Computing Large Sparse Multivariate Optimization Problems with an Application in Biophysics

Emre H. Brookes
Department of Computer Science
University of Texas at San Antonio
ebrookes@cs.utsa.edu

Rajendra V. Boppana
Department of Computer Science
University of Texas at San Antonio
boppana@cs.utsa.edu

Borries Demeler
The University of Texas Health Science Center at San Antonio
demeler@biochem.uthscsa.edu

Abstract

We present a novel divide and conquer method for parallelizing a large scale multivariate linear optimization problem, which is commonly solved using a sequential algorithm with the entire parameter space as the input. The optimization solves a large parameter estimation problem where the result is sparse in the parameters. By partitioning the parameters and the associated computations, our technique overcomes memory constraints when used in the context of a single workstation and achieves high processor utilization when large workstation clusters are used. We implemented this technique in a widely used software package for the analysis of a biophysics problem, which is representative for a large class of problems in the physical sciences. We evaluate the performance of the proposed method on a 512-processor cluster and offer an analytical model for predicting the performance of the algorithm.

1 Introduction

Spectroscopic applications from fields such as astronomy, physics, and biochemistry involve experiments which often generate large amounts of data. Analysis of such data typically involves fitting the experimental data to a parameterized model. A crucial and time-consuming computational task in these analyzes is the determination of a subset of parameters from a much larger parameter space which best model the experimental data. This is achieved using standard optimization techniques. One such approach fits experimental data to a linear combination of vector representations of basis functions on the parameter space. Our motivating application, chosen from the field of biophysics, is analytical ultracentrifugation (AUC) [1, 2, 3]. AUC is a powerful technique for determining hydrodynamic properties of biological macromolecules and synthetic polymers in solution, and one where very large datasets are frequently encountered. Hydrodynamic parameters determined from such experiments allow the investigator to identify shape, molecular weight, and partial concentration of each solute in the mixture.

SC2006 November 2006, Tampa, Florida, USA

0-7695-2700-0/06 $20.00 ©2006 IEEE

The computational task of analyzing the experimental data (denoted as *data* henceforth) to build the best model can be organized into four phases: selecting the appropriate parameter search space, evaluating the basis functions on the parameters, eliminating systematic noise, and selection/identification of the parameters whose basis functions best fit the data. In our case, each basis function is evaluated using a finite element solution of a partial differential equation (PDE). High-performance computational techniques for this final phase of the analysis are well-known and are not the focus of this paper. The resolution of the model is often dictated by time constraints and memory limitations of a high-performance workstation. In this paper, we introduce a new method for optimization (finding the best model of the data) which addresses memory limitations, provides serial speedup and parallelizes with negligible communication overhead and no excess computation.

The contributions of the paper are both in the development of efficient computational techniques and their application to the AUC problem in biophysics. First, we present a divide and conquer method to partition the parameter space and the associated computations. Our method speeds up the computation for both sequential and parallel computing approaches and improves resolution by overcoming memory constraints, allowing larger number of parameters to be searched. We also develop an analytical model to predict the performance and optimal number of processors to use for a given cluster and problem set. Another contribution of the paper is the extensive evaluation of the proposed techniques using data from a large biophysics experiment which is analyzed on high performance cluster computers. Without our techniques, the largest number of parameters that could be searched using sequential techniques on a high-end workstation (Opteron with 4 gigabytes of RAM) was approximately $6.4 * 10^3$ parameters. Using our techniques, we have been able to increase the size to $1.5 * 10^6$ parameters using a cluster of 512 nodes without increase in execution time. This increased resolution provides higher accuracy and higher information content for a given experiment.

This paper is organized as follows: After the background material on AUC, we describe our method in the framework of set functions on the parameter space, using both serial and parallel approaches. We partition the parameter space, apply the function to each set in the partition, union the results and apply the function again. This approach is generally not exactly equivalent to applying the function to the full unpartitioned parameter space, but does allow us to analyze much larger problems. We introduce an iterative variant which is empirically exact for some set functions. Next, we give an analytical model of our method for performance prediction. Finally, we experimentally validate our performance prediction and describe the results for a large biophysics experiment on a 512 node cluster made available to us through TeraGrid.

2 Background

Many of todays' biomedical research projects studying the molecular basis for cancer and other diseases focus on the understanding of dynamic interactions among molecules implicated in the disease process. Sedimentation velocity (SV), one of several AUC techniques, offers a high-resolution approach to study such interactions. SV is a rigorous hydrodynamic technique and essential tool for the characterization of solution properties of biological and synthetic macromolecules, applicable to molecules with mass of just a few thousand Daltons to systems as large as whole virus particles. Analysis of SV data provides hydrodynamic information such as sedimentation and diffusion coefficients which can be used to determine molecular weight and shape, two important metrics for biomedical research. In SV experiments, a solution containing a macromolecular solute or mixture of solutes is sedimented in a centrifugal force field and the transport processes for each molecule are observed over time in order to measure sedimentation and diffusion rates. Experimental data are collected 200-1000 times during the sedimentation process in the form of concentration profiles showing the sedimenting boundaries along the radius of the ultracentrifugation cell. Depending on the optical detector, a total of 500-2000 radial points may be collected for each time point. The concentration profiles change over time due to the two transport processes acting on each solute, sedimentation and diffusion. For the sector shaped geometry of the ultracentrifugation cell, these processes are modeled by the Lamm equation, a second order PDE [4] parameterized by the sedimentation and diffusion coefficients. The Lamm equation is efficiently computed by an adaptive space-time finite element method (ASTFEM) proposed by Cao and Demeler [5]. Using appropriate initialization procedures (described in [6]), a grid is placed on the 2 dimensional parameter space, where each grid point defines the parameters for one basis function of the model. After finding the best fit linear combination of basis functions to the data, the amplitude of each term identifies the partial concentration of each solute. Because a negative concentration does not make physical sense, it is critical to constrain the value of the amplitude to positive values only, or zero if the solute is not present. Our approach to fit SV data to the parameterized models uses the NNLS algorithm [7] to determine the amplitudes. This approach guarantees positively constrained amplitudes and allows us to determine the partial concentration of each solute. In practice this method returns excellent results, but we often needed to analyze systems that overflowed available memory, prompting the improvements described in this work.

