Parallelization and Optimization of a Tax and Benefits Model

Thomas McClintock
s1150361

August 23, 2012

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2012
Abstract

This MSc project seeks to improve the performance of a simulation written by the Institute for Fiscal Studies. The simulation solves the optimal decision problem for 5000 women over the course of their lifetime in order to test the effects of current and proposed tax and benefits policies. We ran the simulation on the Eddie machine maintained by the ECDF. The original runtime of the program was 78 seconds and we were able to reduce this to 10 seconds when running on twelve cores. Parallelization was achieved using OpenMP directives. Further performance gains through parallelization are possible but were difficult to achieve. In this paper we discuss the economic theory behind the model, the difficulties in parallelizing our program, our results, and future work.
Contents

1 Introduction ................................................. 1
   1.1 Motivation ............................................. 2
   1.2 Report Structure ........................................ 2

2 Background .................................................. 4
   2.1 Utility Function .......................................... 4
      2.1.1 Maximizing Utility Using Lagrange Multipliers ........ 5
   2.2 Optimal Decision Problem ............................... 7
      2.2.1 Stochastic Variables ................................. 9
   2.3 Method of Moments ..................................... 9
   2.4 Dynamic Programming .................................. 10
      2.4.1 Parallelism in Dynamic Programming ................ 11

3 Program Structure ......................................... 13
   3.1 Main Subroutine ....................................... 14
   3.2 Three Sections: Environment Set Up, Simulation of Individuals, and Statistics .......... 15
      3.2.1 Environment Set Up .................................. 15
      3.2.2 Simulation ........................................... 16
      3.2.3 Calculation of Moments ............................. 17
   3.3 Porting Our Program .................................... 17

4 Parallelizing our Model .................................. 18
   4.1 Collapsing Nested Loops ............................... 19
   4.2 Module Variables ...................................... 21
      4.2.1 THREADPRIVATE vs PRIVATE ........................ 21
   4.3 Final Parallel Version ................................ 23
   4.4 Running on Eddie ....................................... 25
   4.5 Timing Difficulties and Profiling ..................... 26

5 Results ..................................................... 28
   5.1 Thread Number ......................................... 28
   5.2 Without COLLAPSE .................................... 29

6 Analysis .................................................... 31
## List of Figures

5.1 Speed up of the parallel region and the entire program. Overhead increased as we increased the number of threads. ........................................... 29

5.2 Speed up of the entire collapsed parallel region compared to the fifth and sixth loops only. ................................................................. 30

6.1 Trip count of parallel regions. Parallel regions have trip counts from 6 to 30 in increments of 6, and the number of regions with these trip counts varies. ................................................................................. 32

6.2 Predicted speed up generated from the trip count data. Some numbers of threads are not factors of any of the trip counts and we should expect to see no speed up at that number. ................................................. 34

6.3 Predicted speed up of the entire program compared to just the fifth and sixth loops. ................................................................. 35

6.4 Predicted speed up of just the fifth and sixth loops compared to their measured speed ups. ................................................................. 36

6.5 The dynamic schedule compared to the static schedule and the predicted speed up. ................................................................. 37

7.1 The predicted speed up using COLLAPSE(3) compared to COLLAPSE(2). 39
Acknowledgements

First and foremost I would like to thank my EPCC advisor David Henty. Coming into the program with a physics background I was wholly oblivious to how powerful a tool HPC can be, and David truly captivated me back in the fall during his MPP course. I am extremely grateful that I had the chance to work with him on this project, and I advise any future MSc students beginning projects with David to not hesitate to go to him with any questions. I would also like to thank Jonathan Shaw of the IFS for sharing his research and laying the groundwork for the project. It quickly became clear to me that the scope of the model was immense, however reading papers published by his group quickly convinced me of the importance of the research they conduct. I consider myself lucky to work with a code that is doing research that ultimately impacts the policy decisions of the UK. Lastly, I’d like to thank my fellow students in the MSc. You have all been an inspiration and a motivation, and so often serve to keep my spirits up during the long hours we have spent working together. I wish you all the best of luck as you begin careers in HPC, software development, academia and plenty of other fields, and I am excited to hear about the great things you will accomplish in the next few years.
Chapter 1

Introduction

The Institute for Fiscal Studies sets out to understand the impact of government policies on individuals, families, and businesses. They pursue this goal by modeling the effect of current policies, and any proposed changes, on individuals and the workforce in general. By combining population data over the past several decades with sound economic theories the IFS has been able to advise the UK government on policy that directly affects the workforce in this country. The model in this project specifically focuses on the ability for women to work depending on their family situation, the state of the job market, and most importantly the tax and benefits policies in place.

Tax and benefits policies in the UK serve two purposes. First, they provide a safety net for those that need assistance or are unable to work for themselves. Some tax and benefits policies offer tax breaks to the unemployed while others offer incentives to work (Blundell 2011). The IFS model investigated in this project deals with the Working Tax Credit (WTC) and the Child Tax Credit (CTC) policies, as well as the policy that these replaced called the Working Families Tax Credit (WFTC). The WTC provides subsidies for low-wage workers designed to improve work incentives, while the CTC gives awards based on family composition. These policies replaced the WFTC in 2003 which was effectively a combination of the two but was in general less generous.

An individual’s decision to work depends not only on the tax and benefits policies in place, but also on their family situation and the state of the job market. In our model individuals are assumed to act optimally, however their environment is often subject to stochastic variables. A woman may have a child unexpectedly, couples form and dissolve, and average wages fluctuate. For this reason, the parameters that the IFS are interested in cannot be simply calculated, but must be generated statistically. The IFS has built their model to determine the tendency of women to seek employment. This information will help the UK government make two assessments: whether a tax and benefits policy is providing assistance to women, and also if this assistance favors certain groups over others. Some policies may favor young people while others might do more for those with children, while still others might favor the more educated. It is not necessarily a bad thing that a policy affects one group more than others, but it is
important to establish that a policy has any effect at all.

1.1 Motivation

This dissertation project attempts to improve the runtime of the IFS model of the effects of tax and benefits policies on the tendency of women to seek employment in the UK. The current model runs too slowly to effectively provide results in a timely manner. Researchers using the model at the IFS plan to run the code on shared memory machines, hence the primary goal of this project is to improve the runtime of the model by implementing OpenMP.

The IFS model is not one that fits a standard design pattern. As we will show, the optimal decision problem requires the use of dynamic programming in order to prevent the exponential growth of the problem size. While the simulation of individuals in this model can easily be parallelized, it is not the simulation itself that takes up a majority of the runtime. In fact, it is the preparation and set up of the simulation that comprises roughly 98% of the total runtime. In this report we discuss the difficulties in parallelizing this section of the code, and the results of our eventual success.

1.2 Report Structure

We begin with a discussion of the problem at hand and cover topics including utility functions, the optimal decision problem, and the method of moments, all of which are found in economics texts. Next we cover the principle of dynamic programming and how it relates to the optimal decision problem. We also discuss why dynamic programming algorithms are difficult to parallelize.

In chapter three we dissect the structure of the program provided by the IFS. By looking at all parts of the program, we highlight performance critical areas and how the structure of the program relates to the theory.

The next chapter covers the attempts to implement OpenMP in performance critical areas. The provided program is extremely complex and requires a familiarity with both economics and statistics in order to fully grasp its algorithms. This complexity makes errors in the parallel version of the code very difficult to analyze and repair.

The next two chapters discuss the results of the final, parallelized version of the code. We demonstrate that the parallel version is correct and show the gains in performance by running the program on a shared memory machine.

In chapter seven we discuss the possibility of using MPI in this program as well as other directions this project may take in the future. As we will show, the IFS model requires large data structures in order to simulate individuals’ life choices, and it may be possible
to break these data structures up across multiple processors in order to optimize memory use.

Finally, we conclude with some closing remarks about the project as a whole and the prospective benefits of the improved IFS model.

All data gathered is provided in appendices at the end of the report.
Chapter 2

Background

In order to predict the effect of tax and benefits policies the IFS model touches on a wide range of topics in economics. The optimal decision problem found in microeconomics texts stems from dealing utility functions for individuals or groups of people, while the method of moments comes from econometrics and relies heavily on the use of simulation and statistics. For now, we will focus on defining basic terms before explaining these more daunting problems. After a discussion of the terminology found in the IFS model, we delve into dynamic programming. Dynamic programming is a problem solving technique that prevents the amount of calculations from growing exponentially. Dynamic programming applies directly to the optimal decision problem, and is found in almost all parts of the program provided by the IFS. While we had previously no exposure to the field of economics in general, we relied on multiple sources in order to fully grasp the algorithms employed in our model. These texts include Dynamic Economics by Adda and Cooper, Mathematics for Economists by Pemberton and Rau, Microeconomic Theory: Basic Principles and Extensions by Walter Nicholson, and A Guide to Econometrics by Peter Kennedy.

2.1 Utility Function

The most important quantity that we will deal with is the utility function. At its core, the utility is a characterization of individual’s preferences (Nicholson 66). Utility refers to the overall satisfaction of an individual, and is a dimensionless quantity. Put more quantitatively, it is a ranking of an individual’s situation, and can be expressed mathematically by the equation

\[ Utility = U(X_1, X_2, \ldots, X_n). \]  

(2.1)

Here, the utility is expressed as a consumption of goods, where the \( X \)'s refer to the quantities of the goods chosen, and \( U \) is known as the utility function. The term goods
is used loosely here. Not only does it refer to the consumption of physical commodities, but also by psychological attitudes, peer group pressures, personal experiences, and the general cultural environment (Nicholson 67).

