Performance Optimization of Large Non-Negatively Constrained Least Squares Problems with an Application in Biophysics

Emre H Brookes
Department of Biochemistry,
University of Texas Health Science
Center at San Antonio
7703 Floyd Curl Drive
San Antonio, Texas 78229
1-210-767-3301
emre@biochem.uthscsa.edu

Borries Demeler
Department of Biochemistry,
University of Texas Health Science
Center at San Antonio
7703 Floyd Curl Drive
San Antonio, Texas 78229
1-210-767-3332
demeler@biochem.uthscsa.edu

ABSTRACT
Solving large non-negatively constrained least squares systems is frequently used in the physical sciences to estimate model parameters which best fit experimental data. Analytical Ultracentrifugation (AUC) is an important hydrodynamic experimental technique used in biophysics to characterize macromolecules and to determine parameters such as molecular weight and shape. We previously developed a parallel divide and conquer method to facilitate solving the large systems obtained from AUC experiments. New AUC instruments equipped with multi-wavelength (MWL) detectors have recently increased the data sizes by three orders of magnitude. Analyzing the MWL data requires significant compute resources. To better utilize these resources, we introduce a procedure allowing the researcher to optimize the divide and conquer scheme along a continuum from minimum wall time to minimum compute service units. We achieve our results by implementing a preprocessing stage performed on a local workstation before job submission.

Categories and Subject Descriptors
G.1.6 [**Numerical Analysis**]: Least Squares Methods. G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra - *linear systems; direct and iterative methods; sparse, structured and very large systems*. J.3 [**Life and Medical Sciences**]: Biology and genetics.

General Terms
Algorithms, Performance.

Keywords
Non-negatively constrained least squares, Analytical Ultracentrifugation.

TeraGrid'10, August 2-5, 2010, Pittsburgh, PA, USA.

1. INTRODUCTION
In this paper we describe a procedure to optimize large non-negative least squares problems. Solving these problems can require significant compute resources. Our procedure improves compute resource utilization. We target the inverse problem involved in modeling analytical ultracentrifugation experimental data, but our procedure is general in application. The goal of this paper is to introduce a recipe for anyone wishing to apply our previously published [1] divide and conquer method to other non-negative least squares problems in an efficient manner.

The sections of our introduction describe the necessary background. Non-negatively constrained least squares is a general method used for parameter estimation and is described in Section 1.1. Analytical ultracentrifugation is an important experimental method to which we apply our techniques and is described in Section 1.2. Our scalable high-performance divide and conquer method for solving non-negatively constrained least squares problems is described in Section 1.3. Specifically, important variables and equations that are necessary to understand our procedure for optimizing the divide and conquer method are described in Section 1.3.1. In Section 1.4, we discuss the motivation for the developments presented in this paper.

1.1 Non-negatively constrained least squares
In the physical sciences, researchers often wish to find the most appropriate model possible to describe a given set of experimental data [2]. It is frequently the case that there exists an equation from which one can compute simulated experimental data from a known model. This is known as the forward problem. It is usually much more difficult to determine the model from the experimental data, which is an inverse problem. The limited resolution of experimental measurements requires a best fit solution.

Experimental data typically contain noise from various sources, which includes random noise produced by experimental instrumentation. The presence of noise in the experimental data further complicates the inverse computation and can lead to inaccurate results. Reducing the effects of noise from experimental data often involves using Tikhonov or Maximum-Entropy regularization [2]. These methods introduce a bias that smoothes the solution. To overcome this bias, we have previously introduced a genetic algorithm technique to provide parsimonious regularization [3]. Parsimonious regularization returns the best fit solution with the fewest number of parameters.

For example, in the canonical 'source history reconstruction' problem of ground-water contamination, several polluting factories may be dumping contaminants into the ground-water. By placing test wells at multiple locations and measuring the observed contamination over time, we obtain experimental data. From this data we wish to determine the location of the polluting factories and the amount of pollutants they are dumping. It is straightforward to determine what we will find for data at the test wells if we know the number and location of the factories and their dumping, but finding the polluters from test wells is generally a 'hard' inverse problem of parameter estimation. The problem is mathematically hard because there are issues of uniqueness, existence and instability. Therefore, we look for a 'best fit' solution, often using a constrained least-squares technique [2].

Another application of inverse problems is in astronomical image reconstruction. An earth-based telescope must make observations through the atmosphere which can blur the image. The astronomer wishes to remove the effects of blurring. In this case the blurred image is the experimental data and the model parameters to be estimated are the number, locations and intensity of light sources [4,5].

Inverse problems are often approached using linear modeling: Let **x** be a vector which contains numbers known as model parameters or simply as parameters. Let the matrix **A** be a linear modeling function. Let the vector **b** be the experimental data. A model of the experimental data $\tilde{\mathbf{b}}$ is obtained from the parameters by computing the forward problem of matrix-vector multiplication: $\mathbf{Ax}=\tilde{\mathbf{b}}$. The inverse problem is to compute **x** from experimental data **b**. When the noise contained in **b** is random with a normal distribution, it can be shown that the statistically most likely solution is a least squares solution. This solution is an **x** which minimizes $\|\mathbf{Ax}-\mathbf{b}\|_2$ given A and b, where:

$$\|\mathbf{Ax}-\mathbf{b}\|_2=\sqrt{\sum_i ((\sum_j A_{ij}x_j)-b_i)^2} \qquad (1.1.1)$$

If the model parameters are restricted to positive real values, then a positively constrained least squares algorithm such as NNLS [6] is used.