3 Divide and Conquer Techniques for Optimization

In this section, we provide a divide and conquer approach to facilitate efficient implementation of optimization on sequential as well as large parallel computers. We will also provide an analytical model to predict the execution time complexity of the proposed techniques. We have extensive experience with the NNLS algorithm used in the analysis of AUC data. Therefore, we use NNLS as an example to illustrate our model and also to validate it against actual computation times of NNLS on clusters. However, our analytical model and parallelization techniques may be applied to other optimization problems. Many optimization problems where the majority of the parameters do not contribute to the model could utilize our method. Noise handling and data stabilization techniques such as Tikhonov regularization using NNLS or TGSVD[8] can also be incorporated. In AUC, the method of time invariant noise removal by removing radial averages [9] has been handled successfully by our method.

To describe our method, we introduce the following notation. Let U be the set of all possible parameters. In AUC, each parameter is a pair of real numbers, so in this case U is $\Re\times\Re$. Let R, S and R_T be nonempty subsets of U. S denotes the search space: the set of parameters to our basis functions, which are checked for fitness against the experimental data by the function f. $R_T=f(U)$ denotes our target set, the unknown set of parameters that best model the experimental data. In AUC, R_T would be the set of parameters that model the actual solutes. $R=f(S)$ is the result set and is dependent on the choice of S and the optimization technique used. A perfect solution is achieved when $R = R_T$. This is feasible only when $R_T \subseteq S$, which requires a prescient initialization of S. Therefore, R is the best possible solution for a given S. Additional notation will be introduced as needed.

3.1 Divide and Conquer Method

We may need to search a large number of parameters (we have used $|S| = 10^3$ to 10^6) to find R. Therefore, the computational and memory requirements often exceed the capacities of even high-end workstations. We develop divide and conquer techniques for NNLS calculations to overcome the memory limitations of workstations in

```
1. Select search parameters $S$
2. Partition $S$ into $k$ disjoint sets $S_i$
3. for $i = 1$ to $k$
4.     Apply $f$ to $S_i$ and union the results into a set $R_{int}$
5. Apply $f$ to $R_{int}$ giving the final set of results $R_D$
```

Text 1: A divide and conquer method to speedup optimization techniques. The computation resulting from an application of ***f*** *to an S_i is called the basic computation module.*

sequential computing and to facilitate efficient parallelization of the computational task. We partition S into disjoint sets S_i, apply f to each S_i, union the results and apply f to the union. This procedure is described in *Text 1*. Let R_D denote the result set obtained with the divide and conquer approach. Ideally, R and R_D should be the same, but in practice, they usually differ. If the size of the final union, $|R_{int}|$ in step 5 of *Text 1*, is too large to apply f, then the divide and conquer method can be applied recursively by partitioning R_{int}.

The divide and conquer method improves execution time on both sequential and parallel computers. If a single high-performance workstation is used, the divide and conquer method can be used to overcome the constraint on the size of S due to main memory; the user can choose arbitrarily large S as desired for accurate modeling at the cost of increased execution time as long as each application of f to S_i can fit in the available memory. We show in the analytical modeling section that, for a given S, the computational time is also reduced (dramatically for computationally expensive optimization problems). If a large workstation cluster is used to perform the optimization, then the most computationally intensive step 4 in *Text 1* can be done in parallel, yielding high processor utilization.

3.2 Improving the Accuracy of Results

Arbitrary application of the divide and conquer approach to the optimization problem can lead to unsatisfactory results since f does not 'see' all the parameters of S at once. Based on our analysis of AUC data, we propose the following heuristics to ensure that the quality of the solution with the divide and conquer approach, R_D, is identical or nearly identical to that of the original method, R, in which f is applied to the entire S.

