USING REGRESSION BASED METHODS FOR TIME-CONSTRAINED SCALING OF PARALLEL PROCESSOR COMPUTING APPLICATIONS

by

JEONIFER GARREN

(Under the direction of Jaxk Reeves)

ABSTRACT

There is a need for an automated method that facilitates time-constrained scaling for applications to encourage the wider use of parallel computation for computationally intense problems. In this thesis, we show for the first time a self-generated focal training method that is able to accurately achieve time-constrained scaling using a focused regression. This is demonstrated with six benchmark applications, but can be extended to any application of interest for which time-constrained scaling is needed.

INDEX WORDS: time-constrained scaling, parallel computation, regression

USING REGRESSION BASED METHODS FOR TIME-CONSTRAINED SCALING OF PARALLEL PROCESSOR COMPUTING APPLICATIONS

by

JEONIFER GARREN

B.S., Emory University, 2002

MPH, Boston University, 2005

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

USING REGRESSION BASED METHODS FOR TIME-CONSTRAINED SCALING OF PARALLEL PROCESSOR COMPUTING APPLICATIONS

by

JEONIFER GARREN

Approved:

Major Professor: Jaxk Reeves

Committee: Lynne Seymour

William McCormick

Electronic Version Approved:

Maureen Grasso

Dean of the Graduate School

The University of Georgia

May 2010

DEDICATION

This is for my grandparents, Roy and Betty Garren. Thanks for always believing and accepting me for me. I would also like to dedicate this thesis to all the people in my life who have helped me become who I am today.

ACKNOWLEDGEMENTS

I would like to acknowledge the Statistics department at UGA for allowing me into the program, with special thanks to my committee. To my advisor Jaxk Reeves; I couldn't have done this without him. To the Computer Science departments at UGA and Arizona State University, thanks for allowing me to be part of the research team.

PREFACE

This project was supported by a grant from the National Science Foundation (CSR 08-34356). The project was a collaboration between researchers from the Statistics and Computer Science fields. From the computer science field, this project was assisted by Professor David Lowenthal and Brad Barnes of the University of Arizona and the University of Georgia, respectively.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Parallel computation has become widely popular in many applied sciences. Typically, parallel computation requires a large number of processors and is carried out in a specialized computing center (such as the EITS Research Computing Center (RCC) at UGA) that serves many independent researchers. In order to meet the demands of scientific research for increasing computational complexity, these facilities must increase either the number or the efficiency of the processors they use. We are interested specifically in evaluating what happens to an application when the number of processors is increased. That is, we are not proposing methods to increase efficiency; rather, we wish to forecast how computing time will change if the number of processors were to increase. Other things being held constant, run time tends to decrease as the number of processors increases, but the exact relationship is frequently rather complex. Of even more interest to researchers in the field is how one should adjust ('scale') other input variables such that the total run time (computation time plus communication time) is approximately the same when the application is run with an increased number of processors.

The specific task of prediction is not an easy undertaking for many reasons. Some factors are specific to the application running in parallel, while others are specific to the architecture of the system. Despite these difficulties, there have been numerous attempts with multiple methods to predict performance and a few attempts at forecasting. Although most of these methods are for prediction, they could easily be extended to forecasting. In this thesis, we are using the word 'predict' to mean using a statistical model based on the available data to estimate run time given

a set of input variables within the range for which the model was developed. We are using the word 'forecast' to mean a similar type of time estimation, but for the case where at least one of the input variables (usually number of processors) is outside the range for which the model was developed. In statistical parlance, what we are doing when 'forecasting' is called "extrapolation beyond the range of the data", and is generally considered risky. This thesis attempts to explore how risky this practice is in the setting of parallel processing run time computations, and to offer some guidelines for how to do it effectively. Methods currently available for forecasting are quite crude, with median absolute relative errors for forecasting TIME frequently being well over 10%. If simple new methods could be derived which gave median absolute relative errors of 10% or less, this would be considered quite beneficial to computer scientists.

We are examining six computing applications (BT, LU, SP, CG, Sweep3D, and SMG) as a representative group of computer applications for our research. We believe that results for these six applications can be generalized to many computer applications. The six applications are described in more detail in Section 2.2. The datasets obtained from these six applications for this thesis were collected by Brad Barnes of the Computer Science Department at UGA. Both the "training set" and "forecast set" runs for all six applications were run on *Atlas,* a parallel system located at the Lawrence Livermore National Laboratory.

The communication, computation, and TOTAL run times for each application were recorded to the nearest hundredth of a second. While it should be the case that the sum of computation and communication times is equal to the TOTAL, this is not always the case. In fact, there is not even a uniformly agreed upon way to measure computation and communication times, with some computer scientists preferring to use maximum computation (MAXCOMP) and minimum communication (MINCOMM) times, and others preferring to use the critical path

computation (CPCOMP) and communication (CPCOMM) times. For most of our applications, we used 'TOTAL' as the response variable of interest, but for those for which modeling computation and communication components separately was required, we were more careful in specifying exactly how these separate times were measured.

The input variables which are varied in any set of runs of an application depend upon the application being evaluated. The most important input variable, of course, is the number of processors (P) involved. For most of the applications examined, it was convenient to increase the processors in multiples of powers of two, so training set runs were frequently executed at these seven processor levels: P= 16, 32, 64, 128, 256, 512, and 1024, with forecasts at P=2048. For a few of the applications for which processor numbers were required to be even perfect squares, runs were at similar values. For example, for BT, the processor sizes varied as P= 16, 36, 64, 144, 256, 484, and 1024 in the training set, while P=1936 was used for the forecast set. For a given number of processors, other variables were also varied so as to yield a range of run times. For the simpler applications, there was usually only one other input variable of interest, frequently measuring the 'size' of the problem. For some of the more complex applications, as many as six input variables were varied in some pattern.

Table 1.1 – List of Applications and Characteristics

Application	Key Variables	Training Set					Forecast Set	
		# of Processor levels	Processor Range	Processor Type	Unique Cases	Total Cases	Processor Level	Total Cases
BT	1)Processors 2) Size	7	16-1024	Even Sq	792	792	1936	22
LU	1)Processors 2) Size	7	16-1024	2^n	118	118	2048	31
SP	1)Processors 2) Size	8	16-1024	Even Sq	352	352	1936	124
CG	1)Processors 2) Size 3) Non-zeroes	7	16-1024	2^n	1890	1890	2048	29
Sweep3D	1)Processors 2) Size 3)2 processor dimensions	7	16-1024	2^n	224	224	2048	20
SMG	1)Processors 2) Size 3)3 processor dimensions	9	4-1024	2^n	848	2542	2048	36

Table 1.1 – above lists, for all six applications, the key variables, the number of training processor levels, the training set processor range, the type of processor levels used (power of 2 or even squares), the training set unique cases, the training set total cases, the forecast set processor level, and the forecast set total cases. Many computer scientists do not use replication, since they believe it is an expensive waste of resources to achieve essentially the same answer. This fact helps explain why the unique cases and total cases are identical for all but the last application (SMG), for which two replications (i.e. three total) were run for each combination of input variables. In general, the measurement error when replicating an experiment is relatively small compared to the typical variation between experiments, so the computer science approach of no replication (what statisticians might call 'one observation per cell') is usually justified. However, when replications did occur (in SMG and in other datasets examined for this research but not included in Table 1) we did see occasional extremely large deviations. With only one

observation per experimental condition, such aberrant observations would not be detected unless they were so unusual as to be classified as outlier/influential points and deleted from an analysis.

There are many approaches which one might take in attempting to use the results from the 'training set' of an application to make projections for the 'forecast set'. A number of these are discussed in Section 2.3 of this thesis. Even for a fixed general approach, there are different methods which one might want to use, depending on the data available. We discuss this in Chapter 3, first considering a best-case scenario where one has an actual complete training set available to develop a model, then the more realistic case where one has available a smaller 'focused' subset of the training sample space, and, finally, the most realistic case of all, where one is given output for only one or two points in the 'training space' and must decide how to proceed in making forecasts.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1. BACKGROUND

Parallel computation involves fragmenting a computationally intense application into smaller, more manageable calculations. This is easier said than done. There are several issues that make parallel computation difficult and non-intuitive.

2.1.1. PARALLEL COMPUTATION ISSUES

There are four different types of parallelism: bit-level, instruction-level, data, and task. We are concerned only with data parallelism. Data parallelism refers to how an application's loops distribute the data across the parallel processors [1]. These loops in the application, the architecture of the system, and the interaction between the application and the architecture can create inefficiencies in the flow of the execution of the calculations. If there were no communication or startup issues, and an application was completely efficient, running an application on 'P' parallel processors would take $\frac{1}{P}$ of the time required to run the same job on one processor. However, matters are rarely that simple. Described below are some difficulties that can occur.

2.1.1.1.DEPENDENCIES

Many applications are composed of a series of calculations that are dependent on previous calculations; this is the serial part of an application and cannot be parallelized. The

longest chain of dependent calculations is known as the critical path. An application's execution time cannot be shorter than the critical path execution time [1].

Dependencies can also occur in the parallel part of an application. When dependencies of one loop iteration on previous iterations occur in loops they are called loop-carried dependencies. Loop-carried dependencies prevent the parallelization of loops [1].

Three types of dependencies can occur with loops: 1) true/flow dependent, 2) anti-dependent, and 3) output dependent. All of these arise when two statements in an application are vying to use the same variable. True/flow dependency

> correspond[s] to the first statement producing a result used by the second statement. … [Anti-dependency occurs] when the first statement over-writes a variable needed by the second [statement.] … Output dependency occurs when two statements write to the same location; the final result must come from the last logically executed statement. [1]

For this research, we are not concerned with exactly why these dependencies occur, but we attempt to indirectly model how often they occur.

2.1.1.2. RACE CONDITIONS, MUTUAL EXCLUSION, SYNCHRONIZATION, AND PARALLEL SLOWDOWN

The shift from sequential to parallel computer applications introduced a new set of software problems, thus making parallel computing more difficult. Race conditions, which are one of the most common problems, arise when a variable that is shared between multiple sub-tasks is updated because of lack of synchronization. If the order of the updating and outputting for each sub-task is performed in the incorrect order, errors or even complete failure in the computation can occur. In order to avoid this situation, the programmer can use a *lock* to provide mutual exclusion [1]. While this makes the application safer, it might slow the application considerably.

Another common difficulty with parallel computer applications is the need for communication and synchronization between the different sub-tasks. To force parallel application sub-tasks to act in synchrony requires the use of a *barrier* that creates a point in an application where all processes must arrive before they can depart [1]. The *barrier* will "ensure that two concurrently-executing processes do not execute specific portions of an application at the same time. If one process has begun to execute a serialized portion of the application, any other process trying to execute this portion must wait until the first process finishes." [1] As with the 'lock', this procedure slows an application.

As one increases parallelization, one increases the proportion of time needed for the processors to communicate with each other. If this communication time becomes larger than the computation time, then the parallelization does not result in a speed-up but a slowdown. We refer to this phenomenon as parallel slowdown [1].

2.1.1.3. SPEED –UP

The rate that a parallel application is faster than a sequential application is known as speed-up. Speed-up is defined as $S(P) = \frac{T_1}{T_P}$. That is, it is the ratio of time required to run the application with one processor compared to using P processors. The optimal speed-up would be linear – doubling the number of processing elements should halve the runtime [1]. Unfortunately, "very few parallel [applications] achieve optimal speed-up." [1] Two laws which bound the theoretical speed-up of an application due to parallelization are Amdahl's law and Gustafson's law.

