İSTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

A NEW PARALLEL PROGRAMING LANGUAGE FORTRESS: FEATURES AND APPLICATIONS

M.Sc. Thesis by
Erdem ÜNEY

Department : Informatics Institute

Programme : Computational Science and Engineering

SEPTEMBER 2009

İSTANBUL TECHNICAL UNIVERSITY ★ INFORMATICS INSTITUTE

A NEW PARALLEL PROGRAMING LANGUAGE FORTRESS: FEATURES AND APPLICATIONS

M.Sc. Thesis by
Erdem ÜNEY
(702051007)

Date of submission : 28 August 2009
Date of defense examination: 11 September 2009

Supervisor (Chairman) : Prof. Dr. H. Nüzhet DALFES (İTU)
Members of the Examining Committee : Prof. Dr. Serdar ÇELEBİ (İTU)
Prof. Dr. Hasan DAĞ (KHAS)

SEPTEMBER 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

YENİ BİR PARALEL PROGRAMLAMA DİLİ FORTRESS: ÖZELLİKLERİ VE UYGULAMALARI

YÜKSEK LİSANS TEZİ

Erdem ÜNEY
(702051007)

Tezin Enstitüye Verildiği Tarih : 28 Ağustos 2009
Tezin Savunulduğu Tarih : 11 Eylül 2009

Tez Danışmanı : Prof. Dr. H. Nüzhet DALFES (ITU)
Diğer Jüri Üyeleri : Prof. Dr. Serdar ÇELEBİ (ITU)
Prof. Dr. Hasan DAĞ (KHAS)

EYLÜL 2009

In the loving and constantly illuminating
memory of my father, Tuncer Üney…

FOREWORD

I would like to express my deep appreciation and thanks for my advisor, Prof. Dalfes. Every student presents his regards to his advisor but without the support of my advisor from the days of my undergraduate thesis, I would not be able to seek the academic appretiation and complete the process of graduating. I also would like to thank my good friends İlker Kopan, Sayat Baronyan and lovely Pelin Çallı for their support and motivation during the course of my thesis. Last but not least I would like to thank my little brother and my mother, who is constantly pushing me to go forward. This work is supported by ITU Informatics Institute and National Center for High Performance Computing. Thank you all.

September 2009

Erdem ÜNEY

Meteorology Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

ASCII	**:** American Standard Code for Information Interchange
BSD	**:** Berkeley Software Distribution
CAF	**:** Co-Array Fortran
DARPA	**:** Defense Advanced Research Projects Agency
HPC	**:** High Performance Computing
HPCS	**:** High Productivity Computing Systems
HPF	**:** High Performance Fortran
IPTO	**:** Information Processing Techniques Office
NCHPC	**:** National Center for High Performance Computing
PERCS	**:** Productive Easy-to-use Reliable Computer Systems
UPC	**:** Unified Parallel C

LIST OF TABLES

Page

LIST OF FIGURES

Page

A NEW PARALLEL PROGRAMING LANGUAGE FORTRESS: FEATURES AND APPLICATIONS

SUMMARY

Computer systems are growing in an increasing the phase. DARPA foreseen the requirement and establishing a petascale computer by the year 2010. The project has started with several companies in 2003. Now the project is about to be over and the outcome of the project after millions of dollars spent is three high performance and highly productive computer languages. One of them is Fortress.

Fortress is a mathematical notation based, strictly typed, block based, implicitly parallel computing language. Highly productive and science oriented design is what makes Fortress interesting. In this study we have observed and studied the inner dynamics of Fortress. Benchmarking tests are done to measure the performance of Fortress, and the results are discussed.

YENİ BİR PARALEL PROGRAMLAMA DİLİ FORTRESS: ÖZELLİKLERİ VE UYGULAMALARI

ÖZET

Bilgisayar sistemleri çok hızlı bir şekilde büyümektedirler. DARPA, 2010 yılı için peta-ölçekli bir bilgisayar sisteminin gereksinimi ve yapılabilirliğini öngördü. 2003 yılında bir çok firmayla bir proje başlattı. Şimdi, proje bitmek üzereyken ve milyonlarca dolar projeye aktarılmışken, projenin getirisi üç tane yüksek başarımlı yüksek işlevselli programlama dili oldu. Bu dillerden bir tanesi Fortress.

Fortress, matematik gösterim temelli, değişken tipleri sıkı olarak takip edilen, blok tabanlı ve kesin olarak paralel bir bilgisayar programlama dilidir. Fortress'i ilginç kılan, yüksek işlevsellikli ve bilim yönlü yapısıdır. Bu çalışmada Fortress'in iç dinamiklerini inceledi. Performansını ölçmek için çeşitli testler yapıldı ve sonuçları tartışıldı.

1. INTRODUCTION

The ongoing development of computer and computational technology have led us into the era of the Gigahertz speeds and multiple core machines. In today's world of technology a single home machine can contain 15 gigaflops of computational power with dual core technology [1]. To achieve the best performance from the growing number of high performance machines its required to have newer, more sophisticated programming languages. One of those languages is Fortress, an open sourced, implicitly parallel programming language. A very new language which has reached its 1.0 official release only in April 2008 [2]. In this paper, the development of the language is followed and showcased some of its features.

1.1 Purpose of the Thesis

Main objective of this study is to observe the development of a new High Performance Computing (HPC) language and to set a light to the future of the programming languages for the parallel computing. To accomplish this a new programming language called Fortress was studied. In the first part; this study will show how Fortress has begun, what was the vision laying behind it and how has it evolved to current state. A brief history of the program which has led into the development of the project will be discussed with the competing development programs. The future path of the language will also be presented in the light of the beta version of the language. In the second part, it will be shown the basics of the language with the current capabilities of it, and a comparison with the widely used C programming language will presented. In the last part, the tests of an application written in Fortress will be shown and discussed [3].

1.2 Background

Since the beginning of the computational programming, the languages and the written code to solve problems were always been sequential. A serial code would be written to solve a problem. This way was the result of the architecture. After the dawn of the networking, with the idea of two or more computers communicating led to the idea of computational power and data sharing thus the creation of the parallel programming. The need of making two computers communicate created the communication libraries like PVM and MPI. This was a way of patching the currently serial code, and adding the ability to communicate within several parts. Another way was to balance the load within the CPU with threads and with the introduction of multi core, shared memory machines this led to the OpenMP, pThread and several other load balancing standards [4].

A few other programming languages have been developed based on the currently available code. Some of these examples are Unified Parallel C (UPC), Co-Array FORTRAN (CAF) and Titanium. These languages are respectively based on the currently available code base of ISO C, FORTRAN 95-2003 and Java. Development of these languages is an attempt to create a simple way of parallelization using shared memory address spaces. As can be count many other similar or newly developed languages, most of them do not offer a scalable and robust solution to the parallelization of the growing computer power. This lack of a programming language has led Defense Advanced Research Projects Agency (DARPA), which is an agency of United States of America, to fund a project on high performance computer systems.

2. NEW LANGUAGES FOR PARALLEL PROGRAMING

The Defense Advanced Research Projects Agency (DARPA) is responsible for the development of new technologies for the use of the military. DARPA has been responsible for funding the development of many technologies which have a major impact on the world, including computer networking [5]. DARPA has many projects in several areas of technology. One of the offices which governs an area in research is The Information Processing Techniques Office (IPTO). The mission of the IPTO programs is to create the advanced information processing and exploitation science, technologies, and systems for revolutionary improvements in capability across the spectrum of national security needs. High Productivity Computing is one of the many thrust areas of IPTO, and it contains several projects which are developing the high-productivity, high-performance computer hardware and the associated software technology base required to support future critical national security needs for computationally-intensive and data-intensive applications [6].

One of these projects is High Productivity Computing Systems (HPCS). Conduct a focused research and development program that creates a new generation of high productivity computing systems. The goals of this project are to achieve:

- Performance: Improve the computational efficiency and reduce the execution time of critical national security applications [7].
- Programmability: Reduce cost and time of developing HPCS applications.
- Portability: Insulate HPCS application software from system specifics.
- Robustness: Improve reliability and reduce vulnerability to intentional attacks.

The biggest challenge of all is to design a high productivity computer systems that are optimized for users, not for benchmarks. A new computer system can be designed to reach the petaflops scale, but the main purpose of designing a new system must be productivity not performance. In the light of this task, all the attending groups of the program must design a balanced scalable system architecture with highly effective bandwidth usage. This program has started in 2003 with the vision of the Petaflops scale design be completed during the years 2007-2010. An underlying architecture also with the accompanying tools will be the task points of this project.

2.1 Beginning

HPCS program was planned to be three phases. First phase was designing phase. The companies involved were HP, SGI, IBM, Cray and SUN. The task was to design a HPCS. After the completion of the Phase I, HP and SGI discontinued from the program. In Phase II, the start of building the petaflops machine with a HPCS programming language has started. DARPA has not been funding only one project within each company but a set of several projects. The funded projects in general were:

- Chapel programming language and Cascade; Lustre filesystem with Cray Inc.,
- PERCS (Productive, Easy-to-use, Reliable Computer System) based on POWER7 processor, AIX operating system and General Parallel File System with IBM,
- Proximity communication and research projects of Silicon Photonics, Object-Based Storage, Fortress programming language, interval computing with Sun Microsystems.

Within these programs three new, parallel modeled programming languages has been developed. These are; Chapel of Cray, X10 of IBM and Fortress of Sun. At the end of the Phase II Sun got discontinued from the program. After the departure of Sun from the program, Sun has released the Fortress programming language as an open source project.