NNLS is an iterative algorithm which uses two sets to index the columns of A. Initially, one set Z contains the indices of all columns and the other set P is empty. The algorithm proceeds by moving indices, one at a time, from Z to P, and performing standard (unconstrained) least squares on the system consisting of only the columns indexed by P. The choice of index at each iteration is computed using a projection. Occasional backtracking will occur when the least squares algorithm produces a negative element of x. The interested reader can find further details in [6].

For a general least squares problem, there may be many solutions that have an adequate value for $\|\mathbf{Ax}-\mathbf{b}\|_2$. Considering that **b** contains noise, it is often sufficient to have an approximate solution. Regularization methods have been used to provide approximate solutions in the presence of noise. Tikhonov regularization simultaneously minimizes $\|\mathbf{Ax}-\mathbf{b}\|_2$ and $\|\mathbf{x}\|_2$. The Tiknonov solution introduces a bias, as this procedure penalizes sharp peaks in the vector x. Maximum-Entropy regularization simultaneously minimizes $\|\mathbf{Ax}-\mathbf{b}\|_2$ and maximizes $-\sum x_i \ln(w_i x_i)$, where w_i are weights chosen by the user. The choice of weights causes significant bias to the obtained solution. As with Tikhonov regularization, this method generally penalizes sharp peaks in the vector x. Parsimonious regularization simultaneously minimizes $\|\mathbf{Ax}-\mathbf{b}\|_2$ and the number of nonzero elements of **x**. Further details about regularization can be found in [2,3].

1.2 Analytical ultracentrifugation

In analytical ultracentrifugation (AUC) sedimentation velocity (SV) experiments a sample solution is placed in a high speed centrifuge. The data obtained are pictures of the sample concentration taken over time. From this data we wish to obtain the molecular weights, shapes and partial concentrations of the solutes contained in the sample. Analysis of AUC SV experiments is the target application for our methods.

AUC is a powerful technique for determining hydrodynamic properties of biological macromolecules and synthetic polymers [7,8,9]. AUC can be used to identify the heterogeneity of a sample in both molecular weight and macromolecular shape. Since AUC experiments are conducted in solution, it is possible to observe macromolecules and macromolecular assemblies in a physiological environment, unconstrained by a crystal structure or electron microscope grid. Systems can be studied under high concentrations or under very dilute conditions, and under virtually unlimited buffer conditions. A buffer is a solution which maintains a constant pH and may contain other chemicals to stabilize the solutes[1]. Furthermore, the methods are applicable to a very large range of molecular weights, extending from just a few hundred Daltons to systems as large as whole virus particles. Results of these studies can allow the researcher to follow assembly processes of multi-enzyme complexes, characterize recombinant proteins and assess sample purity before proceeding to NMR [10] or X-ray crystallography experiments [11]. The techniques addressed are currently being used in AUC studies focusing on macromolecular properties of systems related to disease, cancer and aging.

In AUC sedimentation velocity experiments, a sample in solution is contained in a sector shaped cell which is placed in the ultracentrifuge. The ultracentrifuge runs at speeds from 2,000 to 60,000 RPM. At regular time intervals, the instrument records a radial concentration profile of the cell, which is determined from light absorbance at a particular wavelength or by measuring the fluorescence intensity or refractive index of the solution. At the beginning of the experiment, the sample is uniformly distributed throughout the cell and therefore the first observation shows a uniform radial concentration profile. As the experiment progresses, the centripetal force, which can be as high as 290,000g, causes the sample to sediment towards the bottom of the cell. After several hours or more, depending on the sample and the speed of the ultracentrifuge, the sample will be fully sedimented and further observations will contain an unchanging radial concentration profile of an exponential form. The sample may contain several solutes, where each solute is present at some concentration. The behavior of a non-interacting solute is well described by a second order PDE [12] known as the Lamm equation [13]. The parameters describing each solute are the sedimentation coefficient *s*, and the frictional ratio *k*. Given the experimental conditions and solute parameters *s* and *k*, the Lamm equation can be solved by finite element modeling [14,15], providing a model radial concentration profile for the solute.

[1] Solute: a single type molecule in solution.

Mathematically, the experimental data are placed in a vector **b**. The elements of **b** are the observed radial concentration profiles placed end-to-end for each time interval. For example, if each radial concentration profile contains r points, **b**[$2r$ + 1] will contain the second radial concentration of the third observation. Similarly, solutions to the Lamm equation can be placed in vectors and collected into a matrix **A**. Therefore, each column of **A** will be associated with the solute parameters s and k used to solve the Lamm equation. Assuming the data contain normally distributed errors from a distribution of constant variance, the best fit solution vector **x** can be expressed as follows:

$$\min_{\mathbf{x} \geq 0} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \qquad (1.2.1)$$

After solving eq. (1.2.1) for **x**, each element of **x** will contain the concentration of each solute associated with the corresponding column of **A**. Since negative concentrations do not make physical sense, we use NNLS [6] to solve this equation.