Amdahl's Law (1) calculates the theoretical maximum speed-up that can occur when more processors are added to solve an application [2]. It is given by:

$$S(P) = \frac{1}{1 - f(P)} \qquad (1)$$

where S(P) is the speed-up and f(P) is the fraction of the application that is parallelizable when there are P processors in use. The limiting factor for the theoretical maximum speed-up of an application using multiple processors in parallel computing is the time needed for the sequential fraction of the application to run [2]. Amdahl's law depicts a "near-linear speed-up for small numbers of processing elements, [but then] flattens out into a constant value for large numbers of processing elements" (as shown by Figure 2.1) for most applications' theoretical maximum speed-up [1]. Amdahl's law is not used for scaling because it is based on the notion of a fixed problem size, so the sequential part of the application will be the same no matter how many processors are added to the parallel system [3].

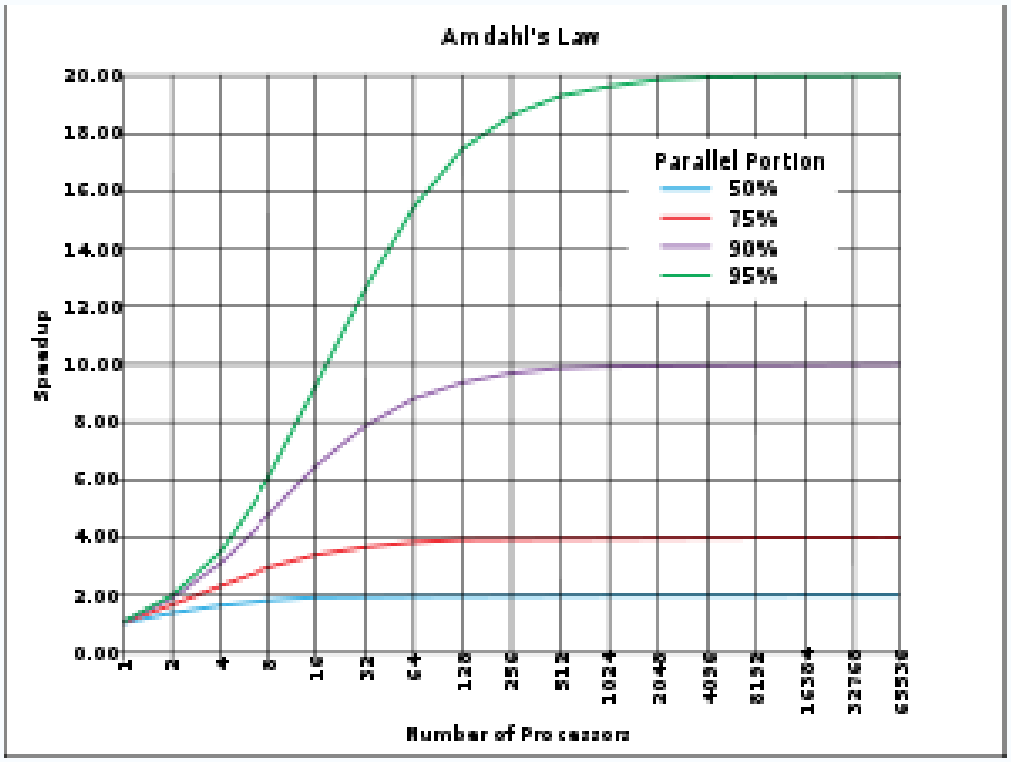

Figure 2.1 – Amdahl's Law of Speed-up due to Parallelization

Gustafson's Law predicts how well an application can be parallelized to speed-up the overall run time [3]. Equation 2 represents Gustafson's Law.

$$S(P) = P - \alpha \cdot (P - 1) \qquad (2)$$

where P is the number of processors, $S(P)$ is the speedup, and α is the non-parallelizable portion of the process. Unlike Amdahl's Law, Gustafson's Law allows scaling. Gustafson proposed using time-constrained scaling when predicting the effectiveness of parallelization.

2.1.2. TYPES OF SCALING

Scalability can refer to many aspects of a parallel system. There are two ways to scale the hardware, horizontally and vertically, and three ways to scale the application: strong, weak, and time-constrained. In our case, we are using horizontal scaling of the hardware and both weak and time-constrained scaling of the application. We describe below both types of scaling that are relevant to our problem. Furthermore, we will discuss problems due to non-linear effects in computation and communication times and unknown relationships between the variables and execution time [4]. These make it difficult to determine the application-specific variables for proper scaling.

2.1.2.1. HORIZONTAL SCALING

Horizontal scaling refers to increasing the number of processors in the system. This is what we wish to model, but, without actually running the application repeatedly at the highest level of processors, since that would be expensive.

2.1.2.2. WEAK SCALING

Weak scaling involves designating a fixed problem size to each processor, so that when the number of processors increases, then, so, too, does the overall problem size [5]. In the case of weak scaling, linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of processors [6]. Unfortunately, under linear scaling, due to added communication time, most run times do not stay constant [4]. When the application is one-dimensional, weak scaling is a relatively trivial problem. However, most applications are not one-dimensional, so weak scaling is difficult in general due to non-linear communication time and multi-dimensionality.

2.1.2.3. TIME-CONSTRAINED SCALING

Time-constrained scaling has recently become more popular than weak scaling, especially since weak scaling is difficult and often unpredictable. Time-constrained scaling occurs when the number of processors used to execute an application increases, but the input variables are adjusted so that the total run time remains the same as it was at a lower processor value. For example, one might be told that for the CG application with P=1024, SIZE=1,000,000 and NZ=14, the run time is 85.20 seconds. Under time-constrained scaling, one would like to determine the value of {SIZE, NZ} at NP=2048 which would also take 85.23 seconds to run.

2.2. APPLICATIONS

This section describes in greater detail the six applications for which training and prediction datasets were created by researchers in the Computer Science Department at UGA.

These applications were chosen because they are well-known applications which are representative of most parallel applications. Included in the descriptions for each of the six applications in subsections 2.2.1-2.2.6 below are the purpose for each application, how it is executed, and the relevant input variables.

2.2.1. BT

The Block Tridiagonal (BT) application performs a synthetic computational fluid dynamics (CFD) calculation to approximate a solution for the three-dimensional Euler/Navier-Stokes equations. This application partitions the three-dimensional space into structured grids with a (5x5) block size to solve multiple block tri-diagonal equations. The application solves the block tri-diagonal equations first in the x, then in the y, and finally in the z direction.

There are two conditions that are inherent in order to run the BT application. First, BT uses a multi-partitioning scheme to create the structured grids of (5x5) block size. This method is more optimal than other possible methods "because it provides good load balance and uses coarse grained communication" [7], which are both desirable qualities.

> In the multi-partition algorithm, each processor is responsible for several disjoint sub-blocks of points ("cells") of the grid. The cells are arranged such that for each direction of the line solve phase, the cells belonging to a certain processor will be evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent. [7]

The second condition that the BT application requires is that only a square even integer number of processors can be assigned to a job. Other than the number of processors (P), the only input variable for the BT application is SIZE, the total volume of the application. It does not

appear that communication time contributes much to the total time for BT to run, so forecasting for BT should be easier than for most other applications examined here.

2.2.2. LU

The Lower-Upper symmetric Gauss-Seidel (LU) application performs a synthetic CFD calculation to approximate a solution for the three-dimensional Euler/Navier-Stokes equations. This application partitions the three-dimensional space into structured grids that are sparse lower and upper (5x5) blocks.

There are two conditions that are inherent if one wants to run the LU application. First, LU performs

> [a] 2-D partitioning of the grid onto processors by halving the grid repeatedly in the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions on the individual processors. The ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively sweep from one corner on a given z plane to the opposite corner of the same z plane, thereupon proceeding to the next z plane. [7]

The second condition that the LU application requires is that only a power-of-two number of processors can be assigned to a job. Communication is known to play a larger relative role in this application than it does for BT, but in the sample region for which we examined LU, it appears that total communication time is small relative to computation time. For LU, as with BT, there is only one variable, SIZE, in addition to P. Because there is only one other variable and the effects of communication time are small, one would expect a linear regression model applied to this data set to perform well.

2.2.3. SP

The Scalar Penta-diagonal (SP) application performs a synthetic computational fluid dynamics (CFD) calculation to approximate a solution for the three-dimensional Euler/Navier-Stokes equations. This application partitions the three- dimensional space into structured grids to solve multiple scalar penta-diagonal equations. The application solves the scalar penta-diagonal equations first in the x, then in the y, and, finally, in the z direction.

There are two conditions that are inherent in running the SP application. First, SP uses a multi-partitioning scheme to create the structured grids. This method is more optimal than other possible methods "because it provides good load balance and uses coarse grained communication" [7], both of which are desirable qualities. Second, as with BT, SP requires an even integer square number of processors and has only one input variable other than P, SIZE. [Note: To be consistent with other applications, we have defined SIZE= Dim1*Dim2*Dim3 to be the volume of the application. For BT, LU, and SP, the application is run on a cube with Dim1=Dim2=Dim3, so our SIZE=Dim1^3. This may be confusing to some users of these applications who refer to the Dim1 variable as 'SIZE'.]

2.2.4. CG

The Conjugate Gradient (CG) application solves a system of linear equations. CG does this by computing an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix through an iterative method. This method will converge rapidly when an appropriate pre-conditioner [8] is used. This application is important because it tests irregular long- distance communication [8].

CG is easily parallelized. Most of the parallelization is done by loops inside the iteration loop. The speed-up due to parallelization goes from sub-linear at 1 to 2 processors, to super-linear at 4 to 8 processors, but with more than 16 processors the speed-up decreases due to communication overhead [9].

> One drawback of [CG] … is that it involves the computation of two *separate* inner products of distributed vectors. Moreover, the first inner product must be completed before the data are available for computing the second inner product. Hence, [CG] … has two separate communication phases for these two inner products. [10]

Perhaps modeling these phases separately will yield better fits, as discussed in Section 4.4.

In addition to the number of processors, P, there are two input variables for CG: the SIZE of the problem (Dim1), and the number of non-zeroes, NZ. Because of the additional variables and potential model complexity due to communication, it is expected that modeling the behavior of CG will be more difficult than for BT, LU, and SP.

2.2.5. SWEEP3D

The SWEEP3D application solves a 1-group time-independent discrete ordinates 3D Cartesian geometry neutron transport problem.

> [I]t uses a mult-dimensional wavefront algorithm for "discrete ordinates" deterministic particle transport simulation. Sweep3D benefits from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three-dimensional space is decomposed onto a two-dimensional orthogonal mesh, where each processor is assigned one columnar domain. Sweep3D exchanges messages between processors as wavefronts propagate diagonally across this 3-D space in eight directions. [11]

In addition to P, there are five input variables for Sweep3D: two processor dimensions, Px and Py, and three grid dimensions, Nx, Ny, and Nz. In fact, this reduces to two independent variables, since Px *Py=P, and Nx*Ny* Nz=WSS is the relevant SIZE variable.

The training set of Sweep3D was perfectly weak scaled. In order to achieve weak scaling of the global grid equally in all three dimensions, each of the grid dimensions (Nx, Ny, and Nz) were increased by a factor of $\sqrt[3]{2}$ as the number of processors doubled.

2.2.6. SMG

The Semi-coarsening Multi-Grid (SMG2000) application is a parallel solver for the Euler/Navier-Stokes equations on logically rectangular grids. The multi-grid "accelerates the convergence of the iterative method by global correction from time to time, accomplished by solving a coarse problem." [12]

> In contrast to other methods, multi-grid methods are general in that they can treat arbitrary regions and boundary conditions. They do not depend on the separability of the equations or on other special properties of the equation. They are also directly applicable to more-complicated non-symmetric and non-linear systems of equations [12].