2.2 The Languages

Chapel, X10 and Fortress are three research level parallel programming models. All three of them have three different and partly similar approaches to parallelization, communication and memory systems.

2.2.1 Chapel

Chapel, the Cascade High Productivity Language, is designed to improve the productivity of high-end computer users while also serving as a portable parallel programming model that can be used on commodity clusters or desktop multi-core systems. Chapel strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI.

Chapel was designed from first principles rather than by extending an existing language. It is an imperative block-structured language, designed to be easy to learn for users of C, C++, FORTRAN, Java, Perl, Matlab, and other popular languages. While Chapel builds on concepts and syntax from many previous languages, its parallel features are most directly influenced by ZPL, High-Performance Fortran (HPF), and the Cray MTA's extensions to C and FORTRAN [8].

Chapel strives to improve the programmability of parallel computers in general and the Cascade system in particular, by providing a higher level of expression than current programming languages do and by improving the separation between algorithmic expression and data structure implementation details. It supports a multithreaded parallel programming model at a high level by supporting abstractions for data parallelism, task parallelism, and nested parallelism. It enables optimizations for the locality of data and computation in the program through abstractions for data distribution and data-driven placement of subcomputations. It allows for code reuse and generality through object-oriented concepts and generic programming features.

2.2.2 X10

X10 is an experimental new language currently under development at IBM in collaboration with academic partners. The X10 effort is part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems) in the DARPA program on High Productivity Computer Systems. The PERCS project is focused on a hardware-software co-design methodology to integrate advances in chip technology, architecture, operating systems, compilers, programming language and programming tools to deliver new adaptable, scalable systems that will provide an order-of-magnitude improvement in development productivity for parallel applications by 2010 [9].

X10 is designed specifically for parallel programming. It is an "extended subset" of the Java programming language, strongly resembling it in most aspects, but featuring additional support for arrays and concurrency. X10 uses a Partitioned global address space model. It supports both object-oriented and non-object-oriented programming paradigms. X10 uses the concept of parent and child relationships for tasks to prevent the lock stalemate that can occur when two or more processes wait for each other to finish before they can complete. A task may spawn one or more child tasks, which may themselves have children. Children cannot wait for a parent to finish, but a parent can wait for a child using the "finish" command.

2.2.3 Fortress

The Fortress programming language is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability. Fortress also supports modular and extensible parsing, allowing new notations and static analyses to be added to the language. The name “Fortress” is derived from the intent to produce a “secure FORTRAN”, i.e., a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles. Despite this etymology, the language is a new language with little relation to FORTRAN other than its intended domain of application. No attempt has been made to support backward compatibility with existing versions of FORTRAN; indeed, many new language features were invented during the design of Fortress.

There are several new features that is shaping Fortress. Two of the reasons governing the choice of this language for this paper are; its mathematical notation aware system and implicitly parallel nature. The features and the general structure of the language will be presented in the next chapter.

3. FORTRESS PROGRAMMING LANGUAGE

Although the name is derived from FORTRAN, it most resembles Scala, Standard ML, and Haskell. Fortress is being designed from the outset to have multiple syntactic style sheets. Source code can be rendered as ASCII text, in UNICODE, or as a prettied image. This allows for support of mathematical symbols and other symbols in the rendered output for easier reading. Fortress is also designed to be both highly parallel and have rich functionality contained within libraries, drawing from Java but taken to a higher degree. In this chapter the specifications of the language will be discussed.

3.1 The main purpose and the current state

Fortress is currently an open source project with BSD Licence. It has an interpreter coded in Java. Currently an interpreted language and also a compiled language but not necessarily. This will be discussed in future plans of the language. Since its interpreter is Java based, the language is platform independent. The main motto of the language is “To Do for Fortran What Java Did for C.” Which is a goal set to accomplish these items:

- Catch simple errors
- Extensive library structure
- Type safety and garbage collection
- Dynamic compilation
- Platform independence
- Multithreading
- Scientific Computing

3.2 Mathematical Notation

One of the main and most noticeable features of the Fortress language is its ability to write in Mathematical Notation. The actual code is very similar to the mathematical formulation of the problem. Here is an example of a code.

$$\text{for } i \leftarrow 1:n, j \leftarrow 1:n \text{ do}$$
$$a_{ij} = \sum_{k \leftarrow 1:n} b_{ik} c_{kj}$$
$$\text{end}$$

Figure 3.1 : Fortress for-loop example code Fortified

The above result is derived from the actual code by using a little application called Fortify. Fortify, converts actual Fortress code into LaTeX environment. Then you can either embed it to your LaTeX document or export in image to use elsewhere. This feature gives much more flexibility when writing and presenting the code. The actual code can be written in two ways. First is the ASCII based and second is the UNICODE based. The Fortress interpreter recognizes both and the code can be written in as similar to the mathematical notation as can be.

```
for i <- 1:n, j <- 1:n do
     a[i,j] = SUM [k <- 1:n] b[i,k] c[k,j]
end
```

Figure 3.2 : Fortress for-loop example code in ASCII

```
for i ⟵ 1:n, j ⟵ 1:n do
     a[i,j] = ∑[k ⟵ 1:n] b[i,k] c[k,j]
end
```

Figure 3.3 : Fortress for-loop example code in UNICODE

Note in the above examples the Summation notation and the left pointing arrows can be either in ASCII or in UNICODE. Not all editors support UNICODE notation so a wiki like notation in ASCII form is defined to help the code writer. All the ways are rendered in the same manner. As can be seen in the above code, juxtaposition is an operator. In Fortress whitespace is meaningful and it helps to define the rendering of the code. Juxtaposition, in the above example between matrices is basically used in its mathematical notation as multiplication. Also the rendering of the variable changes in rendering. For example; `a[i,j]` is rendered as a_{ij} but `a[i, j]` is rendered as $a[i, j]$. There is a set of rules concerning the rendering of the code notations. Here are some are represented in the below table.

Table 3.1: Code Rendering Examples

M is rendered as M	lamda is rendered as λ
v_vec is rendered as $\vec{v}$	LAMDA is rendered as Λ
_v1 is rendered as v_1	PROD is rendered as $\prod$
a_dot_dot is rendered as $\ddot{a}$	ZZ32 is rendered as $\mathbb{Z}32$
x^y is rendered as x^y	`<| x+y | x <- a, y <- b |>` is rendered as $\langle x+y \mid x \leftarrow a, y \leftarrow b \rangle$

Most of the mathematical notation with the Greek letters are predefined in the core with their UNICODE representations. Also the way they behave is defined in the library code, so that the program can be as close to its mathematical representation as it can be. This helps in the productivity of the written code. The programmer does not have to concern herself with the structure of the code and the outcome of it. The easily rendered and displayed code forces the programmer focus more on the problem, rather than trying to write a parallel code solving the desired problem.

Beside the code notation, the way of the Fortress code works as you can expect from a math point of view. For example, one of the most common errors are in other languages is “division by zero” error. In Fortress it is defined that the outcome of division by zero is “INFINITY.”

3.3 Library Coding

Aside from the basic functions and basic types like boolean, float and integer, rest of the code is defined in library. All of the operators and types are defined in the library code. Fortress offers a wide range of functions and definitions to allow a library writer have as much control as possible. So, very little and very basic things are defined in the core. When most of the functions are defined in library, a basic code writer needs not to worry about the types and conditions in the front end. A vast control on everything helps the library writer to left as minimum as possible to the actual programmer.

3.3.1 Overloading

Overloading of functions, methods and operators means that an expression can mean two different things in two different situations. For example, operator plus sign (+) can mean addition of two –one preceding and one following– values together, since every type is declared individually in libraries of Fortress, a plus sign operation can mean sum of two any of types. It is declared how the operator will behave in every type declaration. This leads to the overloading of the operator.

Calls to overloaded operators are resolved first via the fixity of the operators; the way the operator is used by means of the closeness to the operand; based on the context of the calls. Then, among the applicable declarations with that fixity, the most specific declaration is chosen by checking for each meaning of the operator and if the operator is in the declared lists, Fortress runs the code as the matching way the operator has declared. This may lead to a confusion with the programmer and within the interpreter, but since Fortress is a stricly typed language, the code is checked against types, which leads to the clarification and usage of the operator.

3.3.2 Traits, Objects, Operators and Contracts

Basic concepts in Fortress are object and trait. Traits are templates for defining new named types. When declared, a trait specifies a collection of methods. One trait can extend others, which means that it inherits the methods from those traits, and that the type defined by that trait is a subtype of the types of traits it extends. Or a trait can exclude the parts from an extended trait.

Object declarations declare both object values and object trait types. Object declarations extend a set of traits from which they inherit methods. Object declarations do not include excludes clauses, but an object trait type cannot be extended.

Operators are the declarations of the signs and operations in the library code. Operators can be overloaded as explained. Operators can be used as prefix, infix, postfix, or nofix. For example, minus sign (−) can be either infix or prefix in conventional usage. Exclamation mark (!) can be postfix as in computing a factorial, and Fortress already has a declaration of the factorial with the exclamation mark operator. And a nofix operator takes no arguments.

Finally, there are special operators such as juxtaposition. Juxtaposition may be a function application or an infix operator in Fortress. When the left-hand-side expression is a function, juxtaposition performs function application; when the left-hand-side expression is a number, juxtaposition conventionally performs multiplication; when the left-hand-side expression is a string, juxtaposition conventionally performs string concatenation.