Let U be the set of all possible solute parameters s and k. Let G be a finite subset of U. Then given experimental data **b**, we can define a function:

$$f: G \rightarrow (\mathbf{x}, \mathbb{R}) \qquad (1.2.2)$$

which takes the input set G, builds the matrix **A**, computes the NNLS solution of eq. (1.2.1), and returns **x** and the root mean square deviation (RMSD) of the vector difference between **Ax** and **b**, a scalar measure of the goodness-of-fit. Use of this equation will be called an application of f or a basic computation module. When f is applied to a set G, some elements of **x** will be zero. Let G' be the subset of G that only contains the elements associated with a nonzero element of **x** after an application of f.

Since we cannot search all of U, we must select some subset G to search. Good solutions are not obtained if G does not contain representatives of all the solutes present in the sample. For example, if the sample contains two solutes, trying to solve f for just one of the solutes gives erroneous results. Constraining the search space is the first step towards application of all of the methods subsequently described. The range of s can be constrained by the van Holde-Weischet analysis [16]. Physical limits constrain the frictional ratio k to a minimum value of one (a spherical solute) and a maximum limit generally known *a priori* to the researcher which ranges from up to four for proteins, to up to ten or more for elongated or rod shaped molecules like DNA chains or fibrils.

AUC SV experiments contain at least three types of noise. Time-invariant noise produces a constant offset at a fixed radial position and is identically valued at every observation. Similarly, radial-invariant noise has a constant offset at a fixed time and has identical values at every radial position. Time-invariant noise can be caused by imperfections in the optical track of the instrument, for example, a fingerprint on the sample cell window. Methods have been developed to remove time-invariant and radial-invariant noise [17]. The third type of noise is normally distributed random noise. Additional noise sources such as nonlinearity in the optics, intensity fluctuations of the lamp flashes, refractive artifacts, and systematic contributions of unknown source may also be present. These additional sources have not been modeled.

1.3 Divide and conquer

To facilitate the solving of large NNLS systems, we introduced a divide and conquer technique [1]. We start with experimental data and wish to search a large set of possible parameters G (see equation 1.2.2) to discover the subset of G that best fits our data. Since the matrix **A** produced by G is in general too large to be computed on a single workstation, we divide the problem as follows: We partition G into subsets $\{G_1, G_2, \ldots, G_n\}$ and apply f to each subset to obtain a collection $\{G'_1, G'_2, \ldots, G'_n\}$. We subsequently union the results and apply f to the union. This is graphically shown in Figure 1.3.1.

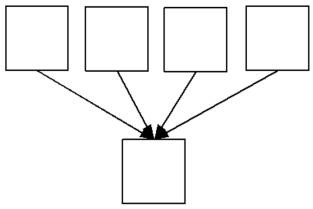

Figure 1.3.1: The top row contains the sets G_1, G_2 G_3 and G_4. The bottom box contains G^1, the union of the sets G'_1, G'_2, G'_3 and G'_4. The final result is $G^{1\prime}$. The top row can be computed on independent processors, but the final result depends on their results.

This basic methodology is complicated by two facts. The result of the union of all sets $G'_i \in \{G'_1, G'_2, \ldots, G'_n\}$ may produce a system too large to compute on one workstation. The solution to this is a recursive application of the method which we name the multistage method. This is shown graphically in Figure 1.3.2.

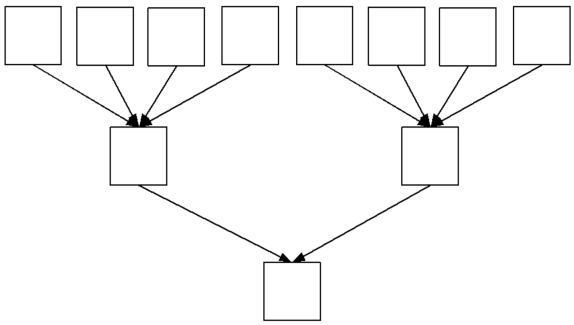

Figure 1.3.2: The multistage method. In this case the size of the unions of the results from the top row are too large to compute on a single processor and must be computed in groups (middle row).

The second complication is whether this divide and conquer method achieves the same result as applying f to the set G. The solution is to apply the multistage divide and conquer method iteratively by taking the result of the multistage method and unioning it to each set and repeating the procedure. This procedure is a contraction which obtains a fixed point and is empirically equivalent to applying f to G. The full iterative multistage procedure is shown in the following Figure 1.3.3.

