languages ubiquitously exists as it is often debated since the evolution of different programming languages. For instance Prechelt L. paper on comparing the programming languages C, C++, Java, Perl, Python, Rexx and Tcl [4]. There is much research on comparing different parallel models as well, such as Artur P. paper on task-based parallel programming frameworks [9].

There is little research on the comparison between different programming languages on how they implement parallelism with each other, specifically on how their respective APIs implement concurrency and how they compete with each other in terms of execution time performance.

1.3 Purpose

The purpose of this research is to perform a comparative analysis on the parallel aspects of the four programming languages C, C++, C# and Java, by performing benchmark tests.

To investigate, compare and finally reach a conclusion on which of the four programming languages implements parallelism most effectively in terms of execution time performance.

1.4 Hypothesis

The expected result from this research is that the imperative programming languages' C and C++ will be more efficient in terms of execution time performance, contrary to the managed programming languages' Java and C# which require a virtual machine.

It is expected because C and C++ are considered low-level languages which means that their code is close to the kernel/hardware and thus able to keep the amount of code needed to a minimum and at the same time maximize efficiency.

Java and C# are considered mid-level languages which means that they have a level of abstraction that make them easier to code but sacrifices the possibility to utilize pointer arithmetic and direct connection to the kernel/hardware.

1.5 Success Criteria

Successfully migrate the benchmarking programs to Java/C# code. Successfully run the benchmarking programs in C/C++ and Java/C#, compare and present the results. Either strengthening our hypothesis or proving it wrong, depending on the results produced from the tests.

1.6 Limitations

Only four of the most popular programming languages C, C++, C# and Java will be covered in this study. One of the more simple benchmarking programs will be used due to the limited knowledge of complex algorithms which more sophisticated benchmarking programs are using.

Parallel programming can be done in a number of ways. Different programming languages have their own APIs for parallel programming. The most used methods in the programming language C and C++ are with the Posix standard Pthreads, OpenMP and MPI. Only OpenMP will be used for C in this study to narrow down the scope to a manageable level and focus on the task-centric version of the API for easy migration to other languages. In C++ this research will focus on two separate benchmarks, one using the latest C++ version C++11 also known as C++x0 and one using C++ with TBB. For the programming language C# only the parallel model TPL (Task Parallel Library) will be tested. For the Java programming language the commonly used fork/join method will be used to make the code as similar as possible to the original benchmarking program.

Chapter 2

Theoretical Background

In the sea of programming languages, there has sprung up a sort of grouping system for categorizing different programming languages. There are three coarse grained categories: low-level, mid-level and high-level programming languages [13].

Low-level programming languages often have superb performance and have little-to-no abstraction level from the hardware. This means that the programmer has total control of what's happening and is able to pinpoint just the right amount of resources to solve some problem, but due to the downside of low abstraction it can be difficult to see the program flow and the programmer must know exactly what he's doing. Some examples of low-level languages are Assembly and C [13].

Mid-level programming languages have a higher level of abstraction and acts as a layer on top of a low-level programming language. Therefore they often come with a virtual machine that acts as a stepping stone for first compiling the mid-level code down to low-level code before it's translated to machine code. The higher level of abstraction makes the use of objects possible which makes it easier to follow the flow of the program and the programmer doesn't need to bother with registers and allocating memory anymore. The downside is that the programmer loses the pinpoint precision of low-level programming languages. Some examples of mid-level languages are Java and C# [13].

High-level programming languages have taken the abstraction level to the extreme. They are therefore very dynamic and give the programmer a huge amount of flexibility in algorithm design that mid- or low-level languages can't provide. Due to this flexibility, it can be hard to follow the flow of the program. Another downside is that due to the many layers of abstraction a few instructions may translate into thousands of machine words, which results in poor performance. Some examples of high-level languages are Ruby and Perl [13].

2.1 Parallel Computing

Unlike traditional software development with sequential programming, parallel programming introduces new challenges and properties.

There are different ways to see parallel programming, either focusing on the structural model of the parallel implementation which includes what type or parallel model e.g. task centric and recursive parallelism or focusing on the behavior of a parallel implemented program. Behavior wise, there are three types of classifications in parallel programming when looking at a parallelized program, fine grained, coarse grained and embarrassing parallelism. Fine grained parallelism is when the paralleled subtasks of the program constantly need to synchronize and communicate with each other every so often in order to function properly. Coarse grained parallelism is similar to fine grained parallelism but does not need to synchronize and communicate as often. Embarrassing parallelism however rarely needs to synchronize and communicate with each other. These classifications give an overhead of how a program with parallel implementation would act like.

The subtasks mentioned above (also called threads) have different types of synchronization and communication methods which are mutual exclusion and barriers. Mutual exclusion is commonly used to avoid race conditions and provides a lock for a thread to handle a critical section safely without any interference from other threads. Barriers are commonly used when threads needs to wait for other threads to finish in order to proceed or if a thread needs the results from another thread and therefore needs to wait for it. These properties make parallel programming both simple and complex.

2.1.1 Parallel model in C

Parallel programming APIs for the programming language C was standardized on October 1998 with the release of OpenMP for C. There were different APIs before the standard but OpenMP gave an official API over parallel programming in C.

There are different parallel models available for the C programming language. Pthreads and OpenMP are a few examples that support multi-platform shared memory parallel programming while MPI support distributed memory parallel programming. The parallel programming model OpenMP will be used for benchmarks tests for the programming language C.

OpenMP gives a simple scalable model that gives programmers a flexible interface in parallel programming for applications ranging from personal computers to research scale supercomputers. OpenMP is an implementation of multithreading, which is a method where a master thread forks a specified number of slave threads and tasks are divided to them. The threads are then distributed by the runtime environment to different processors to run theses threads in parallel. The section of code to be parallelized can be easily marked with a compilation directive. After each thread is done executing, the threads join back to the master thread which then continues.

2.1.2 Parallel models in C++

A parallel model in C++ was officially introduced in 2011 adding multi thread support without the use of parallel models from the programming language C such as OpenMP or MPI.

This new ISO standard for the programming language C++ is called C++11 which introduces new primitives such as atomic declaration of variables and store and load operations. The new standard also introduces sequential consistency, meaning that porting a sequential code to function in parallel without losing its sequential functionality and consistency. While this can cause slowdowns with different barriers and alike, this can be avoided by declaring in the store and load operations that this consistency is not necessary making the language more flexible.

C++ introduces these new standards for mutexes and conditional variables as well as new primitive types for variables so that these parallel implementations done in C++ is guaranteed to work on both today's and future machines. This is because the specifications do not refer to any specific compiler, OS or CPU but instead to an abstract machine which is the generalization of actual systems. This abstract machine, unlike the former versions of C++ fully supports multi-threading in a fully portable manner.

Another Parallel model to be tested in C++ is Intel's Threading Building Blocks (TBB) library which has a more task centric behavior. The advantage of this model is that it implements task stealing to properly balance a parallel workload across a multi core platform. If a processor finishes all the tasks in its queue, C++ TBB can then reassign more tasks to it by taking tasks from other cores with several tasks in its queue still waiting to be executed, this provides a more balanced workload over all processors and makes parallelism more efficient.

2.1.3 Parallel model in C#

The first version of C# already had support for parallel programming and appeared in 2000 [6]. Later releases have got even more support for parallel programming. The most recent version is C# 5.0 and was released in August 15, 2012 [11].

The first version of the .NET Framework was introduced and made parallel programming in C# much easier. The most recent version of the .NET Framework version 4.5 contains a number of parallel programming APIs such as the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). TPL is the preferred way to write parallel applications in C# and will be used for the benchmarking programs.

TPL makes parallel programming in C# easier by simplifying the process of adding parallelism and concurrency to applications. It scales the degree of concurrency dynamically to most efficiently use all the processors available. TPL also takes care of all the low-level details, such as scheduling the threads in the thread pool and dividing the work. By using TPL the programmer can maximize the performance of the code while focusing on the work the program is designed to accomplish.

A thread pool is a collection of threads that execute tasks. The tasks are usually organized in a queue for the thread pool to execute. When a thread is idle or just finished executing a task, a new task from the queue is fetched and executed (if there is any left). This allows the reuse of threads to eliminate the need to create new threads at runtime. Creating new threads is typically a time and resource intensive operation.

2.1.4 Parallel model in Java

Java is an object oriented programming language that is classed as a mid-level programming language. Java uses classes, objects and other more abstract data formats. Java comes with a virtual machine (JVM) that provides an execution environment and automatic garbage collection. As a mid-level language it acts as a layer on top of a low-level language, in fact the Java compiler is bootstrapped from C [13].

Before Java included the first package of concurrency utilities in Java SE 5 (September 30, 2004) [7], developers had to create their own classes to handle the parallel aspects of multi core processors. Thanks to newer versions of Java, developers now have solid ground to stand on when programming parallel applications. Julien Ponge said that:

> *Java Platform, Standard Edition (Java SE) 5 and then Java SE 6 introduced a set of packages providing powerful concurrency building blocks. Java SE 7 further enhanced them by adding support for parallelism [10].*

Java has a number of different methods to implement parallelism. The most used are fork/join and monitors. Recently Java have received OpenMP support in the form of JaMP, an OpenMP solution integrated into Java which have full support for OpenMP 2.0 and partial support for 3.0 [8].

2.2 Benchmarking

Benchmarking is the process of comparing two or more similar products, programs, methods or strategies to determine which one of them has the best performance. The comparison is made by running dedicated benchmarking programs that measure different aspects of performance.

The objectives of benchmarking are to determine what and where improvements are called for, to analyze how other organizations achieve their high performance levels and to use this information to improve performance.

When benchmarking different programming languages the chosen benchmarking programs are coded in each corresponding language and then run on the same machine under the same circumstances. Usually these benchmarking programs perform heavy calculations or processes huge amounts of data and performance is most often measured in execution time or number of operations.

2.2.1 SparseLU - Sparse Linear Algebra

The SparseLU benchmarking program uses sparse linear algebra to compute the LU factorization of a sparse matrix. The sparse matrix is implemented as a block matrix of size 50 x 50 blocks with a block size of 100 x 100 units instead of one relatively large 5000 x 5000 matrix.

A sparse matrix is a matrix which mainly contains zeros in the table. It is used in science or engineering when solving partial differential equations and is also applicable in areas which have a low density of significant data or connections, such as network theory.

LU factorization factors a matrix as the product of a lower and an upper triangular matrix. The product sometimes includes a permutation matrix as well.

Chapter 3

Methodology

The method chosen for testing and comparing the programming languages parallel model is by execution time performance. The benchmarking programs were tested in different programming languages with their respective parallel model.

Execution time performance was chosen as the main testing criteria as todays focus when developing multi core processors is speed. It is then only reasonable to test the execution time performance of different parallel models for the different programming languages.

Other criteria such as power and memory resource management would have been very interesting to take into account while testing different programming languages with their respective parallel model. But the availability of this information is limited and would be difficult to log.

The benchmarks were executed on the Royal Institute of Technology's multi core computer Gothmog with a total of 48 cores divided on 4 sockets. Each socket has 2 NUMA-nodes, each with 6 processors and 8 GB of local RAM. Each of the processors are based on the AMD x86-64 architecture (Opteron 6172) [9].

The benchmarks were executed 5 times to get a median value and then shifted up to the next amount of threads. The number of threads tested were 1, 2, 4, 8, 16, 24, 36 and 48. This was done by a simple script that executed the benchmark several times with different input for the number of threads used. The scripts for each language can be found in Appendix B. The speedup was calculated as $S(n) = T(1)/T(n)$, where n is number of threads, $T(1)$ is the time it takes to execute with 1 thread, $T(n)$ for n threads and $S(n)$ is the achieved speedup.

Analyzing the execution time was done simple. Each test provided a speed result of its parallel and serial sections. The results were then used to compare each programming language parallel execution time performance compared to its sequential counterpart giving a percentual speedup depending on the number of threads used. This was one type of test that provided a simple overhead of the speedup each language obtained when increasing the number of threads. The second type of test was to compare the overall speedup with a specific number of threads, comparing the languages with each other. The third type of test was to find out how well each language could handle different granularities on tasks without loosing its overall speedup.

3.1 Testing C

The benchmark SparseLU from BOTS was originally written in C with the OpenMP 3.0 API for the parallel regions of the code. There was no need to rewrite or change the structure of the program since a task centric version of the code was available.

Minor changes before it was put to use was made for instance the program was ported all into one file for easy reading and future porting to other languages. All the variables and functions were all in one file.

More changes to the benchmark program was made to put in time stamps for speed performance tests. Time stamps were implemented right before the execution of the parallel regions *# pragma omp parallel* and right after all the threads ended in order to only measure the parallel speedups of the code when the number of threads were varied. In regards to Amdahl's law this implementation made it easier to calculate the parallel speedup of the code instead of measuring the whole code's execution time performance. The tools used to make these time stamps are with the help of OpenMP's own clock functions and variables *omp_get_wtime()*.

The program was compiled on the multi core server Gothmog with the gcc compiler. The flags for optimization and OpenMP library (libgomp) were added *gcc -fopenmp -O3 sparselu.c -o sparselu*.

To test the benchmark with different number of threads as well as different granularities, the program was edited once more to fetch input of the amount of threads, matrix size and sub matrix size to be used for easier testing. This led to the program able to run the benchmark program line *sparselu 4 50 100* where 4 is the number of threads, 50 is the matrix size and 100 is the sub matrix size.

3.2 Testing C++ TBB

The TBB version of the benchmark was also available so no porting was made. It used the original testing and compiling principles as the original BOTS package. This automatically provided measurements for the parallel sections of the benchmark.