There are two specifications for functions, which are called function contracts. These include `requires` and `ensures`. An example will explain it better.

$$
\begin{aligned}
&fib(n:\mathbb{Z}32):\mathbb{Z}32\\
&\text{requires}\{n \geq 0\}\\
&\text{ensures}\{result \geq 0\}\\
&= \text{if } n \leq 1 \text{ then } n\\
&\text{else } fib(n-1) + fib(n-2)\\
&\text{end}
\end{aligned}
$$

Figure 3.4 : A rendered example of a Fibonacci function.

As can be observed in the above example that this function requires a variable n to be a positive $(n \geq 0)$ integer $(n : \mathbb{Z}32)$. And it gurantees a `result` variable to be also a positive $(result \geq 0)$ integer $(\mathbb{Z}32)$. This helps a lot of programming where you can guarantee the outcome of a function.

3.3.3 Example Library Code

A `Matrix` trait declaration will be presented and explained. This is an actual library code which is defined and declared in FortressLibrary.fss. FortressLibrary.fss is the only library code that is imported directly to the program code without the need of an import declaration.

```
trait Matrix[\T extends Number, nat s0, nat s1\]
        extends { AnyMatrix, Array2[\T, 0, s0, 0, s1\],
                  AdditiveGroup[\Matrix[\T,s0,s1\]\] }
        excludes { AnyMultiplicativeRing }
    opr +(self, v:Matrix[\T,s0,s1\]): Matrix[\T,s0,s1\] =
            ...
    opr -(self, v:Matrix[\T,s0,s1\]): Matrix[\T,s0,s1\] =
            ...
    opr -(self): Matrix[\T,s0,s1\] =
            ...
    scale(t1: T): Matrix[\T,s0,s1\] =
            ...
    mul[\ nat s2 \](other: Matrix[\T,s1,s2\]):
                      Matrix[\T,s0,s2\] =
            ...
    rmul(v: Vector[\T,s1\]): Vector[\T,s0\] =
            ...
    lmul(v: Vector[\T,s0\]): Vector[\T,s1\] =
            ...
    t(): Matrix[\T,s1,s0\] = ...
end
```

Figure 3.5 : `Matrix` trait declaration from FortressLibrary.fss, the actual calculation code is omitted.

First line shows how the declared `Matrix` trait can be used to call. Three variables must be presented when declaring the `Matrix`; `T`, which extends a number type, two non-negative integers s_0 and s_1. `T` is a number type like $\mathbb{Z}32$ or $\mathbb{R}64$, etc., which is the type of the variables in the matrix, thus the type of the matrix. s_0 and s_1 are the sizes of the matrix. Then trait `Matrix` extends several other traits. One of them is `Array2` trait, which generates a two dimensional array with the type `T` and indices start from zero to s_0 and zero to s_1. Then it excludes the trait `AnyMultiplicativeRing`. All the other extended or excluded traits are declared beforehand.

After the information on declaration, it is then expanded with operators and methods. Since `Matrix` trait extends `Array2` trait, it has all the operators and methods as its own. The operators and methods declared here in the `Matrix` trait are new especially defined for the `Matrix` trait. The already extended methods include; `replica`, `copy`, `put`, `get`, `t` and others. Note that `t` is also declared in the `Matrix` trait, this is an example of overloading of a method. It is declared twice because the previous declared method will return an error, since `Array2` trait requires two more variables like the starting index number of the array. It can be seen that the arrays must not begin with an index number zero. As can be seen that `t` method is the definition of the transpose method of the `Matrix` trait.

The three defined methods are for to define row major or column major type of the matrix. Note that `mul` defines a new type only requiring a new dimension value. The `t` function is getting the transpose of the matrix. There are just inner methods and operators that are defined in the `Matrix` type. Also, note that the calculations which are omitted here are presented in the Appendix.

After all the declarations in the trait are done, the operators and the methods can be defined. The operator plus and minus defined here. Operator plus requires the defined matrix and one more same sized matrix to add their elements together and returns a matrix with the same size. Operator minus in the first definition, requires two matrices same as the plus operator and returns the same size matrix, the elements of the second matrix extracted from the first matrix. Second line with minus operator, requires only the given matrix and returns a same sized matrix with its elements are negated; multiplied with `-1`. This operand is overloaded in the `Matrix` trait. It has already been overloaded since minus sign is already defined as the substraction sign for the several other number types, same as the plus sign. There can also be other operators be defined outside the trait block. One example is the matrix-vector multiplication.

```
opr DOT[\ T extends Number, nat n, nat m \]
    (me:Matrix[\T,n,m\], v:Vector[\T,m\]):Vector[\T,n\]
      = me.rmul(v)

opr juxtaposition[\ T extends Number, nat n, nat m \]
    (me:Matrix[\T,n,m\], v:Vector[\T,m\]):Vector[\T,n\]
      = me.rmul(v)
```

Figure 3.6 : Operators of matrix-vector multiplication.

The operators shown in Figure 3.6 are for matrix-vector multiplication. These are also overloaded for vector-matrix multiplication or even for the simple multiplication. The `DOT` operator renders as " • " and `juxtaposition` is basically the whitespace between the two operands. When these operands are present Fortress checks for the types, calculates and returns the results.

3.4 Implicit Parallelism

The developers of Fortress has foreseen the future of the computational power as multi cored. Steel stated that, "Even if we can build a Petascale machine, somebody will always want to put a bunch of them together" [2]. So the governing idea which shaped Fortress was to establish a path in the multi threaded parallelism. Therefore, the Fortress code is implicitly parallel.

This means every generator, every loop and tuple is parallel where possible. Since Fortress uses a multi threaded approach, it is designed to be parallel, to balance the work in between threads where possible. For example, the 'for' loop is a parallel operation, which will not always iterate in a strictly linear manner depending on the underlying software and hardware. If the programmer wants a sequential generator she has to strictly state when writing the loop. This also forces the programmer more on the problem itself rather than trying to figure out how to parallelize the loops.

All of the parallelization is defined in the library, so the programmar does not need to worry about where to use the parameters for parallelization. Also a process called work stealing which is defined in the libray and compiler code, again takes the load from the programmer. The decision on parallelization is purely based on the availability of the processses in the core. Also a library writer can decide which part of the code will be parallelized as shown in the previous sections.

3.4.1 Work Stealing

The way Fortress parallelizing threads is called Work Stealing. This scheme is based on the work of Blumofe and Leiserson (1999), that allows each process to maintain a work deque and steal an item from others if its deque becomes empty. The stealing occurs from the top of the main deque, so that no elements are added back to minimize the communication overhead. This way if there is an empty thread, it will steal work from the main thread, if a thread completes work, it will steal more work from the main thread. That way the work load will be divided dynamically. Figure 3.7 shows a figurative tree of work stealing, based on a parallelized for loop. Fortress uses Java multithreading to implement Fortress threads but not one to one. Work stealing strategy automatically balances the load [10].

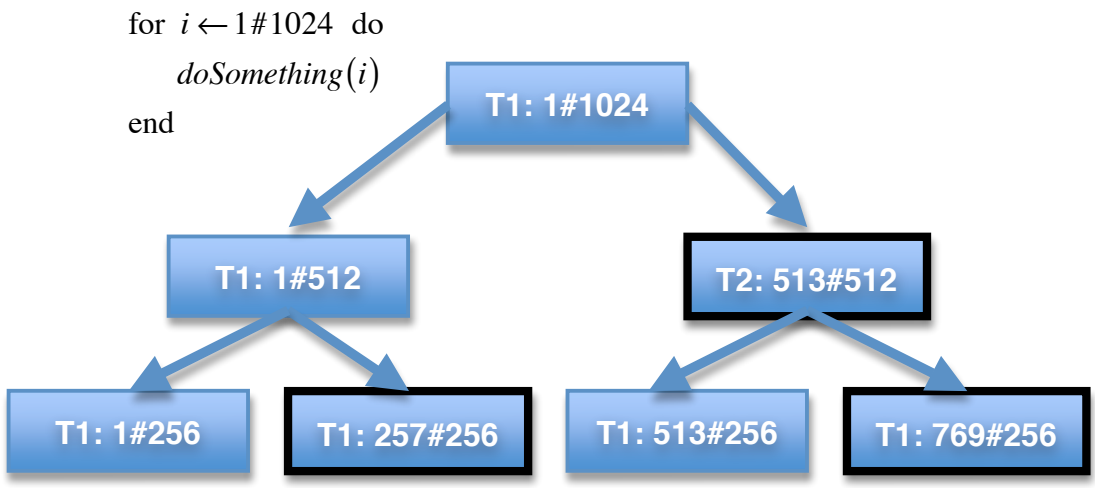

Figure 3.7 : Work stealing tree, thick bordered boxes show the work stealed.

3.5 Memory Model

Fortress programs are highly multithreaded by design; the language makes it easy to expose parallelism. However, without judicious use of synchronization constructs data races will occur and programs can behave in an unpredictable way. The memory model has two important functions:

1. Define a programming discipline for the use of data and synchronization, and describe the behavior of programs that obey this discipline.
2. Define the behavior of programs that do not obey the programming discipline. This constrains the optimizations that can be performed on Fortress programs.

The Fortress memory model has been written with several important guiding principles in mind. Violations of these principles must be taken as a flaw in the memory model specification rather than an opportunity to be exploited by the programmer or implementor. The most important principle is this: violations of the Fortress memory model must still respect the underlying data abstractions of the Fortress programming language. All data structures must be properly initialized before they can be read by another thread, and a program must not read values that were never written. “When a program fails, it must fail gracefully by throwing an exception” states Allen [2].