While economists often narrow down the focus of the utility function in order to deal with known quantities, the IFS model is significant because of its scope. Not only does it incorporate the cost of living for families and individuals, but also on the utility of work experience, levels of education, and familial circumstances. The IFS model deals with a special type of utility function, where an individual seeks to maximize their utility with a budget constraint in place. In this case, the quantity of goods depends on the price $P$ of each commodity and an individual's income $I$. Thus, the utility is now given by

$$ Utility = U[X_1(P_1, P_2, \ldots, P_n, I), X_2(P_1, P_2, \ldots, P_n, I), \ldots, X_n(P_1, P_2, \ldots, P_n, I)]. $$

(2.2)

The utility function now indirectly depends on the price of goods and the income, and an individual must consider both when maximizing their utility. Instead of using this bulky notation, economists often prefer to use the indirect utility function, given by

$$ Indirect Utility Function = IU(P_1, P_2, \ldots, P_n, I). $$

(2.3)

By using the indirect utility function we deal only with prices and income instead of the quantity of commodities.

### 2.1.1 Maximizing Utility Using Lagrange Multipliers

Oftentimes economists want to model an individual by maximize the utility function of that individual with respect to the commodities they deal with. As we will discuss in the next section, it is assumed that individuals always seek to optimize their utility, however we must first show how we can calculate what the maximum utility is and the amount of commodities an individual will consume to achieve that level of utility.

As seen in Eq. 2.1 an individual, let’s call him David, must choose the quantity of each good, $X_i$, so as to maximize the utility $U$. Now suppose David’s budget is constrained by

$$ \sum_{i=1}^{n} X_i P_i = I $$

(2.4)

where $P_i$ is the price of good $X_i$. We assume that in maximizing his utility David cannot consume a negative amount of a commodity and he will always spend his entire income (Pemberton and Rau 330).
Economists refer to Eq. 2.4 as a constraint on Eq. 2.1. In order to maximize the utility we now introduce the Lagrangian function as follows:

\[ L(X, \lambda) = U(X) - \lambda \left( \sum_{i=1}^{n} X_i P_i - I \right). \]  

(2.5)

Here \( \lambda \) is known as a Lagrange multiplier. When solving any maximization problem the Lagrangian function is introduced in order to more easily find the solution. Pemberton and Rau state the relationship as such:

If \((x^*, y^*)\) is a solution of the constrained maximization problem maximize \( f(x, y) \) subject to the constraint \( g(x, y) = 0 \), then there is a real number \( \lambda \) such that the Lagrangian \( L \) has a critical point at \((x^*, y^*, \lambda^*)\).

Here \( g(x, y) \) is Eq.2.4. In order to find the critical point of \( L \) we see that the conditions are

\[ \frac{\partial L}{\partial X_i} = 0, \quad \frac{\partial L}{\partial \lambda} = 0 \]  

(2.6)

where \( i \) ranges from 1 to \( n \). Thus there are \( n + 1 \) first order conditions; one for each \( X \) and one for \( \lambda \). Performing these partial derivatives, we see that the point of maximum utility occurs when

\[ \frac{\partial U}{\partial X_i} = \lambda P_i, \quad \frac{\partial U}{\partial \lambda} = \sum_{i=1}^{n} X_i P_i - I. \]  

(2.7)

Taking the partial derivative of \( L \) with respect to \( \lambda \) just returns the constraint. The \( n \) first order conditions involving the partial derivative of \( X \) along with the constraint can then be solved in order to find the values of \( X \) that yield maximum utility along with the Lagrangian multiplier \( \lambda \). It is important that the left hand side of the equations seen in 2.7 have a special name. The partial derivative of utility with respect to a good is known as the marginal utility of that good. The marginal utility of \( X_i \) is often written as \( MU_{X_i} \) (Nicholson 79). One can interpret the marginal utility to mean the amount of utility gained by consuming one more unit of good \( X_i \). This may seem confusing since a differential assumes continuous variables, however it is convention none the less. This is an important term to remember, as it is central to the optimal decision problem discussed in the following section.

The optimal levels of each \( X_i \) obviously depend on the price of each commodity as well as \( I \). This relation can be written as

\[ X_i = f_i(P_1, P_2, ..., P_n, I) \]  

(2.8)
where \( f_i \) is known as the demand function for \( X_i \). With the demand functions in hand we can finally rewrite the utility function \( U \) in terms of prices and income in order to obtain the indirect utility function \( IU(P_1, P_2, \ldots, P_n, I) \).

### 2.2 Optimal Decision Problem

The IFS model deals with the Optimal Decision Problem. In this problem, an individual’s utility is maximized over the course of their lifetime, and in the case of the IFS model this means that an individual maximizes their utility from one year to the next. Formally, to find the total utility over a person’s lifetime, \( V \), we can write

\[
Total\ Utility = \sum_{i=1}^{n} \left( V(X_{i,0}) \right) = \sum_{i=1}^{n} \left( \sum_{t=0}^{T} \max \left( \beta^{t-1}U(X_{i,t}) \right) \right)
\]  

(2.9)

where \( 0 \leq t \leq T \) and the individual begins by consuming \( X_{i,0} \) of commodity \( i \) at \( t = 0 \). \( \beta \) is known as the discount factor and is \( 0 \leq \beta \leq 1 \). This corresponds to utility diminishing over time. In other words, the maximum utility an individual can attain at the end of their lifetime is less than during their younger years. As seen in Eq. 2.9, from one time period to the next an individual must maximize their utility based on how much they consume in that time period, thus we designate how much of commodity \( i \) is consumed in one time period by \( X_{i,t} \). The utility function is subject to the familiar constraint

\[
\sum_{i=1}^{n} X_{i,t} P_i = I.
\]  

(2.10)

Clearly, for each individual time we can solve for the maximum utility just as we did in the previous section. Proceeding normally we arrive at the same conclusion as Eq. 2.7, however we see something very interesting after dividing by \( P_t \):

\[
\lambda = \frac{\beta^{t-1}MU_{X_{i,t}}}{P_i} = \frac{\beta^t MU_{X_{i,t+1}}}{P_i}.
\]  

(2.11)

After dropping the cumbersome \( i \) notation and a bit of algebra we arrive at

\[
MU_{X_t} = \beta MU_{X_{t+1}}.
\]  

(2.12)

The marginal utility between two time periods is the same up to a coefficient. This means that given this static situation, with prices remaining fixed and no change in income, then the optimal level of consumption during one time period is directly proportional to the optimal level of consumption in the next. Furthermore, in this static case this relation is the same between any subsequent time periods.
In order to take this a step further we have to invoke the Principle of Optimality. First written down by Richard Bellman in 1954, the Principle of Optimality states:

An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.

In terms of our model, this means that given some level of consumption in the current time period, the level of consumption in all subsequent time periods must be optimized with regards to the consumption today. This allows us to recast our equation for total lifetime utility, Eq. 2.9, into something much more useful to us.

To begin this process, let us use the result from Eq. 2.12 to write

\[
\text{Total Utility} = V(X_0) = \max \left( U(X_0) \right) + \sum_{t=1}^{T} \max \left( \beta^{t-1} U(X_t) \right) 
\] (2.13)

It may appear that the problem is now uglier since we have torn today’s utility out of the sum, however we should recognize that the sum on the right is also the total lifetime utility, but now multiplied by \( \beta \) and beginning with commodities \( X_1 \) at \( t = 1 \). Thus, we can drop the \( t \) and write

\[
V(X_0) = \max \left( U(X_0) \right) + \beta V(X_1).
\] (2.14)

We are now left with a recursive function for \( V \) beginning with commodities \( X_0 \) in terms of the maximized utility of those commodities and the maximized utility for the next time period. Going one step further we rewrite \( X_0 \) and \( X_1 \) to simply describe the state today, \( X \), and the state tomorrow \( X' \). We now have the equation

\[
V(X) = \max \left( U(X) \right) + \beta V(X').
\] (2.15)

This is known as a functional equation or a Bellman Equation. The most important observation we can make about Eq. 2.15 is that the total lifetime utility beginning with the commodities tomorrow, \( V(X') \), depends only on the total lifetime utility beginning with the commodities today \( V(X) \). We could, in fact, reverse this statement and say the reverse: \( V(X) \) only depends on \( V(X') \). Either way, the total lifetime utility today does not depend on \( V \) for all future time periods, only the next one. In this sense, we have saved ourselves some work. If we know \( V(X') \) then we just add on \( \max(U) \) and we have \( V(X) \), rather then performing the entire sum in Eq. 2.9. This is the fundamental principle of Dynamic Programming, and we will elaborate on this topic soon.

The link between Eq. 2.15 and the IFS model is in the relation between \( X \) and \( X' \). The link between these two levels of consumption is known as the policy function, that is

\[
X' = \phi(X).
\] (2.16)
This corresponds to the changing value of wealth and goods over time. Inflation and deflation cause the purchasing power of cash to change. Buying a 2012 Toyota Prius today gives you maximum utility, however buying the same car, now used, in five years means you will have less utility. The IFS is interested in studying the effect of policy functions relating to tax and benefits policies. For instance, the utility gained from collecting unemployment benefits may, in some individual’s circumstances, outweigh the utility gained from working for the minimum wage. The policy set by the government is the policy function that relates $X$ and $X'$ in these equations, and by solving the optimal decision problem, the IFS can determine how policies affect how much utility an individual can hope to obtain.

2.2.1 Stochastic Variables

The beauty of the Bellman equation is that uncertainty can be added to the function (Adda and Cooper 20). A natural source of uncertainty for a consumer is the individual’s “appetite” for a given commodity. Let’s say that in any given time period, the appetite of an individual today is $\varepsilon$, a randomly determined variable that can take on the values of $\varepsilon_h$ or $\varepsilon_l$ for a high or low appetite respectively. Without going into full detail, we can write down the new Bellman equation to be

$$V(X) = \max \left( \varepsilon U(X) \right) + \beta \mathbb{E}_{\varepsilon'}|\varepsilon V(X'). \quad (2.17)$$

Here, $\mathbb{E}_{\varepsilon'}|\varepsilon$ is known as the transition function from the appetite today, $\varepsilon$, to the appetite tomorrow, $\varepsilon'$. In this example we already stated that $\varepsilon$ takes on a random value, however we may be able to determine $\varepsilon'$ from $\varepsilon$ using a Markov chain or some other process. While we only deal with variables within a finite set, Bertsekas (1976, sec. 2.1) deals with a more general treatment of uncertainty.