A slight change was made, an extra flag *-w* was added to enable input for the number of threads to be used. As for granularity inputs, this was already implemented with the flags *-n* for the matrix size and *-m* for the sub matrix size.

Compiling and running the complete package provided by BOTS was done in several steps. To compile and run it on the multi core server Gothmog the command *source /opt/intel/bin/compilervars.sh intel64* was used to configure the compiler *tbb(icc)*. TBB(ICC) stands for Intel's C Compiler for the TBB version. A make file was already provided to compile the code. To run it the following file *sparselu.icc.tbb -w* was executed with the flag *-w 48* where 48 is the number of threads to be used.

3.3 Testing C++11

Unfortunately testing C++11 was proven difficult. Porting the sequential code from C to C++11 was easy but implementing this new parallel standard was difficult without changing the parallel model it originally had in C, since C++ does not fully support task centric parallelism. C++11 focused on recursive parallelism and even the asynchronous functions of the language focused on the same functionality. Because of this it was not tested and no results will be presented for C++11. The standard is fairly new and rarely used in the programming community, it is mainly researched on.

3.4 Testing C#

First, sparseLU was ported from C to C# in Visual Studio 2010 where it was built, compiled and debugged. Since C# didn't support pointers and pointer arithmetic's like C, an approach using *out* and *ref* was chosen to minimize the number of data copies. Out and ref are used to send secure pointers to variables as parameters to functions in C#. Another problem was that C# didn't allow free control of the number of threads in the thread pool. It was a crucial problem that was finally solved by restricting the number of tasks run by the thread pool instead of restricting the number of threads. This came with a bit of performance loss as the overhead became bigger. The tasks were created and managed by the Task Parallel Library (TPL).

To be able to measure the parallel region of the code, the *Stopwatch()* method was used to measure the parallel regions execution time. The stopwatch started just before the program entered the parallel region and stopped right after the region ended. This implementation made it easier to calculate the parallel speedup of the code.

After fixing all bugs and obtaining an executable file (.exe), the program was moved to RITs multi core computer Gothmog. There the C# .exe file was run with the help of *mono*. Mono is a runtime implementation of the Ecma Common Language Infrastructure which can be used to run Ecma and .NET applications [1]. Ecma International is an industry association that is dedicated to the standardization of Information and Communication Technology (ICT) and Consumer Electronics (CE) [2].

To test the benchmark with different number of threads, the program was edited to fetch input of the amount of threads to be used for easier testing. To run it on Gothmog, the following line was used: *mono sparselu.exe numThreads*, where numThreads are the number of threads to be used in the parallel version.

Later the benchmark was edited once more to be able to fetch input for different matrix sizes. This was done in order to be able to test the granularity performance of the benchmark with 48 threads running. To run the program with granularity size input the following line was used *mono sparselu.exe numThreads size subsize* where size is the size of the matrix and subsize is the size of the sub matrices.

3.5 Testing Java

C# and Java have similar syntax and when the C# version of sparseLU finished it was easy to port from C# to Java in Eclipse. It was built, compiled and debugged in Eclipse. To counter the loss of pointer arithmetic, the matrix was made global so every thread had access to it. To control the number of threads the ForkJoinPool class in the concurrent package was used.

A *ForkJoinPool* object controls an underlying thread pool with a specified number of threads and has a queue for tasks. If the number of tasks is greater than the number of threads, the rest of the tasks wait in the queue until a thread goes idle and can execute another task.

To get the timestamps for measuring the parallel region of the code, the *System.nanoTime()* method was used. The timestamps was taken right before the parallel region was entered and right after it ended. The execution time was then received by subtracting the start time from the stop time. Thanks to this measuring technique it was very easy to calculate the parallel speedup of the code. After all bugs were fixed, the source code was moved to Gothmog. The benchmark was compiled to Java bytecode with *javac* and then executed with the Java interpreter - the *java* command.

To be able to test the benchmark with different number of threads, the program was edited to fetch input of the amount of threads to be used for easier testing. To run it on Gothmog, the following line was used: *java Main numThreads*, where numThreads are the number of threads to be used in the parallel version. The benchmark was edited once more to be able to fetch input for different matrix sizes. To run the program with granularity size input the following line was used *java Main numThreads size subsize*.

3.6 Comparison

One of the main methods of presenting the results obtained from the test runs was the percentual speedup gained from the increased number of threads. This gave an overall overhead of how each programing language performed based on the number of threads given which made it easily comparable with each other. Another comparison of the same kind was documented, the overall execution time for each programming language.

Another test for comparing the programming languages was changing the size of the benchmark's matrix and sub matrix while keeping the total matrix size of 5000 x 5000. This changed the granularity of each task and the scale of how much the benchmark could utilize the available threads. The different sizes used is shown in table 4.1 on page 21.

3.7 Coding effort

Coding effort is usually measured in LOC (Lines Of Code) written during different stages of a project. In this case only the number of LOC in the final benchmarks were measured. Seen in table 3.1 are the LOC of all four benchmarks. The code was measured in non-commented LOC. A non-commented line of code was defined as any line of text that did not consist entirely of comments and/or whitespace.

Table 3.1: Table showing the final count of LOC (Lines Of Code) for the four benchmarks

Language	Final size in LOC
C	256
C++ with TBB	250
C#	415
Java	381

The complete source codes for both the C and C++ with TBB benchmarks were provided so only the C# and Java benchmarks were written in this research. C# needed extra code to control the number of tasks run at the same time due to the inability to control the number of threads in the thread pool. The thread pool in Java could control the number of threads used but needed duplicate functions for the serial and parallel versions. this was because the parallel functions had to be in its own class that extended the RecursiveAction class in order to be created as tasks for the thread pool.

Chapter 4

Results

This chapter presents the results of the performance tests for C, C++ with TBB, C# and Java. All data is presented in the form of graphs that shows either the relation between speedup and the number of threads or speedup with 48 threads on different granularity.

4.1 Execution Time Performance With Different Number of Threads

The data from the execution time performance tests with different number of threads is presented in graphs 4.1.1 to 4.1.4 that shows the relation between speedup and number of threads used. The performance of the benchmarks are then compared in graphs 4.2 and 4.3, the first one shows the relation between speedup and number of threads and the second one shows the relation between execution time and number of threads.

4.1.1 C

The speedup achieved for C is almost linear to the number of threads up to 24 threads. After that the additional speedup becomes zero regardless of how many threads are used. The abrupt stop in speedup improvement may depend on the fact that the execution time with 24 threads was around 1 second. Execution times around 1 second and less are not reliable as the additional speedup gained by more threads may be canceled out by the increased overhead. Because of that a bigger data-set is needed in order to test the real speedup of 24 threads and above. The speedup is shown in graph 4.1.1.

4.1.2 C++ TBB

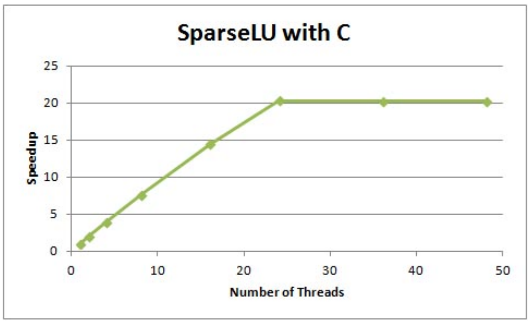
Graph 4.1.2 shows that the achieved speedup for C++ with TBB is almost proportional to the number of threads used. A small decrease in the additional speedup achieved is noticed when the number of threads increase. Worth noting is that C++ with TBB has the highest speedup of all the benchmarks.

4.1.3 C#

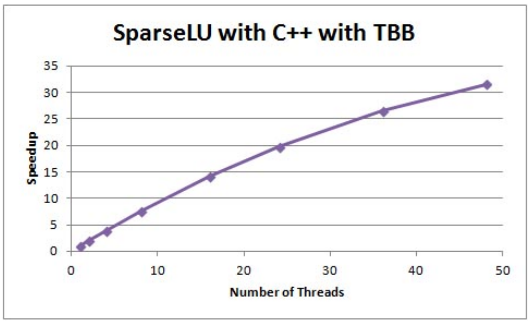
The achieved speedup for C# is not proportional to the number of threads used as seen in graph 4.1.3. In the beginning the additional speedup for C# increases at a steady rate but as the number of threads increase, the additional speedup decreases until it reaches its peak at 36 threads and after that starts to decline in performance.

4.1.4 Java

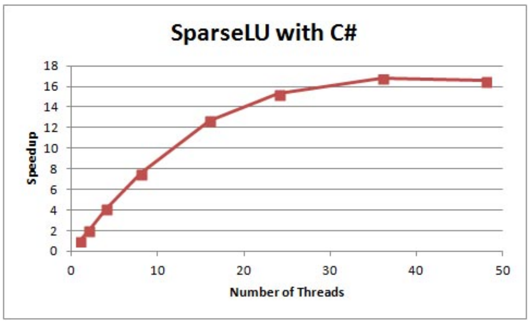
The achieved speedup for Java increases steadily with a slight curve and is almost proportional to the number of threads as seen in 4.1.4. After its peak at 36 threads it suddenly starts to decline. At 36 threads it has an execution time around 1 second and the same phenomena occurs as it did with C. Therefore Java also needs a bigger data-set in order to test the real speedup of 36 threads and above.



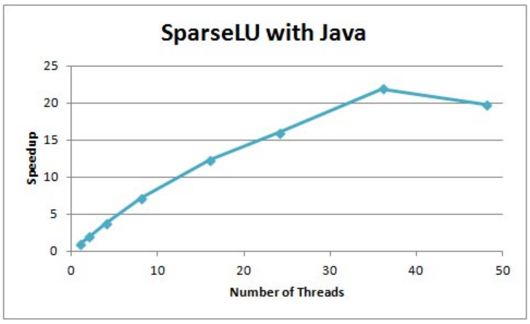
4.1.1: SparseLU written in C



4.1.2: SparseLU written in C++ with TBB



4.1.3: SparseLU written in C#



4.1.4: SparseLU written in Java

Figure 4.1: Graphs showing the achieved speedup with different number of threads on the SparseLU benchmark

4.1.5 Speedup comparison

The converted data from C, C++ with TBB, C# and Java are put together in 4.2 for comparison of their achieved speedup. Remember that the speedup of each of the programs are relative to their single threaded performance and doesn't necessarily mean that their execution times are the same.

The graph show that C++ with TBB has the greatest speedup overall. Before C stops improving its speedup it actually has better performance than C++ with TBB. C# keeps up with the other two in the beginning but as the number of threads increases it begins to drop to finally reach its peak before it starts to decline. Java also keeps up with C and C++ with TBB at the beginning but as the number of threads increases the additional speedup decreases until it reaches its peak at 36 threads then starts to decline in performance.

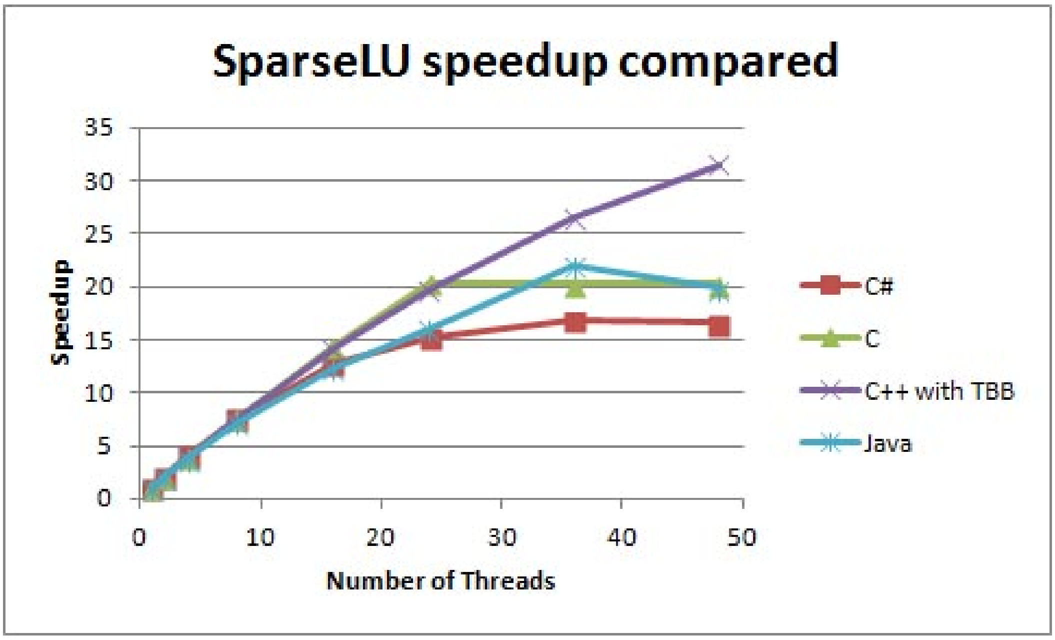


Figure 4.2: A graph comparing the achieved speedup between C, C++ with TBB, C# and Java

4.1.6 Execution time comparison

The earlier graphs in 4.1 showed the relative speedup in contrast to their serial execution times as well as all the runs compared to each other. In graph 4.3 the actual execution time for each of the programs are compared to each other.

The graph shows that the C# benchmark has the by far slowest execution time and stops to improve after 36 threads. The C++ with TBB and C benchmarks improves their execution times really well at the beginning. As the number of threads increase, the C++ with TBB program execution time doesn't improve as it did in the beginning. The C program has a steady improvement of its execution time until it hits 1 second with 24 threads and can't improve further due to the increased overhead canceling out the gains of more threads. The Java benchmark keeps up with the C benchmark and comes down to 1 second in execution time at 36 threads and stops improving.