The second goal is: present a memory model which can be understood thoroughly by programmers and implementors. It must never be difficult to judge whether a particular program behavior is permitted by the model. Where possible, it must be possible to check that a program obeys the programming discipline.

The final goal of the Fortress memory model is to permit aggressive optimization of Fortress programs. A multiprocessor memory model can rule out optimizations that might be performed by a compiler for a uniprocessor. The Fortress memory model attempts to rule out as few behaviors as possible, what is more important is the attempts to make it easy to judge whether a particular optimization is permitted or not [2].

3.5.1 Data and Control Models

Data Model of Fortress is based on shared global address space. It assumes a global memory address space that is logically partitioned and a portion of it is local to each processor. The novelty of shared global address space is that the portions of the shared memory space may have an affinity with a particular thread, thereby exploiting locality of reference. Control model is multithreaded. It is controled by implicitly or explicitly created threads. Impilicitly created threads are generators and reduction operators. Explicitly created threads are basically declared by calling `spawn`, especially for the library writers. Basically by calling spawn you declare a thread in the spawn block.

3.5.2 Transactional Memory

As discussed earlier, all the access policies are written into the library not wired into the core. These can be managed and integrated into the type system. The access to shared variables are managed by;

- Atomic blocks; everything can be in atomic blocks. They run implicitly in separate threads and with `try` and `do` structure, they can explicitly retry the block.
- Read write atomicy, means all the read and write actions to the shared memory are automatically atomic, to avoid a data race condition.
- Lock free structure is maintained by the atomic blocks and there are no blocking and no deadlocks in the Fortress code. Existing of locks can manage parallelism but have problems. Also there occurs the problem of unknowability where there can be a confusion of, which locks protect which data. Let compiler and runtime manage the details. Fortress supports overlapping reads but not writes.
- With transactional memory, a thread can be spawned in a particular region. It is hope that Fortress will seldom need that, but the fact that the local data runs faster is the main issue of the transactional memory.

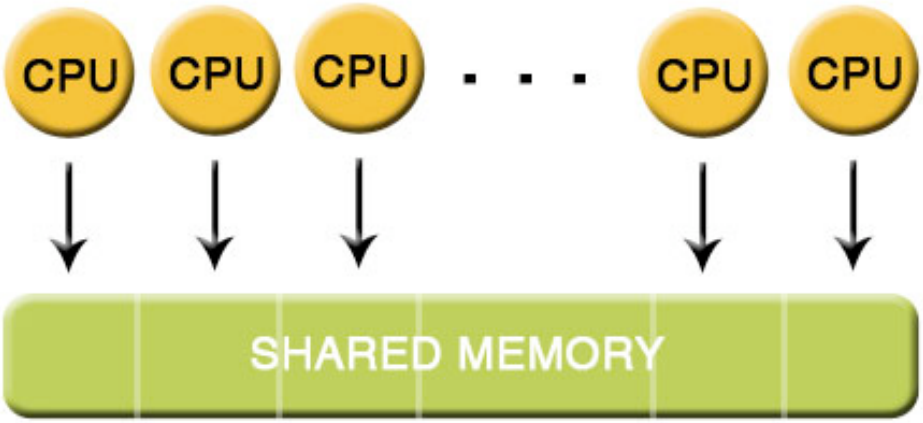

Figure 3.8 : The locality of the data in the memory.

The locality of the data as shown in the Figure 3.8, every CPU is in better reach of a chunk of data which is near to the core. By defining a region in the code. This can be assigned. A thread can be called to work in a particular region of the data, by calling **`spawn`** `x.region` **`do`** `f(x)` **`end`**. Thus the locality and the transactional memory helps, it must not be boldly used, as the core itself has to mange the parallelization, not the library writer.

3.6 Replaceable Components

All the above mentioned rules and the user functions are lead to one thing. The replaceability of the components. As it is dealt through out of this section, it is shown that everything is changeable and manageable within the library code. All the types and generators. The programmer can change all the blocks of the pre declared functions and types. This is a huge plus of the Fortress code.

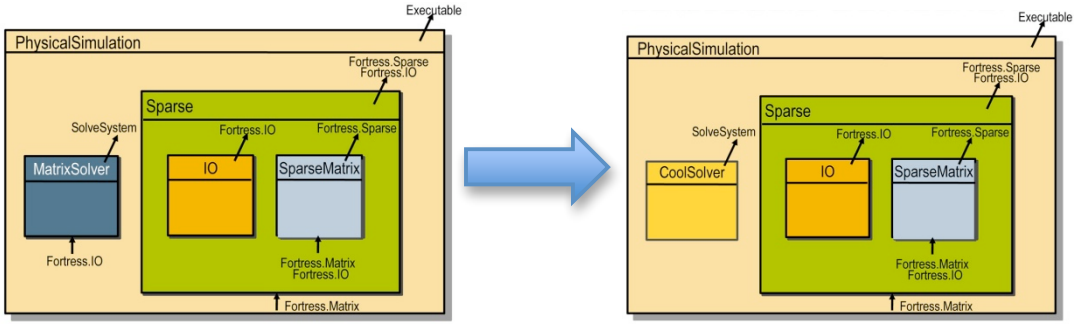

Figure 3.9 : An interchangable scheme of a predefined Matrix Solver.

The Figure 3.9 shows the a matrix solver that uses a library code, pre defined. If the programmer has different needs or simply has a better working matrix solver, she can change it with the current one without breaking the core or writing many overwrite functions. It is as easy as importing the library to the code. Basically, a programmer can change the how the for loop behaves and write another code and use it.

3.7 Growable Language

The outcome of all the mentioned functions and replaceability of the components is a changeable thus a growable language. Since so little is defined in the core compiler, the way the library code is used to define a basic type and simplicity of the language, the mathematical notation makes Fortress as an easy to write and maintain programming language. Everything can be designed according to your needs and changed. The outcome of this is, everybody who contributes code can change and manage Fortress, since it is also an open source project. A new functions can be added trough library coding or new and better functionality can be written to the basic functions. The programmer does not have to wait for the implementation of the code contributed to the core. It can easily be replaced and solved by library code.

3.8 What is missing from the core

There are many things planned for the release of the code, and many of them were presented in the public beta of the Fortress language. Since that will take much work and to work more on the main subjects, some functionality which are defined in the 1.0β of the code had taken out of the official 1.0 release, which was in April 2008. This functionality is not thrown away and resides in the core to be enabled in the future releases forming a path and a plan to the future release of the core. Here are some missing functionality;

- **Getters and setters.** This can be observed in the library code that when a trait is defined getter and setters are declared which reads the current information of a variable and sets to the trait. The use of this functionality adds more robustness to the library code writing.

- **Dimensions and units.** The main problem with the current computer languages is that when the programmer defines a variable, it is only known by the programmer the dimensions or the unit of the variable. With huge projects with many scientists working on it a dimension problem occurs several times. The Fortress code understands the difference between kilometers and miles and acts accordingly. This function has been implemented and tested but removed from the official release for the future releases.

- **Where clause.** This clause runs the code where outcome of the clause is true. This will mostly be used in library code where a type is defined and extended by other types but a special functionality needed in the region. This also helps the parallelization, since it can be define where and when to run the function as parallel or serial.

- **Parallel nested transactions.** This and many others like **Distributions** and **Coercion** definition is helps to improve the parallelization of the code. This improves how the code is parallelized and used. The more and high performing funcionality will be added to the future releases of the Fortress code and everyone can contribute.

4. THE PROBLEM AND THE CODE

To further the study its decided to choose a scientific problem and write a Fortress code to test it. The problem that has been chosen was three dimensional wave equation, it is an important partial differential equation that describes the propagation of waves with speed $\vec{v}$. The form below gives the wave equation in three-dimensional space where ∇^2 is the Laplacian.

$$\nabla^2\Psi = \frac{1}{v^2}\frac{\partial^2\Psi}{\partial t^2} \tag{4.1}$$

$$v^2\nabla^2\Psi = \Psi_{t1} \tag{4.2}$$

Equations (4.1) and (4.2) show the Three Dimensional (3D) Wave Equation with the formulation of Laplacian ∇^2

4.1 The Comparison

To compare the Fortress code we have chosen widely used C code. The purpose of this is to compare the writing process of the code. The above stated problem is written both in C and Fortress programming languages. An initial state given with a Gaussian formed wave to start from. Then the propagation of the initial Gaussian wave is calculated within the given intervals and a step size. Then in every predefined current state is printed to a text document to observe the results. Both codes are available in appendix.