SMG can be used for 2D or 3D problems, but we are concerned with the application in 3D. SMG partitions the three-dimensional space into 27-point grids [13]. The application specifies grid dimensions in terms of a per-processor local grid; one can recover the global grid by taking the product of each grid dimension with the associated processor dimension.

In addition to P, there are six input variables for SMG: three processor dimensions, Px, Py, and Pz, where the product of these must equal the number of processors (P), along with three grid dimensions, Nx, Ny, and Nz, where the product of these equals SIZE. Thus, in addition to P, there are really only three independent input variables: SIZE and two of {Px, Py, Pz}.

The training set of the SMG application was perfectly weak scaled. In order to achieve weak scaling of the global grid equally in all three dimensions, each of the grid dimensions (Nx, Ny, and Nz) were increased by a factor of $\sqrt[3]{2}$ as processor size doubled [4]. In order to achieve

time-constrained scaling in the 'Focal regions' method, Px=1 and one of the other processor dimensions (Py, Pz) is scaled by a factor of two.

Modeling SMG is further complicated in that it is not symmetric in all dimensions. That is, unlike the dimension scale, where the assignment of dimensions is irrelevant as long as SIZE= Nx*Ny*Nz is constant, it **does** make a difference how the Px*Py*Pz=P processors are arranged, and the naïve assumption that Px=Py=Pz=$\sqrt[3]{P}$ would yield the optimal run time is not at all true for the regions which we have examined. SMG is communication intensive and its run-time is known to increase logarithmically with an increase in the problem size.

2.3. POSSIBLE APPROACHES TO PREDICTION/FORECASTING

Various approaches have been tried to predict performance on parallel systems. We will discuss in more detail in subsections 2.3.1 - 2.3.4 the following relevant methods: simulation, non-linear regression on time, artificial neural networks, and general linear models on log (time). Each method has advantages and disadvantages, but we feel that the last will be shown to be the best for projection purposes.

2.3.1. SIMULATION OF APPLICATIONS

As stated previously, there is a difference between ideal (theoretical) speed-up and observed speed-up. Various factors, i.e. latency or synchronization, are the cause for the observed deviation from ideal. These factors are referred to collectively as 'overhead'. Several simulation applications have been developed to run a simulation of a specific application with set variables in order to try and quantify the overhead. These overhead values are then utilized to predict the performance of the application when it is scaled.

Unfortunately, development of a simulation for an application is not an easy task. Complete knowledge of how the application works (i.e. 'white-box' knowledge) must be utilized in order to conduct an accurate application simulation. Knowledge of an application at this level is not to be expected from a typical scientist interested in scaling an application for research, so simulation methods are not examined further in this thesis.

2.3.2. NON-LINEAR RESGRESSION ON TIME

In non-linear modeling, one attempts to model TIME as a non-linear function of parameters related to input variables, such as number of processors (P), size of application, etc. Of course, one could try linear models of the form $T = \beta_0 + \beta_1 * P + \beta_2 * SIZE + \varepsilon$, but they will yield inadequate fits for two reasons. First, the actual relationship is not close to linear at all, with the model $T = SIZE * P^{\alpha} + \varepsilon$ being a more logical theoretical base from which to build more complex models. Secondly, and more importantly, even if the functional form were approximately known, standard methods applied to the above equation all assume that the errors at each point are normally distributed with mean zero and constant standard deviation. However, it is well-known to those in the scalability field that errors in predicting run times are proportional to the size of the time – this is why everyone in this field refers to 'relative error' or 'relative % error' when comparing methods. In such cases, the good done by attempting to fit a non-linear model will be completely worthless unless one also correctly specifies the variation in the errors. While this is possible to do in some cases, it is extremely difficult, in general. A much easier solution, as we shall see below, is not to model TIME directly, but the log of TIME (any scale: ln, $\log_2$, $\log_{10}$, etc. will work – for convenience we use $\log_2$ in this research). If log-transformed TIME is used as the response variable, most applicable models become linear in the

parameters and have approximately constant variance, rendering them much more statistically tractable. This is discussed in more detail in subsection 2.3.4.

2.3.3. ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANNs) are another analytical modeling method for parallel computation. ANNs

> [map input variables to a run time] using a network of neurons, simple processing elements, connected by weighted edges. As in regression, consider an observed sample with known predictor and response vectors x and y. For a particular neuron j in the network, let b_j denote its output and $a = a_1, \ldots, a_p$ denote its p potentially transformed versions of x. If the inputs arrive via edges with weights $v_j = v_{j,1}, \ldots, v_{j,p}$, the neuron computes its output as a weighted sum of inputs in Equation (3).
>
> $$b_j = f\left(v_j \cdot a\right) = f\left(\sum_{k=1}^{p} v_{j,k} a_k\right) \quad (3)$$
>
> An activation function f may transform the weighted sum to increase the set of mappings the network is able to represent. [14]

For example, Lee et al. used a sigmoid function of the form $f(x) = 1/(1+e-x)$ to deal with interaction and non-linearity of the input variables in their ANNs [14]. This eliminates the need for domain specific information, unlike the piecewise polynomial regression and our focused linear regression models.

Support for ANNs is largely based on the fact that they are considered to be a 'black box' method; i.e. a method which will work without requiring any specialized knowledge of the application. However, how this method creates its predictions is not well understood [14]. If properly tuned, ANNs will often create similar predictions to linear or non-linear regression models, for data points in the training region. However, since most ANNs are complex models that incorporate higher level functions and interaction terms, we believe that forecasting with

these models will be extremely unstable when applied to points outside of the training range, which is our ultimate goal.

2.3.4. GENERAL LINEAR MODELS (GLM) ON LOG (TIME)

As noted above, the errors in predicting run time are proportional to TIME. Transforming TIME to log (TIME) tends to solve two problems, non-linearity and non-constant variance, that occur with non-linear regression models. Since the particular log-scale does not matter, for convenience, all of the models in our methodology utilize general linear models on $\log_2$ (TIME).

Furthermore, a general linear model on log (TIME) will tend to be more simplistic than a non-linear regression or neural-network, and thus will be more conducive to forecasting. Since forecasting the run-time when an application progresses to a higher number of processors while maintaining proper scaling is our main goal, we feel that the GLM on $\log_2$(TIME) will be the best of the four general approaches discussed in this section.

CHAPTER 3

PROPOSED DATA ANALYSIS METHODS

This chapter describes three types of proposed data analysis methods which were used within the general linear model framework to make predictions and forecasts for the six applications studied. Which of these methods to use depends upon what assumptions one makes concerning the data set which one will have available from which to make future projections.

3.1 COMPLETE TRAINING SET METHOD

Under this method, one assumes that one has available a large number of runs at processor level P=2^J, 2^(J-1), 2^(J-2), …, where for each run, different combinations of input variables and output (TIME) are recorded. In this case, the complete training set is used to build a linear model (for TIME in $\log_2$ scale). This linear model could be a simple model such as:

$$LT = \beta_0 + \beta_1 * LP + \beta_2 * LSZ + \varepsilon \qquad (4)$$

where LT=$\log_2$(TOTAL), LP=$\log_2$(P), and LSZ=$\log_2$(SIZE) or something much more complex, depending upon the application and the fit. It is often the case, as explained in Section 3.4, that there is not one equation, but separate equations for computation and communication times, or for other sub-components. In any case, once the model's parameters (β_0, β_1, and β_2 in Equation (4)) are estimated, this model is used to make projections for points in the forecast data set (all of which, by definition, will have P=2^(J+1)), where P_0=2^J is the maximum processor size used in the training data set.

The overall fit of a model is typically quantified (by computer scientists) by its median absolute percentage error (MAPE) over the points in the forecast data set. Statisticians would be

more likely to use RMSE (in $\log_2$ scale). If the estimation is unbiased and the distribution of errors (in $\log_2$ scale) is approximately normal with standard deviation estimated by SD=RMSE, then $MAPE = (2^{0.675*RMSE} - 1)*100$, but neither unbiasedness nor normality seem to be justified for the applications we have examined. It should be noted that one usually estimates the linear model's parameters from the training set so as to minimize sum-of-squared-error in log-time scale (which is equivalent to minimizing relative % error under some assumptions), but that the typical error (RMSE) estimated from the training set may not be an accurate estimate of the typical error found in the forecast data set. In fact, as we shall see, overly complex models which yield smaller RMSEs than simple models for the training data sets often perform much worse on the forecast data sets.

3.2 FOCAL REGIONS METHOD

For some applications (typically those with only SIZE and P as input variables, and for which communication time is small relative to computation time in most regions of the data set), the complete data method described in Section 3.1 works fairly well, on average (i.e. the MAPE is less than 10%), but may yield much larger % errors in individual cases. For other applications, even the MAPEs are large, so, of course, individual point predictions in the forecast space might be much worse. One obvious way to combat this problem would be to be less ambitious in attempting to develop a forecast model. In Section 3.1, the model which is developed, in theory, could be used to make forecasts for any set of input variables, and is evaluated over a range of sample points in the forecast data set (i.e. where P=2^(J+1)). A more realistic goal might be to develop a prediction model only for points in the training set whose run-times are 'near' some

focal time of interest, and, then, to evaluate the forecasting ability of the model so derived only 'Focal Regions' method.

The simplest focal region method is based on TIME alone. In such a case, a focal time of interest, T_0, is selected. Then, instead of using all the cases in the training set to develop the predictive model, one would use only those whose run-times fell within a certain region (for example, within +/- 20% of T_0) to develop the model for the training data. Then, once the model was developed, it would be evaluated in the forecast space only over those data points which also met the focal requirements. For example, for the LU application, from Table 1, we see that if we used the complete data method, we would use 118 data points in the training ($P \leq 1024$) set and would evaluate this model over the 31 data points in the forecast ($P=2048$) data set. However, if we set $T_0=40$ seconds, and used a +/-50% focal threshold, we would use only 29 of the 118 points in the training set to develop the model and would evaluate it over only 9 of the 31 points in the forecast data set.

For more complex applications where there are more data points, one might define the focal region not to restrict on the basis of run-time, but so that it restricts the range of other input variables. The advantage of the focal region method is that it uses data points which are more 'similar' to the data point of interest when building the predictive models. The disadvantage, of course, is that if the focal region restrictions are too severe, there will be too few points eligible from which to effectively estimate the parameters needed. A compromise between the complete data method of 3.1 and the focal region described above would be to use some sort of weighted estimator which uses the complete training set data set, but which places higher weights on points with times near T_0 and lesser weights on those with times far from T_0. Although this would be relatively easy to do if T_0 were the only variable affecting inclusion in the focal region,

we have not implemented it in this thesis because determining the weights for other focal regions would be quite difficult.

3.3 SELF-GENERATED FOCAL REGION METHOD

One objection to both the Complete Training Set Method (3.1) and the Focal Region Method (3.2) is that they both assume that 'someone' has previously run many different combinations of the input variables at the processor sizes less than that of interest (P_0=2^J). For the purposes of this thesis, that is true – we are using as our training data set the many runs generated for each of the six applications by Brad Barnes of the UGA CS Department as part of his PhD research. From Table 1.1 of the introductory chapter, we can observe that we have almost 2000 runs available for the CG application, and even the smallest application (LU) has 118 points available in the training data set. One might then wonder about the practicality of our results in a more realistic case, where one had only a few data points initially available. In that case, one would need to generate one's own focal region points, build a model from that, and then evaluate that model at a few points in the forecast region. This is a more complex, but more realistic, undertaking than the two described above. The first two methods mentioned were applied to all six applications, while we attempted to use the self-generated focal training set method for only two applications, BT and CG. BT is the easier of the two, since it involves only one input variable (SIZE) other than P, while CG depends on two (SIZE & NZER). More details on our implementation of this method for BT and CG are given in Section 3.5. Obviously, the self-generated focal region method won't be as successful as the methods which have access to a much larger training base, but the MAPEs observed with this method might be more typical representations of what actual users can expect to observe upon implementation of our proposed forecasting methods.