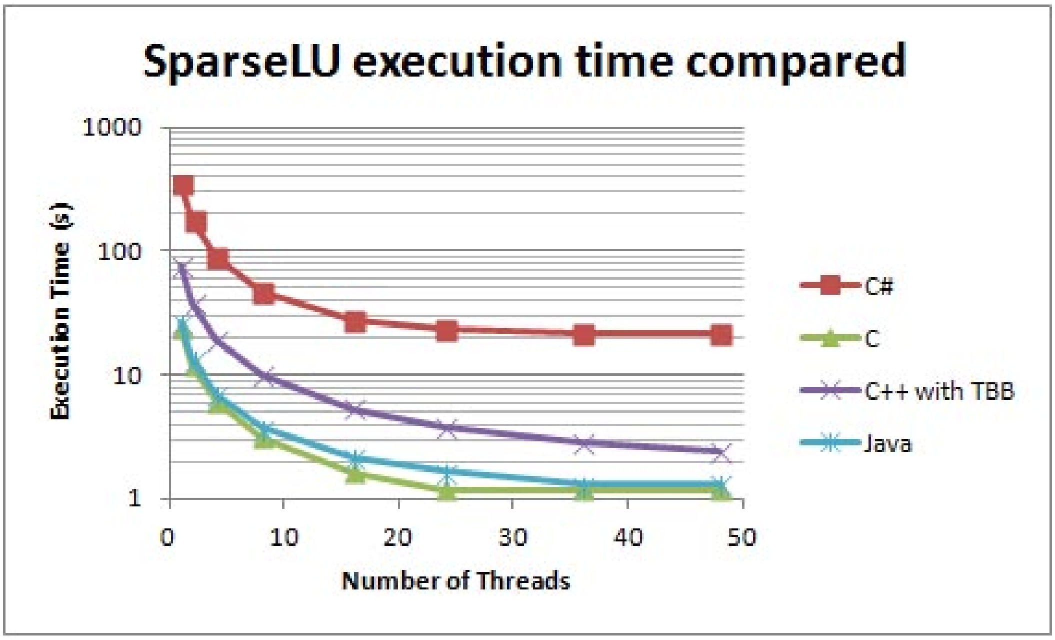


Figure 4.3: A graph comparing execution time for different number of threads between C, C++ with TBB, C# and Java

4.2 Granularity Performance

The data from the execution time performance with different granularity is presented in graphs 4.4.1 to 4.4.4 that shows the relation between speedup with 48 threads on different granularity. The performance of the benchmarks are then compared in graphs 4.5 and 4.6, the first one shows the relation between speedup with 48 threads on different granularity and the second one shows the relation between execution time with 48 threads on different granularity. The different granularity sizes are shown in table 4.1.

4.2.1 C

The speedup for C starts slow but at granularity 6 it suddenly starts to increase drastically until it reaches its peak around granularity 11. Soon after C peaks it

Table 4.1: Table of all different granularity sizes used in the tests

Granularity	Matrix Size	Submatrix Size
1	1	5000
2	2	2500
3	4	1250
4	5	1000
5	8	625
6	10	500
7	20	250
8	25	200
9	40	125
10	50	100
11	100	50
12	125	40
13	200	25
14	250	20
15	500	10

abruptly drops in speedup as it gets too fine grained and then has close to no speedup at all. All this is shown in graph 4.4.1.

4.2.2 C++ TBB

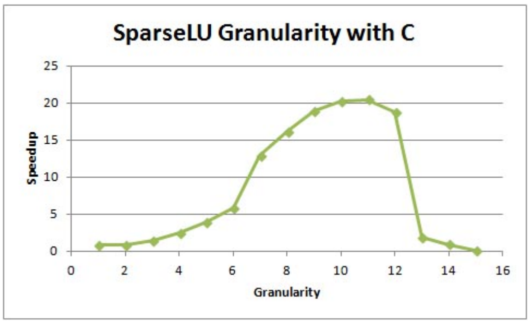
C++ with TBB gets little speedup in the beginning when the granularity is coarse grained. As the granularity gets finer the speedup increases at a steady rate until it peaks around granularity 11 and starts to decline slowly. Unfortunately the data is not complete as the benchmark couldn't execute with too fine grained tasks. That is why the graph stops abruptly at granularity 12. This is shown in graph 4.4.2.

4.2.3 C#

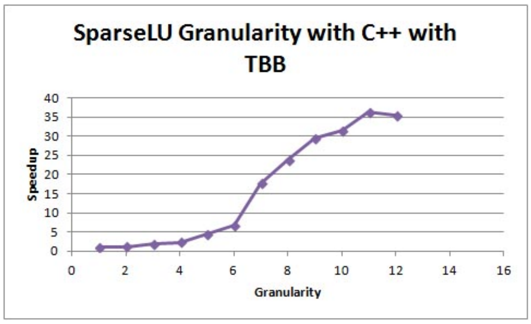
The C# benchmark starts out with little speedup but unlike C, C++ with TBB and Java the speedup increases more quickly and soon reaches its peak. It then drops rapidly as seen in graph 4.4.3. Unfortunately the data is not complete as the benchmark couldn't execute with too fine grained tasks due to an unexpected error. That is why the graph stops abruptly at granularity 10.

4.2.4 Java

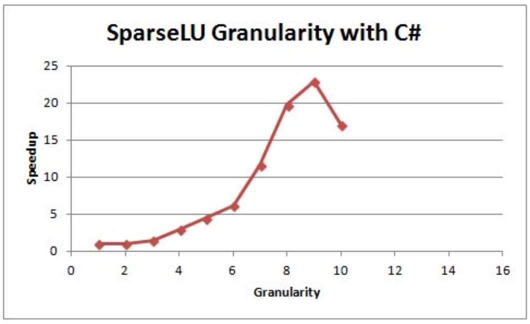
The Java benchmark starts with little speedup but after granularity 8 it abruptly gains considerably in speedup and reaches a plateau. After that it reaches its peak plateau and finally the speedup declines very fast. This is interesting as the curve isn't smooth as the three other benchmark programs. All this is illustrated in graph 4.4.4.



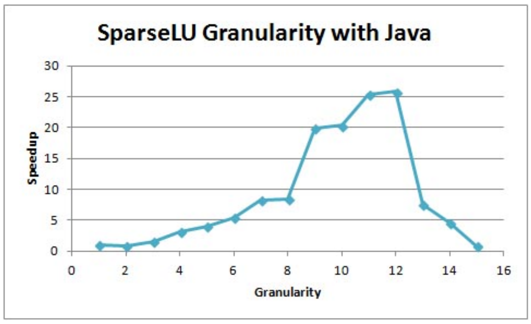
4.4.1: SparseLU written in C



4.4.2: SparseLU written in C++ with TBB



4.4.3: SparseLU written in C#



4.4.4: SparseLU written in Java

Figure 4.4: Graphs showing the achieved speedup with 48 threads on different granularity on the SparseLU benchmark

4.2.5 Speedup comparison

The converted data from C, C++ with TBB, C# and Java are put together in 4.5 for comparison of their achieved speedups on different granularity. The speedup is relative for each of the programs compared with their single threaded performance and doesn't necessarily mean that their execution times are the same.

Shown in graph 4.5 C++ with TBB has the best and quickest speedup for all different granularity until it stops due to incomplete data. The data indicates that C++ with TBB would have continued to have the best speedup throughout the entire granularity tests. The graph also shows that C# has greater speedup than both C and Java in a certain spectrum, after which it drops and then ends due to incomplete data. C has a steady increase in speedup as through the granularity but then drops quickly when it gets too fine grained. Java has a rather uneven speedup curve as it leaps at a certain granularities. It even exceeds the speedup of C but then drops as quickly as C at the same Granularity.

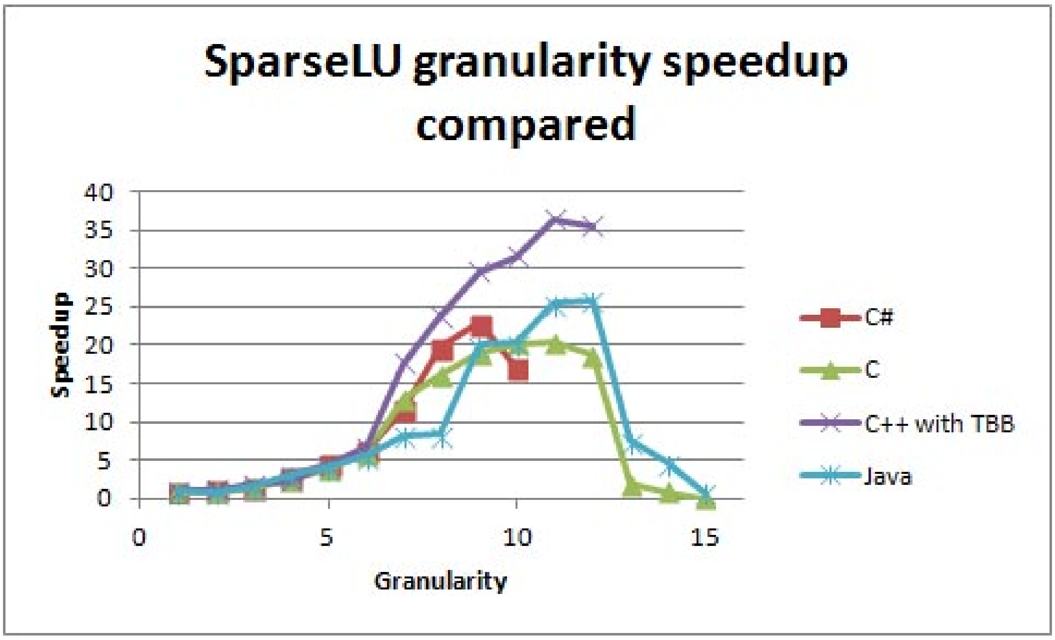


Figure 4.5: A graph comparing the achieved speedup between C, C++ with TBB, C# and Java

4.2.6 Execution time comparison

The earlier graphs in 4.4 showed the relative speedup in contrast to their serial execution times as well as all the runs compared to each other. In graph 4.6 the actual execution time for each of the programs are compared to each other.

The C# benchmark has the by far slowest execution time, even as it speeds up when the granularity gets more fine grained. Unfortunately the data for C# is incomplete and therefore can't show how it performs with very fine grained tasks. The same goes for C++ with TBB as it also has incomplete data. The interesting thing here is that C, C++ with TBB and Java all follow each other's execution time performance in the beginning. As the granularity gets finer the C benchmark execution time decreases faster than C++ with TBB and Java which still follow each other until Java finally gets ahead of C++ with TBB and catches up to C and execute around 1 second. Here the data for C++ with TBB stops but the data for C and Java shows that there is a drastic increase in execution time as the granularity gets too fine grained. Here Java performs better than C but still takes longer to execute.

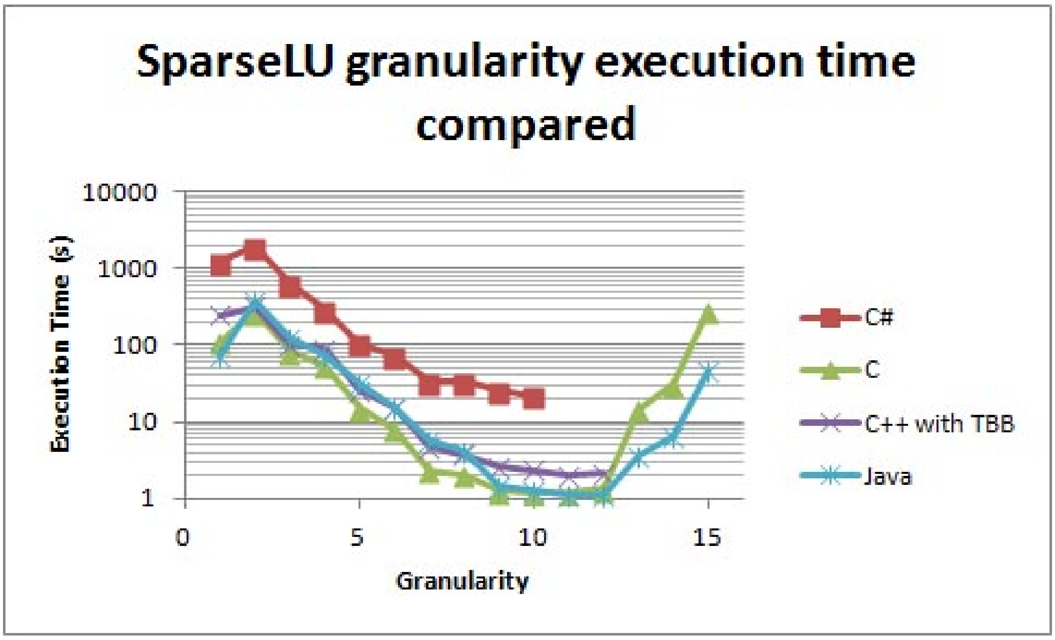


Figure 4.6: A graph comparing execution time on different granularity between C, C++ with TBB, C# and Java

Chapter 5

Discussion and Summary

This chapter takes up discussion about the results and a summary of the research.

5.1 Discussion

The discussion is split in two parts. The first part is about the tests with different number of threads. The second part is about the tests with 48 threads run on different granularity. Both parts focus on the two factors speedup performance and execution time performance as well as some more overall discussion.

Both Java and C# uses garbage collection but C and C++ don't. This is important to take into account when comparing low-level languages to mid- or high-level languages and when comparing mid- or high-level languages with themselves as the garbage collection may differ. Unfortunately the tests in this research couldn't measure and compare the garbage collection on Java and C#.