Table 4.1: Initialization

C Version	Fortress Version

`int main() {`
`int i,j,k,n;`
`int nx,ny,nz;`
`int nsteps, printevery;`
`int tid;`
`double dx,dy,dz;`
`double ***x,***y,***z;`
`double ***f,***g;`
`double ***fp,***gp;`
`hrtime_t stime[TN],ftime[TN],master;`
 	
`x=(double***)malloc(sizeof(double **)*NN);`
`y=(double***)malloc(sizeof(double **)*NN);`
 	
`for (i=0;i<NN;i++) {`
`  x[i]=(double**)malloc(sizeof(double *)*NN);`
`  y[i]=(double**)malloc(sizeof(double *)*NN);`
 	
`  for (j=0;j<NN;j++) {`
`   x[i][j]=(double*)malloc(sizeof(doubl e )*NN);`
`   y[i][j]=(double*)malloc(sizeof(doubl e )*NN);`
`  }`
`}`
`// same for variables z, f, g, fp, gp`
	`nx:ZZ32:=50`
	`fout=1`
	
	`x = array3[\RR64,50,50,50\]()`
	`y = array3[\RR64,50,50,50\]()`
	
	`(* same for variables z, f, g, f', g' *)`

For better performance and for larger matrices, in C we have defined and reserved the memory space with pointers and malloc subroutine. In Fortress, there is a predefined array function to create three dimensional arrays with a given size. To create an array of three dimensional doubles all must be multiplied to reach the size. The array function in Fortress lets an array be initialized only by defining type, which is $\mathbb{R}64$, and size. Also note that you have to initialize all the variables before hand in C. In Fortress you can define later in the program. Note that variables can be defined as `f'` and `g'` which will render to f' and g' respectively. There are two ways to define a variable. To define an integer variable you can declare it by writing `nz:ZZ32:=50` or just declaring with an equal sign, `fout=1`, which will later be checked if you use in any numerical function. We will use to check if it is okay to print so no need to declare the variable type. Fortress allows the programmer not to strictly give a type, this process is mostly the job of a library code writer. As been explained in the previous section.

Table 4.2: For loop for the initial Gaussian wave

C Version	Fortress Version

`for (i=0; i<NX; i++) {`
`  for (j=0; j<NY; j++) {`
`  for (k=0; k<NZ; k++) {`
`  f[i][j][k] =`
`  0.2*exp(-(x[i][j][k]*x[i][j][k]`
`  + y[i][j][k]*y[i][j][k]`
`  + z[i][j][k]*z[i][j][k])`
`  /0.05 );`
`  g[i][j][k] = 0.0;`
`  }`
`  }`
`}`
	`g = array3[\RR64,50,50,50\]().fill(0)`
	
	`for (i,j,k) <- f.indices() do`
	`  f[i,j,k] := 0.2 e^( -((x[i,j,k])^2`
	`  + (y[i,j,k])^2 +`
	`  (z[i,j,k])^2)/0.05 )`
	`end`

In C, to acess all three dimensions three nested for loops must be used. Also to set all the elements of `g` variable, it is placed in the loops to optimize the use of the loops. In Fortress, when initializing an array, a fill function can be added. There are several other functions related to an array, which are already defined with the definition of an array in the library. A tuple is created by calling `f.indices()`, a function of an array which returns the index values of `f`, and the for loop runs accordingly. In C, this part is not parallelized because it is not declared in the code but in Fortress it is already parallelized. The code is also rendered to a portion shown in Figure 4.1.

$$\text{for } (i,j,k) \leftarrow f.indices() \text{ do}$$
$$f_{ijk} := 0.2e^{\left(-\left(x_{ijk}^2 + y_{ijk}^2 + z_{ijk}^2\right)/0.05\right)}$$
$$\text{end}$$

Figure 4.1 : The rendered Fortress code in Table 4.2

The definition of pragma comences the parallel part in the code. The main pragma block will be parallelized during the course of the code. The variables that are going to be used in the parallelized code block has to be defined strictly. This parallelization is basic and it only parallelizes the code equally with the given number of threads. The number of the threads that are going to be used are declared in the shell with the variable `OMP_NUM_THREADS`. There are several other control mechanizm for the OpenMP library in C. The number of the threads, the balancing of the work on threads and other control mechanizm are in the hands of the programmer and the programmer has to decide how to use that.

Table 4.3: Initialization of parallelization in C and main iteration loop

C Version	Fortress Version

`for (n=0; n<nsteps; n++) {`
`  // main for loop of iterations`
`  time1 = time1 + dt;`
 	
`  if (((n+1)%printevery)==0)`
`  printf("time = %5.3f \n",time1);`
 	
`#pragma omp parallel private(i,j,k,tid)`
`{ // main pragma block`
 	
`...`
 	
`#pragma omp for schedule(static)`
`for (i=0; i<NX; i++) {`
` for (j=0; j<NY; j++) {`
`  for (k=0; k<NZ; k++) {`
`  ...`
	`printset = {a | a <- 0#nsteps,`
	`  a MOD printevery = 0}`
	
	`for n <- seq(0#nsteps) do`
	`  (* main for loop of interations *)`
	`  time1 += dt`
	
	`  if ((n+1) IN printset) then`
	`  println("\ntime = " time1) end`
	
	`  ...`
	
	`  for (i,j,k) <- f.indices() do`
	`  ...`

In Fortress since every loop and generator is already parallelized, there is a need to declare a sequential loop explicitly. Therefore, a `seq()` function, must be used to force the generator defined in the for loop run in a sequential way. This is needed because there is data dependency of the previous iterations within the variables. The thread number can also be defined in the shell for the Fortress language by defining the variable `FORTRESS_THREADS`, but the programmer will not interfere with the parallelization. This is already defined in the library as it is been discussed in the previous sections.

There is a section where the time of the iteration is printed to the screen depending on the variable defined at the beginning. We can use the similar `if` section in both codes. However, to demonstrate the capabilities of Fortress we used a SET to calculate the printable times and check against the set when it is time to print. The above set and the if section is rendered shown in the Figure 4.2

$$\text{printset} = \left\{ a \middle| a \leftarrow 0 \# nsteps,\ a \texttt{ MOD } printevery = 0 \right\}$$

...

$$\text{if}\left((n+1) \in printset \right) \text{ then}$$

...

Figure 4.2 : Part of the Fortress code where a set is defined and printed if it is in the created set.

In general, code is cleaner than C. A programmer does not need to know the back end of the works, the Fortress core and the libraries do it for the programmer. Since three dimensional array is used, the calculations needed for loops. For the two dimensional arrays there is a Matrix type defined. Also Vector type for the one dimensional array. For the Matrix type, the code in Figure 4.1 it is easily be written as in Figure 4.3.

$$f := 0.2\ \mathrm{e}^{\left(-\left(x^2+y^2+z^2\right)/0.05\right)}$$

Figure 4.3 : The re-write of the code in Figure 4.1 as a Matrix type.

5. THE BENCHMARKING

There are two ways to run a Fortress code, first one is running a script, an interpreter way. Second one is compiling and then running. Compile mode does not deliver a stand alone application, but a compiled script that can be run again with the interpreter. A stand alone compiler is a work in progress and the current version is also experimental. So the runs are made using only the interpreter version.

All the tests are done at the Trakya machine at NCHPC, which is a 64 core SMP machine with 32 Itanium64 chips installed. As stated in previous section. The code run was done with different maximum threads allowed by setting the value of `FORTRESS_THREADS` in the shell. The code is tested with 2, 4, 8, 16, 32, and 64 threads allowed.

The first test is done with Fortress Threads set to eight, to fix the thread value. First step was to test several two-dimensional arrays with increasing sizes. Figure 5.1 shows the test done with several arrays. The tests are timed using both an internal time function and the time command of the system. Internal time function only times the area where the array is defined; here stated as the core time. The time command times the whole application including the initialization time. Later, the two-dimensional array test is runned again with a fill function, which fills the entire array with a given number.

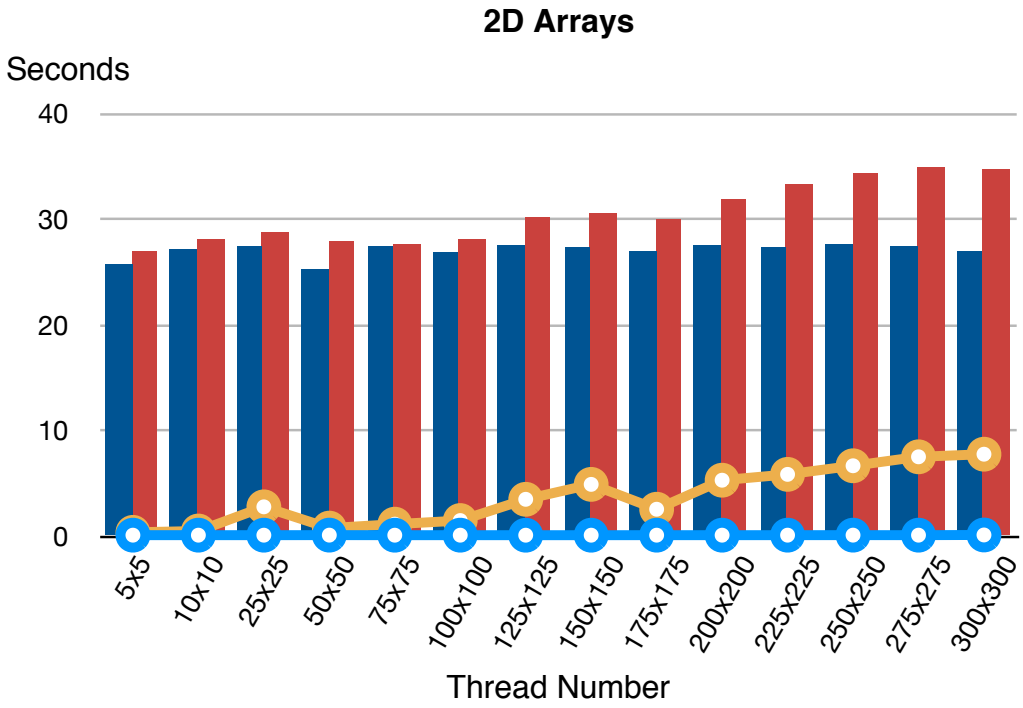

Figure 5.1 : Two-dimensional array tests

As can be observed in Figure 5.1 that the overall time is very high besides the actual time consumed by the definition of the arrays. The time consumed at the core is changing very little or there is no significant change. As the array sizes are growing, the time consumed is increasing. The test of two-dimensional arrays may continue, but to find the limits of the language the three-dimensional array tests are conducted. The same system is used and the following results shown in the Figure 5.2 are obtained.