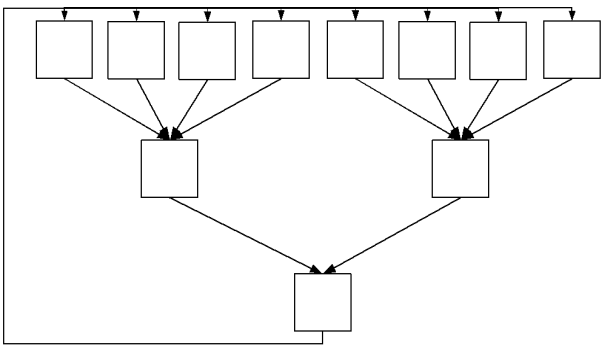

Figure 1.3.3: The iterative multistage method. In this case the result of each multistage solution is unioned back into the initial sets $G, G_1, G_2, \ldots, G_n$ and the process is repeated until the final result is unchanging. This achieves the highest quality result.

There is an alternative to the iterative method. An equivalent goodness of fit as measured by RMSD can be obtained by increasing the resolution of the original set *G*.

The partitioning of *G* into subsets $G_i \in \{G_1, G_2, \ldots, G_n\}$ should be done carefully. The best quality result will be obtained when each subset G_i "covers" the range of the parameter space. This will result in a lower RMSD for the non-iterative multistage method and a fewer number of iterations for the iterative variant. For further details on the divide and conquer method see [1,18].

1.3.1 Performance of divide and conquer

We introduced the divide and conquer method in a 2006 paper [1]. In the paper we also developed and validated an analytic performance prediction model. Additionally, we demonstrated the weak scalability of the method. In this section we discuss the variables and equations from that paper which are pertinent to understanding our optimization procedure.

We start with the experimental data in a vector **b**. It is important to fix this value since the subsequent variables and the resulting performance depend on **b**. *G* is the set of points in our parameter space to be searched. *n* is the number of points in the set *G*. *f* is the basic computational procedure of taking *G* or any subset of *G*, building a matrix, and finding the non-negatively constrained least squares fit of the matrix to **b**. Application of *f* returns a set *G′*, containing the elements of *G* contributing to the best fit solution. We assume the application of *f* on *G* to produce *G′* can be computed in polynomial time, $T_G = \alpha n^a$. This approximation has been validated for the NNLS problems encountered in AUC. The equation for T_G introduces the variables α and *a*. α is a proportionality constant to convert from unitless to measured time. α is dependent on **b**, *f*, and the characteristics of the processor. Conveniently, α divides out of our equations and can be ignored for our discussions. Simply note that α must be determined if one wishes to compute measured time. We fix the variable *a* for a given *f*. *a* is determined by exponential regression. For non-negative least squares with **b** from AUC experiments we have computed $a \approx 1.3$. We also use a variant *f* where $a \approx 2$ and have applied *f* to problems where $a \approx 3$. To "divide and conquer", we partition *G* into *q* equally-sized subsets G_i. Therefore, each subset will contain $m = n/q$ points of *G*. The time to apply *f* to G_i is $T_{Gi} = \alpha m^a$. The set resulting from an application of *f* to G_i is G_i'. The number of points in G_i' is of relevance, since this determines the number of stages required to complete the multistage computation. $w = \mathrm{E}\left|G_i'\right|/m$ is the expected value of the fraction of points returned by an application of *f* on G_i divided by the number of points in the set *G*. The total number of stages the multistage method will require is $r = ceil(-\log q / \log w)$. Given *p* processors, the number of stages where there are at least *p* computational modules without data dependency are $l = floor(-\log q - \log p / \log w)$. Finally, the time to compute a multistage computation is:

$$T_{MP} = \alpha\left[r + \frac{q}{p(1-w)}\right]m^a \qquad (1.3.1.1)$$

The processor efficiency is:

$$\eta = \frac{\frac{q}{p(1-w)}}{\frac{q}{p(1-w)} + r - l} \qquad (1.3.1.2)$$

All of these variables and equations are summarized in Table 1.3.1.1. Further details of the derivation of these equations can be found in [1].

1.3.2 Parallel efficiency

The inefficiencies in parallel algorithms are due to three sources: interprocess communication that cannot be masked by productive computations, excess computations, and processor idling. For the divide and conquer method the interprocess communication consists of sending out and receiving sets of parameters, which are quite small and insignificant compared to the time spent by each worker to perform an application of *f*. There is no excess computation. The major source of inefficiency is processor idling due to data dependency in the multistage method.

1.4 Motiviation

The typical AUC experiment contains 10^7 data points and the number of points of *G* we wish to search are 10^6, creating a matrix **A** of 10^{13} elements. Without optimization of the sizes of the first stage subsets, typical iterative multistage processing on 128 processors takes 30 minutes. Monte Carlo methods multiply this time by 50 to process for 25 hours. New multi-wavelength (MWL) experiments contain 10^{10} data points, requiring significantly more compute resources. A recent iterative Monte Carlo MWL analysis used 200,000 service units[2] of the Texas Advanced Computer Center's Ranger cluster.

Researchers worldwide utilize our method for analyzing AUC data through our TeraGrid Science Gateway at http://uslims.uthscsa.edu. They currently select the sizes of the subsets of *G* without any knowledge of the resulting performance of their choice, which can have significant cost implications for this already expensive computation.

[2] Service units (SU) are a standard unit of allocation granted to researchers on large high performance computation systems. One SU is generally one hour of one processor.