- **Partition *S* carefully:** When partitioning S into disjoint sets, some care must be taken to achieve the best results. Each set should cover the space covered by S, though sparsely. In other words, the range and density of the each set should be similar. The intuition is that each application of f is trying to find the best combination of basis functions to fit the data. Therefore, if f does not get a diverse set of parameters to work with, then it will have a hard time selecting appropriate basis functions. We experimented with different partitioning strategies and found that the suggested approach yields R_D that is nearly identical to R. We now describe the method we use to partition S. Given each parameter in S is of d dimensions, we must first find bounded intervals for each dimension of the parameter. This requires some knowledge about the possible range of the parameters, usually based upon physical limits. For our AUC example, this problem has been solved previously [6]. Given some target number of parameters, we can place a d dimensional grid on S with each point determining a parameter. Then, a subgrid can be defined that consists of every i-th point of the original grid in each dimension. This subgrid defines one of the sets S_i of the partition of S. The subgrid can be moved over the original grid, in each dimension, with each move defining a new set S_i until every point in the original grid is included in some set.
- **Apply *f* on the results obtained iteratively**: After obtaining the final result for the basic method (at the end of step 5 in *Text 1*), all parameters in R_D are unioned back into each S_i and the procedure is repeated until the results no longer vary. We found in our empirical analysis this iterative variant is in exact correspondence to R when f consists of evaluating the basis functions independently and performing NNLS. However, when f has additional processing steps (e.g. noise removal), the iterative results may not converge to a fixed parameter set. In such cases, the iterative procedure can still be applied, but some termination criteria based upon goodness-of-fit should be implemented. Of course, when early termination is applied this way, the result is unlikely to be exact. The iterative phase adds to the execution time and, therefore, should be applied only when necessary. In AUC without noise removal, we have seen convergence ($R_D \rightarrow R$) in 3 to 7 iterations. For systems with noise removal, we have achieved our best fit (no further decrease in residual) in 5 to 10 iterations.
- **Use larger *S***: Instead of applying the iterative technique above, the initial S may be increased to obtain R_D that is close to R. For the NNLS optimization technique, we observed that roughly doubling the number of parameters in the initial S yields the same fitness as would be obtained with the original sequential approach. This would correspond with a grid of increased point density, which of course, comes with a performance penalty. For example, in AUC, instead of using a 2 dimensional grid of $u*v$ parameters, one would use a grid of approximately $\sqrt{2}u*\sqrt{2}v$ parameters.

4 Analytical Model for Performance Prediction

In this section, we develop an analytical model to predict the execution time complexity, speedup over the original sequential method, efficiency of parallelization and optimal partitioning of S for a given computer cluster.

Let $n=|S|$, the number of parameters in S. The optimization problem of finding R can be described as a function f returning the subset of parameters of the input set which contribute to the best fit model. Let P be the power set of U, i.e. the set of all subsets of U. Then f can be defined as a function from P to P, which simply means that f takes a subset of U as input and returns some subset of U. Actually, f will always return a subset of the input set, since these are the only parameters that f knows about. The work done within f consists of evaluating basis functions, one for each parameter, and finding the best fit of the basis functions to the data. If the basis functions are built independently (i.e., one basis function for each parameter, which is the case in AUC when we assume non-interacting molecules), the basis function evaluation phase has time complexity $O(n)$. Finding the best fit in our case uses NNLS, for which a general time complexity formula is unknown, although an unlikely worst-case complexity has been given [10]. We have used exponential regression to approximate a time complexity of $O(n^{1.3})$. In practice, the experimental data can be quite noisy. For example, in AUC, a fingerprint on the cell adds time-invariant noise to the data. One method we use to eliminate time invariant noise consists of removing radial averages from all basis functions, which has a time complexity of $O(n^2)$ [9]. Our divide and conquer approach can also be applied in such instances. Therefore, we model the execution time for the original sequential technique in which f is applied to the entire set S as

$$T_{ORIGINAL} = O(n^a),\ a > 1 \qquad (1)$$

To find R_D, we assume that we start with $|S| >> |R|$. Partition S into k disjoint sets S_i, $i \in \{1,2,...,k\}$. Assume they all have the same cardinality, $m = |S_i|$, therefore, $k = n/m$. If $k=1$ we have simply repeated the initial problem of computing $f(S)$. Therefore, we only consider the instances where $1<k<=n$. In general, the number of parameters returned for each application of f on each S_i is dependent on k, f, S and R. Let $x = \mathrm{E}(|f(S_i)|)/m$, the expected fraction of the number of parameters returned. When x is not known statistically, one can roughly assume

$$x = 1 \text{ for } m<|R| \text{ and } x=|R|/m \text{ for } m>=|R| \qquad (2)$$

This assumption is based on our observations of f for NNLS. When f is given fewer parameters than $|R|$, f generally returns all the parameters, giving a probability of 1. When f is given more points than $|R|$, it returns approximately $|R|$ points. However, $|R|$ is often not known, therefore a statistical determination of x is recommended. For an *a priori* estimate, one can select random samples of S of size m, apply f, and subsequently estimate x. This procedure would require some additional preparatory computation to achieve maximum speedup for the given data. In *Table 1* we compute an average value of x from sample runs with data of different sizes and compare this with equation (2).

Now, we can estimate the total execution time of applying f to each set S_i and applying f to the union of the results. If a single computer is used, the serial execution time is:

$$T_{SEQ} = O(k\, m^a + (x\, n)^a) = O(n\, m^{a-1} + (x\, n)^a) \qquad (3)$$

where the first term is due to the application of f to each S_i and the second term is due to the application of f to the final union. Each application of f to S_i can be easily performed in parallel. If p processors are available, the execution time is:

$$T_{PARALLEL} = O(\lceil k / p \rceil\, m^a + (x\, n)^a), \text{ for } p > 1 \qquad (4)$$

For efficient parallel computing, the time spent the the first term should dominate the time spent on the second term, which must run serially.