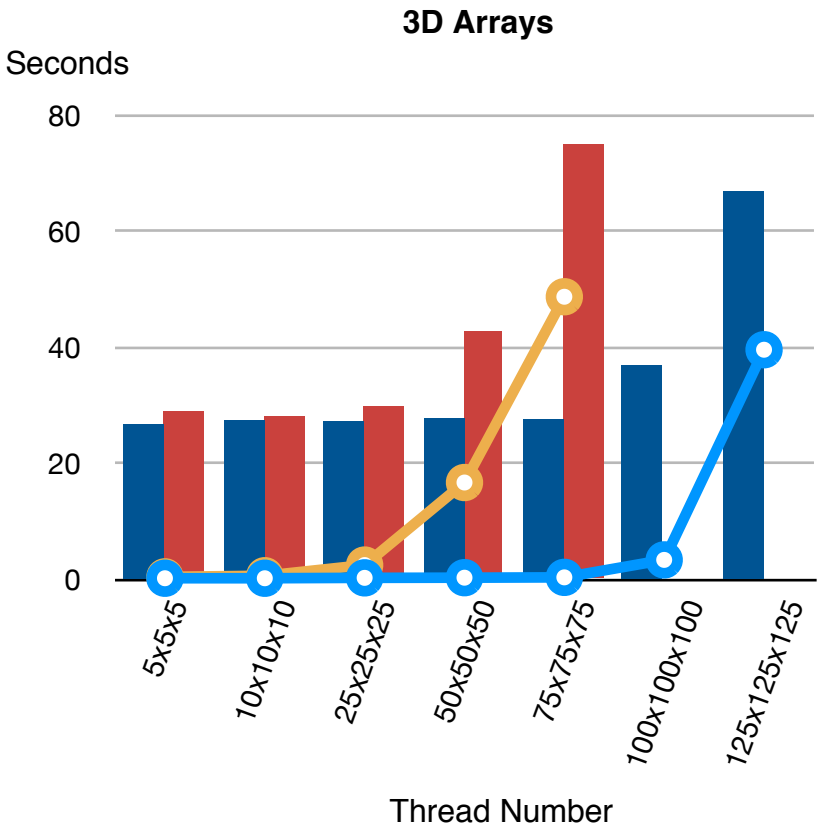

Figure 5.2 : Three-dimensional array tests

The three-dimensional arrays return memory overload error for sizes larger than 100x100x100 for the filled arrays and 150x150x150 for the empty arrays. It gets harder and takes longer just to fill the arrays. The exponential growth in time can be observed due to the addition of another dimension. Significant changes in the core time values occur as the array size reaches a limit value. This can be a limitation of the Java in the system.

The second test was to write an application both in C and Fortress, to compare the code writing process. For this purpose the Three Dimensional Wave Equation has chosen, a very important partial differential application, which can be parallelized in many steps.

First, the code is timed against the C code that has written with OpenMP, just to observe the current capabilities of the code. A real comparision between the two cannot be done since they both use different architectures. The informational run was done using 64 threads and C coded timed about 8.15 seconds versus 8.23 minutes of Fortress code. The run was three dimensional, 10x10x10 in size. This shows that as an initial code it is not very efficient as one can expect considering it was seconds against minutes.

The other tests were done with several array sizes and with several different threads allowed as stated above. The size of the array is 30x30 only using the first two dimensions. The code is timed with inner `nanoTime()` function from the beginning of declaring arrays to the end, where after the output is written to the document. Also the bash environment's `time` command is used to time the whole process, since there is an initial spin up time due to the compilation of the code and importing of the necessary libraries. It is observed that the spin up time is almost around 27 seconds. The measured times can be observed in the Figure 5.3.

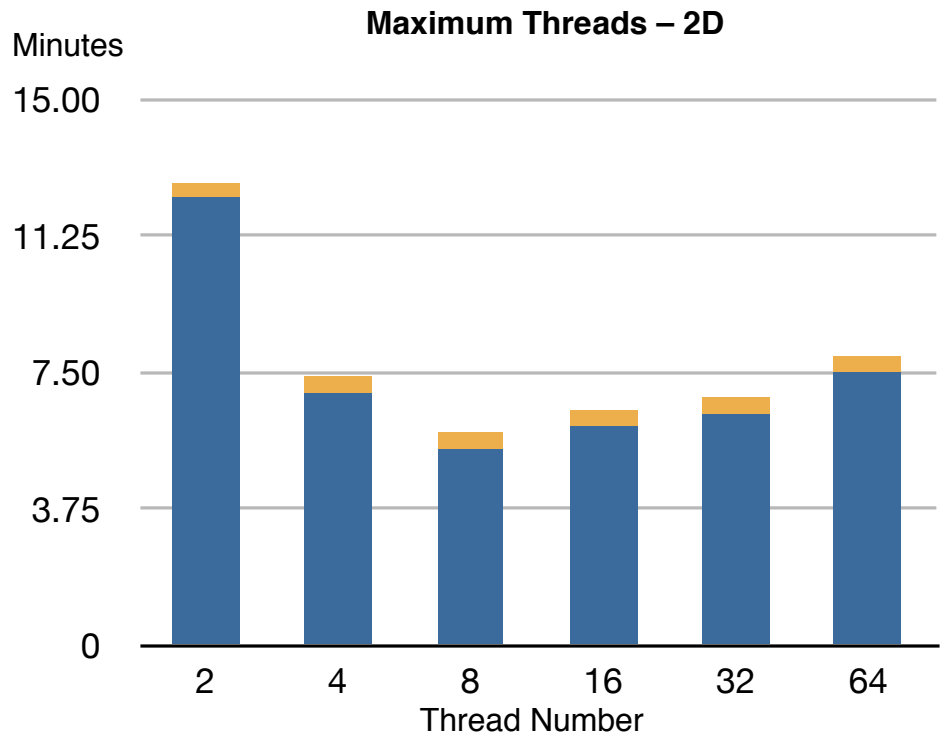

Figure 5.3 : Measured times of the Fortress code by maximum threads allowed.

As can be observed in the chart that the code runs in very long times. The times for the three dimensional matrix are as high as an hour to start. Also can be observed that as maximum threads allowed we do not see an exponential drop in the code timing. The communication time between threads increases as they may have wait each other. The optimum time been observed at the 8 threads mark.

The test of three dimensional to two dimensional is done with multiple threads as well. The outcome is represented in the Figure 5.4. The significant difference is three dimensional code runs more than five times the two dimensional code. The nested for loops or `(i,j,k) ← f.indices` does not in a healthy way. The second thing that can be observed is that the initial time spent for spin up is not changing due to the dimensional differences of the code. Comparing to the total amount of time spent the gap grows in percentage. The spin up time is caused by the Fortress, checking the code acceptability.

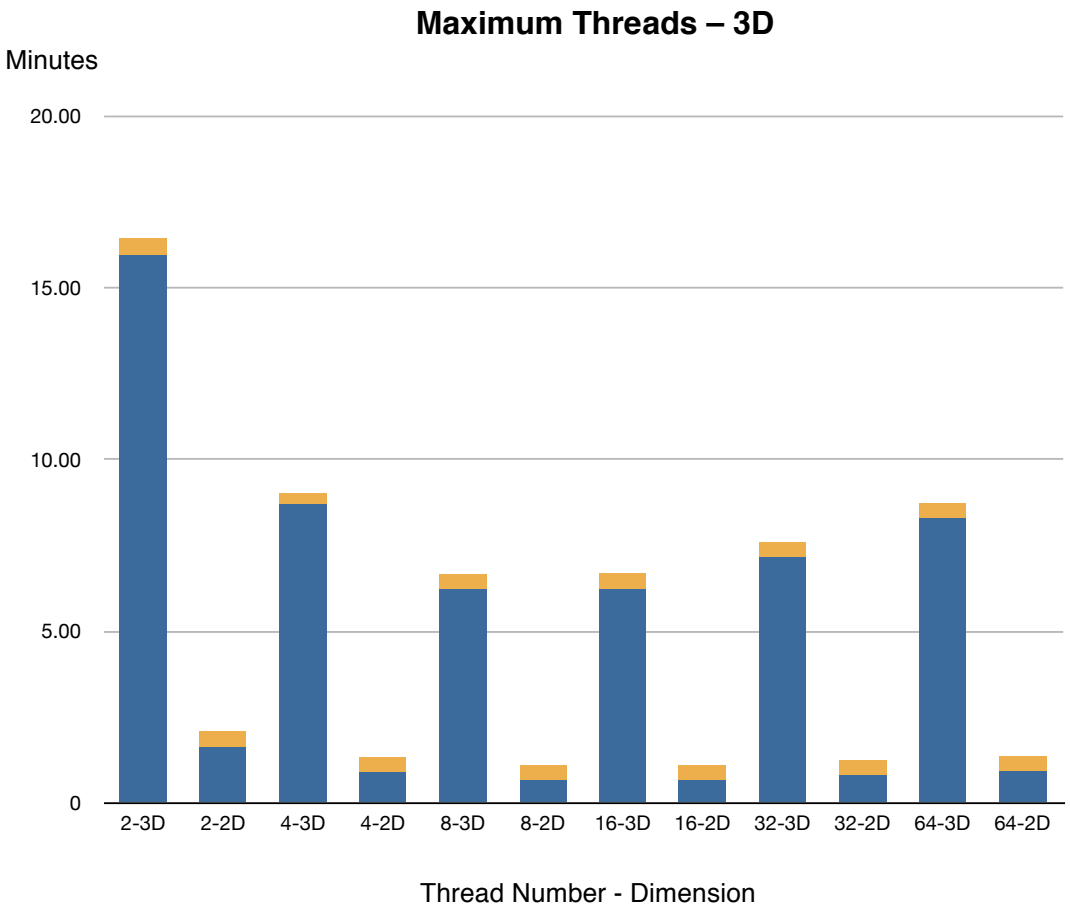

Figure 5.4 : Measured time of the Fortress Code three dimensional versus two dimensional with thread numbers.

