RC24682 (W0810-163) October 29, 2008
Computer Science

IBM Research Report

Efficient High-precision Dense Matrix Algebra on Parallel Architectures for Nonlinear Discrete Optimization

John Gunnels, Jon Lee
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Susan Margulies
Department of Computational and Applied Mathematics
Rice University
Houston, TX 77005
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient high-precision dense matrix algebra on parallel architectures for nonlinear discrete optimization

John Gunnels

IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, New York 10598 USA

Jon Lee *

IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, New York 10598 USA

Susan Margulies

Department of Computational and Applied Mathematics, Rice University, Houston, Texas 77005 USA

Abstract

We provide a proof point for the idea that matrix-based algorithms for discrete optimization problems, mainly conceived for proving theoretical efficiency, can be easily and efficiently implemented on massively-parallel architectures by exploiting scalable and efficient parallel implementations of algorithms for ultra high-precision dense linear algebra. We have successfully implemented our algorithm on the Blue Gene/L computer at IBM's T.J. Watson Research Center. Additionally, we have delineated the necessary algorithmic and coding changes required in order to address problems several orders of magnitude larger, dealing with the limits of scalability from memory footprint, computational, reliability, and interconnect perspectives.

Key words: Nonlinear matroid optimization, model fitting, polynomial model, matrix algebra, Vandemonde system, Stirling numbers, extended precision arithmetic, distributed reduction, parallel Cholesky, limits of scalability, Blue Gene.

* Corresponding author: J. Lee

Email addresses: gunnels@us.ibm.com (John Gunnels), jonlee@us.ibm.com (Jon Lee), susan.margulies@rice.edu (Susan Margulies).

28 October 2008

1 Introduction

Our goal is to provide a proof point for the idea that matrix-based algorithms for discrete optimization problems, mainly conceived by theoretical computer scientists for proving theoretical computational efficiency, can be easily and efficiently implemented on massively-parallel architectures by exploiting efficient (reusable!) parallel implementations of algorithms for ultra high-precision dense linear algebra. In this way, we hope to spark further work on leveraging a staple of high performance computing to efficiently solve discrete optimization problems on modern computational platforms.

Matrix-based methods for combinatorial optimization are mainly conceived to establish theoretical efficiency. There are many examples of such techniques (see e.g., [4,5,14,18]), but they are not typically seen by sober individuals as candidates for practical implementation. Indeed, as far as sequential algorithms go, for combinatorial optimization, there are often better candidates for implementation (see e.g., [15,16]). So our goal was to see if we could take such a matrix-based algorithm and develop a practical parallel implementation leveraging efficient algorithms for dense linear algebra.

2 Background

Our starting point is the paper [4]. Their motivation is the so-called *minimum aberration model fitting problem*. That problem is to find the "best" multivariate polynomial model to exactly fit a data set, using only monomials (in the input "factors") from some given set, that can be uniquely identified from any values of the response variables. The total-degree vector of a polynomial model is a vector of the total degrees of each of the factors (i.e., variables), over the monomials in the model. "Best" can be with respect to any "aberration" function of the total-degree vector (see [12,24]); for example, a simple polynomial model is typically defined to be one where the chosen monomials have low degrees. For example, we may choose to minimize the norm of the total-degree vector, computed over monomials in the model.

For example, we might consider clinical trials designed to determine an effect on patients of various combinations of a small number of drugs. Each patient is given a dose level of each drug. The number of patients and the dose levels are determined by those conducting the trials. We wish to fit a mathematical model that will determine the as-yet-unknown response levels of the patients (e.g., blood pressure). For concreteness and because they are commonly used, we consider polynomial models, where the possible monomials in the model come from some large but finite set. Each summand in the model is a monomial

with the variables standing for the levels of the various drugs. To insure an exact fit, the number of monomials to choose corresponds to the number of patients in the clinical trial. The response from the clinical trials determines the precise model that is fit, through the constants in front of each monomial. We wish to determine a relatively simple polynomial model (i.e., choice of monomials) that will fit whatever responses we might eventually observe. This corresponds to minimizing the aberration of the selected polynomial model.