3.4 TYPES OF MODELS USED

After one has decided which of the three general data analysis methods to use, one must decide what sort of general linear model to use. This section reviews a number of models which were used in the course of this study. All of the statistical models described here have been fit using the statistical package, SAS. The general linear model (PROC GLM) procedure of SAS was used to estimate parameters for some models , although the final forecast models which were developed could all be written (at least at the component level) as linear regressions for $\log_2$(TIME), so all fit output was generated using PROC REG of SAS.

3.4.1 ORDINARY LINEAR REGRESSION MODEL

The most straight-forward models that we developed would be those of the form:

$$LT = \beta_0 + \beta_1 * LP + \beta_2 * LSZ + \varepsilon \qquad (5)$$

where LT is the $\log_2$ transform of TIME. We typically use $\log_2$ (TOTAL) as the TIME variable of interest, but for some sub-component models, we would separately use $\log_2$ of computation and communication time as the response variables. LP=$\log_2$ (P), where P is the number of processors used for the run in question. LSZ is defined by LSZ=$\log_2$ (SIZE), where 'SIZE' is a generic measure of size for the application in question. Sometimes the variable is actually called 'SIZE', whereas for others it is given by WSS=DIM1*DIM2*DIM3 or WSS=Dim1^3. In any case, for most of these applications, even if 'SIZE' is a 2- or 3-dimensional quantity, the individual levels of DIM1, DIM2, and DIM3 do not matter, so their product (or the log thereof) is the key explanatory variable. This, incidentally, is not at all true for the 'PROC DIM' variable which appears as a 2-dimensional variable in 'Sweep3D' and as a 3-dimensional variable in the 'SMG' application. In those cases, it appears that different loadings of the PROC DIMs, even

when their product is constant and other variables are kept constant, have large effects on run time.

Equation 5 above has some theoretical basis under perfect conditions where communication is negligible. In such cases, one would expect β_2 to be near 1 and β_1 to be near -1. If those conditions were exactly true, then it is obvious that one could keep time constant by both doubling the number of processors and doubling the size. For all applications which we examined (except for SMG), $\beta_2 > 0$ and $\beta_1 < 0$, but only rarely was it the case that (β_1, β_2) was extremely close to the $(-1,1)$ values predicted under perfect speed-up conditions.

3.4.2 MODIFICATIONS OF ORDINARY LINEAR REGRESSION MODEL

The ordinary linear regression model given by (5) is a reasonable starting point for modeling the behavior of time in the applications which we examined, but it was usually too crude to yield forecast values of the precision which we desire. Various refinements were tried when needed for various applications. Among these are separate modeling by processor level, use of polynomials of higher order models, and subcomponent models.

3.4.2.1. SEPARATE MODELING BY PROCESSOR LEVEL

Under this modification, a model of the form:

$$LT = \beta_0 + \beta_1 * LSZ + \varepsilon \qquad (6)$$

is used separately for each level of processor. This model tended to yield exceedingly good fits (high R^2, low RMSE's) for each processor level, but would not directly solve the forecasting problem, since there would be no available estimates of (β_0, β_1) for the desired (P=2^(J+1)) number of processors. What could be done is to fit a regression to the sequences (β_0, β_1) for

P=2^J, 2^(J-1), 2^(J-2), …, and try to project this forward to P=2^(J+1). We tried this method both unweighted and with geometric weights which put higher weights on data from larger processors, since those believed to be more relevant to making the forecast at P=2^(J+1) are data points from larger numbers of processors. Of course, this separate method, since it has about seven times as many parameters as the simple model of (5), yields much smaller RMSE's over the points in the training dataset. It usually (but not always) performed better in the forecasting dataset than did the original regression method, but not by nearly the margin predicted by the RMSE's. A slight geometric weighting so as to put more weight on higher processor observations seemed to be slightly better than equal weighting, but not consistently enough that we could uniformly recommend it.

3.4.2.2. POLYNOMIAL OR HIGHER ORDER MODELS

We examined models for LT where we considered quadratic and/or interaction terms for LP, LSZ, or similarly we considered polynomial or interaction terms for the (β_0, β_1) series generated in Equation (6). While these models yielded slight improvements in RMSE's for the points in the training dataset, they were invariably much worse when applied to the forecasting dataset. This is not at all unexpected, and is why we are confident that splines or neural network solutions to this problem will almost surely fail. The polynomial or interaction terms are actually over-parameterizations for the data available. They appear to make the fit better for the data at hand (in the training set), but the improvement due to the extra terms is simply helping explain variability in the training dataset that is not reflected in the forecasting dataset. Both spline and neural network methods, since they involve many more parameters (even if not explicitly

categorizable), will be even more prone to error when applied to the forecast dataset with number of processors (P) outside the range of the data observed in the prediction dataset.

3.4.2.3. SUBCOMPONENT MODELS

For most applications which we examined, the contribution to run time due to computation is larger than that due to communication at all processor sizes, but the relative sizes of these components varies as the number of processors increases. In all cases, it is true that the regression models described earlier work better for describing the behavior of computation time than they do for describing communication time. In the cases where computation time dwarfs communication time at all processors levels, there is not much point in splitting the two components. However, for some applications, splits of the form

$$LTCOMP = \beta_0 + \beta_1 * LP + \beta_2 * LSZ + \varepsilon \quad (7)$$

$$LTCOMM = \gamma_0 + \gamma_1 * LP + \gamma_2 * LSZ + \varepsilon \quad (8)$$

were tried. Currently, we split the regressions for computation and communication only if the percentage of time spent in communication is greater than 50% at P=2^J [4]. We found that with smaller percentages it is sufficient to regress only on total time [4]. When this was done, we could frequently obtain exceedingly good fits and forecasts for the computation time, but models for LTCOMM were not nearly as good, perhaps indicating that the functional form of Equation (8) is not appropriate. It is clear, of course, that while the β_1 of Equation (7) is negative, the γ_1 of Equation (8) is positive – more processors decrease computation time while they increase communication time. However, for many of the applications which we examined, the $(\beta_0 + \beta_2 * LSZ)$ term was so much larger than the $(\gamma_0 + \gamma_2 * LSZ)$ term that the increasing or decreasing effects of the other terms made little difference. For the applications where

communication time was a significant component, we discovered that modeling via Equation (8) was sometimes poor – not necessarily because the model was inadequate but because there were occasionally large unexplained fluctuations in communication time when replicate runs (with all input variables held constant) were generated. This almost never occurred with computation time. There, in log-transformed scale, the standard deviation in response times is almost constant when one examines replications under different input conditions. The occasional large fluctuations in LTCOMM are puzzling, but appear to occur more frequently when the number of processors is large. Possible reasons for this behavior are discussed in Chapter 5, but the short answer is that we have not been nearly as successful at modeling the communication component as we have been with modeling the computation component. Other attempts to even further disaggregate the computation and communication components were attempted for some applications, but none of these were particularly helpful – the best we can do currently is to separately model computation and communication. This can usually be done fairly well (especially in focal regions) for LTCOMP. For LTCOMM, this also works better in the focal regions, but it still does not work particularly well for most applications. The saving grace of the entire forecasting method is that the computation time component typically dominates communication so much that getting the former projection 'right' masks errors in the latter. As is demonstrated later, one of our least successful projections occurs for the SMG application, and SMG is the application for which communication time grows most quickly as processors increases.

3.4.3 FOCUSED REGRESSION MODELS

As mentioned in Section 3.2, the models examined can frequently be improved by restricting the training set to regions of comparable time. This move to narrow our data is

counterintuitive because as statisticians we usually believe that more data is better [4]. However, more data in an irrelevant region simply clutters the analysis. Dataset restriction tends to work relatively well when there is only one variable (SIZE) in addition to processor number (P) allowed to vary. If there are other variables involved, determining a useful focal region is more difficult. For example, for the SMG application we found that processor dimensions were very important, and we focused on regions with $\left(P_x, P_y, P_z\right) = \left(1, P_y, P \middle/ P_y\right)$ and combined points in the same region 'k' if $\log_2 \left(P_y \middle/ P_z\right) = k \pm 1$. This is illustrated in Figure 3.1, with the two highlighted regions being k=0 and k=4. For example, in the k=0 region, if we desire to make forecasts at the $\left(P_x, P_y, P_z\right) = (1,32,32)$ when P=1024, we should use results from the training set with $\left(P_x, P_y, P_z\right) = (1,16,32)$ and $\left(P_x, P_y, P_z\right) = (1,32,16)$ at P=512, $\left(P_x, P_y, P_z\right) = (1,16,16)$ from P=256, $\left(P_x, P_y, P_z\right) = (1,8,16)$ and $\left(P_x, P_y, P_z\right) = (1,16,8)$ at P=128, …, etc. For our focal region, we kept P_x=1, but, we can extend our method to handle the general three dimensional case where P_x is allowed to vary, with some extra modifications.

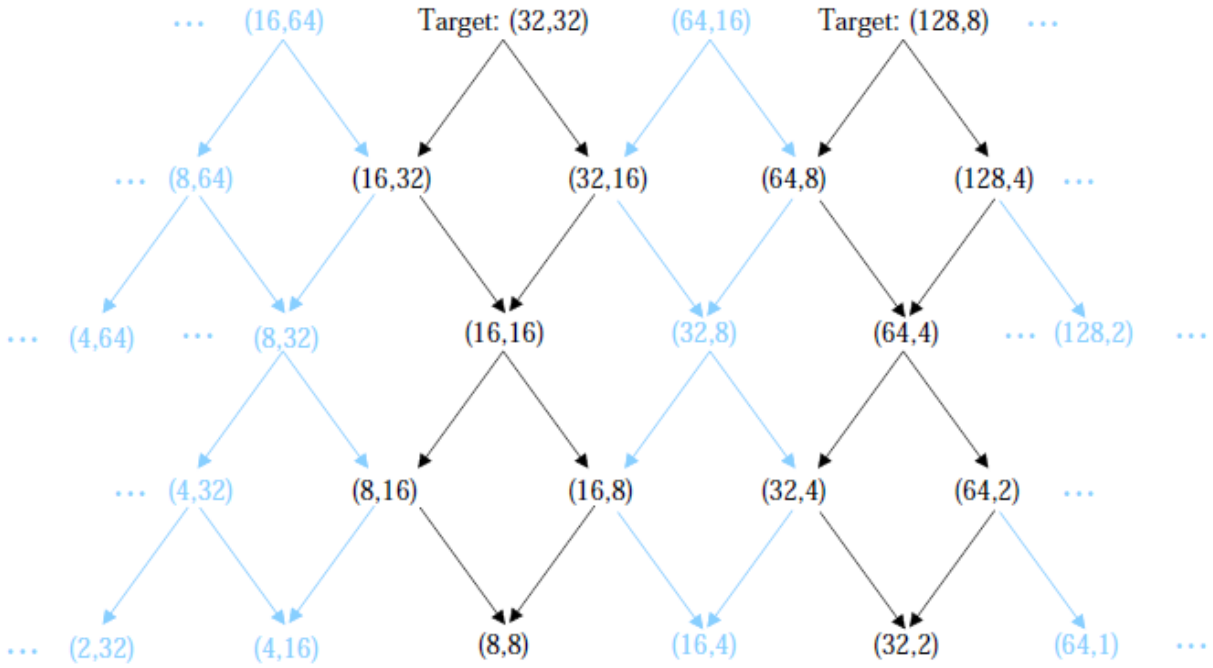

Figure 3.1 SMG focal regions on a processor grid.