5.1.1 Performance with number of threads

The discussion on performance with number of threads will mainly be around the two combined graphs 4.2 and 4.3.

Speedup

The programming language C++ using the TBB API has shown the greatest speedup with just a slight loss after larger number of threads. This is most likely due to the overhead of each task the TBB model has to use to execute them compared to the programming language C where the increasing number of threads doesn't affect its linear speedup.

A peculiar result produced by the programming language C with OpenMP is that after a certain amount of threads available the linear speedup stops flat. This is because at 24 threads it has reached an execution time of 1 second and because of this the increased overhead is canceling out the gains of more threads.

The same goes for the Java benchmark using fork/join. Although it is a bit slower than C it reaches the same speedup and an execution time of 1 second at 36 threads. It then drops a little in speedup due to the increased overhead being bigger than the gains of more threads.

In order to test the true potential of C and Java a bigger data-set is needed because measuring execution times at 1 second and bellow is inaccurate. A bigger data-set would take more time to execute giving more accurate measuring of the speedup gains at 24 threads and above for C and Java.

C# didn't perform as well as the three other in terms of speedup. It may depend on the problem limiting the number of threads running in parallel forcing a solution where the tasks were limited instead. This had an unknown impact on the performance and in order to fairly test the thread pool in C# another approach is needed.

Execution Time

In terms of execution time the OpenMP parallel model for the programming language C dominated the benchmarking tests with the fastest execution time while C# performed the worst.

Comparing the low level programming languages C and C++, C dominated the tests due to the fact that it made use of its pointer arithmetic and with no dependencies each task could handle the matrix without moving any of the sub matrices. For the programming language C++ the TBB API had much extra code compared to C in order to extract each task in a TBB model making it a bit slower than C. The overall speedup is most likely affected by the overhead each task is given by the TBB model.

Comparing the mid-level programming languages Java and C#, Java performed the best. It even performed better than the low level programming language C++. Java has a JIT (Just In Time) compiler which makes optimizations before it is executed on a specific machine. This makes it much faster than C# which also handles a virtual machine in between like Java but with no JIT. This leaves Java in the same result category as C as it also stops improving after a certain amount of threads due to the fast execution time.

Both C and Java performed really well on the test, C being a little faster than Java. The surprise is that Java performed so well when the hypothesis stated that the opposite was expected.

5.1.2 Performance with granularity

The discussion on performance with granularity will mainly be around the two combined graphs 4.5 and 4.6 and the granularity table 4.1.

The granularity is going from coarse grained with a few very large tasks to fine grained with a lot of smaller tasks while keeping the size of the data-set. By testing with this configuration the graphs show at what granularity each benchmark performs the best.

Speedup

All four benchmarks performed almost identically on the most coarse grained granularity from 1 to 6 but after that they spread out in different directions.

C++ with TBB had the greatest speedup of all on the granularity tests. The speedup started to increase at a rapid rate from granularity 6 and peaked around 36 times speedup before it started to decline a little. Due to an unexpected error C++ with TBB couldn't execute with too fine grained tasks so the data is incomplete. By comparing with C and Java it is likely that C++ with TBB would have performed similar and started to drop drastically after granularity 12.

C# achieved greater speedup than C and was the first of the benchmarks to peak. C# was also the first to start declining and did so rather early. Due to an unexpected error C# couldn't execute with tasks that was smaller than granularity 10.

C had a smooth speedup curve that peaked at around granularity 10 and then decline slowly. At granularity 12 a huge drop in speedup occurs and after that the speedup is almost zero for the most fine grained tasks. This clearly shows that C perform better the more fine grained the tasks but if they get too fine grained all the speedup is lost.

Java were slower than the rest to gain considerably in speedup and until granularity 8 it only had around 7 times speedup. Between granularity 8 and 9 the speedup increased from 7 to 20 and another smaller increase in speedup up to 25 between granularity 10 and 11. After that Java had a similar huge drop as C but it stopped at 7 times speedup and from there declined down to zero speedup at granularity 15.

Execution Time

As mentioned earlier both C# and C++ with TBB had incomplete data which made it impossible to know how they would perform with very fine grained tasks.

C# had an execution time of around 1000 seconds which was the absolute worst of all four benchmarks. It improved as the granularity got more fine grained but the execution time never got close to the three other benchmarks.

C, C++ with TBB and Java all had similar execution times from granularity 1 to 8, C being a little faster while C++ with TBB and Java followed each other. After granularity 8 Java jumped down to C and they both executed around 1 second until granularity 12. After that the execution time for C increased rapidly while Java had a slower increase making Java better suited for finer granularity.

5.2 Summary

The programming language C as well as Java dominated the tests in terms of execution time while C++ and C dominated the overall speedup gained. The granularity tests showed that Java could handle small tasks while still keeping its overall speedup in comparison to the other three languages. In all of the tests the programming language C# performed the worst in terms of execution time, overall speedup and the amount of granularity the language could handle before the overhead of the tasks took over.

According to the results the most effective programming language in parallel programming is C with the OpenMP API and Java with the fork/join method. C and Java gave the fastest execution time of all languages and handled granularity in a very efficient way without wasting execution time or overall speed up, especially Java that could handle very small granularity for each task and thread.

Chapter 6

Recommendations and Future Work

6.1 Recommendations

Further studies within this area have great potential, the comparison between programming languages and their respective parallel programming models would prove beneficial to future software development especially when hardware development today are focused in developing multi core processors rather than faster single core processors.

This research could also be beneficial for institutes researching in developing parallel programming APIs. This research gives an overview of how each programming language performs from a parallel point of view, either lacking in support and performance or exceling in these areas.

To grip a better understanding and better overview of the research's benchmark tests larger data sets can be introduced for C and Java. Handling bigger data sets will give a longer execution time and thus easier to measure speedup for even more threads, if this is done the C programming language will not plan out in the graph but continue to improve and the same thing for Java. Another improvement to the benchmark would have to be done for the C# version to get a more efficient code and a more fair comparison to other programming languages.

6.2 Future Work

Future studies specifically following this projects steps would be expanding the amount of different types of benchmarks. The benchmark that was used in this current project was completely independent which showed a simple overhead when comparing the different parallel models. Future benchmarks would include a light and heavy dependent parallel regions or benchmarks where scalability is not limitless e.g. N-Queens problem.

Constant development within parallel programming has led to more parallel programming APIs. Some of these APIs are developed for software developers to make it more easy and efficient to program while other APIs are developed for operating efficiency which would be very interesting to look at within this kind of study.

Another interesting area to study would be how the virtual machines of the compilers of higher level languages affect efficiency when it comes to parallel programming.

Appendix A

Scripts

A.1 Script for performance runs

```
#!/bin/bash
# the script will run 5 times each for
# the following number of threads:
# 1, 2, 4, 8, 16, 24, 36, 48

# writes the time and date for this test run
date > C_output.txt

# Compile the C version of sparseLU
gcc -fopenmp -O3 -o sparselu_C sparselu.c

number=0
threads=0
# Outer loop for changing number of threads
while [ $number -lt 8 ]; do
  case $number in
    0 ) threads=1 ;;
    1 ) threads=2 ;;
    2 ) threads=4 ;;
    3 ) threads=8 ;;
    4 ) threads=16 ;;
    5 ) threads=24 ;;
    6 ) threads=36 ;;
    7 ) threads=48 ;;
    * ) echo "Script failed! Aborting!" >> C_output.txt ; exit 1
  esac
  echo "" >> C_output.txt
```

```
  echo "Running with "$threads" threads" >> C_output.txt
  # Inner loop that executes the benchmark 5 times
  numTestRuns=0
  while [ $numTestRuns -lt 5 ]; do
    # Uncomment (remove #) the line of code that executes
    # the desired benchmark to get the script to run it.
    #./sparselu_C $threads >> C_output.txt
    #./sparselu.icc.tbb -w $threads  >> C_tbb_output.txt
    #mono sparselu.exe $threads >> C_sharp_output.txt
    #java Main $threads >> Java_output.txt
    numTestRuns=$((numTestRuns + 1))
  done
  number=$((number + 1))
done
exit 0
```

A.2 Script for granularity runs

```
#!/bin/bash
# the script will run 5 times each for
# the following different sizes on the
# matrix and submatrix(with 48 threads):
# (1,5000)(2,2500)(4,1250)(5,1000)(8,625)
# (10,500)(20,250)(25,200)(40,125)(50,100)
# (100,50)(125,40)(200,25)(250,20)(500,10)

date >> C_TBB_matrix_output.txt
threads=48
number=0
matrix=0
submatrix=0
# Outer loop for changing sizes of the matrices
while [ $number -lt 15 ]; do
  case $number in
    0 ) matrix=1
       submatrix=5000 ;;
    1 ) matrix=2
       submatrix=2500 ;;
    2 ) matrix=4
       submatrix=1250 ;;
    3 ) matrix=5
       submatrix=1000 ;;
```

```
      4 ) matrix=8
        submatrix=625 ;;
      5 ) matrix=10
        submatrix=500 ;;
      6 ) matrix=20
        submatrix=250 ;;
      7 ) matrix=25
        submatrix=200 ;;
      8 ) matrix=40
        submatrix=125 ;;
      9 ) matrix=50
        submatrix=100 ;;
      10 ) matrix=100
        submatrix=50 ;;
      11 ) matrix=125
        submatrix=40 ;;
      12 ) matrix=200
        submatrix=25 ;;
      13 ) matrix=250
        submatrix=20 ;;
      14 ) matrix=500
        submatrix=10 ;;
      * ) echo "Script failed! Aborting!" >> C_TBB_matrix_output.txt ; exit 1
    esac

    # Inner loop that executes the benchmark 5 times
    numTestRuns=0
    while [ $numTestRuns -lt 5 ]; do
      # Uncomment (remove #) the line of code that executes
      # the desired benchmark to get the script to run it.
      #./sparselu $threads $matrix $submatrix >> C_matrix_output.txt
      #./sparselu.icc.tbb -w $threads -n $matrix -m $submatrix >> C_TBB_matrix_output.txt
      #mono sparselu.exe $threads $matrix $submatrix >> C_sharp_matrix_output.txt
      #java Main $threads $matrix $submatrix >> Java_matrix_output.txt
      numTestRuns=$((numTestRuns + 1))
    done
    number=$((number + 1))
done
exit 0
```

Appendix B

Source Code

B.1 Source code for C

```
/* ****************************************************************************** */
/*  This program is part of the Barcelona OpenMP Tasks Suite */
/*  Copyright (C) 2009 Barcelona Supercomputing Center - Centro Nacional de Supercomputacion  */
/*  Copyright (C) 2009 Universitat Politecnica de Catalunya */
/*                                                                                */
/*  This program is free software; you can redistribute it and/or modify */
/*  it under the terms of the GNU General Public License as published by */
/*  the Free Software Foundation; either version 2 of the License, or */
/*  (at your option) any later version. */
/*                                                                                */
/*  This program is distributed in the hope that it will be useful, */
/*  but WITHOUT ANY WARRANTY; without even the implied warranty of */
```

```
/*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the          */
/*  GNU General Public License for more details.                          */
/*                                                                        */
/*  You should have received a copy of the GNU General Public License     */
/*  along with this program; if not, write to the Free Software           */
/*  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA  */
/* ************************************************************************ */

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <libgen.h>
#include <sys/time.h>
#include <omp.h>

#define EPSILON 1.0E-6

double start_time, end_time, start_seq, end_seq;

#define MAXWORKERS 24    /* maximum number of workers */
#define MAXMINMATR 1000    /* maximum number of workers */
#define MAXMAXMATR 1000    /* maximum number of workers */


int numWorkers;

unsigned int bots_arg_size = 50;
unsigned int bots_arg_size_1 = 100;

#define TRUE 1
#define FALSE 0

#define BOTS_RESULT_SUCCESSFUL 1
```

```
#define BOTS_RESULT_UNSUCCESSFUL 0

/* ***********************************************************************
 * checkmat:
 *********************************************************************** */
int checkmat (float *M, float *N)
{
    int i, j;
    float r_err;

    for (i = 0; i < bots_arg_size_1; i++)
    {
        for (j = 0; j < bots_arg_size_1; j++)
        {
            r_err = M[i*bots_arg_size_1+j] - N[i*
                bots_arg_size_1+j];
            if (r_err < 0.0 ) r_err = -r_err;
            r_err = r_err / M[i*bots_arg_size_1+j];
            if(r_err > EPSILON)
            {
                fprintf(stderr,"Checking failure: A[%d][%d]=%f  
                   B[%d][%d]=%f; Relative Error=%f\n",
                        i,j, M[i*bots_arg_size_1+j], i,j, N[i*
                           bots_arg_size_1+j], r_err);
                return FALSE;
            }
        }
    }
    return TRUE;
}
/* ***********************************************************************
 * genmat:
 *********************************************************************** */
void genmat (float *M[])
{
    int null_entry, init_val, i, j, ii, jj;
    float *p;