Of particular importance is finding the optimum value of m which gives the minimum execution time, for fixed n, a and x. In general, this seems to be a difficult problem, since x is dependent on k, f, S and R. First, we determine x for various m. Then, using a fixed constant of proportionality to remove the big-O (assuming the data is fixed), we can divide equation equation (1) by equation (3) or (4) to compute the best possible serial or parallel speedup. In *Table 2* we see the maximum theoretical serial speedups that are achievable for a given x and a, after fixing $n=2048$. It is interesting to note that the optimal value of m for the best speedup is not very dependent on a, though a does primarily determine maximum serial speedups that are achievable.

4.1 Modeling Recursive Use of Divide and Conquer in the Combining Phase

Another issue that must be addressed is the size of the final union, $|\cup\, i\ (f(S_i))|$ which may exceed the available memory and practical execution time limits. This can be alleviated by further partitioning the results. Let $S^{(1)} = \cup\, i\ (f(S_i))$, then we can repeat the procedure for S by substituting $S^{(1)}$, keeping m constant from S to $S^{(1)}$, (note that m will most likely not divide $n^{(1)} = |S^{(1)}|$), so $k^{(1)} = \lceil n^{(1)} / m \rceil$. We call each recursive application of the divide and conquer technique a stage, so $S^{(1)}$ is the first stage of recursion (the initial procedure for S is the zeroth stage). Continuing on, we may have additional stages $S^{(2)}$, $S^{(3)}$, ..., until we have a stage with $|S^{(r)}| <= m$. To achieve this, we have to take some care choosing m, since if $m < |R|$, we may never have $|S^{(r)}| <= m$. This condition can be checked and m increased appropriately, alternatively, m can be increased at each stage up to some predefined limit based upon available memory and maximum allowed execution time. Note that x will likely vary somewhat from S to $S^{(1)}$, $S^{(2)}$, etc. In practice, the average x for the zeroth stage is a good predictor of x on subsequent stages. We can assume x is constant at each stage and $x < 1$. (If $x = 1$, then there will

m	***Relative data size***				***Average***	***|R|=10***
	1	***2***	***4***	***8***		
1	1.0000	1.0000	1.0000	N/A	1.0000	1.0000
2	0.7290	0.8320	0.7803	0.7832	0.7811	1.0000
4	0.6562	0.7744	0.7759	0.7730	0.7449	1.0000
8	0.4712	0.6602	0.6738	0.6685	0.6184	1.0000
16	0.2554	0.5601	0.5405	0.5366	0.4731	0.6250
32	0.1309	0.4697	0.4131	0.3989	0.3531	0.3125
64	0.0654	0.2822	0.2285	0.2129	0.1973	0.1563
128	0.0522	0.2017	0.1338	0.1299	0.1294	0.0781
256	0.0381	0.0967	0.0854	0.0938	0.0785	0.0391
512	0.0264	0.0483	0.0503	0.0566	0.0454	0.0195
1024	0.0146	0.0278	0.0269	0.0293	0.0247	0.0097

Table 1: The observed values of x for data of various sizes. The N/A entry is because this system was too large to compute the final application of f for m=1 on a single machine. All the data were taken from an AUC experiment of an identical mixture. Larger data sets were obtained by combining results from multiple experiments of various centrifugal rotational speeds. Other mixtures will result in different average x. The final column is an estimate of x for |R|=10, a number we believed might be representative of our data. More general study of the relationship of x to the data would likely lead to better estimators.

be no decrease in the problem size and the computation will not terminate). First, we analyze the serial multi-stage case. For the first stage, $n^{(1)} = xn$. For stage j, $n^{(j)} = xn^{(j-1)} = x^j n$, and the number of basic computation modules $k^{(j)} = n^{(j)}/m = \lceil x^j k \rceil$. We will have a stage with only one set (a final result union) when $k <= x^{-r}$ for some r calculated as follows:

$$r = \lceil \log k / -\log x \rceil \qquad (5)$$

Therefore, the total number of times the basic computation performed is

$$\sum (j = 0..r) \lceil x^j k \rceil \cong k * (1 - x^{r+1}) / (1 - x) \cong k/(1-x), \qquad (6)$$

since either $x<<1$ or r is large, and $(1-x^{r+1}) \cong 1$. Since the execution time complexity of each basic computation module is $O(m^a)$, the overall serial execution time complexity is approximately

$$T_{MULTISTAGE\text{-}SEQ} = O((k/(1-x))\, m^a) \qquad (7)$$

4.2 Efficiency of Parallel Processing

The organization of the parallel implementation proceeds as follows. We begin at the zeroth stage with a master process partitioning the parameters *S*. These sets of the partition are distributed to available workers. Each worker applies *f* to their set of the partition, performing the basic computational module. When the worker is done, the subset of their set which contributes to the solution are returned to the master. When sufficient zeroth stage results are available (a union of parameters with cardinality not greater than *m*), the master can begin distributing first stage jobs to the workers. This procedure continues until the final stage is complete. For the iterative variant, the entire process will repeat multiple times. In this analysis, we consider the overheads and efficiency of a single iteration of our method.