In the IFS model, uncertainty manifests itself as stochastic variables that determine how an individual’s life circumstances may change. Economists often refer to these variables as taste shocks, and in the IFS model they include marriage, divorce, the arrival of children, job loss, and a changing labor supply, just to name a few. The Bellman equation for the IFS model written down would look similar to Eq. 2.17, but would be much more complex as there are many more stochastic variables.

2.3 Method of Moments

Oftentimes economists are concerned with establishing the relationship between variables in a model. The IFS model for instance investigates the relationship between the tendency of women to join the labor force depending on the tax and benefits policies in place. Oftentimes this relationship is difficult to determine and there may be a measure
of uncertainty involved. Associated with these types of relationships are unknown con-
stants, called parameters (Kennedy 3). For example, the relationship between the utility
$U$ and the prices of goods $P$ may be

$$ U = AP + B + \varepsilon $$

(2.18)

where $A$ and $B$ are parameters and $\varepsilon$ is the uncertainty in the relationship. In order
to learn the values of these parameters, economists employ various techniques and al-
gorithms depending on the problem at hand. In the IFS model, the parameters are
determined using the Method of Moments (MoM) technique.

MoM is an iterative technique in which we estimate the parameters in the relationship
between variables in a model. The steps are: guess the parameters, measure the mo-
mants, use the resulting moments in order to make a better estimate of the parameters.
Moments are metrics used to describe the shape of a distribution of a set of points and
include the mean, variance, skewness, and kurtosis of a distribution. We will not delve
into the nitty gritty statistics behind the MoM, however it serves an important role in
the IFS model: it links the measurable variables, such as the tax and benefits policies, to
the unmeasurable variables, such as the tendency for an individual to seek employment.

### 2.4 Dynamic Programming

Equations of the form 2.15 are known as functional equations or Bellman equations.
This means that the solution for a given time period depends on the solution to the
previous time period. The unknown in a Bellman equation is the function itself, and
the purpose of working with these equations is to find a function $V(X)$ that satisfies
the equation for all $X$ (Adda and Cooper 17). Dynamic programming is the term used
to differentiate between Eq. 2.15 and Eq. 2.14. The former is our Bellman equation,
while in the latter equation the total utility for time $t = 0$ explicitly depends on the total
utility for all other time periods since $V(X_1)$ will follow the same equation. As Richard
Bellman pointed out in 1958 when he was developing dynamic programming, leaving
these dependencies within a program will quickly result in a computer running out of
memory.

In order to more clearly demonstrate the advantages of dynamic programming, let us
look at a simple example of its use: finding the $n$th Fibonacci number. Mathematically,
the Fibonacci numbers are defined as:

$$ F_n = F_{n-1} + F_{n-2}, $$

(2.19)

with seed values

$$ F_0 = 0, F_1 = 1. $$
If we were to naïvely write a recursive program to calculate the \( n \)th Fibonacci number, it may look something like

\[
\text{Fib}(n) \{
    \text{if} (n == 0) \text{ return } 0;
    \text{if} (n == 1) \text{ return } 1;
    \text{return Fib}(n-1) + \text{Fib}(n-2);
\}
\]

As Bellman pointed out, this program will quickly use all available memory to even modern day computers as \( n \) becomes very large. The number of calls to the \( \text{Fib} \) function grows exponentially with \( n \). Dynamic programming avoids the pitfalls of this approach by recognizing that the problem at hand is in fact a collection of overlapping subproblems. That is, when calculating \( F_{n-2} \), we are at the same time calculating the solution for \( F_{n-1} \). In order to reflect this in our program we might instead have

\[
\text{Fib\_array}[\text{Large Number}];
\text{Fib\_array}[0] = 0;
\text{Fib\_array}[1] = 1;

\text{Fib}(n) \{
    \text{if} (\text{Fib\_array}[n] == \text{null}) \text{Fib\_array}[n] =
    \text{Fib}(n-1) + \text{Fib}(n-2);
    \text{return Fib\_array}[n];
\}
\]

Rewriting our program this way the Fibonacci numbers are recorded in \( \text{Fib\_array} \) each time they are calculated, and are returned immediately each time \( \text{Fib} \) is called for that number. This process is also known as \textit{memoization}. While the call tree of our original recursive program required \( O(e^n) \) memory, the dynamic programming method only requires \( O(n) \) memory.

The IFS model uses the principles of dynamic programming in order to solve for the total lifetime utility of individuals. Fundamentally, Eq. 2.15 closely resembles our treatment of the Fibonacci numbers. Instead of repeatedly calculating the total utility for a given time period, we instead recall this value. This is what is meant when the time index is dropped from Eq. 2.14 in favor of a functional equation. The functional equation reflects the fact that the only dependence of the current time period is on the previous time period, and no previous values need to be calculated since it is assumed that they are already known.

\subsection*{2.4.1 Parallelism in Dynamic Programming}

While dynamic programming makes certain problems tractable by eliminating exponential growth, the algorithms that dynamic programming apply to are not easily parallelized. In our Fibonacci number example for instance, there is no obvious way to
parallelize the calculation of the \( n \)th number. Each number is calculated in a linear fashion. In fact, if we were to allow two different threads calculate \( \text{Fib}(n-1) \) and \( \text{Fib}(n-2) \) then this would likely result in repeated calculation of the values in \( \text{Fib\_array} \), which is precisely the behavior we wish to avoid.

In the IFS model, the parallelism will not occur at the same level that the dynamic programming takes place. The total utility of each time period must be calculated serially. Instead, we will attempt to find opportunities to employ parallelization techniques within Eq. 2.15 itself. We should recall that \( X \) is a vector of quantitates of goods, and these goods may not depend on one another. If they do not, then we can divide the calculation of \( \max(U(X)) \) by independent goods over multiple processors. In the next section we will discuss the structure of the program itself, and highlight where exactly these opportunities present themselves.
Chapter 3

Program Structure

The IFS model is written in Fortran 95 and contains over six thousand lines of code. The total runtime is approximately 78 seconds, broken down into three sections: setting up the decision problem environment, simulating the decision of individuals, and computing the moments. The performance critical section was the environment set up, which took up 98% of the runtime and was the focus of our efforts. The table below shows the runtime of the three sections we deal with in our code: the environment set up, the simulation, and the statistics step, as well as the initialization. This timing data was gathered by calling \texttt{cpu\_time} around the subroutine calls for these sections.

\begin{center}
\begin{tabular}{|c|c|}
\hline
Section & Run Time (sec) \\
\hline
Initialization & 0.71 \\
Environment Set Up & 75.68 \\
Simulation & 1.28 \\
Statistics & 0.29 \\
\hline
\end{tabular}
\end{center}

Although we do not discuss the initialization, this step does not grow with the problem size and does very little. We can see that the environment set up is the most costly section, however as we will discuss it does not share some of the complexities that go into the other sections of the code. The simulation, for instance, links with the Numerical Algorithms Group (NAG) library in order to call various maths functions and generate arrays of random numbers. Had the random number generator been used in a section of the code that had to be parallelized we would have had to deal with issues such as reproducibility and generating random numbers in parallel.

The code behind the IFS model is extremely complex and uses a sophisticated naming system. For simplicity, we will refer to only crucial functions and variables used in the program, otherwise we will rely on high level, abstract descriptions of sections of the program. Critical functions include \texttt{solve} and \texttt{slnWL} while the variables we were interested in are \texttt{EV} and \texttt{EDU}. These are labels for arrays that have very similar names and are used in very similar ways, however for our purposes we are only interested in how large these arrays are. An example of two of these arrays can be seen below.
Here we see a declaration for the variables $lEV_1$ and $lEDU_1$. These are two largest arrays used for storing all of the value and marginal utility values generated in the environment set up and used in the simulation. The type of the arrays are of type $male_1$, meaning that they apply to women who are married to men. We see that the arrays themselves are essentially eleven dimensional arrays of real values. These arrays make up the backbone of the program and account for the tremendous amount of memory used. The size of all eleven dimensions are given in the tables below.

<table>
<thead>
<tr>
<th>Type: $male_1$</th>
<th>mina:maxa</th>
<th>nts</th>
<th>ntth</th>
<th>1-doCC:1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type: real(wp)</th>
<th>ngpk</th>
<th>maxngpe</th>
<th>ngpp</th>
<th>1:maxmaxakd</th>
<th>ntsm</th>
<th>ngplm</th>
<th>ngppm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>12</td>
<td>19</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

The first table shows the dimensions that are of type $male_1$ while the second table shows the dimensions of type $real(wp)$, which is double precision. In total this amounts to 78796800 reals, or 601 MB for each array. These two arrays make up a vast majority of the memory used during the execution of our program, so the total memory used is approximately 1.2 GB.

### 3.1 Main Subroutine

The main subroutine begins by loading routines from the FORTAX library, a library written by the IFS containing routines related to the optimal decision problem and the MoM, by looking at supplied .csv files. These files also include initial data such as the prices $P$ of commodities bought by families. Again, commodities are not only goods but also the value of accumulating work experience, personal experiences, general cultural influences, and most importantly the tax and benefits policies in place. Next it initializes constant values used during the simulation and computation of the moments. Many of these constants are simply used to define the shape of the data structures used throughout the program, thus the constants include the number of education levels, the age that individuals begin schooling, and the average lifespan of individuals. The program then calculates certain parameters that depend only on these constants, such as the average cost of childcare for a given family size and the probability of families transforming either through child arrival or couple formation/dissolution. For parameters
and moments that must be calculated after the simulation, the program loads initial estimates from external files. This concludes the preparation steps and allows the program to begin the subroutines that contain the primary sections of the program that take up a majority of the runtime.