```

```
    init_val = 1325;

    /* generating the structure */
    for (ii=0; ii < bots_arg_size; ii++)
    {
       for (jj=0; jj < bots_arg_size; jj++)
       {
          /* computing null entries */
          null_entry=FALSE;
          if ((ii<jj) && (ii%3 !=0)) null_entry = TRUE;
          if ((ii>jj) && (jj%3 !=0)) null_entry = TRUE;
    if (ii%2==1) null_entry = TRUE;
    if (jj%2==1) null_entry = TRUE;
    if (ii==jj) null_entry = FALSE;
    if (ii==jj-1) null_entry = FALSE;
          if (ii-1 == jj) null_entry = FALSE;
          /* allocating matrix */
          if (null_entry == FALSE){
             M[ii*bots_arg_size+jj] = (float *) malloc(
                bots_arg_size_1*bots_arg_size_1*sizeof(float)
                );
       if ((M[ii*bots_arg_size+jj] == NULL))
             {
                fprintf(stderr,"Error: Out of memory\n");
                exit(101);
             }
             /* initializing matrix */
             p = M[ii*bots_arg_size+jj];
             for (i = 0; i < bots_arg_size_1; i++)
             {
                for (j = 0; j < bots_arg_size_1; j++)
                {
               init_val = (3125 * init_val) % 65536;
                     (*p) = (float)((init_val - 32768.0) /
                        16384.0);
                     p++;
                }
             }
          }
          else
          {
             M[ii*bots_arg_size+jj] = NULL;
          }
       }
```

```
    }
}
/***********************************************************************
 * print_structure:
 **********************************************************************/
void print_structure(char *name, float *M[])
{
    int ii, jj;
    fprintf(stderr,"Structure for matrix %s @ 0x%p\n",name, M);
    for (ii = 0; ii < bots_arg_size; ii++) {
      for (jj = 0; jj < bots_arg_size; jj++) {
         if (M[ii*bots_arg_size+jj]!=NULL) {fprintf(stderr,"x");}
         else fprintf(stderr," ");
      }
      fprintf(stderr,"\n");
    }
    fprintf(stderr,"\n");
}
/***********************************************************************
 * allocate_clean_block:
 **********************************************************************/
float * allocate_clean_block()
{
  int i,j;
  float *p, *q;

  p = (float *) malloc(bots_arg_size_1*bots_arg_size_1*sizeof(float));
  q=p;
  if (p!=NULL){
     for (i = 0; i < bots_arg_size_1; i++)
        for (j = 0; j < bots_arg_size_1; j++){(*p)=0.0; p++;}

  }
  else
```

```
  {
      fprintf(stderr,"Error: Out of memory\n");
      exit (101);
  }
  return (q);
}

/*
   ***********************************************************************
 * lu0:
 ***********************************************************************
   */
void lu0(float *diag)
{
   int i, j, k;

   for (k=0; k<bots_arg_size_1; k++)
      for (i=k+1; i<bots_arg_size_1; i++)
      {
         diag[i*bots_arg_size_1+k] = diag[i*bots_arg_size_1+
            k] / diag[k*bots_arg_size_1+k];
         for (j=k+1; j<bots_arg_size_1; j++)
            diag[i*bots_arg_size_1+j] = diag[i*
               bots_arg_size_1+j] - diag[i*bots_arg_size_1+k
               ] * diag[k*bots_arg_size_1+j];
      }
}

/*
   ***********************************************************************
 * bdiv:
 ***********************************************************************
   */
void bdiv(float *diag, float *row)
{
   int i, j, k;
   for (i=0; i<bots_arg_size_1; i++)
      for (k=0; k<bots_arg_size_1; k++)
      {
         row[i*bots_arg_size_1+k] = row[i*bots_arg_size_1+k]
             / diag[k*bots_arg_size_1+k];
         for (j=k+1; j<bots_arg_size_1; j++)
```

```c
                row[i*bots_arg_size_1+j] = row[i*bots_arg_size_1+j] - row[i*bots_arg_size_1+k]*diag[k*bots_arg_size_1+j];
      }
}
/***********************************************************************
 * bmod:
 ***********************************************************************/
void bmod(float *row, float *col, float *inner)
{
   int i, j, k;
   for (i=0; i<bots_arg_size_1; i++)
      for (j=0; j<bots_arg_size_1; j++)
         for (k=0; k<bots_arg_size_1; k++)
            inner[i*bots_arg_size_1+j] = inner[i*bots_arg_size_1+j] - row[i*bots_arg_size_1+k]*col[k*bots_arg_size_1+j];
}
/***********************************************************************
 * fwd:
 ***********************************************************************/
void fwd(float *diag, float *col)
{
   int i, j, k;
   for (j=0; j<bots_arg_size_1; j++)
      for (k=0; k<bots_arg_size_1; k++)
         for (i=k+1; i<bots_arg_size_1; i++)
            col[i*bots_arg_size_1+j] = col[i*bots_arg_size_1+j] - diag[i*bots_arg_size_1+k]*col[k*bots_arg_size_1+j];
}


void sparselu_init (float ***pBENCH, char *pass)
{
   *pBENCH = (float **) malloc(bots_arg_size*bots_arg_size*sizeof(float *));
   genmat(*pBENCH);
```

```
    //print_structure(pass, *pBENCH);
}

void sparselu_par_call(float **BENCH)
{
    int ii, jj, kk;

    //fprintf(stderr,"Computing SparseLU Factorization (%dx%d
        matrix with %dx%d blocks) ", bots_arg_size,
        bots_arg_size,bots_arg_size_1,bots_arg_size_1);

    start_time = omp_get_wtime();

    #pragma omp parallel
    #pragma omp single nowait
    #pragma omp task untied
    for (kk=0; kk<bots_arg_size; kk++)
    {
        lu0(BENCH[kk*bots_arg_size+kk]);
        for (jj=kk+1; jj<bots_arg_size; jj++)
            if (BENCH[kk*bots_arg_size+jj] != NULL)
                #pragma omp task untied firstprivate(kk, jj)
                    shared(BENCH)
            {
                fwd(BENCH[kk*bots_arg_size+kk], BENCH[kk*
                    bots_arg_size+jj]);
            }
        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
                #pragma omp task untied firstprivate(kk, ii)
                    shared(BENCH)
            {
                bdiv (BENCH[kk*bots_arg_size+kk], BENCH[ii*
                    bots_arg_size+kk]);
            }

        #pragma omp taskwait

        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
                for (jj=kk+1; jj<bots_arg_size; jj++)
                    if (BENCH[kk*bots_arg_size+jj] != NULL)
                    #pragma omp task untied firstprivate(kk, jj,
                        ii) shared(BENCH)
```

```
                    {
                            if (BENCH[ii*bots_arg_size+jj]==NULL)
                               BENCH[ii*bots_arg_size+jj] =
                               allocate_clean_block();
                            bmod(BENCH[ii*bots_arg_size+kk], BENCH[
                               kk*bots_arg_size+jj], BENCH[ii*
                               bots_arg_size+jj]);
                    }

                #pragma omp taskwait
    }
    end_time = omp_get_wtime();
    //fprintf(stderr," completed!\n");
}


void sparselu_seq_call(float **BENCH)
{
    int ii, jj, kk;

    start_seq = omp_get_wtime();

    for (kk=0; kk<bots_arg_size; kk++)
    {
        lu0(BENCH[kk*bots_arg_size+kk]);
        for (jj=kk+1; jj<bots_arg_size; jj++)
            if (BENCH[kk*bots_arg_size+jj] != NULL)
            {
                fwd(BENCH[kk*bots_arg_size+kk], BENCH[kk*
                    bots_arg_size+jj]);
            }
        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
            {
                bdiv (BENCH[kk*bots_arg_size+kk], BENCH[ii*
                    bots_arg_size+kk]);
            }
        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
                for (jj=kk+1; jj<bots_arg_size; jj++)
                    if (BENCH[kk*bots_arg_size+jj] != NULL)
                    {
                            if (BENCH[ii*bots_arg_size+jj]==NULL)
                               BENCH[ii*bots_arg_size+jj] =
```

```
                                allocate_clean_block();
                        bmod(BENCH[ii*bots_arg_size+kk], BENCH[
                            kk*bots_arg_size+jj], BENCH[ii*
                            bots_arg_size+jj]);
                }

    }
    end_seq = omp_get_wtime();
}

void sparselu_fini (float **BENCH, char *pass)
{
    //print_structure(pass, BENCH);
}

int sparselu_check(float **SEQ, float **BENCH)
{
    int ii,jj,ok=1;

    for (ii=0; ((ii<bots_arg_size) && ok); ii++)
    {
        for (jj=0; ((jj<bots_arg_size) && ok); jj++)
        {
            if ((SEQ[ii*bots_arg_size+jj] == NULL) && (BENCH[ii
                *bots_arg_size+jj] != NULL)) ok = FALSE;
            if ((SEQ[ii*bots_arg_size+jj] != NULL) && (BENCH[ii
                *bots_arg_size+jj] == NULL)) ok = FALSE;
            if ((SEQ[ii*bots_arg_size+jj] != NULL) && (BENCH[ii
                *bots_arg_size+jj] != NULL))
                ok = checkmat(SEQ[ii*bots_arg_size+jj], BENCH[ii
                    *bots_arg_size+jj]);
        }
    }
    if (ok) return BOTS_RESULT_SUCCESSFUL;
    else return BOTS_RESULT_UNSUCCESSFUL;
}


int main ( int argc, char *argv[])
{
      float **SEQ,**BENCH;

    numWorkers = (argc > 1)? atoi(argv[1]) : MAXWORKERS;
    bots_arg_size = (argc > 2)? atoi(argv[2]) : MAXMINMATR;
```

```
    bots_arg_size_1 = (argc > 3)? atoi(argv[3]) : MAXMAXMATR;
    if (numWorkers > MAXWORKERS) numWorkers = MAXWORKERS;

    omp_set_num_threads(numWorkers);

      //fprintf(stderr,"Startar parallel version...\n");
      sparselu_init(&BENCH,"benchmark");
      sparselu_par_call(BENCH);
      sparselu_fini(BENCH,"benchmark");

      //fprintf(stderr,"Startar seriell version...\n");

      sparselu_init(&SEQ,"serial");
      sparselu_seq_call(SEQ);
      sparselu_fini(SEQ,"serial");

      //fprintf(stderr,"Testar om Parallel och Seriell version stÃ¤mmer med varandra...\n");
      sparselu_check(SEQ,BENCH);
    printf("Completed! Sequential took %g seconds. Parallel took %g seconds.\n", end_seq - start_seq, end_time - start_time);
}
```

B.2 Source code for C++ with TBB

```
/* ****************************************************************************** */
/*  This program is part of the Barcelona OpenMP Tasks Suite                      */
/*  Copyright (C) 2009 Barcelona Supercomputing Center - Centro Nacional de Supercomputacion  */
/*  Copyright (C) 2009 Universitat Politecnica de Catalunya                       */
/*                                                                                */
/*  This program is free software; you can redistribute it and/or modify          */
/*  it under the terms of the GNU General Public License as published by          */
```

```
/*  the Free Software Foundation; either version 2 of the License, or             */
/*  (at your option) any later version.                                            */
/*                                                                                 */
/*  This program is distributed in the hope that it will be useful,                */
/*  but WITHOUT ANY WARRANTY; without even the implied warranty of                 */
/*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the                  */
/*  GNU General Public License for more details.                                   */
/*                                                                                 */
/*  You should have received a copy of the GNU General Public License              */
/*  along with this program; if not, write to the Free Software                    */
/*  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA */
/* ******************************************************************************* */

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <libgen.h>
#include <sys/time.h>

#define EPSILON 1.0E-6

//double start_time, end_time, start_seq, end_seq;

#define MAXWORKERS 24    /* maximum number of workers */

//int numWorkers;