With the goal of solving the minimum aberration model fitting problem, and indeed any so-called nonlinear matroid-base optimization problem, the authors of [4] developed an algorithm based on matrix methods. That algorithm was designed to be theoretically efficient, in the worst-case sense, when the number of factors is fixed. Indeed, the problem is provably intractable when the number of factors is not fixed. The dependence of their algorithm on the number of factors is exponential, so it could only be practical only for a very small number of factors.

Our goal was to implement their method on a massively-parallel architecture by exploiting efficient parallel implementations of algorithms for dense linear algebra. In doing so, we hope to demonstrate that their algorithm is practical, on such an architecture, for a modest number of factors. Moreover, the algorithm is relatively simple to implement and indeed parallelize, so there is yet another advantage from the standpoint of practicality.

It is worth mentioning that a serial implementation of the algorithm using standard floating-point arithmetic is not remotely practical. In particular, for problem instances of interest, the algorithm requires the solution of Vandermonde systems of order in the thousands. Such systems are inherently extremely badly conditioned, and the state-of-the-art for their solution on a serial machine in floating point is well below our needs.

3 Notation

We are given an $m \times d$ *design matrix* P of floating point numbers and an $n \times d$ *monomial degree matrix* β (of small nonnegative integers). Connecting this with the description from the previous section:

- d = number of factors (i.e., variables);
- n = number of monomials to select from;
- m = number of design points = number of monomials to select.

Each row $p_{i\cdot}$ of P is a design point, specifying a setting of the d factors. Each row $\beta_{i\cdot}$ of β specifies the degrees of each factor in the monomial corresponding

to that row. That is, $\beta_{i\cdot}$ describes the monomial $\prod_{k=1}^{d} x_k^{\beta_{i,k}}$.

Let A be the $m \times n$ matrix A defined by

$$a_{i,j} \;:=\; \prod_{k=1}^{d} p_{i,k}^{\beta_{j,k}} \;, \quad i = 1, \ldots, m \;,\; j = 1, \ldots, n \;.$$

A polynomial model

$$\pi(x) := \sum_{i \in S} c_i \prod_{k=1}^{d} x_k^{\beta_{i,k}}$$

is determined by a set $S \subset \{1, 2.., n\}$ indexing some of the monomials. Let A_S be the submatrix of A comprising its columns with indices from S . The set S is *identifiable* if $A_S\, z \;=\; y$ has a unique solution for all ("response vectors") y (see [19], for example). This is exactly the condition that is needed for there to be a unique choice of the c_i above so that $\pi(p_{i\cdot}) = y_i$, for $i = 1, \ldots, m$, for every possible y . Typically A has full row rank (and we do assume that for convenience), so S is identifiable if and only if A_S is square $(m \times m)$ and $\det(A_S) \neq 0$.

The *total-degree vector* of the model (indexed by) S is

$$x(S) := \sum_{i \in S} \beta_{i\cdot} \;\in \mathbb{Z}_+^d \;.$$

The ultimate goal is to select an identifiable model that minimizes some function of the total-degree vector. As there can be many reasonable functions f , we seek to enumerate the possible total-degree vectors. Our goal is to demonstrate that this can be made practical even in situations when the number of identifiable models cannot be practically enumerated.

We consider a concrete practical example. For a positive integer ω , we look at the matrix β having all possible rows of nonnegative integers satisfying $\beta_{i,k} \leq \omega$. Such a matrix β has $n = (\omega + 1)^d$ rows. The number of potential identifiable models is then

$$\binom{n}{m} = \binom{(\omega+1)^d}{m} \;,$$

while the number of potential total-degree vectors is only

$$(m\omega + 1)^d \;.$$

For example, with $d = 2$ (2-factor experiments), $\omega = 9$ (maximum degree of 9 in each monomial), we have that the number of monomials to choose from is $n = 100$. If we have say $m = 12$ (design points), then the number of potential

identifiable models is

$$1,050,421,051,106,700 \ \sim 1.05 \times 10^{15}$$

while the number of potential total-degree vectors is only $11,881$ (many potential models obviously must have the same total-degree vector). So we seek to determine which of these total-degree vectors is realizable, without running through all $\sim 1.05 \times 10^{15}$ potential identifiable models. In this way, we can optimize any aberration function over these at most $11,881$ solutions.