3.2 Three Sections: Environment Set Up, Simulation of Individuals, and Statistics

The three sections of the calculation are implemented in various ways within the program, allowing the user to specify when data is written to file and if any additional parameters need to be estimated. The critical function call which takes up the majority of the runtime in the environment setup section is a subroutine called solve. The name might sound misleading since the problem for a particular individual is not solved at this point in the program, however it is in this part of the code that the solution of the optimal decision problem is found before determining stochastic variables in an individual’s life. That is, the solution is found for Eq. 2.17 before incorporating random variables coinciding with decisions. Thus, when we speak about setting up the environment, we mean we are solving Eq. 2.17 up to the point at which individual decisions factor in.

In the IFS model, finding the solution to Eq. 2.17 essentially means creating arrays to contain the total utility evaluated at all time periods for all known commodities. Here, all time periods correspond to the point at which an individual attends school, through their working life, through retirement and up to death. The total utility is stored in the array EV. The model simultaneously calculates and stores the marginal utility and stores this in the array EDU that is the same shape as EV. Filling in these arrays within the solve function comprises the bulk of the runtime of the program.

3.2.1 Environment Set Up

The utility arrays are filled in within the solve function in three stages: the period of life when an individual attends university, an individual’s working years, and their retirement years. Calculating the utility from the university stage is simply a matter of adding up the value of the gained education. It is assumed at this time that the individual is not accumulating assets of her own and consumption dedicated to finishing the degree. Similarly, the calculation for the retirement stage is fairly simple, although both the utility and marginal utility must be calculated. During the retirement stage the only uncertainty is partnering, since the individual is unaffected by factors such as the labor supply or average wages.

Finding the utility during the working years takes up a vast majority of the runtime, as the most number of parameters come into play. This occurs in a function called slnWL,
standing for working life solution, which takes as inputs the age, level of schooling, family characteristics, level of experience, and assets. We are able to know the assets of the individual pending randomness since we know at the end of the person’s life they will have no assets remaining and it is assumed that they expend their assets during the retirement years in a steady way.

The end result of the call to slnWL is the calculation of the total utility from all goods, effectively calculating Eq. 2.17 without accounting for uncertainties. The first step towards this goal is the integration over the individual’s productivity. The second is to integrate over the family productivity, and then final step is to sum the utility and marginal utility of each good to determine the total utility and total marginal utility for a given time. These values are stored in the EV and EDU arrays, and once these are completely filled the program has completed the solution section.

The complexity of the slnWL function makes implementing OpenMP very difficult. The use of module variables, calls to functions and subroutines contained within other parts of the program, and a lack of documentation make this the least intuitive part of the program. As we will discuss later, the main issue with the slnWL function is that it is very difficult to track side-effects when calling external functions and subroutines, thus causing erratic behavior within memory.

3.2.2 Simulation

The next section is the simulation, which takes up approximately 1.5% of the total runtime. Here, the program simulates the life choices of 5620 individuals with varying levels of education at the start of their lives. As discussed in the Optimal Decision Problem subsection, individuals make optimal decisions based on their knowledge of their present conditions, their past conditions, and what may happen in the future. As discussed earlier, the total utility arrays were filled in backwards during the solve function because in the model the final conditions are known. That is, at an individual’s death they have no utility and will have spent all of their assets. In this sense, the utility arrays are filled in backwards in time during the environment set up, while the simulation section runs forward in time. With the utility and marginal utility arrays at hand, each individual must decide whether to join the labor force at that time after all random variables are presented to them.

For each individual, the process of simulation begins by choosing initial variables for that individual such as education level and the age at which employment begins, then stepping through each time period in that individual’s life to solve Eq. 2.17. At each time period the individual decides what level of consumption and what course of action will be most optimal for them. Each individual also accounts for the future by anticipating how their actions today will increase utility tomorrow. At the conclusion of the simulation section, each individual’s choices are recorded in arrays referred to as the state space of the problem. These arrays do not constitute a significant part of the memory required and are much smaller than the EV and EDU arrays.
3.2.3 Calculation of Moments

In the final section of the program the moments are calculated from the state space recorded during the simulation section. Here, the moments of interest to the user are calculated and either written to file or printed to screen. Calculating the moments takes up an insignificant proportion of the runtime, however if the model is ever expanded then this part may become more expensive. Additionally, the current implementation does not print out all of the state variables and arrays used during calculation such as the EV and EDU arrays. If these are eventually included in the output then this would greatly increase the runtime of this section.

3.3 Porting Our Program

The serial version of the IFS program did not initially run on the ECDF’s Eddie machine. To make our code work properly we had to perform numerous small fixes. First, in our makefile we had to change the path to the NAG library. Our program also required an extensive file tree with certain parts of the FORTAX library placed in various folders in order for the program to compile correctly.

Undoubtedly the biggest problem we had in making our program work at all were related to memory. As stated previously, the EV and EDU arrays took up just over 1 GB of memory while our program ran. Because of this, we could not run our program on either the front end or back end of Eddie, and whenever we tried it would crash after informing us that there was insufficient memory available. There was not enough space on the stack by default to allow for the creation of these arrays. We first attempted to address this issue by setting the ulimit to 2 GB. This allowed our program to run in serial. When we first tried to run in parallel however, it crashed after yielding the same error. It turned out that ulimit only adjusts the stack size on the master thread, and that any threads spawned had the default stack size. To change this we had to add a line to the job script adjusting the stack size of all threads. Normally this would involve changing the environment variable OMP_STACKSIZE, however since we were using the Intel implementation the variable we had to change was called KMP_STACKSIZE instead. This allowed our program to run on any number of threads on Eddie.
Chapter 4

Parallelizing our Model

As it was requested that we tamper with the code as little as possible, our first priority was to identify sections of the code that may benefit from the use of OpenMP. Not only is the use of OpenMP directives significantly less intrusive than say parallelization using MPI, but as we saw in the previous section much of the runtime of the program is taken up in a very small part of the code. Thus, we sought to implement OpenMP in only a small section while still gaining a large speedup. The only optimization in place prior to the start of this project was the use of compiler flags. In the ‘release’ mode of the makefile, as opposed to the debug mode, the -O3 flag is used. Aside from this there is very little.

As discussed in the previous section, finding parallelism in the solve subroutine is critical for significantly reducing the runtime, especially when calculating the utility function during the working life of individuals. This is done in a function called slnWL. The slnWL function is called within six nested DO loops that run over environmental factors that are independent of one another inside of the solve function. A snippet of the code appears below showing how these loops appear below.

```fortran
do dCC=1,1-doCC,-1  
do s=1,nts                      
do a=merge(maxa,maxabirth+10,dCC==1),as(s),-1  
   if (a<maxa) call nxtebds(a,s)  
do th=1,ntth  
do ei=1,ngpe(a,s)  
do ki=1,ngpk  
call slnWL(th,s,a,ei,ki,dCC)  
end do !ki  
end do !ei  
end do !th  
end do !a  
end do !s  
end do !dCC
```
The seven loop variables, or environmental factors, are $dCC$ for presence of children, $s$ for level of education, $a$ for age, $th$ for permanent unobserved heterogeneity, $ei$ for work experience, and $ki$ for assets.

The trip counts of the loops over these factors are 2, 3, 30 to 40, 2, 1 to 5, and 6 in that order. The trip count for age and for work experience depend on the index of the loops they are nested within. The ending age depends on the presence of children, while the starting age depends on level of education. The maximum index of the work experience depends on both age and level of education, and is found in the array $ngpe$. One thing that may appear out of place is that the loop over age goes in reverse. This is because, as stated in the discussion of the optimal decision problem, it does not matter if we are going forward or backwards in time; the Bellman equation can be written either way.

4.1 Collapsing Nested Loops

When we first investigated our program one thing was immediately apparent: the loops which we had to parallelize over had very small trip counts, and we were afraid that we would not be able to take advantage of all of the cores available to us. Thankfully, the loops we were working with were nested, so we recognized that we may be able to use the NESTED or the COLLAPSE clauses. The former creates nested parallel regions while the latter combines perfectly nested loops in order to create a single parallel region with a larger trip count. For our purposes, it was much simpler to use COLLAPSE as it required fewer OpenMP directives.

The first task when parallelizing our code was to decide which loops to parallelize. Complications arose from dependencies within the looped variables. While each of these variables, which we should remember are environmental factors themselves, are independent of each other within `solve`, the trip count of some of these loops depended on the indices of the outer loops. For instance, the trip count of the loop over work experience, the fifth loop, depends explicitly on the age and level of education; the second and third loops respectively. Similarly, the ending age depended on the presence of children, while the starting age depended on the level of education. We were able to immediately see that would we not be able to collapse the loop over age or work experience, the third and fourth loops respectively, with any above loops. This left us three ways to achieve parallelization: collapse the first and second loops together, parallelize over the age only, or collapse the fourth, fifth and sixth loops together. These regions would have trip counts of 6, 30 to 40, and 12 to 60 respectively.

We immediately ruled out the first option since we knew we would have more than six threads available to us when we inserted OpenMP directives. Parallelizing the loop over the age turned out to be impossible, however this was not immediately obvious. We realized this after further investigation of the `solve` function. As previously stated, each of the six environmental factors we work with are independent of one another within `solve`. However, the value of the $EV$ and $EDU$ arrays for a given age depended
on the values of those arrays at the previous age. This made sense, since we are working with Eq. 2.17 after all, where the value at one time depends explicitly on the value at the previous time period. Because of this dependency, we were not able to parallelize over age at all. Thus we were left with one option: to attempt to collapse over the inner three loops over permanent unobserved heterogeneity, work experience, and assets.

In order to guarantee that these factors did not behave in a manner similar to age within the solve function we ran these loops in reverse in the serial program. Running these three loops in reverse generated identical results. Unfortunately, we were unable to prevent our program from crashing when all three of these loops were collapsed into a single parallel region. We were only able to collapse the inner two loops, work experience and assets, in order to make a single parallel region with a trip count ranging from 6 to 30.

Once we chose which loops to work with we had to tackle the errors causing the program to crash. The performance critical section of the environment set up is six nested DO loops. Initially we tried to use a PARALLEL DO and simply identify variables with SHARED and PRIVATE clauses, however this did not function correctly. Each time we ran the program we would encounter an arithmetic error suggesting that a floating point value was being divided by zero. One thing to note is that these crashes were unrelated to the crashes associated with insufficient memory discussed earlier.