6. THE CONCLUSION AND THE RECOMMENDATIONS

It is generally observed that the Fortress has an easy to write, mathematical notation oriented, scientist derived language. It is ease of writing, makes it very productive to the people who would like to take less time in coding and more time in creating the problem in the code. As the Fortress is a newly formed language, it has much way to grow. With current developments as a standalone compiler and more efficient libraries it can grow into a bigger more efficient language.

Fortress and other languages need more push in the way of developing. We may not write our next big application in Fortress but I believe future looks bright for Fortress and scientists in our department with accesible multi core machines can code few of their problems in Fortress and help the development of the language.

REFERENCES

[1] **Eadline D.,** What Are You Going To Do With Your FLOPS?, *Linux Magazine,* http://www.linux-mag.com/id/4088, accessed at April 2009

[2] **Allen E., et al.**, March 31, 2008. *The Fortress Language Specification*. Sun Microsystems, Inc.

[3] **URL-1** Programming Language Popularity, http://www.langpop.com/, accessed at May 2009

[4] **Wilkinson, B. and Allen M.**, 1999: Parallel Programming—Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall, New Jersey.

[5] **URL-2** DARPA Home, <*http://www.darpa.mil/index.html#learn*>, accessed at May 2009

[6] **URL-3** IPTO Home, <*http://www.darpa.mil/ipto*>, accessed at May 2009

[7] **URL-4** HPCS Home, <*http://www.darpa.mil/ipto/programs/hpcs/hpcs.asp*>, accessed at May 2009

[8] **URL-5** Chapel Home, <*http://chapel.cs.washington.edu/*>, accessed at May 2009

[9] **URL-6** X10 Home, <*http://x10-lang.org/*>, accessed at May 2009

[10] **Blumofe, R. D. and Leiserson, C. E.,** 1999: Scheduling Multithreaded Computations by Work Stealing. J. ACM 46(5), 720-748.

APPENDICES

APPENDIX A.1 : Trait Matrix code from the Fortress Library

```
trait Matrix[\T extends Number, nat s0, nat s1\]
        extends { AnyMatrix, Array2[\T, 0, s0, 0, s1\],
AdditiveGroup[\Matrix[\T,s0,s1\]\] }
        excludes { AnyMultiplicativeRing }
    opr +(self, v:Matrix[\T,s0,s1\]): Matrix[\T,s0,s1\] =
        ivmap[\T\](fn (i:(ZZ32,ZZ32),e:T):T => e + v.get(i))
    opr -(self, v:Matrix[\T,s0,s1\]): Matrix[\T,s0,s1\] =
        ivmap[\T\](fn (i:(ZZ32,ZZ32),e:T):T => e - v.get(i))
    opr -(self): Matrix[\T,s0,s1\] = map[\T\](fn (e:T):T => - e)
    scale(t1: T): Matrix[\T,s0,s1\] = map[\T\](fn (e:T):T => t1 e)
    mul[\ nat s2 \](other: Matrix[\T,s1,s2\]): Matrix[\T,s0,s2\] = do
        res = matrix[\T,s0,s2\]()
        mma(a:ZZ32,i:ZZ32,b:ZZ32,j:ZZ32,c:ZZ32,k:ZZ32):() =
            if k>=i AND k>=j then
              if k=1 then
                pr : T = get(a,b) other.get(b,c)
                (* If this were atomic, we could parallelize j-partition. *)
                res.put((a,c), res.get(a,c) + pr)
              else
                (k0,k1) = partition(k)
                (mma(a,i,b,j,c,k0),mma(a,i,b,j,c+k0,k1))
              end
            elif j>=i then
                (j0,j1) = partition(j)
                mma(a,i,b,j0,c,k)
                mma(a,i,b+j0,j1,c,k)
            else
                (i0,i1) = partition(i)
                (mma(a,i0,b,j,c,k),mma(a+i0,i1,b,j,c,k))
            end
        mm(a:ZZ32,i:ZZ32,b:ZZ32,j:ZZ32,c:ZZ32,k:ZZ32):() =
            if k>=i AND k>=j then
              if k=1 then
                res.put((a,c), get(a,b) other.get(b,c))
              else
                (k0,k1) = partition(k)
                (mm(a,i,b,j,c,k0),mm(a,i,b,j,c+k0,k1))
              end
            elif j>=i then
                (j0,j1) = partition(j)
                mm(a,i,b,j0,c,k)
                mma(a,i,b+j0,j1,c,k)
            else
                (i0,i1) = partition(i)
                (mm(a,i0,b,j,c,k),mm(a+i0,i1,b,j,c,k))
            end
        if s0=0 OR s1=0 OR s2=0 then
          res
        else
          mm(0,s0,0,s1,0,s2)
          res
        end
      end
    rmul(v: Vector[\T,s1\]): Vector[\T,s0\] = do
        row(i:ZZ32):T =
            SUM[(j,v_j)<-v.indexValuePairs] get(i,j) v_j
```

```
        vector[\T,s0\]().fill(row)
      end
    lmul(v: Vector[\T,s0\]): Vector[\T,s1\] = do
        col(i:ZZ32):T =
            SUM[(j,v_j)<-v.indexValuePairs] v_j get(j,i)
        vector[\T,s1\]().fill(col)
      end
    t(): Matrix[\T,s1,s0\] = TransposedMatrix[\T,s1,s0\](self)
end
```