3.5 ALGORITHM FOR SELF-GENERATED DESIGN POINTS

The procedure described below gives an algorithm for generating design points within the training region in the simple case where there is only one variable (SIZE) in addition to P. For our applications, this procedure should work for self-generation of points for BT, LU, and SP, and we performed the algorithm on BT as reported in Section 4.3. Modifications would be needed to run this on a more complex application. In Section 4.3, we also describe how we performed this self-generation for the CG application.

As mentioned in Section 3.3, this procedure is more realistic than those in 3.1 or 3.2, in that we do not assume that there is a large training set from which to choose points on which to build our model. Rather, we assume that the situation is such that a client

> [has run] an existing application with input variable [‘SIZE=SZ_0’] on a system with number of [processors P=2^J], [such that] the application takes [TIME=$TOTAL_0$] seconds to run. The client wishes to know, based on runs that

> can be performed with [P<=2^J], approximately what value of [SIZE] should be used if the processor capacity were increased to [P=2^(J+1)] such that the [TOTAL] time for the application to run would be approximately the same ([TIME=$TOTAL_0$]) as currently. We assume that the client also has available results of a run with {[SIZE=SZ_X, P=2^(J-1), TIME=$TOTAL_X$]} such that [$TOTAL_X$] is approximately equal to [$TOTAL_0$].[15]

We consider the two client-provided results as central values. To create upper and lower bounds for those central values we use +/- 10% of the central value SIZE as input values. This will give us an error bound around our points and also be a sample in the input variable (LSZ*LP) space. We will run the four new sample runs to obtain the time (TOTAL) needed for each of the runs. These four runs along with the two client-provided runs will allow us to generate a regression model to regress backwards, as shown in Table 3.2 below.

Table 3.2 – Six Initial Sample Runs

LSZ	LP	LT
LSZ_0U	J	$TOTAL_0U$
LSZ_0*	J*	$TOTAL_0$*
LSZ_0L	J	$TOTAL_0L$
LSZ_XU	J-1	$TOTAL_XU$
LSZ_X*	J-1*	$TOTAL_X$*
LSZ_XL	J-1	$TOTAL_XL$

* Provided by client

After these six values are generated, a regression equation can be created to predict a high, medium, and low run-time at lower processor values. For example, in Table 3.2 above, if J=10, the client gave us points at P=512 and P=1024, and we then generated four additional points. From these six points, we can create a regression model which would allow us to predict three more points each at (say) P=16, 32, 64, 128, and 256. Because the initial regression based on six points will likely be crude, it is quite possible that the fifteen newly generated points may have times that are not that close to the desired focal time, T_0. That should not matter too much, as we are simply trying to create an approximate focal region training set. Finally, using all

twenty-one points, we can generate regression (or adjusted regression models, as discussed above) to make projections for what would happen with P=2^(J+1) (P=2048 in the example above). In particular, we can use the model to project what value of SIZE, when P=2^(J+1) would yield the desired time (T_0).

We used this algorithm with T_0=101 seconds and J=10 (P_0=1024) on the BT application. We used a modification of this algorithm with T_0=100 seconds, NZ=14, and J=10 (P_0=1024) on the CG application. In both cases, the algorithm yielded twenty-one training points, (three each at P=16, 32, …, 1024), and we used these to build a model which would allow us to project the value of SIZE needed at P=2048 so that the total run-time was equal to the desired T_0 (101 seconds for BT, 100 seconds for CG). One difficulty encountered here, but not in using the methods of Sections 3.1 and 3.2, is that in those cases , there were at least a few runs available at P=2^(J+1) on which we could evaluate our forecasting performance. In the more realistic case of Section 3.3 for which the above algorithm is specified, there are, of course, no runs at P=2^(J+1) – that is why the client came to us in the first place! For evaluation purposes in this thesis, we used our created regression model to predict the values of SIZE which would yield the five expected times of ($0.8*T_0$, $0.9*T_0$, T_0, $1.1*T_0$, $1.2*T_0$), and then ran these five runs at P=2048 to see how good the fit was. The results for both BT and CG using this algorithm are reported in Section 4.3.

CHAPTER 4

ANALYSES

This chapter presents and discusses the results for three proposed forecasting methods on the applications. We desire a median absolute percentage error below 20%, with a MAPE below 10% being our ultimate goal. We expect that the complete training set method should produce good results for simple applications, but will yield much less satisfactory results for more complex applications. The focal regions method should improve our results for both simple and complex applications. We hope that the self-generating focal training set method will produce acceptable results for our representative simple and complex applications, BT and CG.

4.1. RESULTS USING COMPLETE TRAINING SET METHOD

For the six applications, the complete training set method does not automatically exclude any points from the analyses, since there is no replication. However, for CG, were replication occurred, 68 of the 2542 points (2.7%) were excluded from all analyses because they were obvious outliers. Even though these blatant outliers (in CG) were removed prior to any analyses, there were still a few other points for all applications' analyses that would be considered outliers (observations far away from the rest of the observed data) or influential points (extreme observations that affect the slope of the regression line) for particular analyses. The points which were flagged as extreme outliers or as influential points when evaluated over the training set for a particular application were subsequently deleted from the analysis and not included in construction of the final prediction model used for making forecasts. However, this exclusion of

points was rare; the vast majority of the points are neither outliers nor influential points, so the impact of the few deleted points should be marginal.

Table 4.1 lists the median absolute % error (MAPE) for each application when the simple linear regression which best fits the training data set is applied to the forecast data set. In addition to the MAPE value, the last 5 columns of Table 4.1 display the distribution of the relative % error over all data points used in evaluating the forecast data set. Note that these relative % errors (calculated as ((PRED-OBS)/OBS)*100%) may be positive or negative. The fact that most are positive, especially for the complex applications, indicates that the complete data method is moderately to seriously biased toward over-estimating the time required for the application to run at the higher levels of nodes used in the forecasting data sets. We see from Table 4.1 that the median absolute percent error (MAPEs) for BT, LU, and SP are 9.95%, 12.08%, and 18.25%, respectively. The complete training set method results are excellent for BT and good for LU and SP because they fall below target thresholds of 10% and 20% relative error, respectively. Table 4.1 also shows us that the MAPEs for CG, SWEEP3D, and SMG are all poor, over 65%. This is exactly what we suspected would happen. In order to forecast complex applications accurately, conditions must be controlled for more carefully than using the additive regression models shown in Table 4.2. We more adequately control for other conditions in the complex applications in the 'Focal Regions' method explained in Section 4.2.

Table 4.1 – MAPE and Distribution of Relative % Error for Complete Training Method by Application

Application	PredictN	ForecastN	MAPE%	Min	25%-ile	50%-ile	75%-ile	Max
BT	792	22	9.95	-25.89	0.56	8.38	10.21	12.52
LU	118	31	12.08	-48.72	-20.72	-2.44	9.13	17.10
SP	352	124	18.25	-48.96	4.63	17.53	20.03	31.10
CG	1890	29	145.75	79.30	121.78	145.75	192.00	1385.83
Sweep3D	224	20	69.74	40.60	61.55	69.74	75.31	81.96
SMG	2542	36	65.17	-0.17	55.83	65.17	82.28	110.93

Table 4.2 – Regression Coefficients for Complete Training Method by Application

Application	TrainN	Intercept, β_0	LP, β_1	LSZ, β_2	X, β_3	R^2	RMSE	E(MAPE%)
BT	792	-12.69	-0.92	0.95	n/a	0.997	0.154	7.74
LU	118	-13.37	-0.95	0.97	n/a	0.993	0.260	12.93
SP	352	-12.97	-0.87	0.92	n/a	0.989	0.306	15.39
CG	1890	4.76	-0.68	1.34	0.65	0.936	0.465	24.29
Sweep3D	224	-13.81	-0.76	2.78	0.03	0.829	0.459	23.95
SMG	2542	-11.01	0.22	0.78	0.04	0.702	0.522	27.66

The estimated regression equation coefficients under the complete data method for the six applications are listed in Table 4.2. This assumes a generic model of the form:

$$LT = \beta_0 + \beta_1 * LP + \beta_2 * LSZ + \beta_3 * X + \varepsilon \qquad (9)$$

where 'X' is non-existent for the simple applications (BT,LU,SP). For CG, Sweep3D, and SMG, respectively, the 'X' utilized was $\log_2(NZ)$, $\log_2(PD1/PD2)$, and $\log_2(PD2/PD3)$. Clearly, as seen from the poor fits in Table 4.1, for those applications, either those were not the appropriate factors to use as 'X', or there are very strange unaccounted-for interactions. On the other hand, for the simple models, the Complete Training Method appears to be reasonable, as seen in Table 4.1. From Table 4.2, for these models, we see that the estimated (β_1, β_2) are near -1.0 and +1.0. It should be noted that while (β_1, β_2) are near (-1, +1) for the simple applications, they are not extremely close. That is, even for the simple applications, we cannot keep time constant simply by both doubling processors and doubling size; we do not observe perfect speed-up conditions for any of the applications. It should also be noted, by comparing the E(MAPE) of Table 4.2 with the actual MAPE of Table 4.1 that even when the training model fits incredibly well, as it does for BT, LU, and SP, the observed MAPE's are larger than the E(MAPE)'s. This is the inherent danger in attempting to extrapolate beyond the range of the observed data. Of course, if the training set fit is not extremely good in the first place, it is unlikely that the prediction set MAPE will be at all good.

4.2. RESULTS USING FOCAL REGIONS METHOD

Figure 4.3, Figure 4.4, and Figure 4.5 show plots of LT*LP, with selected focal regions highlighted, for the BT, LU, and SP applications, respectively. We create a focal region for simple applications by attempting to find a constant LT value that intersects the LT range encountered for all the LP values. The range of the LT values chosen is the focal region, and is used to maintain time-constrained scaling. For BT, we chose $LT = 6.5 \pm 0.3$, which is equivalent to the asymmetric TOTAL interval (73, 101, 111) seconds. For LU, we chose the three points closest to LT=6, which is equivalent to the three points closest to TOTAL=64 seconds. For SP, we chose $LT = 6.65 \pm 0.2$, which is equivalent to the asymmetric TOTAL interval (87, 100, 115) seconds.