The inefficiencies in parallel computing can be due to three sources: (a) interprocess communication that cannot be masked by productive computations, (b) excess computations, often used to reduce communication among processes, and (c) processor idling. For the problems we investigated, the interprocess communication involved 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 the basic computation module. Additionally, the initial data set must be distributed to all the processors once. It may be that for some applications, the basis function evaluation phase is expensive, and some caching of vector representations may be of benefit. There is no excess computation in the parallel algorithm, as it is identical to the serial divide and conquer algorithm with the addition of the communication of parameters. The major source of inefficiency in our

$\lvert R\rvert$	*a*						
	1.1	*1.3*	*1.5*	*2*	*2.5*	*3*	*3.5*
2	1.41 32	3.33 16	7.54 16	51.20 32	341.33 32	2048.00 32	10922.67 32
4	1.33 64	2.82 32	5.91 32	32.00 32	153.83 64	819.20 64	4279.56 64
8	1.24 128	2.38 64	4.53 64	21.33 64	90.51 64	341.33 64	1158.52 64
16	1.16 128	1.99 128	3.40 128	12.80 128	47.28 128	170.67 128	599.85 128
32	1.09 256	1.67 128	2.67 128	8.00 128	21.33 128	56.89 256	160.91 256
64	1.03 512	1.43 256	2.09 256	5.33 256	13.25 256	32.00 256	74.98 256
128	0.97 1024	1.21 512	1.60 512	3.20 512	6.40 512	12.80 512	25.60 512
Data from Table 1	1.16 128	2.11 64	3.78 64	14.25 64	46.19 128	164.66 128	569.48 128

Table 2: Maximum theoretical serial speedups and the associated optimal value of m (below each speedup) for various values of a. The comparison is between equations (1) and (3) for n=2048. For the last row, x is obtained from Table 1. For the other rows, |R| is given and x is computed with equation (2). Note the speedups increase with increasing a and the optimal values of m remain relatively unaffected by a. Also, the optimal values of m increase monotonically with increasing |R|.

parallel method is processor idling in the combining stage. The remainder of the section discusses this issue in depth.

The execution time is simply the number of stages times the execution time of a stage, which is $O(m^a)$, assuming sufficient number of processors are available. So the minimum execution time in the parallel case with sufficient processors is

$$T_{MULTISTAGE\text{-}PARALLEL} = O(r\, m^a) \qquad (8)$$

If we have *p* processors, then the execution time of stage *j* is

$$O(\lceil k^{(j)} / p \rceil\, m^a) = O(\lceil \lceil x^j k \rceil / p \rceil m^a) \qquad (9)$$

Summing over all stages

$$\Sigma\ (j = 0..r) \lceil \lceil x^j k \rceil / p \rceil\, m^a \ \cong\ [r + (k/p)/(1-x)]\ m^a$$

Assuming one stage must complete before the next stage can begin, the execution time in the parallel case with limited processors is

$$T_{MULTISTAGE\text{-}PARALLEL} = O([r + (k/p)/(1-x)]\, m^a) \quad (10)$$

Since $(1-x)$ is $\Theta(1)$ values of $m >> |R|$ it can be omitted from equations (7) & (10). In practice, once sufficient results are available from one stage, the next stage processing can be performed simultaneously, thus increasing processor utilization and lowering the value of (10). Equation (9) provides insight into processor utilization. Note $k^{(r)}$ decreases at each stage until the final stage when $k^{(r)} = 1$. Therefore, at the final stages, processors become idle, until the final stage is performed on one processor. This places a limit on overall processor utilization.

The overall speedup and efficiency can be estimated as follows. Let *l* be the number of stages up to which the number of basic computation modules is at least *p*. So $k^{(l)} \geq p$; and

$$l = \lfloor log(k/p) / -log(x) \rfloor.$$

Using the total computation in units of basic computation modules given in equation (6) and the execution times given in *Table 3* for various stages of parallel computation, the speedup, γ, and processor utilization, η, are calculated as follows:

$$\gamma = [k/(1-x)] \,/\, [(k/p)/(1-x) + r - l] \qquad (11)$$
$$\eta = [(k/p)/(1-x)] \,/\, [(k/p)/(1-x) + r - l] \qquad (12)$$

To maximize processor utilization, it is best to set *k* to some multiple of the number of processors. For the application below, we used a multiplier of 30. This means each processor will be busy with 30 zeroth stage applications of *f*, before it is required to apply *f* to any further stages. If there are 4 additional stages, then the global system utilization will be approximately 90%. In principle, one could target arbitrarily high global system utilizations by increasing the multiplier of *p* to determine *k*.

To further elucidate, say that an application of *f* on a set of size *m* takes 1 minute. If each processor must compute 30 applications of *f* , then this will take 30 minutes, with all processors fully occupied. If there are 4 additional stages and $x = .1$, then the first stage will require 3 applications of *f* by each processor, for an additional 3 minutes, again with full processor utilization, then the second stage will then only keep 30% of the processors busy for 1 minute, the third stage, 3% for 1 minute and the final stage will keep 1 processor busy for 1 minute. So during approximately 33.3 minutes of a 36 minute calculation, all processors are busy, giving a utilization of 92.5%. In this example, $r=4$, $l=1$, $x=0.1$, and $k/p=30$; plugging these values in equation (12) gives $\eta = 91.7\%$. The discrepancy is due to the approximation made in the calculation of total time spent in the first $(l+1)$ stages.

We assumed that number of parameters given to each computation module is the same. This is true for the first stage where each S_i contains *m* parameters. However, subsequent stages are given parameters of cardinality less than or equal to *m*. Therefore, the execution time of those modules will be less. This can improve the efficiency over the formula given in equation (12).