APPENDIX A.2 : The runned C code.

```
/*
3D Wave Equation Problem
*/

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

#define NN 50

#define NX NN
#define NY NN
#define NZ NN

#define TN atoi(getenv("OMP_NUM_THREADS"))

#define FOUT 1

/*
 evolve a 3D scalar wave equation
 the two fields we'll evolve are:
 f - the field
 g - the time derivative of the field
*/

int main() {

  int i,j,k,n;

  int nx,ny,nz;

  int nsteps, printevery;

  int tid;

  double dx,dy,dz;
  double ***x,***y,***z;//x[NX][NY][NZ],y[NX][NY][NZ],z[NX][NY][NZ];
  double ***f,***g;//f[NX][NY][NZ],g[NX][NY][NZ];
  double ***fp,***gp;//fp[NX][NY][NZ],gp[NX][NY][NZ];
  hrtime_t stime[TN],ftime[TN],master;

  master=gethrtime();

   for (i=0;i<TN;i++){
      stime[i]=0.0;
      ftime[i]=0.0;
   }

  x=(double***)malloc(sizeof(double **)*NN);
  y=(double***)malloc(sizeof(double **)*NN);
  z=(double***)malloc(sizeof(double **)*NN);

  f=(double***)malloc(sizeof(double **)*NN);
  g=(double***)malloc(sizeof(double **)*NN);

  fp=(double***)malloc(sizeof(double **)*NN);
  gp=(double***)malloc(sizeof(double **)*NN);

  for (i=0;i<NN;i++) {
   x[i]=(double**)malloc(sizeof(double *)*NN);
   y[i]=(double**)malloc(sizeof(double *)*NN);
   z[i]=(double**)malloc(sizeof(double *)*NN);

   f[i]=(double**)malloc(sizeof(double *)*NN);
   g[i]=(double**)malloc(sizeof(double *)*NN);
```

```
 fp[i]=(double**)malloc(sizeof(double *)*NN);
 gp[i]=(double**)malloc(sizeof(double *)*NN);

 for (j=0;j<NN;j++) {
    x[i][j]=(double*)malloc(sizeof(double )*NN);
    y[i][j]=(double*)malloc(sizeof(double )*NN);
    z[i][j]=(double*)malloc(sizeof(double )*NN);

    f[i][j]=(double*)malloc(sizeof(double )*NN);
    g[i][j]=(double*)malloc(sizeof(double )*NN);

    fp[i][j]=(double*)malloc(sizeof(double )*NN);
    gp[i][j]=(double*)malloc(sizeof(double )*NN);
 }
}

double time1, dt;

FILE *fPtr;

/* Start Timing */

/* ******************** */
/* user defined variable */
nsteps = 100;
printevery = 10;
dt = 0.01;
/* in order for the system to be numerically dt < dx!!! */
/* ******************** */

time1 = 0.0;

printf("time = %5.3f \n",time1);

nx = NX;
ny = NY;
nz = NZ;

/* initialize the grid to run from -1 to 1 in each direction */
dx = 2.0/(nx-1);
dy = 2.0/(ny-1);
dz = 2.0/(nz-1);

/* initialize the grid to run from -1 to 1 in each direction */
for (i=0; i<NX; i++) {
  for (j=0; j<NY; j++) {
    for (k=0; k<NZ; k++) {
 x[i][j][k] = -1.0 + (i)*dx;
 y[i][j][k] = -1.0 + (j)*dy;
 z[i][j][k] = -1.0 + (k)*dz;
    }
  }
}

/* initialize the field to be a gaussian */
for (i=0; i<NX; i++) {
  for (j=0; j<NY; j++) {
    for (k=0; k<NZ; k++) {
 f[i][j][k] = 0.2*exp( - ( x[i][j][k]*x[i][j][k] +
            y[i][j][k]*y[i][j][k] +
            z[i][j][k]*z[i][j][k] )/0.05 );
 g[i][j][k] = 0.0;
    }
  }
}

/* output the initial data
   when there are an even number of points,
   pick a line closest to a coordinate axis */
if (FOUT==1) {
fPtr = fopen("wave3d.c.txt", "w");
```

```
  fprintf(fPtr,"initial data output is below\n");
  for (i=0; i<NX; i++) {
    fprintf(fPtr,"%5.3f %10.6e\n",x[i][NY/2][NZ/2],f[i][NY/2][NZ/2]);
  }
  fprintf(fPtr,"---------------------------------\n");
  }

  for (n=0; n<nsteps; n++) {

    time1 = time1 + dt;

    if (((n+1)%printevery)==0)
      printf("time = %5.3f \n",time1);

#pragma omp parallel private(i,j,k,tid)
   {
   tid = omp_get_thread_num();
    stime[tid]=gethrtime();
    /* predictor */
#pragma omp for schedule(static)
    for (i=0; i<NX; i++) {
      for (j=0; j<NY; j++) {
      for (k=0; k<NZ; k++) {
        fp[i][j][k] = f[i][j][k] + dt*g[i][j][k];
      }
      }
    }

    /* static boundaries */

    for (j=0; j<NY; j++) {
      for (k=0; k<NZ; k++) {
      gp[0][j][k] =  g[0][j][k];
      gp[NX-1][j][k] = g[NX-1][j][k];
      }
    }

    for (i=0; i<NX; i++) {
      for (k=0; k<NZ; k++) {
      gp[i][0][k] =  g[i][0][k];
      gp[i][NY-1][k] = g[i][NY-1][k];
      }
    }

    for (i=0; i<NX; i++) {
      for (j=0; j<NY; j++) {
      gp[i][j][0] =  g[i][j][0];
      gp[i][j][NZ-1] = g[i][j][NZ-1];
      }
    }


    /* use the predictor to update gp */
#pragma omp for schedule(static)
    for (i=1; i<NX-1; i++) {
      for (j=1; j<NY-1; j++) {
      for (k=1; k<NZ-1; k++) {
        gp[i][j][k] = g[i][j][k] +
         dt*(
         (fp[i+1][j][k]-2.0*fp[i][j][k]+fp[i-1][j][k])/dx/dx  +
         (fp[i][j+1][k]-2.0*fp[i][j][k]+fp[i][j-1][k])/dy/dy +
         (fp[i][j][k+1]-2.0*fp[i][j][k]+fp[i][j][k-1])/dz/dz );
      }
      }
    }

    /* use the average g's to update f */
#pragma omp for schedule(static)
    for (i=0; i<NX; i++) {
      for (j=0; j<NY; j++) {
      for (k=0; k<NZ; k++) {
        fp[i][j][k] = f[i][j][k] +
```

```
          dt*( 0.5*(g[i][j][k]+gp[i][j][k]));
        }
        }
      }

      /* now update all the variables */
#pragma omp for schedule(static)
      for (i=0; i<NX; i++) {
        for (j=0; j<NY; j++) {
        for (k=0; k<NZ; k++) {
          f[i][j][k] = fp[i][j][k];
          g[i][j][k] = gp[i][j][k];
        }
        }
      }
     ftime[tid]+=gethrtime()-stime[tid];
      }

      if (FOUT==1) {
        if (((n+1)%printevery)==0) {
           for (i=0; i<NX; i++) {
           fprintf(fPtr,"%5.3f %10.6e \n", x[i][NY/2][NZ/2], f[i][NY/2][NZ/2]);
           }
           fprintf(fPtr,"\n");
         }
      }
    }

    /* Stop Timing */
     printf("\n");
     for (i=0;i<TN;i++) {
        printf("tid %d = %lf \n",i,ftime[i]*1.0e-9);
     }

    //printf("\nVersion 3 Computation Time: %ld\n",(end-start));
     printf("\nMASTER= %lf\n",(gethrtime()-master)*1.0e-9);

     exit(0);
}
```

APPENDIX A.3 : Measured Fortress Code.

```
(*
*3D Wave Equation
*)

component tdwave50
import Constants.{e}
import File.{...}
import Set.{...}
export Executable

    (* Start Master Timing *)
    master = nanoTime()

    nx:ZZ32:=50
    fout=1

    x = array3[\RR64,50,50,50\]()
    y = array3[\RR64,50,50,50\]()
    z = array3[\RR64,50,50,50\]()

    f = array3[\RR64,50,50,50\]()
    g = array3[\RR64,50,50,50\]().fill(0)

    fp = array3[\RR64,50,50,50\]()
    gp = array3[\RR64,50,50,50\]()

    run(args:String...)=do
        (*outfile:FileWriteStream = FileWriteStream("waves.txt")*)

        (* Timing of the Wave
         * These variables won't change during program' *)
        (nsteps,printevery) = (200,100)

        dt:RR64:=0.01
        time1:RR64:= 0.00
        println("time = " time1)

        dx:RR64:=2.0/(nx-1)

        (* Initialize the grid to run from -1 to +1 in each direction *)
        for (i,j,k) <- x.indices() do
            x[i,j,k] := -1.0 + i dx
            y[i,j,k] := -1.0 + i dx
            z[i,j,k] := -1.0 + i dx
        end

        (* initialize the field to be a gaussian *)
        for (i,j,k) <- f.indices() do
            f[i,j,k] := 0.2 e^( -((x[i,j,k])^2 + (y[i,j,k])^2 +
(z[i,j,k])^2)/0.05 )
        end


        if (fout=1) then
            outfile.println( "initial data output is below" )
            for i <- seq(0#nx) do
                outfile.println((x[i,(nx/2),(nx/2)]).asString "  "
(f[i,(nx/2),(nx/2)]).asString)
                println ("file written")
            end
            outfile.println("---------------------------------")
        end

        printset:Set[\ZZ32\]
        printset = { a | a <- 0#nsteps , a MOD printevery = 0}

        for n <- seq(0#nsteps) do
```

```
            time1 += dt
            print (".")
            if ((n+1) IN printset) then
                println("\ntime = " time1)
            end

            (* Predictor *)
            for (i,j,k) <- f.indices() do
                fp[i,j,k] := f[i,j,k] + (dt g[i,j,k])
            end
            (* Static Boundaries *)
            for j <- 0#nx, k <- 0#nx do
                gp[0,j,k]:=g[0,j,k]
                gp[nx-1,j,k]:=g[nx-1,j,k]
            end
            for i <- 0#nx, k <- 0#nx do
                gp[i,0,k]:=g[i,0,k]
                gp[i,nx-1,k]:=g[i,nx-1,k]
            end
            for i <- 0#nx, j <- 0#nx do
                gp[i,j,0]:=g[i,j,0]
                gp[i,j,nx-1]:=g[i,j,nx-1]
            end

            (* Use the Predicator to update gp *)
            for i <- 1#(nx-2), j <- 1#(nx-2), k <- 1#(nx-2) do
                gp[i,j,k]:=g[i,j,k] +
                (dt TIMES
                (((fp[i+1,j,k]) + ((-2.0) (fp[i,j,k])) + (fp[i-1,j,k]))/(dx^2) +
                ((fp[i,j+1,k]) + ((-2.0) (fp[i,j,k])) + (fp[i,j-1,k]))/(dx^2) +
                ((fp[i,j,k+1]) + ((-2.0) (fp[i,j,k])) + (fp[i,j,k-1]))/(dx^2)))
            end

            print ("*")
            (* use the average g's to update f ' *)
            for (i,j,k) <- g.indices() do
                fp[i,j,k]:=f[i,j,k] + dt TIMES (0.5 (g[i,j,k]+gp[i,j,k]))
            end

            print (".")
            (* now update all the variables *)
            for (i,j,k) <- g.indices() do
                f[i,j,k]:=fp[i,j,k]
                g[i,j,k]:=gp[i,j,k]
            end


            if (fout=1) then
                if (((n+1) MOD printevery) = 0) then
                    for i <- seq(0#nx) do
                        outfile.println((x[i,(nx/2),(nx/2)]).asString "  "
(f[i,(nx/2),(nx/2)]).asString)
                        println(" file written ")
                    end
                end
            end
        end

        (* Stop Master Time*)
        mastertime:RR64:=(nanoTime()-master) 10^(-9)
        println("\nMaster = " mastertime)
        outfile.close()
    end
end
```

CURRICULUM VITÆ

Candidate's full name:	Erdem ÜNEY
Place and date of birth:	Ankara, 17.09.1980
Permanent Address:	Doğa Sokağı Doğakent Sitesi A-1 Blok D:1 Ulus, 34340, İstanbul - TÜRKİYE
Universities and Colleges attended:	Şişli Terakki High School, 1998
	İstanbul Technical University, Department of Meteorology Enginering, 2005
Publications:	Erdem ÜNEY and H. Nüzhet DALFES, August 2009: A New Parallel Programing Language Fortress: Features and Applications. *ICSCCW 2009*.