```

```
static const int num_threads = 48;
std::thread t[num_threads];

struct input {
  float **benchinp1;
  float **benchinp2;
  float **benchinp3;
};

unsigned int bots_arg_size = 50;
unsigned int bots_arg_size_1 = 100;

#define TRUE 1
#define FALSE 0

#define BOTS_RESULT_SUCCESSFUL 1
#define BOTS_RESULT_UNSUCCESSFUL 0

/* ********************************************************************
 * checkmat:
 ******************************************************************** */
int checkmat (float *M, float *N)
{
    int i, j;
    float r_err;

    for (i = 0; i < bots_arg_size_1; i++)
    {
        for (j = 0; j < bots_arg_size_1; j++)
        {
            r_err = M[i*bots_arg_size_1+j] - N[i*bots_arg_size_1+j];
            if ( r_err < 0.0 ) r_err = -r_err;
            r_err = r_err / M[i*bots_arg_size_1+j];
            if(r_err > EPSILON)
            {
                fprintf(stderr,"Checking failure: A[%d][%d]=%f  B[%d][%d]=%f; Relative Error=%f\n",
                        i,j, M[i*bots_arg_size_1+j], i,j, N[i*bots_arg_size_1+j], r_err);
                return FALSE;
```

```
            }
        }
    }
    return TRUE;
}
/*
    ************************************************************************

 * genmat:
 **********************************************************************
     */
void genmat (float *M[])
{
    int null_entry, init_val, i, j, ii, jj;
    float *p;

    init_val = 1325;

    /* generating the structure */
    for (ii=0; ii < bots_arg_size; ii++)
    {
        for (jj=0; jj < bots_arg_size; jj++)
        {
            /* computing null entries */
            null_entry=FALSE;
            if ((ii<jj) && (ii%3 !=0)) null_entry = TRUE;
            if ((ii>jj) && (jj%3 !=0)) null_entry = TRUE;
    if (ii%2==1) null_entry = TRUE;
    if (jj%2==1) null_entry = TRUE;
    if (ii==jj) null_entry = FALSE;
    if (ii==jj-1) null_entry = FALSE;
            if (ii-1 == jj) null_entry = FALSE;
            /* allocating matrix */
            if (null_entry == FALSE){
               M[ii*bots_arg_size+jj] = (float *) malloc(
                   bots_arg_size_1*bots_arg_size_1*sizeof(float)
                   );
        if ((M[ii*bots_arg_size+jj] == NULL))
                {
                    fprintf(stderr,"Error: Out of memory\n");
                    exit(101);
                }
                /* initializing matrix */
                p = M[ii*bots_arg_size+jj];
```

```
            for (i = 0; i < bots_arg_size_1; i++)
            {
               for (j = 0; j < bots_arg_size_1; j++)
               {
              init_val = (3125 * init_val) % 65536;
                    (*p) = (float)((init_val - 32768.0) / 16384.0);
                    p++;
               }
            }
         }
         else
         {
            M[ii*bots_arg_size+jj] = NULL;
         }
      }
   }
}
/* ************************************************************************
 * print_structure:
 ********************************************************************** */
void print_structure(char *name, float *M[])
{
   int ii, jj;
   //fprintf(stderr,"Structure for matrix %s @ 0x%p\n",name, M);
   for (ii = 0; ii < bots_arg_size; ii++) {
     for (jj = 0; jj < bots_arg_size; jj++) {
        if (M[ii*bots_arg_size+jj]!=NULL) {fprintf(stderr,"x");}
        else fprintf(stderr," ");
     }
     fprintf(stderr,"\n");
   }
   fprintf(stderr,"\n");
}
/* ************************************************************************
 * allocate_clean_block:
```

```
 *************************************************************************
    */
float * allocate_clean_block()
{
  int i,j;
  float *p, *q;

  p = (float *) malloc(bots_arg_size_1*bots_arg_size_1*
     sizeof(float));
  q=p;
  if (p!=NULL){
     for (i = 0; i < bots_arg_size_1; i++)
        for (j = 0; j < bots_arg_size_1; j++){(*p)=0.0; p
           ++;}

  }
  else
  {
      fprintf(stderr,"Error: Out of memory\n");
      exit (101);
  }
  return (q);
}

/*
   *************************************************************************

 * lu0:
 *************************************************************************
    */
void lu0(float *diag)
{
   int i, j, k;

   for (k=0; k<bots_arg_size_1; k++)
      for (i=k+1; i<bots_arg_size_1; i++)
      {
         diag[i*bots_arg_size_1+k] = diag[i*bots_arg_size_1+
            k] / diag[k*bots_arg_size_1+k];
         for (j=k+1; j<bots_arg_size_1; j++)
            diag[i*bots_arg_size_1+j] = diag[i*
               bots_arg_size_1+j] - diag[i*bots_arg_size_1+k
               ] * diag[k*bots_arg_size_1+j];
      }
```

```
}

/***********************************************************************
 * bdiv:
 ********************************************************************/
void bdiv(float *diag, float *row)
{
    int i, j, k;
    for (i=0; i<bots_arg_size_1; i++)
        for (k=0; k<bots_arg_size_1; k++)
        {
            row[i*bots_arg_size_1+k] = row[i*bots_arg_size_1+k]
                / diag[k*bots_arg_size_1+k];
            for (j=k+1; j<bots_arg_size_1; j++)
                row[i*bots_arg_size_1+j] = row[i*bots_arg_size_1
                    +j] - row[i*bots_arg_size_1+k]*diag[k*
                    bots_arg_size_1+j];
        }
}
/***********************************************************************
 * bmod:
 ********************************************************************/
void bmod(float *row, float *col, float *inner)
{
    int i, j, k;
    for (i=0; i<bots_arg_size_1; i++)
        for (j=0; j<bots_arg_size_1; j++)
            for (k=0; k<bots_arg_size_1; k++)
                inner[i*bots_arg_size_1+j] = inner[i*
                    bots_arg_size_1+j] - row[i*bots_arg_size_1+k
                    ]*col[k*bots_arg_size_1+j];
}
/***********************************************************************
 * fwd:
 ********************************************************************/
```

```
void fwd(float *diag, float *col)
{
    int i, j, k;
    for (j=0; j<bots_arg_size_1; j++)
        for (k=0; k<bots_arg_size_1; k++)
            for (i=k+1; i<bots_arg_size_1; i++)
                col[i*bots_arg_size_1+j] = col[i*bots_arg_size_1+j] - diag[i*bots_arg_size_1+k]*col[k*bots_arg_size_1+j];
}


void sparselu_init (float ***pBENCH, char *pass)
{
    *pBENCH = (float **) malloc(bots_arg_size*bots_arg_size*sizeof(float *));
    genmat(*pBENCH);
    //print_structure(pass, *pBENCH);
}

void sparselu_par_call(float **BENCH)
{
    int ii, jj, kk;


    //fprintf(stderr,"Computing SparseLU Factorization (%dx%d matrix with %dx%d blocks) ", bots_arg_size, bots_arg_size,bots_arg_size_1,bots_arg_size_1);

    //start_time = omp_get_wtime();

    //#pragma omp parallel
    //#pragma omp single nowait
    //#pragma omp task untied

    for (kk=0; kk<bots_arg_size; kk++)
    {
        lu0(BENCH[kk*bots_arg_size+kk]);
        for (jj=kk+1; jj<bots_arg_size; jj++)
            if (BENCH[kk*bots_arg_size+jj] != NULL)
                //#pragma omp task untied firstprivate(kk, jj) shared(BENCH)
            {
        input in;
```

```
        in.benchinp1 = BENCH[kk*bots_arg_size+kk];
        in.benchinp2 = BENCH[kk*bots_arg_size+jj];
        if (i < num_threads){
          t[i] = std::thread(fwd, in);
        } else {
          fwd(in);
        }
              //fwd(BENCH[kk*bots_arg_size+kk], BENCH[kk*
                  bots_arg_size+jj]);
          }
        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
              //#pragma omp task untied firstprivate(kk, ii)
                  shared(BENCH)
            {
              bdiv (BENCH[kk*bots_arg_size+kk], BENCH[ii*
                  bots_arg_size+kk]);
            }

        //#pragma omp taskwait

        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
              for (jj=kk+1; jj<bots_arg_size; jj++)
                  if (BENCH[kk*bots_arg_size+jj] != NULL)
                  //#pragma omp task untied firstprivate(kk, jj
                      , ii) shared(BENCH)
                  {
                          if (BENCH[ii*bots_arg_size+jj]==NULL)
                              BENCH[ii*bots_arg_size+jj] =
                              allocate_clean_block();
                          bmod(BENCH[ii*bots_arg_size+kk], BENCH[
                              kk*bots_arg_size+jj], BENCH[ii*
                              bots_arg_size+jj]);
                  }

              //#pragma omp taskwait
    }
    //end_time = omp_get_wtime();
    //fprintf(stderr," completed!\n");
}


void sparselu_seq_call(float **BENCH)
```

```
{
    int ii, jj, kk;

    //start_seq = omp_get_wtime();

    for (kk=0; kk<bots_arg_size; kk++)
    {
        lu0(BENCH[kk*bots_arg_size+kk]);
        for (jj=kk+1; jj<bots_arg_size; jj++)
            if (BENCH[kk*bots_arg_size+jj] != NULL)
            {
                fwd(BENCH[kk*bots_arg_size+kk], BENCH[kk*
                    bots_arg_size+jj]);
            }
        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
            {
                bdiv (BENCH[kk*bots_arg_size+kk], BENCH[ii*
                    bots_arg_size+kk]);
            }
        for (ii=kk+1; ii<bots_arg_size; ii++)
            if (BENCH[ii*bots_arg_size+kk] != NULL)
                for (jj=kk+1; jj<bots_arg_size; jj++)
                    if (BENCH[kk*bots_arg_size+jj] != NULL)
                    {
                            if (BENCH[ii*bots_arg_size+jj]==NULL)
                                BENCH[ii*bots_arg_size+jj] =
                                allocate_clean_block();
                            bmod(BENCH[ii*bots_arg_size+kk], BENCH[
                                kk*bots_arg_size+jj], BENCH[ii*
                                bots_arg_size+jj]);
                    }

    }
    //end_seq = omp_get_wtime();
}

void sparselu_fini (float **BENCH, char *pass)
{
    //print_structure(pass, BENCH);
}

int sparselu_check(float **SEQ, float **BENCH)
{
```

```
    int ii,jj,ok=1;

    for (ii=0; ((ii<bots_arg_size) && ok); ii++)
    {
        for (jj=0; ((jj<bots_arg_size) && ok); jj++)
        {
            if ((SEQ[ii*bots_arg_size+jj] == NULL) && (BENCH[ii
                *bots_arg_size+jj] != NULL)) ok = FALSE;
            if ((SEQ[ii*bots_arg_size+jj] != NULL) && (BENCH[ii
                *bots_arg_size+jj] == NULL)) ok = FALSE;
            if ((SEQ[ii*bots_arg_size+jj] != NULL) && (BENCH[ii
                *bots_arg_size+jj] != NULL))
                ok = checkmat(SEQ[ii*bots_arg_size+jj], BENCH[ii
                    *bots_arg_size+jj]);
        }
    }
    if (ok) return BOTS_RESULT_SUCCESSFUL;
    else return BOTS_RESULT_UNSUCCESSFUL;
}


int main ( int argc, char *argv[])
{
      float **SEQ,**BENCH;

    //numWorkers = (argc > 1)? atoi(argv[1]) : MAXWORKERS;
    //if (numWorkers > MAXWORKERS) numWorkers = MAXWORKERS;

    //omp_set_num_threads(numWorkers);

      fprintf(stderr,"Startar parallel version...\n");
      sparselu_init(&BENCH,"benchmark");
      sparselu_par_call(BENCH);
      sparselu_fini(BENCH,"benchmark");

      fprintf(stderr,"Startar seriell version...\n");

      sparselu_init(&SEQ,"serial");
      sparselu_seq_call(SEQ);
      sparselu_fini(SEQ,"serial");

      fprintf(stderr,"Testar om Parallel och Seriell version
          stÃ¤mmer med varandra...\n");
      sparselu_check(SEQ,BENCH);
```

```
    printf("Completed!");
}
```