	Stage $j=0, \ldots, l$	Stage $j=l+1, \ldots, r$
Processors used	p	$< p$ and $\geq p\, x^{j-l}$
Time spent in a stage (in units of basic module computation times)	$k\, x^j/p$	1
Total time spent	$\Sigma\, (j = 0..l) \lceil \lceil x^r k \rceil / p \rceil$ $=(k/p) * (1-x^{l+1})/(1-x)$ $\cong (k/p) * 1/(1-x)$	$(r-l)$

Table 3: The time spent in various stages of the multistage parallel method.

5 Evaluation of Analytical Model

In this section, we describe the implementation of the proposed divide and conquer method for NNLS and noise removal algorithms in a production software called UltraScan, used by researchers worldwide. Using the actual data obtained in AUC experiments and analyzes of the same in the software package, we validate the accuracy of the proposed performance prediction model.

For our observational results we analyzed a simulated SV experiment. Our method has been implemented in the UltraScan software [11] since 2005 and is heavily used in the AUC community. The parallelization of this method was implemented in 2006 and announced at the Analytical Ultracentrifugation Workshop in San Antonio [12] and demonstrated at the recent AUC conference [13]. Job submission to the parallel version is made available through a convenient web interface [14]. The UltraScan software is written in C++, and parallel versions use MPI [15,16]. We implemented the allocation of basic computation modules to various processors using a master-worker model. A master process keeps track of the basic computation modules. Each free worker processes queries and receives a few computation modules; upon completing the execution, it returns the result parameter set to the master and receives additional computational modules. It is noteworthy that when the multistage method is used, all the basic computation modules in one stage need not be finished to start the computations in the next stage. This helps maximize processor utilization.

All observed execution times are from running an instrumented version of UltraScan, which offers the capability to globally analyze multiple experiments simultaneously [17]. The *f* in UltraScan consists of building basis functions for each parameter, optionally processing for time-invariant noise, and running NNLS. Our results reflect simulations without time-invariant noise. We have observed corresponding results for experiments containing time-invariant noise (data not shown). UltraScan partitions *S*, a 2 dimensional parameter space bounded by physical constraints, by starting with a regularly spaced grid over this space. This grid serves as the initial set of the partition. The next set is created by moving the grid by a fraction of the grid spacing in each dimension. Each movement of the

	Relative data size		
	1	*2*	*4*
$T_{ORIGINAL}$	46.61	301.67	425.42

Table 4: $T_{ORIGINAL}$ *computed for different data sizes. The data sets are the same as those used in Table 1, we kept n=2048. We believe the large* $T_{ORIGINAL}$ *between the data size 1 and 2 is due to caching. Note the increase from 2 to 4 is not as significant.*

m	*Relative data size*								
	1			*2*			*4*		
	x	*Predicted*	*Observed*	*x*	*Predicted*	*Observed*	*x*	*Predicted*	*Observed*
1	1	46.76	49.25	1	302.61	268.15	1	426.75	387.16
2	0.729	31.09	31.60	0.832	238.69	223.43	0.7803	309.81	281.51
4	0.6562	27.18	28.18	0.7744	217.82	204.34	0.7759	307.94	296.82
8	0.4712	17.80	20.63	0.6602	177.62	155.76	0.6738	257.15	231.33
16	0.2554	8.24	9.83	0.5601	144.20	135.55	0.5405	194.29	188.45
32	0.1309	3.73	3.93	0.4697	115.66	115.74	0.4131	138.62	134.66
64	0.0654	**1.86**	**2.04**	0.2822	61.58	58.22	0.2285	67.13	67.16
128	0.0522	2.27	3.41	0.2017	45.85	46.12	0.1338	**42.71**	**47.02**
256	0.0381	3.79	5.48	0.0967	**34.68**	**40.68**	0.0854	45.86	53.96
512	0.0264	8.10	16.60	0.0483	55.63	58.20	0.0503	78.89	84.95
1024	0.0146	19.12	23.86	0.0278	125.38	114.30	0.0269	176.64	168.28

Table 5: Comparison of theoretical predictions with observed times in seconds for the parallel method for data of varying sizes. All observed times are the average of 5 separate runs. The jobs were run using a maximum of 34 processors. For all cases n=2048, a=1.3. The theoretically predicted times were computed from equation (4) using observed values of x from the test runs as shown in this table and Table 1. Equation (4) was scaled by $T_{ORIGINAL} / n^a$ where $T_{ORIGINAL}$ is taken from Table 4. There is good correspondence between the predicted and observed values. Importantly, the optimal value of m is matched for all predictions and observations for each data size are indicated in bold.

grid creates an additional set of the partition. This way, *S* and the partition of *S* are built up from an initial partition set which is moved around to eventually cover the entire parameter space *S*. This method limits *k* to be a perfect square. We report here results from runs performed without the iterative method.

We conducted two sets of computer runs. The first set is used to validate the performance prediction model for the serial and parallel single stage case. The second set of computer runs is used to demonstrate the highly efficient parallel execution of the multistage optimization algorithm in the UltraScan software.