4 The algorithm

We will not give a detailed justification of the basic algorithm in this paper — this can be found in [4] — but we do give all details to fully specify it.

Let $\omega := \max \beta_{j,h}$ (the maximum degree of a factor, over all of the monomials that we consider), let $s - 1 := m\omega$ (the maximum possible entry in the total-degree vector of an identifiable polynomial model), and let $Z := \{0, 1, \ldots, m\omega\}^d$ (the set of potential total-degree vectors for identifiable polynomial models).

Let X be the $n \times n$ diagonal matrix whose j-th diagonal component is the monomial $\prod_{k=1}^{d} x_k^{\beta_{j,k}}$ in the variables $x_1, \ldots, x_d$; that is, the matrix of monomials defined by

$$X \quad := \quad \mathrm{diag}\left(\prod_{k=1}^{d} x_k^{\beta_{1,k}}, \ldots, \prod_{k=1}^{d} x_k^{\beta_{n,k}} \right) \ .$$

So, X is just the diagonal matrix of monomials that we consider for inclusion in our polynomial model.

Let $X(t)$ be the matrix obtained by substituting $t^{s^{k-1}}$ for x_k , $k = 1, 2, \ldots, d$, in X . Thought of another way, we are simply evaluating each of the monomials (diagonal elements of X) at the point $(t^{s^0}, t^{s^1}, \ldots, t^{s^d}) \in \mathbb{R}^d$.

for $t = 1, 2, \ldots, s^d$ **do**
    Compute $\det(AX(t)A^T)$;
end
Compute and **return** the unique solution g_u , $u \in Z$, of the square linear system:
    $\sum_{u \in Z} t^{\sum_{k=1}^{d} u_k s^{k-1}} g_u = \det(AX(t)A^T)$, $\quad t = 1, 2, \ldots, s^d$;

For each potential total-degree vector u, we have $g_u \geq 0$. Moreover, $g_u > 0$ if and only if u is the total-degree vector of an identifiable model. In fact, $g_u = \sum \det^2(A_S)$, where the sum is over the models S having total degree u (see [4]). The trick is to compute each g_u without explicitly carrying out this sum, and that is what the algorithm does.

So, we have to make $s^d = (m\omega + 1)^d$ determinant calculations (to get the right-hand sides), followed by a solve of a square system of $s^d = (m\omega + 1)^d$ linear equations.

The optimal value of the *minimum aberration model fitting problem* is just

$$\min \{ f(u) \ : \ u \in Z \ , \ g_u > 0 \} \, ,$$

and for any given $f : \mathbb{Z}^d :\rightarrow \mathbb{R}$, we can scan through the $u \in Z$ having $g_u > 0$ to find an optimal solution. So the main work is in identifying the $u \in Z$ having $g_u > 0$, because such u are precisely the total-degree vectors of the *identifiable* polynomial models.

Unfortunately the numbers in the algorithm can get extremely large, and this makes working in double or even extended precision grossly insufficient. One possibility is to try working in infinite or very high precision. Stable and tuned numerical libraries in this realm do not currently exist, but we needed very limited matrix/vector functionality, so we fabricated our own utilities, employing ARPREC (a C++/Fortran-90 arbitrary precision package; see [2]). We note that working in very high precision is a growing trend in scientific computation (see [3]).

In Figure 1, we present a plot of solution values for one of our Vandermonde systems for a typical large example. Our variables are indexed by points in $u \in Z := \{0, 1, ..., mw\}^d$, and we number them conveniently as $n(u) := 1 + \sum_{k=1}^{d} u_k s^{k-1}$. For example, at the extremes, the point $u = (0, 0, ..., 0)$ gets numbered 1, and the point $u = (mw, mw, ..., mw)$ gets numbered by $(mw+1)^d = s^d$. So, in Figure 1, we have simply plotted the points $(n(u), g_u)$. It is easy to see that we have a very large range of solution values, further justifying our use of high-precision arithmetic. It is also interesting to note that the nonzeros are confined to a limited range of variable indices — we will return to and exploit this fact later.