Our first concern was that module variables from another part of the program was being modified. More specifically we believed that a variable was being shared that should have been private. This was extremely difficult to deal with since according to the OpenMP standard module variables are automatically shared, and the complexity of our program made it difficult to tell whether or not variables were being modified. We knew that the EV and EDU arrays were modified by many threads at once, but this was not a problem because the values were not being used again until the simulation which was performed in serial.

The only clue that we had was the error message. It stated that at some point there was a floating point divide by zero occurring. We found the error in a function written in another file, where the input to the function was exponentiated to a negative number. Thus, it would appear that the input was incorrectly being set to zero. To add to our confusion it appeared that the program would crash after a seemingly random number of iterations. We were finally able to reason that some of the variables we were declaring as PRIVATE had to be THREADPRIVATE. The purpose of this clause will be discussed in the following two sections, but it appeared that our program was suffering from a subtle problem that can only be repaired by using THREADPRIVATE. That is, referencing privatized module variables in a subroutine caused the thread to use the global copy rather than its own copy.

Thankfully this was the only significant error to be found in the code. Once it was repaired, most errors found afterwards followed a similar pattern and were relatively easy to deal with. In order to track the value of variables we inserted print statements at regular intervals within the code. Each thread printed to its own file and we compared
where the progress of the two threads ended from this output. This process took a significant amount of time due to the size of the call tree and the fact that we generally had to go line by line through a function to determine where the error occurred. This was probably the most instructive part of the project for both me and David, as neither of us had previously seen the use of THREADPRIVATE or known how to apply it.

4.2 Module Variables

Module variables, similar to global file scope variables in C/C++, are by default treated as SHARED in a parallel region. Our program uses module variables for two purposes: to store information used during the calculation of the $EV$ and $EDU$ values, and to hold temporary values about the outer three loops within $slnWL$. Dealing with the latter case was simple enough. These variables depended on the outer loops only which were not parallelized, so we just had to add these variables to the SHARED clause. The variables containing information used during the calculation proved to be more complicated. Each thread requires its own copy of the variable, however only the call to $slnWL$ was explicitly within the OpenMP directives. So when a thread entered $slnWL$ and any subroutines after that it had no information about the module variables used if they were not explicitly past in as arguments. Because of this, each time a thread referenced a module variable it was in fact referencing the single global copy. To remedy this we had to declare these module variables as THREADPRIVATE.

Since the THREADPRIVATE clause is often not necessary, the following subsection discusses the differences between THREADPRIVATE and PRIVATE, and demonstrates the necessity of the former in certain circumstances.

4.2.1 THREADPRIVATE vs PRIVATE

The primary difference between THREADPRIVATE and PRIVATE is in where the two clauses can be used. PRIVATE can be appended to the start of a parallel region, while THREADPRIVATE must be used where module variable are declared. In C/C++ this would be a static region.

David Henty wrote a short program in order to test the differences between the two clauses. The code is reproduced below.

```fortran
program main
    use testscope
    call doit()
end program main

module testscope
    use omp_lib
```
implicit none
integer, parameter :: n = 4
integer, dimension(n) :: iarray
!$omp threadprivate(iarray)
contains
subroutine doit()
  implicit none
  integer :: ithread, i
  iarray(:) = -13
  !$omp parallel private(ithread)
  ithread = omp_get_thread_num()
  call setit(ithread)
  call printit(ithread)
  !$omp end parallel
  write(*,*) 'tag = ', -1, ', iarray = ', (iarray(i), i = 1, n)
  !$omp parallel private(ithread)
  ithread = omp_get_thread_num()
  call printit(ithread)
  !$omp end parallel
end subroutine doit

subroutine setit(value)
  implicit none
  integer :: value
  iarray(:) = value
end subroutine setit

subroutine printit(tag)
  implicit none
  integer :: tag, i
  write(*,*) 'tag = ', tag, ', iarray = ', (iarray(i), i = 1, n)
end subroutine printit
end module testscope

In this example program the module variable we are interested in is iarray. At the moment, iarray is a THREADPRIVATE variable however it is fairly simple to change iarray to PRIVATE and test that behavior. We see from the output below that iarray retains its information when a thread enters the printit subroutine:
This is the same behavior that was necessary for our IFS model to function properly. If instead we commented out the THREADPRIVATE clause and instead let \textit{iarray} be PRIVATE then the behavior would be much different. When the threads enter the \textit{printit} subroutine they are referencing the global array. This can be seen from the output below:

Thread Number
0 \text{iarray} = 0 0 0 0
1 \text{iarray} = 1 1 1 1
2 \text{iarray} = 2 2 2 2
3 \text{iarray} = 3 3 3 3
-1 \text{iarray} = 0 0 0 0
0 \text{iarray} = 0 0 0 0
1 \text{iarray} = 1 1 1 1
2 \text{iarray} = 2 2 2 2
3 \text{iarray} = 3 3 3 3

We can see that thread 2 has set the values of \textit{iarray} to 2, and this is printed by both threads 2 and 0. Next, thread 1 sets the values to 1 and this is printed by 1 and 3 before the second parallel region when each thread prints 1. This was the behavior was causing problems in the parallel version of our code. Threads would clear the value of module variables at each iteration, but they were in fact both referencing the global copy of the variable and not their own PRIVATE copy. Turning inline may solve this problem, however in our case it did not help since the level of inlining performed by the \texttt{-O3} compiler flag was not enough to cover all references of the module variables. In addition, relying on inlining to solve this problem is really only a temporary fix, since the program would fail if inlining were ever removed. Using THREADPRIVATE solves this problem by providing each thread with its own separate copy of the module variable.

### 4.3 Final Parallel Version

The final version of our code required only a few lines of OpenMP directives to function properly, however as we have shown here it was not obvious to us from the start what
these directives had to be. We provide the final version of the important parts of the code below.
At the top of the file is a THREADPRIVATE clause, which as we stated was necessary in order to hold information that persisted from one parallel region to the next and assure that threads referred to their own copy of a data structure when inside a subroutine or function. In the nested loops we also see that the COLLAPSE clause makes a larger parallel region out of the inner two loops. One change from the original version is the ngpeTEMP variable. PARALLEL DO loops using COLLAPSE cannot be created around loops for which one of the bounds is a function call, as was the case originally. Thus, we created a temporary variable named ngpeTEMP to hold this value, allowing us to parallelize the loop over work experience and collapse this with the loop over assets.

4.4 Running on Eddie

We ran the IFS model on the HPC cluster Eddie maintained by the Edinburgh Compute and Data Facility (ECDF). All of our information related to Eddie can be found on the ECDF website at: www.wiki.ed.ac.uk/display/ecdfwiki/Eddie+and+the+ECDF.
We chose this machine because of the availability of the Intel Fortran compiler; the same compiler used at the IFS. According the ECDF website, Eddie is in its second incarnation - dubbed "Eddie Mark 2" - and is comprised of two main parts:

1. Mark2Phase1 - 130 IBM dx360M3 iDataPlex servers, each with two Intel Xeon E5620 quad-core processors. All nodes connected by a GigabitEthernet network with a 10 Gigabit network core.

2. Mark2Phase2 - 156 IBM dx360M3 iDataPlex servers, each with two Intel Xeon E5645 six-core processors. All nodes connected by Gigabit Ethernet network, and 68 of them are also connected by a QDR QLogic Infiniband network- for Message Passing Interface (MPI) jobs.

We tested our program on both sets of nodes. Unfortunately the Phase1 nodes did not run our program using more than 5 threads at a time. Instead of including this incomplete set of data in our discussion we included it in our appendix. We present the completed results from running on Phase2 in the following chapter. We were able to choose which nodes to run our program on by adding flags to our job scripts. We did not test the effect of enabling the Infiniband network because we did not submit MPI jobs, and we did not use Eddie’s GPGPU resources.

The PGI implementation of OpenMP used during the MSc course and the Intel implementation used in this project differ slightly. The only difference that affected our project at all was the need to use the environment variable $KMP_STACKSIZE$ as opposed to $OMP_STACKSIZE$. The variables are exactly the same except for how they are written, however they may seem strange to someone investigating our job scripts that had not previously used $ifort$.

### 4.5 Timing Difficulties and Profiling

In our efforts to implement a functioning parallel version of the IFS model we encountered smaller challenges that took some time to complete. These challenges fell into two categories: timing our program effectively, and confirming correct parallel behavior. We present both of these issues here, as we felt that neither was significant enough to warrant a full section.

The original code contained statements that timed the three major parts of the program: the environment set up, the simulation, and the statistics step. All three of these were called in the main subroutine, and so did not give us specific information pertaining to the $slnWL$ function. In addition, when our parallel version originally began functioning we were surprised to see that no matter what number of threads we ran on, the timing statements in the main subroutine always returned nearly the same time. This, of course, was because it was calling $cpu\_time$, which returns the total amount of time spent by all processors summed together, and not $omp\_get\_wtime$, which returns the wall-clock time.
time elapsed by the program. In order to time the parallel region of our program as well as the entire environment set up we inserted calls to the `omp_get_wtime` function.

While timing our program we also decided that it would be best to profile the parallel version of our code and compare this to the profile of the serial version. For this we used both gprof and the Intel Thread Profiler (ITP). The gprof output let us know that the call tree was identical to the serial version. Unfortunately we did not have access to all ITP features, but the version used on Eddie still gave detailed information on the parallel regions. The full output of the ITP is provided in the appendix, however the important part of ITP are shown below.

Program execution time (in seconds):

```
    cpu : 0.01 sec
  elapsed : 44.20 sec
      serial : 2.95 sec
      parallel : 41.25 sec
   cpu percent : 0.02 %
```

Summary over all regions (has 2 threads):

```
 # Thread   #0    #1
  Sum Parallel : 41.253  41.251
  Sum Imbalance : 1.024   0.010
  Min Parallel : 3.1e-03  3.1e-03
  Max Parallel : 0.170    0.170
  Max Imbalance : 0.007   2.4e-03
```

We see that ITP provides us with most of the information we are interested in. This ITP output was from a trial running on two threads. The total execution time of the program was 44.20 seconds and the time executed while in parallel regions was 41.25 seconds. While there was some imbalance between the two threads this time was significantly less than the total parallel execution time, and so we may conclude that the parallel version was well balanced.