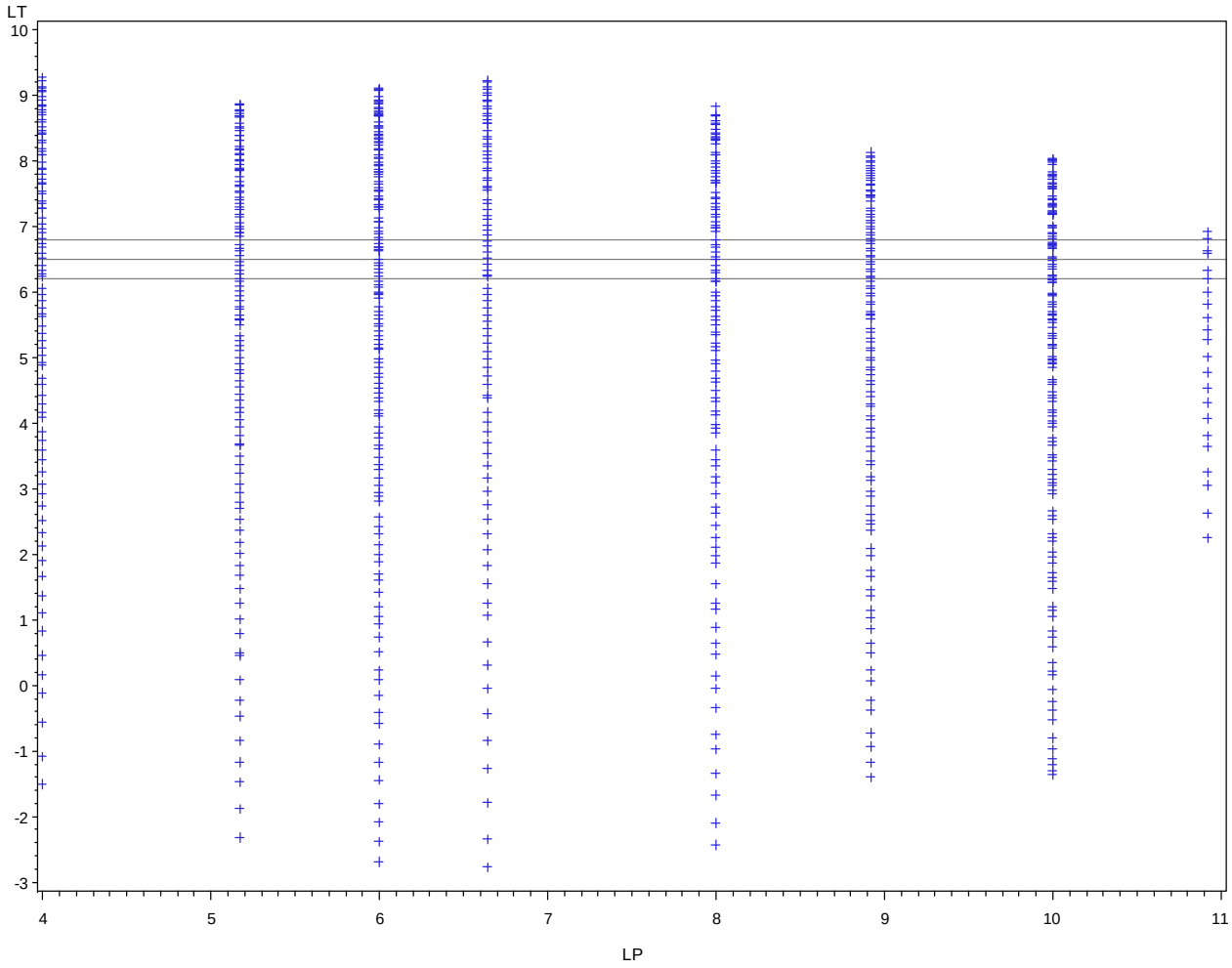

Figure 4.3 LT*LP for the BT application with ‘Focal Region’ 6.2<LT<6.8

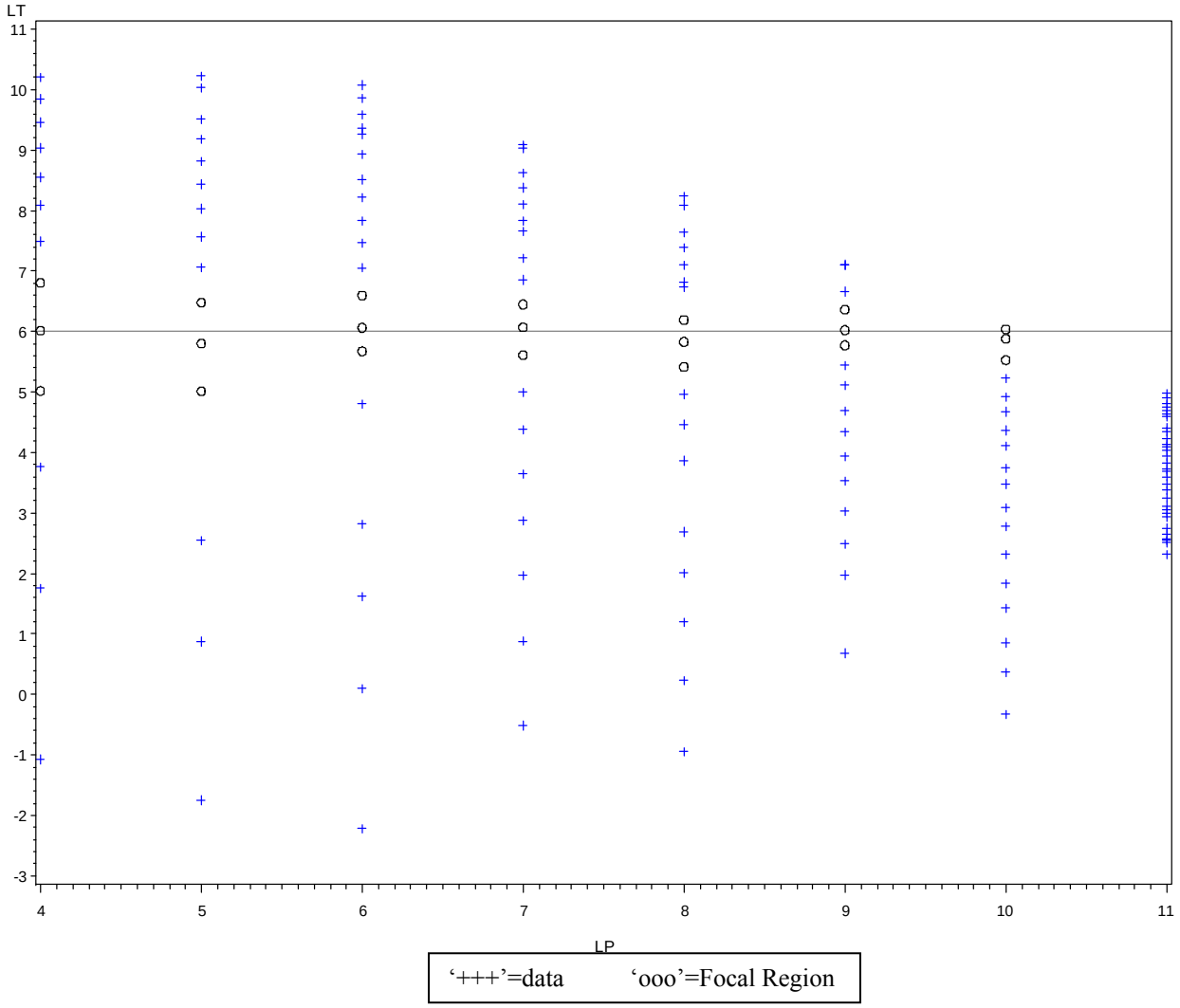

Figure 4.4 LT*LP for LU application with 'Focal Region' close to LT=6.00

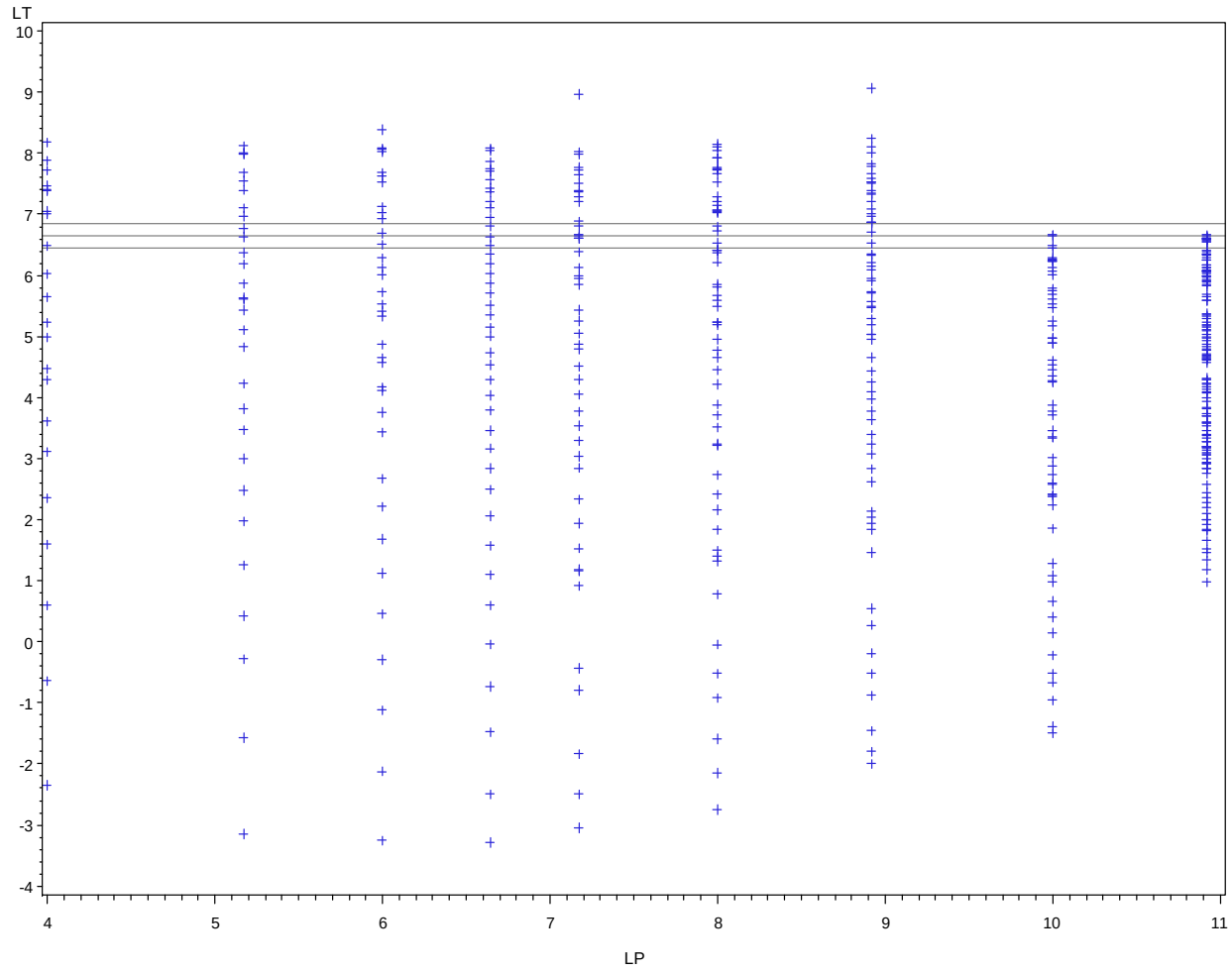

Figure 4.5 LT*LP for SP application with 'Focal Region' of 6.45<LT<6.85

For BT, LU, and SP, the fit of the focal region method model could be negatively affected by three aspects of the data. First, as seen from Figure 4.4 for the LU application, the upper and lower bounds of the focal region might not be uniform for all the processor levels and the focal region might not appear in all processor levels. Second, the few points at each processor level may not be sufficient to obtain a good regression equation. Finally, there could be an outlier that falls within the focal region.

When creating our focal regions for more complex applications, we cannot use a constant LT that intersects the observed LT range for all processor levels. Thus, we must be more careful

about outliers and influential points within the focal region. After a regression model is fit, points which have a Cook's D value greater than $\frac{2p}{n}$ (where p is the number of parameters and n is the number of observations) are deleted from the analysis, even if they fall within the focal region. For SMG, because the data are run in replicates, an errant point shows that there are problems with the parallel system. Since we are not interested in the efficiency of the parallel system in this thesis, we are ignoring those types of problems and discarding data points that we believe are a result of those difficulties. Unfortunately, since replicates were run only rarely for the other five applications, it is likely that a few gross outliers have been accidentally included in the data sets of these five applications.