B.3 Source code for C#

```
ï»¿using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace sparselu
{
    class Program
    {
        public static double EPSILON = 1.0E-6;
        public static int bots_arg_size = 50;
        public static int bots_arg_size_1 = 100;
        public static int BOTS_RESULT_SUCCESSFUL = 1;
        public static int BOTS_RESULT_UNSUCCESSFUL = 0;
        public static double par_time, seq_time;
        public static int numberOfConcurrentTasks;

        static void Main(string[] args)
        {
            if (args.Length != 0)
            {
                numberOfConcurrentTasks = Convert.ToInt32(
                    args[0]);
                if (args.Length >= 2) bots_arg_size =
                    Convert.ToInt32(args[1]);
                if (args.Length >= 3) bots_arg_size_1 =
                    Convert.ToInt32(args[2]);
            }
            else numberOfConcurrentTasks = 10000;

            float[,][,] SEQ, BENCH;
            //Console.Error.WriteLine("Startar parallel
                version...\n");
            sparselu_init(out BENCH, "benchmark");
```

```
            BENCH = sparselu_par_call(BENCH);
            //sparselu_fini(ref BENCH, "benchmark");

            //Console.Error.WriteLine("Startar seriell version...\n");
            sparselu_init(out SEQ, "serial");
            sparselu_seq_call(ref SEQ);
            //sparselu_fini(ref SEQ, "serial");

            int success;
            //Console.Error.WriteLine("Testar om Parallel och Seriell version stÃ¤mmer med varandra...\n");
            success = sparselu_check(ref SEQ, ref BENCH);

            if (success == 1)
            {
                Console.WriteLine("SUCCESS! Time in seconds: Parallel: {0}   Seriell: {1}", par_time, seq_time);
            }
            else
            {
                Console.WriteLine("DEAD!");
            }

            /* Just so the terminal doesn't close to early */
            //Console.Error.WriteLine("Press Enter to finish");
            //Console.ReadLine();
        }

        static void sparselu_init(out float[,][,] BENCH, string pass)
        {
            BENCH = new float[bots_arg_size,bots_arg_size][,];
            genmat(ref BENCH);
            //print_structure(pass, ref BENCH);
        }

        static void genmat(ref float[,][,] BENCH)
        {
```

```
            bool null_entry;
            int init_val = 1325;

            /* generating the structure */
            for (int ii = 0; ii < bots_arg_size; ii++)
            {
                for (int jj = 0; jj < bots_arg_size; jj++)
                {
                    /* computing null entries */
                    null_entry = false;
                    if ((ii < jj) && (ii % 3 != 0)) null_entry = true;
                    if ((ii > jj) && (jj % 3 != 0)) null_entry = true;
                    if (ii % 2 == 1) null_entry = true;
                    if (jj % 2 == 1) null_entry = true;
                    if (ii == jj) null_entry = false;
                    if (ii == jj - 1) null_entry = false;
                    if (ii - 1 == jj) null_entry = false;
                    /* allocating matrix */
                    if (null_entry == false)
                    {
                        BENCH[ii, jj] = new float[bots_arg_size_1,bots_arg_size_1];
                        if (BENCH[ii, jj] == null)
                        {
                            Console.Error.WriteLine("Error: Out of memory\n");
                            /* ERROR_NOT_ENOUGH_MEMORY 8 (0x8) */
                            Environment.Exit(8);
                        }
                        /* initializing matrix */
                        for (int i = 0; i < bots_arg_size_1; i++)
                        {
                            for (int j = 0; j < bots_arg_size_1; j++)
                            {
                                init_val = (3125 * init_val) % 65536;
                                BENCH[ii, jj][i, j] = (float)((init_val - 32768.0) / 16384.0);
```

```
                            }
                        }
                    }
                    else
                    {
                        BENCH[ii, jj] = null;
                    }
                }
            }
        }

        static void print_structure(string name, ref float[,][,] BENCH)
        {
            Console.Error.WriteLine("Structure for matrix {0} @ 0x{1}\n", name, BENCH);
            for (int ii = 0; ii < bots_arg_size; ii++)
            {
                for (int jj = 0; jj < bots_arg_size; jj++)
                {
                    if (BENCH[ii, jj] != null) Console.Error.Write("x");
                    else Console.Error.Write(" ");
                }
                Console.Error.Write("\n");
            }
            Console.Error.Write("\n");
        }

        static float[,][,] sparselu_par_call(float[,][,] BENCH)
        {
            //Console.Error.WriteLine("Computing SparseLU Factorization ({0}x{1} matrix with {2}x{3} blocks) ",
            //      bots_arg_size, bots_arg_size, bots_arg_size_1, bots_arg_size_1);

            CountdownEvent numberOfActiveTasks = new CountdownEvent(1);
            List<Task> tasks = new List<Task>();
            CountdownEvent countdown = new CountdownEvent(1);
            Stopwatch stopwatch1 = new Stopwatch();
```

```

            stopwatch1.Start();

            for (int kk = 0; kk < bots_arg_size; kk++)
            {
                lu0(ref BENCH[kk, kk]);
                for (int jj = kk + 1; jj < bots_arg_size; jj
                   ++)
                {
                    while (((numberOfActiveTasks.
                        CurrentCount - 1) <
                        numberOfConcurrentTasks) && (tasks.
                        Count != 0))
                    {
                        Task task = tasks.ElementAt(0);
                        tasks.RemoveAt(0);
                        countdown.AddCount();
                        numberOfActiveTasks.AddCount();
                        task.Start();
                    }
                    if (BENCH[kk, jj] != null)
                    {
                        int temp1 = kk, temp2 = jj; /* To
                            avoid race conditions with for-
                            loop */
                        Task task = new Task(() =>
                        {
                            fwd(ref BENCH[temp1, temp1], ref
                                BENCH[temp1, temp2]);
                            numberOfActiveTasks.Signal();
                            countdown.Signal();
                        }
                            );
                        if ((numberOfActiveTasks.
                            CurrentCount - 1) <
                            numberOfConcurrentTasks)
                        {
                            countdown.AddCount();
                            numberOfActiveTasks.AddCount();
                            task.Start();
                        }
                        else
                        {
                            tasks.Add(task);
```

```
                    }
                }
            }
            for (int ii = kk + 1; ii < bots_arg_size; ii++)
            {
                while (((numberOfActiveTasks.CurrentCount - 1) < numberOfConcurrentTasks) && tasks.Count != 0)
                {
                    Task task = tasks.ElementAt(0);
                    tasks.RemoveAt(0);
                    task.Start();
                }
                if (BENCH[ii, kk] != null)
                {
                    int temp1 = kk, temp2 = ii; /* To avoid race conditions with for-loop */
                    Task task = new Task(() =>
                    {
                        bdiv(ref BENCH[temp1, temp1], ref BENCH[temp2, temp1]);
                        numberOfActiveTasks.Signal();
                        countdown.Signal();
                    }
                        );
                    if ((numberOfActiveTasks.CurrentCount - 1) < numberOfConcurrentTasks)
                    {
                        countdown.AddCount();
                        numberOfActiveTasks.AddCount();
                        task.Start();
                    }
                    else
                    {
                        tasks.Add(task);
                    }
                }
            }

            while (true)
```

```
            {
                if (tasks.Count == 0)
                {
                    break;
                }
                if ((numberOfActiveTasks.CurrentCount - 1) < numberOfConcurrentTasks)
                {
                    Task task = tasks.ElementAt(0);
                    tasks.RemoveAt(0);
                    countdown.AddCount();
                    numberOfActiveTasks.AddCount();
                    task.Start();
                }
            }

            countdown.Signal();
            countdown.Wait();
            countdown.Reset(1);

            for (int ii = kk + 1; ii < bots_arg_size; ii++)
            {
                if (BENCH[ii, kk] != null)
                {
                    for (int jj = kk + 1; jj < bots_arg_size; jj++)
                    {
                        while (((numberOfActiveTasks.CurrentCount - 1) < numberOfConcurrentTasks) && tasks.Count != 0)
                        {
                            Task task = tasks.ElementAt(0);
                            tasks.RemoveAt(0);
                            task.Start();
                        }
                        if (BENCH[kk, jj] != null)
                        {
                            int temp1 = kk, temp2 = ii, temp3 = jj; /* To avoid race conditions with for-loops */
```

```
                                Task task = new Task(() =>
                                {
                                    if (BENCH[temp2, temp3]
                                        == null) BENCH[temp2,
                                        temp3] =
                                        allocate_clean_block
                                        ();
                                    bmod(ref BENCH[temp2,
                                        temp1], ref BENCH[
                                        temp1, temp3], ref
                                        BENCH[temp2, temp3]);
                                    numberOfActiveTasks.
                                        Signal();
                                    countdown.Signal();
                                }
                                    );
                                if ((numberOfActiveTasks.
                                    CurrentCount - 1) <
                                    numberOfConcurrentTasks)
                                {
                                    countdown.AddCount();
                                    numberOfActiveTasks.
                                        AddCount();
                                    task.Start();
                                }
                                else
                                {
                                    tasks.Add(task);
                                }
                            }
                        }
                    }
                }
                while (true)
                {
                    if (tasks.Count == 0)
                    {
                        break;
                    }
                    if ((numberOfActiveTasks.CurrentCount -
                        1) < numberOfConcurrentTasks)
                    {
                        Task task = tasks.ElementAt(0);
                        tasks.RemoveAt(0);
```

```
                    countdown.AddCount();
                    numberOfActiveTasks.AddCount();
                    task.Start();
                }
            }

            countdown.Signal();
            countdown.Wait();
            countdown.Reset(1);
        }
        stopwatch1.Stop();
        par_time = (double)stopwatch1.
            ElapsedMilliseconds/1000; /* cast to seconds
            from milliseconds */
        return BENCH;
    }

    static void sparselu_fini(ref float[,][,] BENCH,
        string name)
    {
        print_structure(name, ref BENCH);
    }

    static void sparselu_seq_call(ref float[,][,] SEQ)
    {
        //Console.Error.WriteLine("Computing SparseLU
            Factorization ({0}x{1} matrix with {2}x{3}
            blocks) ",
        //      bots_arg_size, bots_arg_size,
            bots_arg_size_1, bots_arg_size_1);

        Stopwatch stopwatch2 = new Stopwatch();

        stopwatch2.Start();

        for (int kk = 0; kk < bots_arg_size; kk++)
        {
            lu0(ref SEQ[kk, kk]);
            for (int jj = kk + 1; jj < bots_arg_size; jj
                ++)
            {
                if (SEQ[kk, jj] != null)
                {
```

```
                        fwd(ref SEQ[kk, kk], ref SEQ[kk, jj]);
                    }
                }
                for (int ii = kk + 1; ii < bots_arg_size; ii++)
                {
                    if (SEQ[ii, kk] != null)
                    {
                        bdiv(ref SEQ[kk, kk], ref SEQ[ii, kk]);
                    }
                }
                for (int ii = kk + 1; ii < bots_arg_size; ii++)
                {
                    if (SEQ[ii, kk] != null)
                    {
                        for (int jj = kk + 1; jj < bots_arg_size; jj++)
                        {
                            if (SEQ[kk, jj] != null)
                            {
                                if (SEQ[ii, jj] == null) SEQ[ii, jj] = allocate_clean_block();
                                bmod(ref SEQ[ii, kk], ref SEQ[kk, jj], ref SEQ[ii, jj]);
                            }
                        }
                    }
                }
            }
            stopwatch2.Stop();
            seq_time = (double)stopwatch2.ElapsedMilliseconds / 1000; /* cast to seconds from milliseconds */
        }

        static void lu0(ref float[,] diag)
        {
            for (int k = 0; k < bots_arg_size_1; k++)
            {
```

```
                for (int i = k + 1; i < bots_arg_size_1; i++)
                {
                    diag[i, k] = diag[i, k] / diag[k, k];
                    for (int j = k + 1; j < bots_arg_size_1; j++)
                    {
                        diag[i, j] = diag[i, j] - diag[i, k] * diag[k, j];
                    }
                }
            }
        }

        static void fwd(ref float[,] diag, ref float[,] col)
        {
            for (int j = 0; j < bots_arg_size_1; j++)
            {
                for (int k = 0; k < bots_arg_size_1; k++)
                {
                    for (int i = k + 1; i < bots_arg_size_1; i++)
                    {
                        col[i, j] = col[i, j] - diag[i, k] * col[k, j];
                    }
                }
            }
        }

        static void bdiv(ref float[,] diag, ref float[,] row)
        {
            for (int i = 0; i < bots_arg_size_1; i++)
            {
                for (int k = 0; k < bots_arg_size_1; k++)
                {
                    row[i, k] = row[i, k] / diag[k, k];
                    for (int j = k + 1; j < bots_arg_size_1; j++)
                    {
                        row[i, j] = row[i, j] - row[i, k] * diag[k, j];
                    }
```

```
                }
            }
        }

        static float[,] allocate_clean_block()
        {
            float[,] p = new float[bots_arg_size_1,
                bots_arg_size_1];
            if (p != null)
            {
                for (int i = 0; i < bots_arg_size_1; i++)
                {
                    for (int j = 0; j < bots_arg_size_1; j
                        ++)
                    {
                        p[i, j] = (float)0.0;
                    }
                }
            }
            else
            {
                Console.Error.WriteLine("Error: Out of
                    memory\n");
                /* ERROR_NOT_ENOUGH_MEMORY 8 (0x8) */
                Environment.Exit(8);
            }
            return p;
        }

        static void bmod(ref float[,] row, ref float[,] col,
            ref float[,] inner)
        {
            for (int i = 0; i < bots_arg_size_1; i++)
            {
                for (int j = 0; j < bots_arg_size_1; j++)
                {
                    for (int k = 0; k < bots_arg_size_1; k
                        ++)
                    {
                        inner[i, j] = inner[i, j] - row[i, k
                            ] * col[k, j];
                    }
                }
            }
```

```
        }

        static int sparselu_check(ref float[,][,] SEQ, ref float[,][,] BENCH)
        {
            bool ok = true;

            for (int ii = 0; ((ii < bots_arg_size) && ok); ii++)
            {
                for (int jj = 0; ((jj < bots_arg_size) && ok); jj++)
                {
                    if ((SEQ[ii, jj] == null) && (BENCH[ii, jj] != null))
                    {
                        ok = false;
                    }
                    if ((SEQ[ii, jj] != null) && (BENCH[ii, jj] == null))
                    {
                        ok = false;
                    }
                    if ((SEQ[ii, jj] != null) && (BENCH[ii, jj] != null))
                    {
                        ok = checkmat(ref SEQ[ii, jj], ref BENCH[ii, jj]);
                    }
                }
            }
            if (ok) return BOTS_RESULT_SUCCESSFUL;
            else return BOTS_RESULT_UNSUCCESSFUL;
        }

        static bool checkmat(ref float[,] M, ref float[,] N)
        {
            float r_err;

            for (int i = 0; i < bots_arg_size_1; i++)
            {
                for (int j = 0; j < bots_arg_size_1; j++)
                {
                    r_err = M[i, j] - N[i, j];
```

```
                    if (r_err < 0.0) r_err = -r_err;
                    r_err = r_err / M[i, j];
                    if (r_err > EPSILON)
                    {
                        Console.Error.WriteLine("Checking␣failure:␣A[{0}][{1}]={2}␣␣B[{3}][{4}]={5};␣Relative␣Error={6}\n",
                          i, j, M[i, j], i, j, N[i, j], r_err);
                        return false;
                    }
                }
            }
            return true;
        }
    }
}
```