We also used ITP to confirm that Eddie was parallelizing OpenMP code correctly. To do this we ran the Mandelbrot set calculation, studied during the MSc course, on Eddie and observed the ITP output. This appeared similar to that shown above for the IFS model, and let us know that the program was running properly. Since the Mandelbrot code was so much smaller, we also took this opportunity to get more familiar with ITP in general. For instance, the Program events generated by ITP are not particularly intuitive, however using the Mandelbrot program we were able to determine what most of these events meant, giving us a better understanding of the output from our own code.
Chapter 5

Results

Here we present all results gathered while running on Eddie. The analysis of these results is left to the following chapter.

5.1 Thread Number

Running on the Phase2 nodes on Eddie we were able to vary the number of threads from 1 to 12. This resulted in speed up of the parallel regions as well as the entire program as seen below in Fig. 5.1.
We see that the speed up did not simply level off as the number of threads increased, but the speed up seemed to exhibit step-like behavior.

The simulation and statistics steps incurs additional overhead as seen by the speed up of the entire program in Fig. 5.1, however together they do not yet comprise a majority of the runtime. The speed up of the parallel region highlights some interesting behavior: the largest performance gains after using 6 threads were made on even numbers of threads.

Clearly, the environment set up in the IFS model benefits from a larger thread count. Since the maximum trip count of the parallel regions is 30, it is possible that the program will continue to exhibit speed up to that point, however we only had access to 12 threads at once.

5.2 Without COLLAPSE

As stated in the previous chapter we use the COLLAPSE clause to create one large parallel region out of two smaller ones. Here we present the speed up of those loops independently. That is, we removed the COLLAPSE clause and parallelized the fifth and sixth loops on their own. We compare these results with the speed up of the parallel region with COLLAPSE included in Fig. 5.2 presented below.
Figure 5.2: Speed up of the entire collapsed parallel region compared to the fifth and sixth loops only.

Not only is the speed up of the individual loops worse than with COLLAPSE on place, but we also see these loops plateau very quickly. This is because the fifth and sixth loops have maximum trip counts of 5 and 6, respectively, and using more threads than that provides no additional benefit.

It may have been possible for us to use nested PARALLEL DO clauses on these loops instead of COLLAPSE, but this would likely have had no overall benefit.
Chapter 6

Analysis

In this chapter we analyze the results presented previously. We also investigate the dynamic schedule, and discuss how the parallel regions varied in size and how this affected the speed up.

6.1 Investigating Trip Counts

As discussed in previous chapters, the trip counts of the parallel regions vary in size from 6 to 30. Here we attempt an analysis of this variation in order to explain the behavior of the speed up when increasing the number of threads. As we saw, speed up jumped significantly for even number of threads, however this gain was not constant. Our parallel regions contained trip counts from 6 to 30 and increased in increments of 6. To make our analysis, we counted the number of parallel regions for each trip count, as seen in Fig. 6.1. We did this by creating an array with fifty elements, and just before the start of each parallel region we added one to the element of the array that corresponded to the trip count of that region.
Figure 6.1: Trip count of parallel regions. Parallel regions have trip counts from 6 to 30 in increments of 6, and the number of regions with these trip counts varies.

According to the figure, a vast majority of the parallel regions have trip counts of either 24 or 30. Assuming that overhead from creating more threads remains constant, our program may exhibit speed up all the way up to 30 threads. However this would have to be tested on a different machine, since 12 the maximum number of threads possible on Eddie.

Fig. 6.1 also explains the trends seen in the speed up from Fig. 5.1. Significant speed up occurred up to 6 threads and then at each even number of threads from then on. We can see that these gains were made when the number of threads was a factor of the trip counts of the parallel region. In other words, significant gains were made up to six threads because 2, 3, 4, 5, and 6 are all multiples of either 24, 30, or both. This same trend explains the step wise behavior from 7 up to 12 threads. Since both 7 and 11 are not factors of any of the trip counts, none of the parallel regions benefitted from using that many threads. It may seem that using 9 threads would offer some benefit since there are parallel regions with trip counts of 18, but Fig. 6.1 explains why this is not the case. There are too few regions with 18 iterations to make 9 threads provide a significant benefit.

In our MSc program we were not faced with situations similar to this. When confronted with this type of program, it is critical to gather as much information about these parallel regions as possible. As demonstrated here, analyzing the trip counts for these regions provides insight into what happens when more threads are added. Indeed, we see that if we had access to say a 14 core machine it may not be worthwhile to use 14 threads since it is not a multiple of 24 or 30.
6.2 Predicting Performance Gains

We were able to predict the speed up of our program. In order to do this we first needed to know how many iterations a given thread performed for some number of threads \( T \). This was given by

\[
\text{Iterations}_T = \sum_{i=1}^{5} \left( \left\lceil \frac{6 \times i}{T} \right\rceil \times R_i \right).
\]  

(6.1)

The summed variable \( i \) runs from 1 to 5 because there are 5 different sizes of parallel regions increasing in size by 6 each time. \( R_i \) is the number of parallel regions that have the trip count \( 6 \times i \) and is given by the data seen in Fig. 6.1. Taking the ceiling of the trip count divided by the number of threads gives the maximum number of iterations done by a given thread for a given sized parallel region. We then multiply this by the number of parallel regions of that trip count, \( R_i \), and we arrive at the maximum number of iterations for a given thread. For \( T = 1 \) we just had the number of iterations in our parallel regions, 10332.

Dividing the total number of iterations in our parallel regions by the maximum number of iterations done by a given thread gives us the predicted speed up. This can be seen in Fig. 6.2 below.
Figure 6.2: Predicted speed up generated from the trip count data. Some numbers of threads are not factors of any of the trip counts and we should expect to see no speed up at that number.

The predicted speed up follows our measured speed up very closely, however we notice a few differences. At certain numbers of threads we should see no gain in performance, for instance at $T = 7, 9, \text{ and } 11$, however we see some small gains in our measured speed up. This suggests that not all iterations require the same amount of work and that there may have been some load imbalance. This was confirmed when we tested a dynamic schedule as discussed in the next section. In addition, we see that there is some serial overhead present for any number of threads. This is likely due to the threads being created just before each parallel region, or perhaps the small amount of computation required when evaluating the trip counts of the parallel region created by the COLLAPSE clause.

We performed a similar analysis of the speed up for the fifth and sixth loops individually. The equations for calculating the maximum number of iterations for these two loops are given by

$$\text{Loop Five Iterations}_T = \sum_{i=1}^{5} \left( \left\lceil \frac{i}{T} \right\rceil \times R_i \right) \quad (6.2)$$

and
\[ Loop \text{ Six Iterations}_T = \left\lfloor \frac{6}{T} \right\rfloor \times R. \] (6.3)

For the fifth loop the trip counts run from 1 to 5, so \( i \) does not need to be multiplied by 6, also the number of iterations for each parallel region \( R_i \) changes. For the sixth loop all parallel regions have a trip count of 6 and the total number of parallel regions \( R \) is just 10322. In the following two figures we compare the predicted speed up of just the two loops to the predicted speed up of parallel regions when using COLLAPSE, as well as a comparison of the predicted speed up of just the two loops to the measured speed up of the loops.

Figure 6.3: Predicted speed up of the entire program compared to just the fifth and sixth loops.
Figure 6.4: Predicted speed up of just the fifth and sixth loops compared to their measured speed ups.

From Fig. 6.3 it is abundantly clear that using COLLAPSE should always yield better results in theory than a parallelization of either loop by themselves. We can also see from Fig. 6.4 that the predicted speed up is better than the measured speed up of the individual loops. This further supports our theory that there is some serial overhead involved.

6.3 Schedules

In this section we compare the static schedule to the dynamic schedule. Chunk sizes were left as their default values because of the small value of the trip counts. Fig. 6.5 below compares the measured speed up using the static schedule to the measured speed up of the dynamic schedule, along with the theoretical speed up discussed in the previous section.
Fig. 6.5 certainly shows that the static schedule performs better overall than the static schedule. This implies that the overhead seen in the previous section is due to a load imbalance between threads. The dynamic schedule addresses this by giving out chunks of iterations for threads to perform and giving out new chunks when a thread completes its current work. If a thread is bogged down by a particularly lengthy calculation in a given iteration then other threads can pick up the slack. Additionally, we see that at certain points, at 7 and 9 threads, the dynamic schedule appears to perform better than ideal. This can be associated with the performance gain from each thread dealing with a smaller problem size. When this occurs a thread must access memory less frequently, and oftentimes can store more of the calculation in the cache. Since this performance gain is not associated with the equations we wrote it is unaccounted for. Although throughout this project we used the static schedule to make measurements, it is fair to assume that all results would likely have been very similar had we used a dynamic schedule, although we would have likely observed less serial overhead.
Chapter 7

Future Work

The primary goal of this project has been satisfied; to improve the runtime of the IFS model. The code we worked with during the project called the *solve* function only once, however there are variations on the model where *solve* must be called multiple times. In those cases, our results become even more important and the runtime continues to be dominated by the environment set up. That being said there remains work to be done both to further improve the performance of the set up step and to parallelize other parts of the program.

7.1 Increasing Trip Count

First and foremost, the most important work remaining is increasing the trip count of the parallel regions within *slnWL*. As previously stated, out of the six nested DO loops that surround the call to *solve* we collapse the fifth and sixth loops together. We also showed that it may be possible to parallelize the fourth loop when we were able to run the loop backwards in the serial version of the program. However, we were not able to collapse this fourth loop along with the fifth and the sixth without the program generating a floating point divide by zero error. Although we were not able to find the root of this error it is identical to the errors generated earlier in the project when we misused SHARED and PRIVATE clauses. It may very well be possible to correct this behavior and increase the trip count of the parallel region, thus increasing the benefit of using a large number of threads. Solving this problem requires the same approach used to get the parallel version working initially; print out the state of the environment set up and compare this to the serial version in order to pin point the exact location of the mathematical error.