Table 1.3.1.1: Summary of Variables and Equations Used. The variables used are listed in the first column in partial order of dependence. For referencing our SC06 paper[1] the cross reference is listed in the second column. "not defined" means the specific variable was not defined. The equation column contains "n/a" if the variable is not defined by an equation.

Variable	Cross reference to our SC06 paper	Equation	Description
b	"the data"	n/a	The experimental data
G	S	n/a	The set of points in our parameter space.
n	n	$n=\|G\|$	Total number of points in G.
f	f	$f: G \to G'$	f takes a set of points, builds models to populate columns of a matrix**,** computes the non-negative least squares best fit to the experimental data, and returns a subset of G containing those elements of G which contribute to the best fit solution.
α	not defined	n/a	α is a constant of proportionality which is used to determine execution times. α is a property of the data **b**.
a	a	n/a	a is a property of f. a can be determined from exponential regression of f for different sizes of sets G on various **b**.
T_G	$T_{ORIGINAL}$	$T_G = \alpha n^a$	Time to compute an application of f on G.
q	k	n/a	Number of sets in the partition of G
G_i	S_i	n/a	The *i-th* set of the partition of G.
m	m	$m=\frac{n}{q}$	Number of points in each set in the partition of G.
T_{Gi}	not defined	$T_{Gi} = \alpha m^a$	Time to compute a application of f on G_i
w	x	$w=\frac{\mathrm{E}(\lvert f(G_i)\rvert)}{m}$	Expected value of the fraction of points returned by an application of f.
r	r	$r=ceil(-\frac{\log q}{\log w})$	Maximum number of stages of the multistage method.
p	p	n/a	Number of processors.
l	l	$l=floor(-\frac{\log q-\log p}{\log w})$	Number of stages where the number of computational modules is at least p.
T_{MP}	$T_{MULTISTAGEPARALLEL}$	$T_{MP} = \alpha[r+\frac{q}{p(1-w)}]m^a$	Time to execute the multistage applications up to a proportionality factor which is fixed given experimental data.
γ	γ	$\gamma = \frac{\frac{q}{1-w}}{\frac{q}{p(1-w)}+r-l}$	Speedup of computing *f(G)* versus a single iteration of our multistage method given a fixed m
η	η	$\eta = \frac{\frac{q}{p(1-w)}}{\frac{q}{p(1-w)}+r-l}$	Processor utilization.

2. METHOD

In this section we describe the procedure to optimize the performance of the divide and conquer method. There is a continuum of possibilities for this optimization. The researcher may wish to receive the results with a minimum of execution time or may wish to minimize the cost of the analysis in terms of service units. The researcher may also wish to select a point on the continuum from minimum execution time to minimum compute service units.

The question can be stated as: Given experimental data we wish to analyze at a predetermined resolution and some available compute resource with *max-p* processors, how do we determine the optimal run variables? Using the terminology described in Section 1.3.1 and summarized in Table 1.3.1.1 the problem can be stated precisely: Given max-*p*, **b**, *a*, *f*, and *G*: how does one determine the optimal number of processors *p* and the number of sets *q* or equivalently, the size of the each G_i, *m*? To minimize execution time, one might naïvely assume to use all the available processors, by setting both *p* and *q* to *max-p*, yet it will be shown that this is quite wasteful of compute resources. To minimize cost in service units, one may assume to set *p* to 1, but *m* must still be determined. It will be shown that for a small increase over the minimum cost, a significant improvement in execution time can be obtained.

Our optimization procedure begins in Section 2.1 with the estimation of the variable *w*. Given this result, we show how to optimize the non-iterative multistage case in Section 2.2 and the iterative multistage case in Section 2.3. For instructive examples we have chosen seven data sets obtained from AUC experiments. These are taken from a variety of researchers' analyses of DNA and proteins. For *G* we use a grid of 129,600 points representative of the parameter space appropriate to each experiment. For our examples, we used values of *m* in the range of 25 to 650. Values of *m* smaller than 25 result in G_i being trivially small and as the number of points in each G_i get too small, the overall performance suffers. Values of *m* larger than 650 result in G_i that exceed the available memory of individual processors to compute a single application of *f*. These values are for our experience with AUC data. The investigator applying the method to other problems should determine the ranges appropriate to their problems.

2.1 Estimating *w*

Recall from Section 1.3.1 and Table 1.3.1.1 that *w* is the expected fraction of points returned from an application of *f* on a set G_i. This important variable determines the total number of stages *r*. As *m* increases, *w* and *r* decrease. The time to compute an application of *f* to G_i is minimized with small *m*, but the number of stages *r* required increases dramatically. To demonstrate this fact we sampled applications of *f* to G_i on our seven example data sets with varying *m*. For each sample we computed *w* and *r*. The results are shown in Figures 2.1.1 and 2.1.2.

An estimate for *w* as a function of *m* is dependent on *f*, *G*, and **b**. Given the dependencies and *a priori* knowledge of the range of interest of *m*, it is a simple matter to select a set of equidistant points in the range of *m*. For each point in the selected set, create a few sets G_i and apply *f* to each, giving an average *w*(*m*). We have obtained good results with approximately 3 to 5 such sets. From the *w*(*m*) computed, *w*(*m*) can be extended to a function on the range of interest of *m* by linear interpolation.