The top panel of Table 4.6 shows the results of using the focal regions method on BT, LU, and SP. The focal time tolerances were set fairly wide so as to get a reasonable number of points in the focused regression. Using the focal regions method, their median absolute percentage error (MAPE) rates are 4.64%, 5.31%, and 6.74%, respectively, for BT, LU, and SP. These values are excellent because they all fall below our target of 10% and all three show an improvement over the global method. SP also shows us that our assumption of the effects of influential and outlier points as marginal for simple applications is correct, because there are two influential points within the focal region of SP but the focal regions method still showed improvement over the global method. LU does show an improvement when compared with the global method but there are no points in the LU focal region for P=2048. For some programming reason, the LU application was unable to be run at a size larger than 1000. When we calculated the sizes needed for the 2048 processors to retain time-constrained scaling, we found that the necessary size would be greater than 1000. Thus, we are unable to validate our results for LU at P=2048 with the correct predicted SIZEs. However, we did use the focal region

regression model to forecast results for the three largest LT values available in the LU forecast dataset, yielding the 5.31% MAPE noted.

The bottom panel of Table 4.6 shows the MAPEs of CG, Sweep, and SMG, using the focal regions method, to be 29.30%, 3.44%, and 11.77%, respectively. These values are excellent for both Sweep3D and for SMG. However, CG still does not fit well, even after controlling for non-zeros (NZ). We controlled NZ in CG because it affects the run times with increasing NZ increasing TOTAL. However, the exact nature of how NZ behaves is unknown and could be affecting the run times in a non-linear way. Future work must be done to extend our attempt at controlling the effects of NZ. Despite the lingering of some error in CG, we see a marked improvement over the complete training set method, which is encouraging. Controlling for $IX=\log_2\left(\frac{\text{Procdim1}}{\text{Procdim2}}\right)$ in Sweep3D and for Procdim1 (P1) and $J=\log_2\left(\frac{\text{Procdim2}}{\text{Procdim3}}\right)$ in SMG, respectively, worked well. These factors affect the run times, but the precise manner of the effect is unknown. Thus, we decided to keep their levels approximately constant, which apparently is enough to garner excellent fit. It should be noted that it is sometimes impossible to keep the condition constant over processor levels. For example, we illustrated in Figure 3.1, the 'J' of SMG is either odd or even in alternating processor sizes.

Table 4.6 – Conditions Used to Create Focal Regions for Applications and Resultant MAPE's

Application	Focal T_0 or condition	Train N	Tolerance%	Forecast N	MAPE%
BT	LT_0=6.50	74	23.11	3	4.64
LU	LT_0=6.00	21	40.64	3	5.31
SP	LT_0=6.65	21	14.87	10	6.74
CG	NZ=(14,16,18,20,22,24)	161 or 183	n/a	29	29.30
Sweep3D	$IX=\{-3,-1,1,3\}$	12	n/a	20	3.44
SMG	P1=(1,2,4)*J=(-3,-2,-1,0,1,2,3)	51 or 56	n/a	36	7.50

Table 4.7 – Regression Coefficients for Focused Regressions by Application and Condition

Application	Train N	Intercept, β_0	LP, β_1	LSZ, β_2	condition	R^2	RMSE
BT	74	-13.06	-0.95	0.97	T_0+/-23%	0.973	0.031
LU	21	-13.87	-1.00	1.00	T_0+/-41%	0.988	0.054
SP	21	3.72	-0.13	0.41	T_0+/-15%	0.120	0.105
CG	183	6.28	-0.63	1.30	NZ=14	0.969	0.234
	183	6.73	-0.68	1.31	NZ=16	0.969	0.240
	183	7.15	-0.72	1.32	NZ=18	0.970	0.247
	183	7.55	-0.76	1.34	NZ=20	0.970	0.247
	161	8.07	-0.79	1.31	NZ=22	0.970	0.267
	161	8.40	-0.82	1.32	NZ=24	0.971	0.267
Sweep3D	12	-14.91	-0.90	2.88	IX=-1	0.999	0.032
	12	-13.88	-0.88	2.83	IX=1	0.999	0.035
	12	-14.76	-0.96	3.00	IX=-3	0.998	0.046
	12	-12.65	-0.81	2.68	IX=3	0.995	0.079
SMG	51	-14.23	0.13	0.94	P1=0, J=-3	0.980	0.131
	56	-14.46	0.11	0.96	P1=0, J=-1	0.991	0.086
	56	-14.25	0.11	0.95	P1=0, J=1	0.987	0.103
	51	-13.59	0.14	0.92	P1=0, J=3	0.964	0.174
	51	-12.18	0.12	0.85	P1=1, J=-2	0.982	0.110
	56	-12.11	0.11	0.85	P1=1, J=0	0.984	0.107
	51	-11.88	0.12	0.84	P1=0, J=2	0.980	0.116
	51	-10.61	0.12	0.78	P1=2, J=-1	0.961	0.153
	51	-10.62	0.12	0.79	P1=2, J=1	0.968	0.140

The estimated regression equation coefficients under the focused regression method for the six applications are listed in Table 4.7. This assumes a generic model of the form:

$$LT = \beta_0 + \beta_1 * LP + \beta_2 * LSZ + \varepsilon \qquad (10)$$

at each condition level, where LSZ=log2(size)for all applications. The condition covariates for CG, Sweep3D, and SMG are the levels of $\log_2$(NZ), IX=$\log_2$(PD1/PD2), and (P1 and J=$\log_2$(PD2/PD3)), respectively. Clearly, as seen from the good fits in Table 4.6 for most of these applications, conditioning on those factors is sufficient to control any undefined interaction or non-linearity. From Table 4.7, for BT and LU, we see that the estimated (β_1, β_2) are extremely near -1.0 and +1.0, with the reasoning for this being the same as in Section 4.1. The SP equation appears almost flat due to the effect of 2 extreme outliers among the 21 points in the

focal training set. Sweep3D is, in fact, behaving similarly to BT and LU after conditioning on IX, with the reason the β_2 coefficient is nearer to +3.0 than +1.0 related to the way in which SIZE is measured in Sweep3D. For SMG, the fit is extremely good although the equations are completely different from other applications. Since $\beta_1 > 0$ for SMG, this means that increasing processors increases run time. As noted in equations (7) and (8) of Section 3.4.2, this can occur only when the communication time is larger than computation time. For SMG, this occurs around P=128, so over the entire focal region from P=16 to P=1024, the estimated β_1 coefficient is positive. Of course, if one had only the data from P=16 to P=64 and attempted to use such data to predict SMG at P=128, there would be no possible way to predict what would happen – that is always a potential problem with "extrapolating beyond the range of data" methods such as those which we are using here. Perhaps something similar occurred for CG, as the separate regression equations for each NZ have exceptionally good fits as shown in Table 4.7, but the overall MAPE (Table 4.2.1) is 29.3%.

4.3. RESULTS USING SELF-GENERATED FOCAL REGION METHOD

For the BT application, we generated the six initial points in Table 4.8 by altering SIZE by +/- 10% (± 106 for SIZE=1060 at P=1024 and ± 85 for SIZE=850 at P=484). In this case, our two starting points are T0=101 seconds (at P=1024, SIZE=1060) and T=101.98 seconds (at P=484, SIZE=850). We used these SIZE/P combinations to find the points' run times, as shown in Table 4.8.

Table 4.8 – Six Initial Points for BT

SIZE	P	TIME
1166	1024	116.00
1060	1024	101.10
954	1024	69.34
935	484	124.34
850	484	101.98
765	484	69.16

We then used the six points in Table 4.8 to find the coefficients of a regression equation of the form:

$$LT=\beta_0+\beta_1*LP+\beta_2*LSZ+\varepsilon \qquad (11)$$

Next, we inverted equation (11) to find the 'SIZE' necessary to yield T=101 for all processor levels less than P=484. For BT, the coefficients $(\beta_0, \beta_1, \beta_2)$ to equation (11) are (-12.71, -0.85, 2.76). If we set TOTAL=101, then the SIZE corresponding to P for each P<484 are listed as the middle line in Table 4.9. The other two rows for each P in Table 4.9 have altered SIZE by +/- 10%. The TIME values shown are those observed when we ran these SIZE/P combinations.

Table 4.9 – Fifteen Points for BT at P<484

SIZE	P	TIME
334	16	169.47
304	16	125.29
273	16	90.05
428	36	155.57
389	36	117.12
350	36	84.30
511	64	148.58
464	64	110.09
418	64	82.49
586	100	145.86
533	100	108.77
479	100	78.81
782	256	134.39
711	256	100.26
639	256	74.86

Table 4.9 shows the fifteen SIZE/P combinations that are run and the points' run time. We then use the twenty-one points from Tables 4.8 and 4.9 in equation (11). The coefficients for $(\beta_0, \beta_1, \beta_2)$ to equation (11) using all twenty-one points are (-13.36, -0.95, 2.92). Finally, we invert equation (11) to find the estimated 'SIZE' necessary to yield T_0=101 for P=1936. For the BT application, the three points in Table 4.10 we found by ± 138 to SIZE=1380 for P=1936.

Table 4.10 – Three Forecasting Points for BT at P=1936

SIZE	P	Pred_Time	Actual_Time
1518	1936	132.37	149.59
1380	1936	100.21	115.97
1242	1936	73.67	85.56

When we compare the OBS and PRED times in Table 4.10 we obtain a MAPE=13.59%. This is a good result but not as good as the focal region method.

The self-generating focal method for the BT application has three complications that have the potential to cause lack of fit. First,

> [t]he points at lower [processors] (obtained from the initial regression of 6 data points at LP=J and LP=J-1) [are not] centered at the right places. [The points tend to have a larger LT time then the LT_0]. [Second,] the three points at each [processor] level may not be sufficient to obtain a good regression equation. (In fact, the fits for P=256 and P=1024 are much worse than at the other 5 processors. … [Third] the initial 'known' point (P=1024, SIZE=1060, TOTAL=101.10) appears to be an outlier. That is, the true value for time for BT when P=1024 and SIZE=1060 is probably closer to 96-97 than 101. [15]

Surprisingly, these complications are marginal and we would get a similar MAPE for the self-generated method as with the focal region method if we did not see an extreme jump in the communication times for the P=1936 self-generated focal method runs. This suggests that for simple applications the self-generated focal method may perform almost as well as it would if there were a large training set from which to select the focal region points.

For the CG application, the six initial points in Table 4.11 are generated in the same way as for the BT application, but rather than conditioning on time alone, we are also conditioning on the fact that NZ=14 at our focal points of interest. (These two points are {P=512, NZ=14, SIZE=1810K, TIME=99.23} and {P=1024, NZ=14, SIZE=2050K, TIME=99.21}). We run the SIZE/P combinations shown in Table 4.11 to find the other four points' run times.

Table 4.11 – Six Initial Points for CG

SIZE	P	TIME
1570K	512	88.57
1810K	512	99.23
1930K	512	112.47
1870K	1024	88.90
2050K	1024	99.21
2230K	1024	112.66

We next used the six points in Table 4.11 to estimate the coefficients of equation (11). Then, we inverted equation (11) to find the 'SIZE' necessary for all processor levels less than P=512. For CG, the coefficients $(\beta_0, \beta_1, \beta_2)$ to equation (11) are (2.87, -0.25, 1.20). If we set TOTAL=100 +/-10% then the estimated SIZE and P for P<512 are listed in the first two columns of Table 4.12, with last column displaying the TIME actually observed when theses runs were executed (all with NZ=14).