In the previous chapter we were able to predict the speed up of our program fairly accurately when using COLLAPSE and when parallelizing individual loops. We can apply the same method in order to predict the speed up when using COLLAPSE over the fourth loop in addition to the fifth and six loops. In this case, the equation telling us
the maximum number of iterations for a given thread is

$$\text{Iterations}_T = \sum_{i=1}^{5} \left( \left\lceil \frac{12 \times i}{T} \right\rceil \times R_i \right). \quad (7.1)$$

Here, the trip counts are multiples of 12 instead of 6, and the $R_i$ values will be half of the values seen in our histogram in Fig. 6.1. This is because there are effectively the same number of iterations, however we are grouping them into parallel regions of twice the trip count. Dividing the total number of iterations, 10332, by the numbers generated from this equation gives of the predicted speed up just as before. In Fig. 7.1 we show the predicted speed up using COLLAPSE(3) over the fourth, fifth, and sixth loops as opposed to the predicted behavior of our current implementation that only uses COLLAPSE(2) over only the fifth and sixth.

Figure 7.1: The predicted speed up using COLLAPSE(3) compared to COLLAPSE(2).

We see that collapsing over three loops performs better than collapsing over only two at every number of threads. It is important to note that in this example the number of threads is extended to 24. Using 12 threads, the maximum allowed on Eddie, and collapsing over three loops we see that the predicted speed up is ideal. This makes sense because all of the trip counts in this case are multiples of 12, whereas when collapsing over 2 loops all trip counts were multiples of 6.

One feature of Fig. 7.1 that is concerning is that using the current implementation our program will see no performance gain when using between 15 and 23 threads. This is unacceptable if the IFS wishes scale the program to many more cores at once. It is clear
that incorporating the third loop into the COLLAPSE clause is critical if this program is to run on larger machines and further reduce the runtime of the environment set up.

7.2 Parallelizing Simulation and Statistics

In the future it may be the case that the set up step no longer dominates the runtime of the IFS model. If the problem size gets larger or the set up is able to scale to many more threads, then both the simulation step and the statistics step may need performance improvements. Thankfully, both of these steps may be easy to parallelize. The environment set up constructed the arrays containing the value of individuals’ decisions before actually considering any random variables. This step took most of the time because there is a significant amount of maths involved in calculating the value for each of the many environmental factors. The simulation step loops through each of the five thousand individuals and calculates the result of the individual’s decision, of which there are very few. Similarly, in the statistics step the program loops through individuals in order to collect data and then calculates the moments associated with that data. Both of these steps loop over individuals independent of one another, and so this loop can be parallelized and have a trip count of five thousand.

One thing that may be an issue if we were to parallelize the simulation would be the use of random number generators. At the moment a single random number generator is used when the program calls the NAG library in order to make individuals’ decisions. If we were to parallelize this process we may run into trouble. If two threads are accessing the random number generator at the same time then we would not necessarily know in which order in which the threads generate these numbers. It may be that our results no longer become reproducible. There are a few ways around this. One possibility is to give each thread a random number generator, which would likely have to be THREAD-PRIVATE since it would be called from many places in the code, however in that case you may not produce the same results when changing the number of threads. Another possibility is to generate all of the required random numbers before the parallel region and keep these in a look up table, so that the threads are looking up the same random numbers every time. Either one of these would require modifications to the code in its current form and are thankfully not necessary yet.

7.3 Hybrid Implementation

Originally the work plan for this project included an investigation into an MPI implementation of the IFS model. While the immediate goals of improving runtime were achieved using OpenMP directives, a hybrid implementation may still offer significant benefits. Critically, the data structures used in the current implementation are extremely large. The arrays for the value and marginal utility require over 1 GB of memory.
Adding in any additional environmental factors will vastly increase the memory required to the point that the program becomes impossible to run. To remedy this, we must consider implementing MPI in order to take advantage of memory available on multiple nodes of a distributed memory machine. One way to accomplish this may be to break apart the $EV$ and $EDU$ arrays according to independent environmental factors, however this would require significant modification of the code in its current form.

Another use of MPI would not be in the environment set up, but in the statistics step. If I/O ever becomes a critical source of slowdown then MPI I/O may be necessary in order to write data to file in a reasonable amount of time. This would certainly come after the model grows significantly in size though, since at the moment writing the results to file takes up only a small fraction of the runtime.
Chapter 8

Conclusion

Our project was a success overall. We met our main goal of improving the performance of the IFS tax and benefits model using OpenMP directives. We determined that using the COLLAPSE clause on as many of the loops as possible surrounding the $lnWL$ function within solve yields parallel regions that can be easily divided among at least 12 threads. Using this approach we reduced the overall runtime of the program from 78 seconds to 10 seconds.

While we met our goal of using OpenMP to improve performance we expended a considerable amount of effort to reach this point. The complexities of the IFS model obscured the steps required to correctly distinguish between SHARED, PRIVATE, and THREADPRIVATE variables. Even getting the program to run properly was a challenge. Memory problems, setting up the NAG and FORTAX libraries, and collecting timing and profiling data were all issues that required effort to achieve.

Looking back on our project, I can say it might have been helpful to do a few things slightly differently. First, in terms of understanding the problem at hand, I would have asked for guidance on elementary economics subjects before diving into the more complex subject matter directly related to our problem. Second, when parallelizing the code I would have been more systematic in keeping track of variables and what their purpose is. Instead of working top down by attempting to parallelize and then figuring out how to repair the program, I would have first made an exhaustive list of variables altered in the crucial sections of the code and assessed how to use OpenMP directives from there. I suppose this was the philosophy preached during our Software Development class, and at least I can say I appreciate that outlook much more now. Third, I would have liked to have known from the onset of the project that the work done was just as important as the results gathered. From the beginning I believe I was only seeing the end goal: parallelize the program and write a report on the gains that we made. In fact, I believe I spent more time discussing the work that I performed over these past six months than on the results, and I think that I failed to realize this and take extensive notes from the beginning. Despite these hurdles, I am very proud of how the project turned out.
Working in a field completely alien to me was rewarding in a variety of ways. It gave me a better appreciation for the maths I have seen in physics that also appear in econometrics and microeconomics. Not only that, but I was also distinctly proud of the fact that I was working on a project that may directly benefit people in the UK. As seen in Blundell 2011, opportunities for women to find employment in the UK is closely tied to tax and benefits policies. Improving the performance of the IFS model allows researchers and policy makers to gather data much more quickly and make more informed decisions on what is best for this group of individuals. My project may not have been as extensive, groundbreaking, or revolutionary as other MSc dissertations, but I am extremely grateful for the opportunity to work with the code, Jonathan Shaw, and David Henty on this project.
Bibliography


Appendix A

Trip Count of Parallel Regions

<table>
<thead>
<tr>
<th>Trip Count</th>
<th>Number of Parallel Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>120</td>
</tr>
<tr>
<td>30</td>
<td>234</td>
</tr>
</tbody>
</table>
Appendix B

Runtime for Different Numbers of Threads

B.1 Two Intel Xeon E5620 Quad-Core Processors

<table>
<thead>
<tr>
<th>Threads</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Mean Time (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>74.39</td>
<td>74.15</td>
<td>74.47</td>
<td>74.34</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>39.83</td>
<td>39.82</td>
<td>39.24</td>
<td>39.63</td>
<td>1.88</td>
</tr>
<tr>
<td>3</td>
<td>26.30</td>
<td>25.30</td>
<td>26.77</td>
<td>26.12</td>
<td>2.85</td>
</tr>
<tr>
<td>4</td>
<td>20.91</td>
<td>20.61</td>
<td>20.66</td>
<td>20.73</td>
<td>3.59</td>
</tr>
<tr>
<td>5</td>
<td>16.24</td>
<td>16.42</td>
<td>16.42</td>
<td>16.30</td>
<td>4.56</td>
</tr>
</tbody>
</table>

B.2 Two Intel Xeon E5645 Six-Core Processors

<table>
<thead>
<tr>
<th>Threads</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Mean Time (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>74.97</td>
<td>83.88</td>
<td>74.83</td>
<td>77.89</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>42.32</td>
<td>42.13</td>
<td>39.05</td>
<td>41.17</td>
<td>1.89</td>
</tr>
<tr>
<td>3</td>
<td>27.95</td>
<td>27.26</td>
<td>27.01</td>
<td>27.41</td>
<td>2.84</td>
</tr>
<tr>
<td>4</td>
<td>20.64</td>
<td>22.09</td>
<td>21.18</td>
<td>21.30</td>
<td>3.66</td>
</tr>
<tr>
<td>5</td>
<td>16.44</td>
<td>16.5</td>
<td>16.82</td>
<td>16.59</td>
<td>4.70</td>
</tr>
<tr>
<td>6</td>
<td>13.82</td>
<td>13.71</td>
<td>13.47</td>
<td>13.67</td>
<td>5.70</td>
</tr>
<tr>
<td>7</td>
<td>13.42</td>
<td>13.24</td>
<td>13.11</td>
<td>13.26</td>
<td>5.88</td>
</tr>
<tr>
<td>8</td>
<td>10.95</td>
<td>10.59</td>
<td>11.01</td>
<td>10.85</td>
<td>7.18</td>
</tr>
<tr>
<td>9</td>
<td>10.59</td>
<td>10.51</td>
<td>10.59</td>
<td>10.56</td>
<td>7.37</td>
</tr>
<tr>
<td>10</td>
<td>8.75</td>
<td>8.73</td>
<td>8.76</td>
<td>8.75</td>
<td>8.91</td>
</tr>
<tr>
<td>11</td>
<td>8.55</td>
<td>8.72</td>
<td>8.79</td>
<td>8.69</td>
<td>8.97</td>
</tr>
<tr>
<td>12</td>
<td>7.70</td>
<td>7.80</td>
<td>7.75</td>
<td>7.75</td>
<td>10.05</td>
</tr>
</tbody>
</table>
Appendix C