In our first experiment we compare the theoretical predictions of equations (1) and (3) with the observed execution times. In this case AUC software was run on a cluster of 40 Opterons with 2 gigabytes of RAM per processor running Linux and located at the University of Texas Health Science Center at San Antonio. The motivation for this experiment was to compare our theoretical predictions with real cases of varying data sizes. For both the theoretical and observed cases we fixed *n*=2048. We have empirical data giving an approximate *a* of 1.3. To determine our speedup, we computed $T_{ORIGINAL}$ for each data set as reported in *Table 4*.

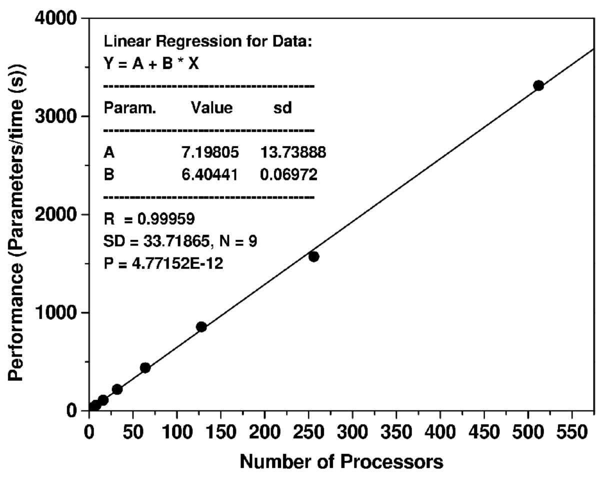

Figure 1: Performance profile for the Divide and Conquer scheme. Shown are number of parameters evaluated per second as a function of number of processors.

Given the values of *x* from *Table 1*, we can now compare the theoretical predictions with the observations. These results are summarized in *Table 6* for the serial case and *Table 5* for the parallel case. We ran tests for varying *m* and found that the optimum theoretical *m* coincided with

m	*x*	*Relative data size*					
		1		*2*		*4*	
		Predicted	*Observed*	*Predicted*	*Observed*	*Predicted*	*Observed*
1	1	51.34	73.03	332.30	332.75	468.61	490.65
2	0.7811	39.63	42.98	256.51	252.56	361.74	391.52
4	0.7449	38.96	38.36	252.14	230.58	355.57	358.90
8	0.6184	33.78	32.35	218.66	188.21	308.36	286.63
16	0.4731	28.49	22.96	184.38	174.85	260.02	247.74
32	0.3531	25.43	18.20	164.58	164.81	232.09	232.01
64	0.1973	**22.13**	**17.60**	**143.23**	**128.49**	**201.99**	**201.99**
128	0.1294	23.55	23.60	152.45	144.26	214.98	206.72
256	0.0785	26.68	31.83	172.70	155.84	243.54	252.68
512	0.0454	31.59	42.54	204.44	185.97	288.31	286.15
1024	0.0247	38.24	43.63	247.49	220.33	349.01	288.18

Table 6: Comparison of theoretical predictions with observed times in seconds for the serial method for data of varying sizes. All observed times are the average of 5 separate runs. For all cases n=2048, a=1.3. The theoretically predicted times were computed from equation (3) taking values of x from Table 1. Equation (3) was scaled by $T_{ORIGINAL} / n^a$ where $T_{ORIGINAL}$ is taken from Table 4. There is good correspondence between the predicted and observed values using average values for x. If one uses the observed values of x from Table 1 instead of the average, even closer correspondence is obtained. Importantly, the best speedups are achieved with m=64 for all predictions and observations as indicated in bold.

No of processors	***Actual time in seconds***	***Number of parameters searched***
4	396	12,100
8	435	25,600
16	443	48,400
32	439	96,100
64	441	193,600
128	449	384,400
256	494	774,400
512	452	1,537,600

Table 7: Implementation results for running on the LoneStar cluster of TeraGrid. This demonstrates the increased resolution possible for the method keeping time constant and increasing the number of processors. In each case m=100. The problem was the analysis of AUC experimental data which contained 80,640 data points.

the observed values. We believe these results provide validation for our method.

For our second experiment, we chose to see how many parameters we could search in a fixed time by increasing the number of processors. The motivation was primarily to validate the ability to scale the algorithm. This was run on the LoneStar cluster of 1024 Intel Xeon processors with 1 gigabyte RAM per processor made available to us through NSF TeraGrid. We tested up to 512 processors since all 1024 processors were not functional during our testing phase. For this experiment we fixed *m=100* and varied *k* to be approximately 30 times *p*, the number of processors. *k* couldn't be exactly 30**p* since *k* must be a perfect square in UltraScan. One may note that the system appears to become overdetermined, but this does not effect NNLS, because NNLS actually performs an incremental sequence of unconstrained linear least squares problems starting with a single basis function (for more details see [7]). The results are shown in *Table 7* and a linear regression of the number of parameters searched per second is shown in *Figure 1*. We conclude that this algorithm scales linearly with the number of processors and can be used to search an extremely large number of parameters.

6 Conclusion

We have presented a new method for solving large problems in optimization. Our method gives serial speedup and parallelizes efficiently. The results are not generally in exact correspondence with the original problem, but can be made sufficiently close through careful use of the given heuristics. Exact correspondence can be obtained for some problems with our iterative variant. A performance prediction model was given which has been validated by experimental runs. The scalability of this method has been demonstrated on a large cluster of 512 processors.