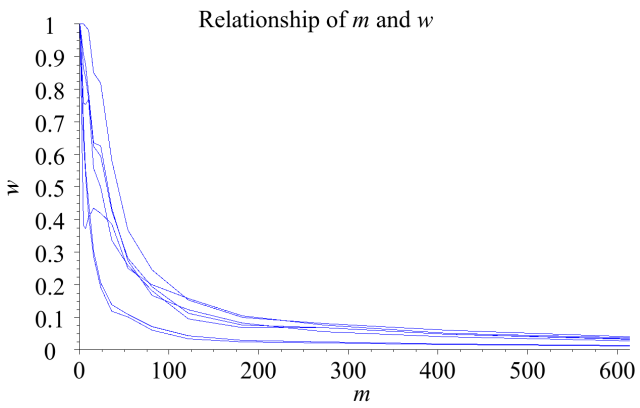

Figure 2.1.1: The relationship of the size of each set *m* and *w*, the expected fraction of parameters returned by an application *f*. Seven AUC example data sets were analyzed. As *m* increases the size of each G_i, increases, *w* decreases.

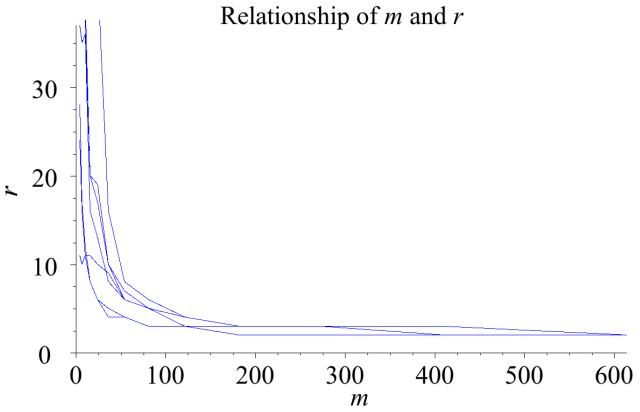

Figure 2.1.2: The relationship of the size of each set *m* and the number of stages required, *r*. Seven AUC example data sets were analyzed. As *m* increases, *r* decreases.

2.2 Optimizing: Non-Iterative Multistage

In this section we explain how to determine optimal values of *m* and *p*. Given *w*(*m*) described in Section 2.1, we can then compute arrays $T_{MP}(m,p)$ and *Cost*(*m*,*p*), which contain the execution times and costs, using the equations of Section 1.3.1.1 and Table 1.3.1.1. This is detailed in the procedure described in Text 2.2.1.

1. foreach *m*
2. $q \leftarrow \frac{n}{m}$
3. $r \leftarrow ceil(-\frac{\log q}{\log w(m)})$
4. foreach *p*
5. $T_{MP}(m,p) \leftarrow [r+\frac{q}{p(1-w(m))}]m^a$
6. $Cost(m,p) \leftarrow p \cdot T_{MP}(m,p)$

Text 2.2.1: The procedure for computing optimal *m* and *p* for a given non-iterative multistage problem.

Note that we set $\alpha = 1$ in step 5 for unitless computations of T_{MP}. In practice, we "normalize" the matrix T_{MP} by dividing all the T_{MP} by the minimum value found in the matrix. We similarly "normalize" $Cost(m,p)$ by dividing by the minimum cost found. To determine actual computational times, the applications of f should be timed on the target system to compute α. Once $T_{MP}(m,p)$ and $Cost(m,p)$ have been computed it is a trivial matter to search these matrices for the minimum execution time and minimum cost. To provide the user with an intuitive choice between these limits, we produce a continuum of costs from the minimum computational cost to the cost of the minimum execution time. This is achieved by creating equivalence classes containing points (m,p) with identical or nearly identical costs. Each equivalence class is assigned the minimum execution time of all points within the equivalence class. The procedure is shown in Text 2.2.2.

```
1. foreach m
2.    foreach p
3.       c ← floor( Cost(m,p) / 1000 ) · 1000
4.       if not defined BestTime(c)
5.          BestTime(c) ← ∞
6.       if BestTime(c) > T_MP(m,p)
7.          BestTime(c) ← T_MP(m,p)
8.          Bestm(c) ← m
9.          Bestp(c) ← p
```

Text 2.2.2: The second step of the procedure for computing optimal m and p for a given multistage non-iterative problem. The division and multiplication by 1000 in step 3 controls the resolution of the costs.