As for the linear system solve, this is a Vandermonde system, therefore the number of arithmetic operations needed to solve it is quadratic in its dimensions (see [6], for example). In fact, as we indicate in the next section, we have chosen a very special Vandermonde system (successive powers of the points $1, 2, \ldots, s^d$) which has a closed form inverse. Though there are approaches for parallelizing a general Vandermonde system solve (see [21], for example), we selected our particular Vandermonde system to have this relatively simple

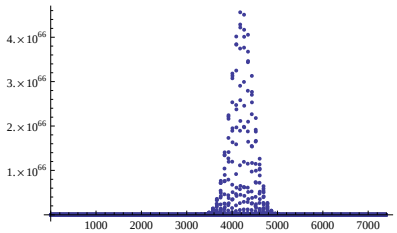

Fig. 1. Solution values

form because it enables a straightforward parallelization. Several extensions and enhancements to the base algorithm are also possible in the parallel realm; we will touch upon these later in the paper.

We remark that for other purposes, one may choose others points than $1, 2, \ldots, s^d$ (e.g., Chebyshev points) to generate a Vandermonde system, when one uses such a system to fit a polynomial. One driving concern can be the fit of a polynomial near the extremes of a range. We emphasize that we are not really *fitting* a polynomial — we have a particular polynomial implicitly defined, and we are seeking an efficient manner to determine its nonzero coefficients.

5 A special Vandermonde inverse

Let $N \times N$ matrix V be defined by

$$V_{i,j} := j^{i-1} \ , \ \text{for } 1 \leq i, j \leq N \ .$$

(in our application, we have $N := s^d$). We wish to solve a so-called "dual problem" of the form

$$V^T g = b \ ,$$

simply by evaluating V^{-1} and letting $g := V^{-T} b$. We apply this directly to the task of solving the linear system in the algorithm of the last section.

Our Vandermonde matrix is chosen to be a very special one. As such, it even has a closed form for the inverse V^{-1} :

$$V^{-1}_{i,j} := \begin{cases} (-1)^{i+N} \frac{1}{(i-1)!(N-i)!} \ , & j = N \ ; \\ i \ V^{-1}_{i,j+1} + \begin{bmatrix} N+1 \\ j+1 \end{bmatrix} V^{-1}_{i,N} \ , \ 1 \leq j < N \ , \end{cases}$$

where $\begin{bmatrix} N+1 \\ j+1 \end{bmatrix}$ denotes a Stirling number of the first kind (see [11], though they define things slightly differently there). The form for $V^{-1}_{i,j}$ indicates how each row of V^{-1} can be calculated independently, with individual entries calculated from right to left, albeit with the use of Stirling numbers of the first kind. We note that the Stirling number used for $V^{-1}_{i,j}$ does not depend on the row i , so

the needed number can be computed once for each column j . The (signed) Stirling numbers of the first kind can be calculated in a "triangular manner" as follows (see [20]). For $-1 \leq j \leq N$, we have

$$\begin{bmatrix} N+1 \\ j+1 \end{bmatrix} := \begin{cases} 0 \ , & N \geq 0 \ , \ j = -1 \ ; \\ 1 \ , & N \geq -1 \ , \ j = N \ ; \\ \begin{bmatrix} N \\ j \end{bmatrix} - N \begin{bmatrix} N \\ j+1 \end{bmatrix} \ , & N > j \geq -1 \ . \end{cases}$$

We remark that a Matlab code to calculate our desired Vandermonde inverse is available as VanInverse.m (see [23]), and a C code to calculate Stirling numbers of the first kind is available as mStirling.c (see [17]). Of course, we could not work with the Stirling numbers in double precision or long ints — even for the smallest problem that we consider ($N = 1369$), we encounter Stirling numbers of magnitude around 10^{3700}.