The developments described here have far reaching significance for the field of biophysics, and other areas where multivariate optimization is applied. For our example, due to restrictions in computer processing power and memory, a non-stochastic, model-independent optimization approach has so far not been possible. Because of this limitation, SV fitting has been relegated to either model dependent, non-linear approaches [18], which introduce user bias and often fail to converge, or model independent approaches that are either limited in resolution and information content [19], or approaches that are not suitable for the general case [20] where heterogeneity in mass and shape may exist simultaneously. This latter requirement is crucial for many systems in biophysics and biochemistry. Another important aspect is the generality of the application. A large class of multivariate optimization problems can be conveniently solved with this method. Many linear problems (whether optimized with NNLS or a generalized least squares approach, or by other fitting metrics) can now be reformulated to match our approach and benefit from parallelization.

Acknowledgments

We would like to express our appreciation to Jay Boisseau and Warren Smith, and the staff from the Texas Advanced Computing Center for assistance with the LoneStar computations, and thank Dr. Hugh Maynard for discussions in the preparation of this manuscript. This work was supported by grant#10001642 to B.D., R.B., and Weiming Cao from the San Antonio Life Science Institute and National Science Foundation grant EIA-0117255 to R.B. We also acknowledge a TeraGrid allocation to B.D. by the National Science Foundation through grant TG-MCB060019T. The development of the UltraScan software was supported by the National Science Foundation through grant DBI-9974819 to B.D., and Internet2 connectivity was made possible through grant ANI-228927 from the National Science Foundation to B.D. and R.B.

References

[1] COLE, J.L. and HANSEN, J.C. 1999. Analytical Ultracentrifugation as a Contemporary Biomolecular Research Tool. *J. Biomolecular Techniques* 10, 163-174.

[2] VAN HOLDE, K.E. 1985. *Physical Biochemistry, 2nd Ed.* Prentice Hall, NJ. 110-136

[3] DEMELER, B. 2005. Hydrodynamic Methods. *Bioinformatics Basics: Applications in Biological Science and Medicine. 2nd Edition.* H. Rashidi and L. Buehler, Eds. CRC Press LLC. 226-255

[4] LAMM, O. 1929. Die Differentialgleichung der Ultrazentrifugierung. *Ark. Mat. Astron. Fys.* 21B:1-4

[5] CAO, W. and DEMELER, B. 2005. Modeling analytical ultracentrifugation experiments with an adaptive space-time finite element solution of the Lamm equation. *Biophys J.* 89(3):1589-602.

[6] BROOKES, E. and DEMELER, B. 2006. Genetic Algorithm Optimization for obtaining accurate Molecular Weight Distributions from Sedimentation Velocity Experiments. *Analytical Ultracentrifugation VIII, Progr. Colloid Polym. Sci.* 131:78-82. C. Wandrey and H. Cölfen, Eds. Springer

[7] LAWSON, C. L. and HANSON, R. J. 1974. *Solving Least Squares Problems.* Prentice-Hall, Inc.

[8] ASTER, R., BORCHERS, B., and THURBER, C. 2005. *Parameter Estimation and Inverse Problems*, Elsevier Academic Press, London.

[9] SCHUCK, P., and DEMELER, B. 1999. Direct Sed. Analysis of Interference Optical Data in Analytical Ultracentrifugation, *Biophys. J.* 76:2288-2296.

[10] HANSON, R. 2006. *Personal communication*.

[11] DEMELER, B. 2005. *UltraScan - A comprehensive software package for the analysis of analytical ultracentrifugation experiments.* The University of Texas Heath Science Center at SA, Dept. of Biochem. http://www.ultrascan.uthscsa.edu

[12] DEMELER, B. 2006. *Analytical Ultracentrifugation Workshop: Focus on Problem Solving - From Experimental Design to Data Analysis.* http://www.ultrascan.uthscsa.edu/workshop/

[13] BROOKES, E. and DEMELER, B. 2006. A 2 D Spect. Anal. for Shape and MW Dist. from Sed. Vel. Exp. *15th International Symposium on Analytical Ultracentrifugation,* London.

[14] WILSON, J. 2006. *A web interface for the parallel 2-dimensional spectrum analysis.* The Center for Analytical Ultracentrifugation of Macromolecular Assemblies (CAUMA) at the University of Texas Health Science Center at SA. http://cauma.uthscsa.edu

[15] GRAMA A., GUPTA A., KARYPIS, G., and KUMAR, V. 2003. *Introduction to Parallel Computing, 2nd Edition.* Addison-Wesley, London.

[16] MPI FORUM. *Message Passing Interface Forum*. http://www.mpi_forum.org

[17] BROOKES, E. and DEMELER, B. 2006. Global Sed. Vel. Anal. using the 2-D Spectrum Anal.. *15th Int. Symposium on Analytical Ultracentrifugation,* London.

[18] DEMELER, B. and SABER, H. 1998. Determination of Molecular Parameters by Fitting Sedimentation Data to Finite Element Solutions of the Lamm Equation. *Biophysical Journal*. 74, 444-454

[19] DEMELER, B. and VAN HOLDE, K.E. 2004. Sed. velocity analysis of highly heterogeneous systems. *Anal. Biochem. Vol* 335(2):279-288

[20] SCHUCK P. 2000. Size-distribution anal. of macromolecules by sed. vel. ultracentrifugation and Lamm equation modeling. *Biophys. J.* 78(3):1606-19