The procedure of Text 2.2.2 produces three arrays which are functions of cost c: $BestTime(c)$ is the best relative execution time for cost c, $Bestm(c)$ is the associated value of m, and $Bestp(c)$ is the associated value of p. There is no guarantee after this procedure that $BestTime(c)$ is a strictly increasing function. Increasing the divisor of step 3 (we used 1000) alleviates this issue. In practice, steps 3 through 9 of Text 2.2.2 can be inserted after step 6 of Text 2.2.1. For illustration of these procedures we processed our example AUC datasets and plotted $BestTime(c)$. Since we "normalized" $T_{MP}(m,p)$ and $Cost(m,p)$ the minimum time and minimum cost are both 1. Interestingly, we had improved execution times for relative costs as high as 10 times the minimum cost, but with insignificant improvement in relative time. For this reason we clipped the relative cost at 2. Importantly, paying the cost of many service units did little to improve the execution time. Also, paying a minor 10 or 20 percent additional cost over the minimum significantly improves the execution time. The relative cost versus p shows a linear scaling. The divide and conquer method has been shown to be weakly scalable, so to make good use of more processors, a larger G is needed. The results are shown in Figures 2.2.1 and 2.2.2.

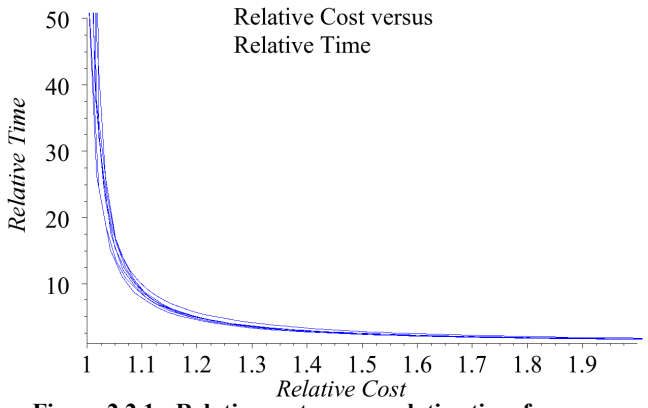

Figure 2.2.1: Relative cost versus relative time for seven AUC experiments.

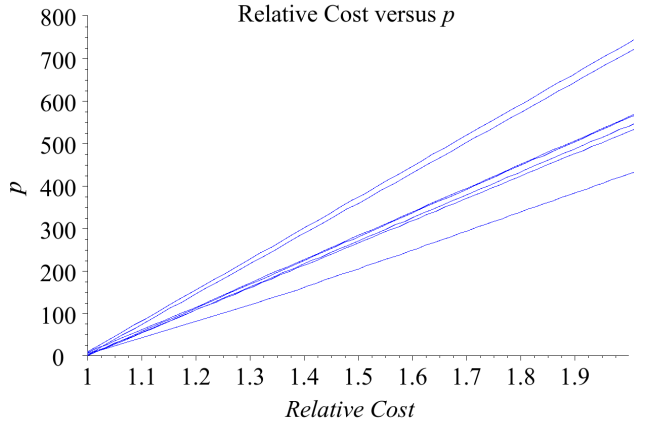

Figure 2.2.2: Relative cost versus p for seven AUC experiments.

2.3 Optimizing: Iterative Multistage

The iterative method, which provides the highest quality result introduces the further complication that the entire multistage procedure is repeated until a fixed point is reached. The iterations vary as a function of m. We computed the number of iterations required for the multistage divide and conquer method to converge for each of our seven AUC experiments. We used values of m in {24,36,54,121,182,273,410,614}. The results are shown in Figure 2.3.1.

Notice that the minimum number of iterations is two. The iterative method requires a minimum of two passes to determine convergence. Small m require more iterations, which we believe makes intuitive sense in the following way: since each G_i "sees" less of the solution space, it can only return an approximate result. These approximate results will be brought together at the end of the multistage process and must be unioned back into each original G_i during the subsequent iteration. This "expands the sight" of each G_i on each subsequent iteration, allowing the quality of the result to improve.

To apply the cost of iterations we must modify our execution time equation by including two factors. One factor is the multiple of the number of iterations. We define $I(m)$ to be the number of iterations required for m. Since $I(m)$ cannot be known without actually performing the full iterative multistage process, a record should be kept of prior iterative runs to be used as a predictor of future runs. The second factor increases the size of each G_i in subsequent iterations due to the union of the previous multistage result. The number of points added to each G_i is approximately the number of points we expect in the final result, $n \cdot w(n)$. Due to time and memory restrictions, $w(m)$ is only computed for values of $m << n$. We therefore let m_{MAX} be the largest value of m for which we have a $w(m)$ computed. The estimated number of points added to each G_i in the subsequent stages becomes $m_{MAX} \cdot w(m_{MAX})$. With these additions, we can state the equation for the time to execute the iterative multistage parallel divide and conquer method T_{IMP} as follows:

$$T_{IMP} = \alpha[r + \frac{q}{p(1-w(m))}] I(m)(m + m_{MAX} w(m_{MAX}))^a \quad (2.3.1)$$

This equation is substituted for T_{MP} in the procedures of Text 2.2.1 and 2.2.2.

As an example computation of $I(m)$ for AUC experiments, we take an average of the 7 experimental systems iteration counts and provide an $I(m)$ which is a linear interpolation of the average values. We force $I(m)$ to be a decreasing function of m based upon empirical observations and our belief that larger systems should iterate fewer times. It is difficult to predict the actual number of iterations a specific analysis will require. For example, if by chance, one of the sets G_i of our partition of G contains the best fit solution, the system will converge in the minimum two iterations.