Table 4.12 – Fifteen Points for CG at P<512

SIZE	P	TIME
790K	16	893.75
850K	16	978.63
910K	16	1061.35
850K	32	377.75
910K	32	426.86
970K	32	477.80
940K	64	261.49
970K	64	275.17
1000K	64	288.74
850K	128	94.81
910K	128	105.44
970K	128	113.35
1330K	256	135.29
1390K	256	139.16
1450K	256	150.91

The twenty-one SIZE/P combinations in Tables 4.11 and 4.12 combined were used in a regression of the form given in (11). This yielded an equation with RSQR=0.971 and RMSE=0.219 with coefficients $(\beta_0, \beta_1, \beta_2)$ of (4.41, -1.06, 2.45). If we project this equation forward to P=2048 and solve for the SIZEs needed to yield T={80, 90, 100, 110,120} seconds, the forecast points are as shown in Table 4.13.

Table 4.13 – Five Forecasting Points for CG at P=2048, NZ=14

SIZE	P	Pred_Time	Actual_Time
2770K	2048	81.44	24.91
2890K	2048	90.16	25.78
3010K	2048	99.47	26.29
3190K	2048	114.41	27.76
3250K	2048	119.67	28.16

When we compare the OBS and PRED times in Table 4.13 we obtain a MAPE=278.30 %. The self-generated method MAPE is much worse than the focal region method and this is extremely disappointing. We believe that the poor results are a reflection of the lower processors not being

centered in the right place. As can be seen in Table 4.12, the times for the fifteen points are not anywhere near the T={ 90, 100, 110} seconds we want, except for P=128. If we had the time, we would have re-centered our fifteen points rather than continued the process using these points.

So, based on these two examples, it appears that the self-generated focal method might yield results similar to those which could be obtained from selecting a focal region from a larger training set. How good those results are, however, seems to depend on the application.

4.4 SUBDIVISION INTO COMPONENTS

We can separate the TOTAL run time into the two subcomponents, MAXCOMP and MINCOMM (or CPCOMP and CPCOMM). As mentioned in Chapter 1, computer scientists disagree on which subdivision is best. In addition, unless one is extremely careful in synchronizing timers, it is usually the case that the sum of the two components, whether one uses the Max/Min or the critical path measures, is less than TOTAL time which we have used heretofore. The difference is usually about 1%-2%, but can be as high as 10% or more. If one subdivides time into components, one must be careful in resolving this discrepancy. We believe that subdivision into computation and communication components may be beneficial to in improving the forecast errors for applications which did not achieve the 10% relative error threshold when using TOTAL as the time variable. The two subcomponents behave in very different ways. We know that MAXCOMP behaves in a linear fashion (in log-TIME scale) while MINCOMM behaves in a non-linear fashion. We also know that MAXCOMP decreases as P increases (with other variables held constant), while MINCOMM increases as P increases. Separation of the two subcomponents will allow the regression equations to better fit the

respective behaviors of MAXCOMP and MINCOMM. Despite the extra steps involved in splitting and recombining the two subcomponents to make forecasts, we suggest that all linear regression models should split the TOTAL into MAXCOMP and MINCOMM when communication time is a significant amount of the TOTAL. Of the applications which we examined, splitting would appear to be most beneficial in the case of SMG.

The estimated regression equation coefficients under the focused regression method for the two components for SMG are shown in Table 4.14. This assumes a generic model of the form:

$$LT = \beta_0 + \beta_1 * LP + \beta_2 * LSZ + \varepsilon \qquad (12)$$

at each condition level, where LSZ= $\log_2$ (Dim1*Dim2*Dim3). The condition covariates for SMG are the levels of P1 and J=$\log_2$ (PD2/PD3. Clearly, as seen from the exceedingly good fits in Table 4.4.1 for the computational component of this application, subdivision into components improves the computational time fit. However, the subcomponent model's MAPE=8.07% is not better than the focal regions method MAPE=7.50%. This is unexpected, but could be due to the fact that MAXCOMP and MINCOMM do not actually equal TOTAL. Or, it could be due the fact that the generic model in (12) is satisfactory for log-computation time, but not for log-communication time.

Table 4.14 – Regression Coefficients for Component Focused Regressions

Application	TIME	Train N	Intercept, β_0	LP, β_1	LSZ, β_2	condition	R^2	RMSE
SMG	LTCOMP	51	-16.12	0.06	1.04	P1=0, J=-3	0.992	0.086
	LTCOMM	51	-12.95	0.44	0.67	P1=0, J=-3	0.992	0.360
	LTCOMP	56	-16.27	0.04	1.05	P1=0, J=-1	0.996	0.060
	LTCOMM	56	-12.36	0.42	0.64	P1=0, J=-1	0.969	0.226
	LTCOMP	56	-16.13	0.04	1.04	P1=0, J=1	0.995	0.065
	LTCOMM	56	-12.12	0.38	0.65	P1=0, J=1	0.951	0.270
	LTCOMP	51	-15.70	0.06	1.02	P1=0, J=3	0.989	0.098
	LTCOMM	51	-12.27	0.38	0.69	P1=0, J=3	0.897	0.378
	LTCOMP	51	-15.55	0.03	1.02	P1=1, J=-2	0.996	0.060
	LTCOMM	51	-8.79	0.39	0.49	P1=1, J=-2	0.973	0.176
	LTCOMP	56	-15.60	0.03	1.02	P1=1, J=0	0.997	0.053
	LTCOMM	56	-8.58	0.37	0.49	P1=1, J=0	0.972	0.186
	LTCOMP	51	-15.29	0.04	1.01	P1=0, J=2	0.994	0.069
	LTCOMM	51	-9.10	0.36	0.53	P1=0, J=2	0.950	0.231
	LTCOMP	51	-14.99	0.01	1.00	P1=2, J=-1	0.997	0.054
	LTCOMM	51	-7.28	0.37	0.44	P1=2, J=-1	0.960	0.207
	LTCOMP	51	-14.90	0.02	1.00	P1=2, J=1	0.996	0.056
	LTCOMM	51	-7.83	0.36	0.48	P1=2, J=1	0.963	0.195

We conducted further research during which we subdivided the MAXCOMP and MINCOMM subcomponents into eight phases. However, some preliminary work with these eight phases of computation and communication for the CG application showed erratic behavior at higher processor levels. We believe these errant points were caused by a hardware issue. To date, the hardware issue has not been resolved, so we have ceased our efforts in this direction for now. Nevertheless, if these hardware/timing issues can be resolved, we believe that subdivision of MAXCOMP and MINCOMM into phases could be beneficial to predictions and, thus, to forecasting.

CHAPTER 5

CONCLUSION

We conclude this thesis with a chapter that condenses our results, discusses important issues that can affect our results, and summarizes the effect of our efforts to solve the important computer science problem of accurate time-constrained scaling with a linear regression model.

We see that the results for simple applications are good for the complete training set method. This occurs because the applications BT, LU, and SP, "have a high computation-to-communication ratio and have a single input parameter."[4] When the application run time is dominated by the computation time, a simple linear regression will forecast and predict well because the log-computation time is an approximately linear function of the log-problem size. However, the complete training set method is not so good for complex applications.

We see that the results for the focal region method are good for all applications except CG. The results of the complex applications are much better under this method than they are under the complete training set method, and even the simple applications, except LU, show an improvement versus the complete training set method.

Based on the sample results for BT and CG, we believe that our self-generating focal region method will work nearly as well the focal region method. For BT, this should be the case because BT is a simple application and the self-generated points fall within the focal region. For CG, the focal region method results are a combination of those for NZ=(14, 16, 18, 20, 22, 24), but for the self-generating focal training set method, we examined only the NZ=14. When looking at the focal region method at NZ=14, the MAPE=124.61%. This is much worse than

overall CG application MAPE of 29.30%, so perhaps NZ=14 was not the best condition to use in creating the CG example.

One fundamental question that permeates all methods described in Chapter 3 is the accuracy of forecasting. Forecasting, especially when one is extrapolating beyond the data range to make predictions of future observations, is inherently unreliable. Nonetheless, we believe that extrapolating to the P=2^(J+1) processor level will be relatively accurate because points that fall into our narrow focus region and obtain time-constrained scaling should behave fairly similarly to such points with more processors. .

Furthermore, we found that forecasting of interaction and higher order polynomial models were unstable when forecasted. All forecasts for these models had relative error much worse than the simple linear regression forecasts. This we believe is due to the interaction and higher order polynomial models over-fitting the data. Even if we had carried these models further in our methodology, we would still expect the models to display the unstable behavior exhibited in the global model forecasts.

One problem that is specific to our self-generating focal training set method described in Section 3.5 is double dipping. Double dipping, using the data for selection and analysis, can create bias in our results. However, we believe that the effect of double dipping will be minor because an independent dataset of points that falls into our narrow focus region and obtains time-constrained scaling will be relatively similar or exactly the same as the points that we used to create the model.

A preliminary look at the self-generated focal method with BT suggests that the method should work adequately for simple applications. When we extended the method to more complex applications, for example CG, the success of the self-generated focal method was poor.

The self-generated focal method seems to perform similar to the focal region method, in some cases for better or worse.

The difficulties exhibited in the preliminary analysis of BT could pose a greater problem for more complex applications. Overall, failure to choose points centered at the right places, having few points at each processor level, and/or having the initial 'known' point as an outlier are all problems that would affect the controlled setting that we are attempting to create. In spite of these problems, the BT application was unaffected. Thus, we might logically believe that more complex applications may be affected, but might not be affected drastically. Thus, we believe the self-generated focal region method should yield results close to the focused linear regression method results.

For more complex applications, the self-generating focal method might be difficult to implement. First, there are more factors that must be controlled or imputed in the model. Second, although we control some factors in our self-generating focal method, it might not be feasible or make sense for the user running the application to also hold those same factors constant.

Despite the limitations discussed above, we feel that our methods work and that they represent a step forward in the quest for a method to perform accurate time-constrained scaling of parallel-processing computer applications. We believe that our estimation procedures may help encourage the wider use of parallel computation for computationally intense problems. We feel that these methods and our future work will bring us closer to a ‘black box’ method for time estimation. This work is ongoing and will continue to produce valuable information along this path.

REFERENCES

[1] *Parallel computing*. (2002, November 8). Retrieved January 20, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Parallel_computing

[2] *Amdahl's Law*. (2002, February 25). Retrieved January 25, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Amdahl%27s_law

[3] *Gustafson's Law*. (2006, March 1). Retrieved January 25, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Gustafson%27s_Law

[4] Barnes et al. (2010). Using Focused Regression for Accurate Time-Constrained Scaling of Scientific. *23rd IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS).* Atlanta.

[5] *Scalability*. (2003, February 20). Retrieved January 23, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Scalability

[6] *Measuring Parallel Scaling Performance*. (2010, February 11). Retrieved February 24, 2010, from Sharcnet: https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance

[7] Bailey et.al. (1995). *NAS Parallel Benchmark 2.0.* NAS.

[8] Davd H. Bailey, E. B. (1993). NAS Parallel Benchmark Results. *IEEE Concurrecy* , 43-51.

[9] H. Jin, M. F. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance.* NAS.

[10] E.F. D'Azevedo, C. R. (1992). *Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors.* Oak Ridge: Oak Ridge Natiional Lab.

[11] J. Vetter, A. Y. (2002). An Empirical Performance Evaluation of Scalable Scientific Applications. . *IEEE* .

[12] *Multigrid method*. (1995, April 16). Retrieved February 26, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Multigrid_method

[13] *The SMG2000 General README File*. (2001, September 19). Retrieved February 26, 2010, from Lawrence Livermore National Labatories: https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/smg2000_readme.html

[14] Lee et. al.. (2007). Methods of inference and learning for performance modeling of parrallel applications. *Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming*. San Jose: ACM: Association for Computing Machinery .

[15] Reeves, J. (2009, December 28). Algorithm. Athens, GA, USA.

[16] Reeves, J. (2009, December 3). Focal Regions for CG. Athens, GA, USA.