Individual Loop Runtimes

C.1 Fifth Loop Only

<table>
<thead>
<tr>
<th>Threads</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Mean Time (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79.42</td>
<td>79.84</td>
<td>78.81</td>
<td>79.36</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>45.35</td>
<td>45.15</td>
<td>45.91</td>
<td>45.47</td>
<td>1.75</td>
</tr>
<tr>
<td>3</td>
<td>33.31</td>
<td>33.49</td>
<td>32.41</td>
<td>33.07</td>
<td>2.40</td>
</tr>
<tr>
<td>4</td>
<td>27.64</td>
<td>28.38</td>
<td>27.81</td>
<td>27.94</td>
<td>2.84</td>
</tr>
<tr>
<td>5</td>
<td>17.87</td>
<td>18.02</td>
<td>17.88</td>
<td>17.92</td>
<td>4.43</td>
</tr>
<tr>
<td>6</td>
<td>17.73</td>
<td>17.76</td>
<td>17.54</td>
<td>17.68</td>
<td>4.49</td>
</tr>
<tr>
<td>7</td>
<td>17.58</td>
<td>17.64</td>
<td>17.68</td>
<td>17.63</td>
<td>4.50</td>
</tr>
<tr>
<td>8</td>
<td>17.7</td>
<td>17.79</td>
<td>17.73</td>
<td>17.74</td>
<td>4.47</td>
</tr>
<tr>
<td>9</td>
<td>17.63</td>
<td>17.68</td>
<td>17.63</td>
<td>17.65</td>
<td>4.50</td>
</tr>
<tr>
<td>10</td>
<td>17.67</td>
<td>17.83</td>
<td>17.65</td>
<td>17.72</td>
<td>4.48</td>
</tr>
<tr>
<td>11</td>
<td>17.66</td>
<td>17.74</td>
<td>17.71</td>
<td>17.70</td>
<td>4.48</td>
</tr>
<tr>
<td>12</td>
<td>17.82</td>
<td>17.78</td>
<td>17.74</td>
<td>17.78</td>
<td>4.46</td>
</tr>
</tbody>
</table>
### C.2 Sixth Loop Only

<table>
<thead>
<tr>
<th>Threads</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Mean Time (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77.66</td>
<td>74.76</td>
<td>75.44</td>
<td>75.95</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>40.17</td>
<td>43.57</td>
<td>38.97</td>
<td>40.90</td>
<td>1.86</td>
</tr>
<tr>
<td>3</td>
<td>26.98</td>
<td>26.64</td>
<td>26.65</td>
<td>26.76</td>
<td>2.84</td>
</tr>
<tr>
<td>4</td>
<td>26.7</td>
<td>25.7</td>
<td>25.38</td>
<td>25.93</td>
<td>2.93</td>
</tr>
<tr>
<td>5</td>
<td>25.7</td>
<td>25.89</td>
<td>25.46</td>
<td>25.68</td>
<td>2.96</td>
</tr>
<tr>
<td>6</td>
<td>13.44</td>
<td>13.59</td>
<td>13.46</td>
<td>13.50</td>
<td>5.63</td>
</tr>
<tr>
<td>7</td>
<td>14.16</td>
<td>13.55</td>
<td>13.45</td>
<td>13.72</td>
<td>5.54</td>
</tr>
<tr>
<td>8</td>
<td>13.54</td>
<td>13.44</td>
<td>13.36</td>
<td>13.45</td>
<td>5.65</td>
</tr>
<tr>
<td>9</td>
<td>13.55</td>
<td>13.55</td>
<td>13.51</td>
<td>13.54</td>
<td>5.61</td>
</tr>
<tr>
<td>10</td>
<td>14.22</td>
<td>13.48</td>
<td>13.51</td>
<td>13.74</td>
<td>5.53</td>
</tr>
<tr>
<td>11</td>
<td>13.6</td>
<td>14.02</td>
<td>13.54</td>
<td>13.72</td>
<td>5.54</td>
</tr>
<tr>
<td>12</td>
<td>13.58</td>
<td>13.49</td>
<td>13.46</td>
<td>13.51</td>
<td>5.62</td>
</tr>
</tbody>
</table>
# Appendix D

## Dynamic Runtimes

<table>
<thead>
<tr>
<th>Threads</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Mean Time (sec)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79.46</td>
<td>82.08</td>
<td>76.32</td>
<td>79.29</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>40.32</td>
<td>38.38</td>
<td>40.79</td>
<td>39.83</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>27.45</td>
<td>28.29</td>
<td>27.11</td>
<td>27.62</td>
<td>2.87</td>
</tr>
<tr>
<td>4</td>
<td>20.08</td>
<td>20.35</td>
<td>21.9</td>
<td>20.78</td>
<td>3.82</td>
</tr>
<tr>
<td>5</td>
<td>16.50</td>
<td>16.67</td>
<td>16.38</td>
<td>16.52</td>
<td>4.80</td>
</tr>
<tr>
<td>6</td>
<td>13.44</td>
<td>13.5</td>
<td>13.62</td>
<td>13.52</td>
<td>5.86</td>
</tr>
<tr>
<td>7</td>
<td>13.36</td>
<td>13.03</td>
<td>12.67</td>
<td>13.02</td>
<td>6.09</td>
</tr>
<tr>
<td>8</td>
<td>10.65</td>
<td>10.75</td>
<td>10.77</td>
<td>10.72</td>
<td>7.39</td>
</tr>
<tr>
<td>9</td>
<td>10.27</td>
<td>10.32</td>
<td>10.33</td>
<td>10.31</td>
<td>7.69</td>
</tr>
<tr>
<td>10</td>
<td>8.67</td>
<td>8.71</td>
<td>8.62</td>
<td>8.67</td>
<td>9.15</td>
</tr>
<tr>
<td>11</td>
<td>8.63</td>
<td>8.77</td>
<td>8.57</td>
<td>8.66</td>
<td>9.16</td>
</tr>
<tr>
<td>12</td>
<td>7.75</td>
<td>7.82</td>
<td>7.80</td>
<td>7.79</td>
<td>10.18</td>
</tr>
</tbody>
</table>
Appendix E

Intel Thread Profiler Data

*** KAI statistics library  k3301

*** Begin Task 0

Environment variables:

OMP_NUM_THREADS : 2
OMP_SCHEDULE : static
OMP_DYNAMIC : FALSE
OMP_NESTED : FALSE
KMP_STATSFILE : guide.gvs
KMP_STATSCOLS : 80
KMP_INTERVAL : 0
KMP_BLOCKTIME : 200
KMP_PARALLEL : 2
KMP_STACKSIZE : 2147483648
KMP_STACKOFFSET : 0
KMP_SCHEDULING : <unknown>
KMP_CHUNK : <unknown>
KMP_LIBRARY : throughput

end

System parameters:

start : Thu Jul 26 19:40:36 2012
host : eddie315
ncpu : 8

end

Unix process parameters:
maxrss : 1503940
minflt : 381239
majflt : 0
nswap : 0
inblock : 0
oublock : 0
nvcs : 606
nivcs : 4798
end

Region counts:
  serial regions : 3
  barrier regions : 0
  parallel regions : 1
end

Program execution time (in seconds):
cpu : 0.01 sec
elapsed : 44.20 sec
  serial : 2.95 sec
  parallel : 41.25 sec
cpu percent : 0.02 %
end

Summary over all regions (has 2 threads):
# Thread #0 #1
Sum Parallel : 41.253 41.251
Sum Imbalance : 1.024 0.010
Min Parallel : 3e-03 3e-03
Max Parallel : 0.170 0.170
Max Imbalance : 0.007 2.4e-03
end

Region #1 (has 2 threads) at slnmod_mp_solve_/108 in "/exports/home/s1150361/dissertation/code/solution1.f90":
# Thread #0 #1
Sum Parallel : 41.253 41.251
Sum Imbalance : 1.024 0.010
Min Parallel : 3e-03 3e-03
Max Parallel : 0.170 0.170
Max Imbalance : 0.007 2.4e-03
end

Region #1 (has 2 threads) profile:
# Thread Incl Excl Routine

51
Serial program regions:
Serial region #1 executes for 0.81 seconds
begins at START OF PROGRAM
ends before region #1 (using 2 threads) at slnmod_mp_solve_/108 in "/exports/home/s1150361/dissertation/code/solution1.f90"
Serial region #2 executes for 0.00 seconds
begins after region #1 (using 2 threads) at slnmod_mp_solve_/108 in "/exports/home/s1150361/dissertation/code/solution1.f90"
ends before region #1 (using 2 threads) at slnmod_mp_solve_/108 in "/exports/home/s1150361/dissertation/code/solution1.f90"
Serial region #3 executes for 2.14 seconds
begins after region #1 (using 2 threads) at slnmod_mp_solve_/108 in "/exports/home/s1150361/dissertation/code/solution1.f90"
ends at END OF PROGRAM
Serial region #1 profile:
# Thread Incl Excl Routine
Serial region #2 profile:
# Thread Incl Excl Routine
Serial region #3 profile:
# Thread Incl Excl Routine
Program events (total):
# Thread #0 #1
mppbeg : 1 0
mppend : 1 0
serial : 439 0
mppfrk : 438 0
mppjoi : 438 0
mppadj : 1 0
mpptid : 68900545 68900544
mpttpa : 28 28
Region #1 (has 2 threads) events:
# Thread #0 #1
mppfrk : 438 0
mppjoi : 438 0
mpptid : 68900544 68900544
mpttpa : 28 28
end
Serial section events:
# Serial #1 #2 #3
mppbeg : 1 0 0
mppend : 0 0 1
serial : 1 437 1
mppfkd : 1 437 0
mppadj : 1 0 0
mpptid : 1 0 0
end

*** end