2.4 Optimization Recipe

In summary, the procedure to optimize the processing of a divide and conquer analysis is as follows: We start with our experimental data **b**, a desired resolution G, and a compute resource of max-p processors. We estimate w by analysis of a random sampling of G_i for a range of m as described in Section 2.1. For the non-iterative multistage method, we apply the procedures of Section 2.2 in Texts 2.2.1 and 2.2.2. For the iterative multistage method, we must also have the iteration estimation function $I(m)$ in order to apply the procedures of Texts 2.2.1 and 2.2.2 modified by Equation 2.3.1. These procedures compute the data necessary to display an optimal relative cost and relative execution time graph. From the graph, the user can choose an optimal cost and an associated execution time. The user's choice determines the optimal values of m and p needed to submit the job to the compute resource.

3. CONCLUSION

Solving non-negative least squares is an important problem in the physical sciences. Analysis of AUC experiments relies on large non-negative least squares problem solving. The divide and conquer method can solve large non-negative least squares problems in a parallel environment, but requires careful selection of run variables to achieve optimal results in terms of minimum execution time or minimum service unit cost. Naïve selection of these parameters can result in an extreme waste of service units or excessive computational time. Our procedure is suitable for computation on a workstation prior to job submission. The procedure computes the run variables to achieve optimality along a continuum of costs. It applies to both the multistage and iterative multistage divide and conquer non-negative least squares method. This method has been validated on multiple datasets from AUC experiments.

4. ACKNOWLEDGMENTS

The development of the UltraScan software is supported by the National Institutes of Health (RR022200 to BD), supercomputer allocations were provided through National Science Foundation (TG-MCB070038, TG-MCB070040N to BD). We thank the staff of the Texas Advanced Computer Center, Warren Smith for discussions, and Joyce and Suzanne Tufek for editing assistance.

5. REFERENCES

[1] Brookes, E. H., Boppana, R.V. and Demeler, B. 2006 Computing large sparse multivariate optimization problems with an application in biophysics. SuperComputing 2006. DOI=http://doi.acm.org/10.1145/1188455.1188541

[2] Aster, C., Borchers, B. and Thurber, C. H. 2005 Parameter estimation and inverse problems. Elsevier Academic Press. London.

[3] Brookes, E. and Demeler, B. 2007 Parsimonious regularization using genetic algorithms applied to the analysis of analytical ultracentrifugation experiments. GECCO Proceedings ACM 978-1-59593-697-4/07/0007. DOI=http://doi.acm.org/10.1145/1276958.1277035

[4] Briggs, D.S. 1995 High fidelity deconvolution of moderately resolved sources. Doctoral Thesis. New Mexico Institute of Mining and Technology.

[5] Puetter, R. C. and Yahil, A. 1999 The pixon method of image reconstruction. Astronomical data analysis software and systems VIII, ASP Conf. Ser. Vol 172.

[6] Lawson, C. L. and Hanson, R. J. 1995 Solving least squares problems, 2nd Edition. SIAM, Philadelphia.

[7] Cole, J. L. and Hansen, J. C. 1999 Analytical ultracentrifugation as a contemporary biomolecular research tool. J. Biomolecular Techniques, 10:163-74.

[8] Demeler, B. 2005 Hydrodynamic methods. Bioinformatics basics: applications in biological science and medicine. 2nd Edition. Pages 226-255. CRC Press LLC.

[9] van Holde, K. E. 1985 Physical biochemistry, 2nd Edition. Prentice Hall, New Jersey.

[10] Claridge, T. D. W. 1999 High-resolution nmr techniques in organic chemistry. Pergamon, Oxford.

[11] Ladd, M., and Palmer, R. 2003 Structure determination by x-ray crystallography, 4th Edition. Kluwer Academic/Plenum Publishers, New York.

[12] Strauss, W. A. 1992 Partial differential equations, an introduction. Wiley, New York.

[13] Lamm, O. 1929 Die Differentialgleichung der Ultrazentrifugierung. Ark. Mat. Astrol. Fys., 21B:1-4.

[14] Cao W. and Demeler B. 2008 Modeling analytical ultracentrifugation experiments with an adaptive space-time finite element olution for multi-component reacting systems. Biophys. J. 95(1):54-65

[15] Cao, W. and Demeler, B. 2005 Modeling analytical ultracentrifugation experiments with an adaptive space-time finite element solution of the Lamm equation. Biophysical Journal. 90:4651-61.

[16] Demeler, B. and van Holde, K. E. 2004 Sedimentation velocity analysis of highly heterogeneous systems. Anal. Biochem. 335: 279-88.

[17] Schuck, P. and Demeler, B. 1999 Direct sedimentation analysis of interference optical data in analytical ultracentrifugation, Biophys. J. 76:2288-2296.

[18] Brookes, E., Cao, W., and Demeler, B. 2010 A two-dimensional spectrum analysis for sedimentation velocity experiments of mixtures with heterogeneity in molecular weight and shape. Eur. Biophys. J. 39(3):405-14