B.4 Source code for Java

```
import java.util.concurrent.ForkJoinPool;


public class Main {

  public static float[][][][] BENCH, SEQ;

  public static double EPSILON = 1.0E-6;
  public static int bots_arg_size = 50;
  public static int bots_arg_size_1 = 100;
  public static int BOTS_RESULT_SUCCESSFUL = 1;
  public static int BOTS_RESULT_UNSUCCESSFUL = 0;
  public static double time_start, par_time, seq_time;
  public static int numberOfThreads;
  public static boolean isBENCH;
  public static ForkJoinPool threadpool;

  public static void main(String[] args){

    //read input on how many threads to use
    if (args.length != 0){
```

```
            numberOfThreads = Integer.parseInt(args[0]);
            if(args.length >= 2) bots_arg_size = Integer.parseInt(args[1]);
            if(args.length >= 3) bots_arg_size_1 = Integer.parseInt(args[2]);
        }
        else numberOfThreads = 48;


        threadpool = new ForkJoinPool(numberOfThreads);

        isBENCH = true;
        sparselu_init("benchmark");

        sparselu_par_call();

        //sparselu_fini("benchmark");

        isBENCH = false;
        sparselu_init("serial");

        sparselu_seq_call();

        //sparselu_fini("serial");

        int success;
        success = sparselu_check();

        if (success == 1)
        {
            System.out.println("SUCCESS! Time in seconds: Parallel: "+par_time+"   Seriell: "+seq_time);
        }
        else
        {
            System.out.println("DEAD!");
        }
    }

    static void sparselu_init(String pass)
    {
        if(isBENCH) BENCH = genmat();
        else        SEQ = genmat();
        //print_structure(pass);
```

```
    }

    static float[][][][] genmat()
    {
      float[][][][] matrix = new float[bots_arg_size][
          bots_arg_size][][];
      boolean null_entry;
      int init_val = 1325;

      /* generating the structure */
      for (int ii = 0; ii < bots_arg_size; ii++)
      {
        for (int jj = 0; jj < bots_arg_size; jj++)
        {
          /* computing null entries */
          null_entry = false;
          if ((ii < jj) && (ii % 3 != 0)) null_entry = true;
          if ((ii > jj) && (jj % 3 != 0)) null_entry = true;
          if (ii % 2 == 1) null_entry = true;
          if (jj % 2 == 1) null_entry = true;
          if (ii == jj) null_entry = false;
          if (ii == jj - 1) null_entry = false;
          if (ii - 1 == jj) null_entry = false;
          /* allocating matrix */
          if (null_entry == false)
          {
            matrix[ii][jj] = new float[bots_arg_size_1] [
                bots_arg_size_1];
            if (matrix[ii][jj] == null)
            {
              System.out.println("Error: Out of memory\n");
              /* ERROR_NOT_ENOUGH_MEMORY 8 (0x8) */
              System.exit(1);
            }
            /* initializing matrix */
            for (int i = 0; i < bots_arg_size_1; i++)
            {
              for (int j = 0; j < bots_arg_size_1; j++)
              {
                init_val = (3125 * init_val) % 65536;
                matrix[ii][jj][i][j] = (float)((init_val -
                    32768.0) / 16384.0);
              }
            }
```

```
            }
            else
            {
               matrix[ii][jj] = null;
            }
         }
      }
      return matrix;
   }

   static void print_structure(String name)
   {
      float[][][][] matrix;
      if(isBENCH) matrix = BENCH;
      else        matrix = SEQ;
      System.err.println("Structure for matrix "+name+" @ 0x"+matrix+"\n");
      for (int ii = 0; ii < bots_arg_size; ii++)
      {
         for (int jj = 0; jj < bots_arg_size; jj++)
         {
            if (matrix[ii][jj] != null) System.err.print("x");
            else System.err.print(" ");
         }
         System.err.print("\n");
      }
      System.err.print("\n");
   }

   static void sparselu_par_call()
   {
      //Console.Error.WriteLine("Computing SparseLU Factorization ({0}x{1} matrix with {2}x{3} blocks) ",
      //      bots_arg_size, bots_arg_size, bots_arg_size_1, bots_arg_size_1);

      //Stopwatch stopwatch2 = new Stopwatch();

      //stopwatch2.Start();
      time_start = System.nanoTime();

      for (int kk = 0; kk < bots_arg_size; kk++)
      {
         lu0(BENCH[kk][kk], kk, kk);
```

```
            for (int jj = kk + 1; jj < bots_arg_size; jj++)
            {
                if (BENCH[kk][jj] != null)
                {
                    threadpool.execute(new Task("fwd", BENCH[kk][kk], BENCH[kk][jj], kk, jj));
                }
            }
            for (int ii = kk + 1; ii < bots_arg_size; ii++)
            {
                if (BENCH[ii][kk] != null)
                {
                    threadpool.execute(new Task("bdiv", BENCH[kk][kk], BENCH[ii][kk], ii, kk));
                }
            }

            while (!threadpool.isQuiescent()) {}

            for (int ii = kk + 1; ii < bots_arg_size; ii++)
            {
                if (BENCH[ii][kk] != null)
                {
                    for (int jj = kk + 1; jj < bots_arg_size; jj++)
                    {
                        if (BENCH[kk][jj] != null)
                        {
                            if (BENCH[ii][jj] == null) BENCH[ii][jj] = allocate_clean_block();
                            threadpool.execute(new Task("bmod", BENCH[ii][kk], BENCH[kk][jj], BENCH[ii][jj], ii, jj));
                        }
                    }
                }
            }
            while (!threadpool.isQuiescent()) {}
        }
        par_time = (System.nanoTime() - time_start)/1000000000;
        //stopwatch2.Stop();
        //seq_time = (double)stopwatch2.ElapsedMilliseconds / 1000; /* cast to seconds from milliseconds */
    }

```

```
static void sparselu_fini(String name)
{
  print_structure(name);
}

static void sparselu_seq_call()
{
  isBENCH = false;
  //Console.Error.WriteLine("Computing SparseLU
      Factorization ({0}x{1} matrix with {2}x{3} blocks) ",
  //      bots_arg_size, bots_arg_size, bots_arg_size_1,
      bots_arg_size_1);

  //Stopwatch stopwatch2 = new Stopwatch();

  //stopwatch2.Start();
  time_start = System.nanoTime();

  for (int kk = 0; kk < bots_arg_size; kk++)
  {
    lu0(SEQ[kk][kk], kk, kk);
    for (int jj = kk + 1; jj < bots_arg_size; jj++)
    {
      if (SEQ[kk][jj] != null)
      {
        fwd(SEQ[kk][kk], SEQ[kk][jj], kk, jj);
      }
    }
    for (int ii = kk + 1; ii < bots_arg_size; ii++)
    {
      if (SEQ[ii][kk] != null)
      {
        bdiv(SEQ[kk][kk], SEQ[ii][kk], ii, kk);
      }
    }
    for (int ii = kk + 1; ii < bots_arg_size; ii++)
    {
      if (SEQ[ii][kk] != null)
      {
        for (int jj = kk + 1; jj < bots_arg_size; jj++)
        {
          if (SEQ[kk][jj] != null)
          {
```

```
                if (SEQ[ii][jj] == null) SEQ[ii][jj] =
                    allocate_clean_block();
                bmod(SEQ[ii][kk], SEQ[kk][jj], SEQ[ii][jj], ii
                    , jj);
              }
            }
          }
        }
      }
      seq_time = (System.nanoTime() - time_start)/1000000000;
      //stopwatch2.Stop();
      //seq_time = (double)stopwatch2.ElapsedMilliseconds /
          1000; /* cast to seconds from milliseconds */
    }

    static void lu0(float[][] diag, int x, int y)
    {
      for (int k = 0; k < bots_arg_size_1; k++)
      {
        for (int i = k + 1; i < bots_arg_size_1; i++)
        {
          diag[i][k] = diag[i][k] / diag[k][k];
          for (int j = k + 1; j < bots_arg_size_1; j++)
          {
            diag[i][j] = diag[i][j] - diag[i][k] * diag[k][j];
          }
        }
      }
      if(isBENCH) BENCH[x][y] = diag;
      else      SEQ[x][y] = diag;
    }

    static void fwd(float[][] diag, float[][] col, int x, int
        y)

    {
      for (int j = 0; j < bots_arg_size_1; j++)
      {
        for (int k = 0; k < bots_arg_size_1; k++)
        {
          for (int i = k + 1; i < bots_arg_size_1; i++)
          {
            col[i][j] = col[i][j] - diag[i][k] * col[k][j];
          }
```

```
            }
        }
        if(isBENCH) BENCH[x][y] = col;
        else      SEQ[x][y] = col;
    }

    static void bdiv(float[][] diag, float[][] row, int x, int
        y)
    {
        for (int i = 0; i < bots_arg_size_1; i++)
        {
            for (int k = 0; k < bots_arg_size_1; k++)
            {
                row[i][k] = row[i][k] / diag[k][k];
                for (int j = k + 1; j < bots_arg_size_1; j++)
                {
                    row[i][j] = row[i][j] - row[i][k] * diag[k][j];
                }
            }
        }
        if(isBENCH) BENCH[x][y] = row;
        else      SEQ[x][y] = row;
    }

    static float[][] allocate_clean_block()
    {
        float[][] p = new float[bots_arg_size_1][bots_arg_size_1
            ];
        if (p != null)
        {
            for (int i = 0; i < bots_arg_size_1; i++)
            {
                for (int j = 0; j < bots_arg_size_1; j++)
                {
                    p[i][j] = (float)0.0;
                }
            }
        }
        else
        {
            System.err.println("Error: Out of memory\n");
            /* ERROR_NOT_ENOUGH_MEMORY 8 (0x8) */
            System.exit(1);
        }
```

```
        return p;
    }

    static void bmod(float[][] row, float[][] col, float[][]
        inner, int x, int y)
    {
        for (int i = 0; i < bots_arg_size_1; i++)
        {
            for (int j = 0; j < bots_arg_size_1; j++)
            {
                for (int k = 0; k < bots_arg_size_1; k++)
                {
                    inner[i][j] = inner[i][j] - row[i][k] * col[k][j];
                }
            }
        }
        if(isBENCH) BENCH[x][y] = inner;
        else     SEQ[x][y] = inner;
    }

    static int sparselu_check()
    {
        boolean ok = true;

        for (int ii = 0; ((ii < bots_arg_size) && ok); ii++)
        {
            for (int jj = 0; ((jj < bots_arg_size) && ok); jj++)
            {
                if ((SEQ[ii][jj] == null) && (BENCH[ii][jj] != null)
                    )
                {
                    ok = false;
                }
                if ((SEQ[ii][jj] != null) && (BENCH[ii][jj] == null)
                    )
                {
                    ok = false;
                }
                if ((SEQ[ii][jj] != null) && (BENCH[ii][jj] != null)
                    )
                {
                    ok = checkmat(SEQ[ii][jj], BENCH[ii][jj]);
                }
            }
```

```
    }
    if (ok) return BOTS_RESULT_SUCCESSFUL;
    else return BOTS_RESULT_UNSUCCESSFUL;
  }

  static boolean checkmat(float[][] M, float[][] N)
  {
    float r_err;

    for (int i = 0; i < bots_arg_size_1; i++)
    {
      for (int j = 0; j < bots_arg_size_1; j++)
      {
        r_err = M[i][j] - N[i][j];
        if (r_err < 0.0) r_err = -r_err;
        r_err = r_err / M[i][j];
        if (r_err > EPSILON)
        {
          System.err.println("Checking failure: A["+i+"]["+j
              +"]="+M[i][j]+"  B["+i+"]["+j+"]="+N[i][j]+"; 
              Relative Error="+r_err+"\n");
          return false;
        }
      }
    }
    return true;
  }

}
```

```
import java.util.concurrent.RecursiveAction;


public class Task extends RecursiveAction{

  private float[][] input1, input2, input3;
  private int x, y;
  String function;

  Task(String function, float[][] input1, float[][] intput2,
      int x, int y){
    this.function = function;
    this.input1 = input1;
    this.input2 = intput2;
```

```
    this.x = x;
    this.y = y;
  }

  Task(String function, float[][] input1, float[][] input2,
      float[][] input3, int x, int y){
    this.function = function;
    this.input1 = input1;
    this.input2 = input2;
    this.input3 = input3;
    this.x = x;
    this.y = y;
  }

  @Override
  protected void compute() {
    // TODO Auto-generated method stub
    if (function.equals("fwd")) fwd(input1, input2, x, y);
    if(function.equals("bdiv")) bdiv(input1, input2, x, y);
    if(function.equals("bmod")) bmod(input1, input2, input3,
        x, y);
  }


    static void fwd(float[][] diag, float[][] col, int x,
        int y)

    {
      for (int j = 0; j < Main.bots_arg_size_1; j++)
      {
        for (int k = 0; k < Main.bots_arg_size_1; k++)
        {
          for (int i = k + 1; i < Main.bots_arg_size_1; i++)
          {
            col[i][j] = col[i][j] - diag[i][k] * col[k][j];
          }
        }
      }
      if(Main.isBENCH) Main.BENCH[x][y] = col;
      else         Main.SEQ[x][y]   = col;
    }

    static void bdiv(float[][] diag, float[][] row, int x,
        int y)
```

```
    {
        for (int i = 0; i < Main.bots_arg_size_1; i++)
        {
            for (int k = 0; k < Main.bots_arg_size_1; k++)
            {
                row[i][k] = row[i][k] / diag[k][k];
                for (int j = k + 1; j < Main.bots_arg_size_1; j++)
                {
                    row[i][j] = row[i][j] - row[i][k] * diag[k][j];
                }
            }
        }
        if(Main.isBENCH) Main.BENCH[x][y] = row;
        else        Main.SEQ[x][y]   = row;
    }

    static void bmod(float[][] row, float[][] col, float[][] inner, int x, int y)
    {
        for (int i = 0; i < Main.bots_arg_size_1; i++)
        {
            for (int j = 0; j < Main.bots_arg_size_1; j++)
            {
                for (int k = 0; k < Main.bots_arg_size_1; k++)
                {
                    inner[i][j] = inner[i][j] - row[i][k] * col[k][j];
                }
            }
        }
        if(Main.isBENCH) Main.BENCH[x][y] = inner;
        else        Main.SEQ[x][y]   = inner;
    }
}
```

Bibliography

[1] Eli Dow. Mono brings .net apps to linux, September 2005.

[2] Ecma. What is ecma international, June 2013.

[3] Paul Jansen and Bram Stappers. Tiobe programming community index, April 2013.

[4] Prechte L. An emperical comparison of seven programming languages. Technical report, Karlsruhe Univ., Germany, August 2002.

[5] Calvin Lin and Larry Synder. *Principles of Parallel Programming*. Addison-Wesley Publishing Company, USA, first edition, 2008.

[6] Microsoft. Microsoft introduces highly productive .net programming language: C sharp. June 2000.

[7] Sun Microsystems. Sun ships new version of java platform. September 2004.

[8] Michael Philippsen. Openmp/java. October 2009.

[9] Artur Podobas. Performance-driven exploration using task-based parallel programming frameworks. Licentiate thesis, KTH Royal Institute of Technology, Information and Communication Technology, Sweden.

[10] Julien Ponge. Fork and join: Java can excel at painless parallel programming too! July 2011.

[11] Kathleen Richards. Visual studio 2012 and windows 8 released to developers, August 2012.

[12] Philip E. Ross. Why cpu frequency stalled, April 2008.

[13] Daniel Spiewak. Defining high, mid and low-level languages, February 2008.

[14] Xavier Teruel. The barcelona openmp task suite (bots) project. Technical report, University of North Carolina, the Ohio State University and the University of Maryland, November 2011.

TRITA-ICT-EX-2013:157