6 Limiting the range

From a completely naïve standpoint, it might appear that every variable g_u , $u \in Z := \{0, 1, ..., m\omega\}^d$, could be positive (i.e., nonzero) in the solution. But the plot of Figure 1 indicates otherwise. A simple idea is to try to narrow the range for which variables could be positive. Recall that our variables are numbered $1, 2, ..., (m\omega + 1)^d = s^d$, according to the map $n(u) := 1 + \sum_{k=1}^{d} u_k s^{k-1}$, for $u \in Z$.

Notice for example that $u = (0, 0, ..., 0)$ and $u = (m\omega, m\omega, ..., m\omega)$ are actually not achievable by adding up $m > 1$ distinct rows of β (after all, β itself has distinct rows). Therefore, already we see that $g_{(0,0,...,0)} = g_{(m\omega,m\omega,...,m\omega)} = 0$. Rather than continue in this direction with combinatorial reasoning, we cast the problem of determining the minimum and maximum $n(u)$ for which $g_u > 0$ as a pair of optimization problems. We will see that this pair of optimization problems can be solved very easily.

Let

$$\begin{aligned} I_{\min} := 1 + \min \ & y^T(\beta c) \ , \\ \text{subject to} \\ & \det(A_y) \neq 0 \ ; \\ & y^T e = m \ ; \\ & y \in \{0, 1\}^n \ , \end{aligned}$$

where $c := (s^0, s^1, ..., s^{d-1})$, and A_y is the matrix comprising the m columns of the $m \times n$ matrix A indicated by the vector y of binary variables.

This is a *linear* objective matroid optimization problem. As such, it is exactly solvable by the "greedy algorithm" (see [15] for example). We simply select variables to include into the solution, in a greedy manner, starting from the minimum objective-coefficient value $(\beta c)_i$, working up through the greater values. We only include the i-th column A_i of A in the solution (ie., set $y_i = 1$) if the columns from A already selected, together with A_i , are linearly independent. We stop once we get m columns.

We define and calculate $I_{\max}$ similarly. We simply replace min with max in the definition, and in the greedy algorithm we start with *maximum* objective-coefficient value $(\beta c)_i$, working *down* through the *lesser* values.

Note that this can be done with no numerical difficulties. We only use the objective vector βc above to order the variables. The linear algebra (mentioned above) only involves the matrix A (which has modest coefficients). Moreover, we do not need to even explicitly form the objective vector βc ; we just observe that this vector lexically orders the rows of β , and so in the greedy algorithm for determining $I_{\min}$ (respectively, $I_{\max}$), we simply choose index i before i' $(1 \leq i, i' \leq n)$ if $\beta_{i\cdot}$ is lexically less (respectively, greater) than $\beta_{i'\cdot}$.

Finally, referring back to the last section where we represented our sought after solution as $g := V^{-T}b$, it is easy to see that armed with the values $I_{\min}$ and $I_{\max}$, we only need take the dot product of rows numbered between $I_{\min}$ and $I_{\max}$ with the right-hand side b , as all other dot products would be zero. In particular, we do not need all columns of V^{-1} . As columns of V^{-1} are calculated from right to left, we can halt that computation once we have the column numbered $I_{\min}$.

7 Generation of the right-hand side

In this section, we clarify how the right-hand side of our linear system is generated. As described in the last line of the algorithm (Section 4), the right-hand side elements are $\det(AX(t)A^T)$, for $t = 1, \ldots, s^d$. The matrix X is a diagonal matrix of monomials described as follows:

$$X_{jj} := \prod_{k=1}^{d} x_k^{\beta_{j,k}} \ .$$

The monomials are in d variables. A given X is evaluated at a point t by substituting $t^{s^{k-1}}$ for the variable x_k , with k ranging from 1 to d , in each of the monomials. Thus, when $\det(AX(t)A^T)$ is evaluated, X is no longer a

diagonal matrix of monomials; rather, it is a diagonal matrix of real numbers corresponding to the evaluations of each of the monomials in X at a point $(t^{s^0}, t^{s^1}, \ldots, t^{s^{d-1}}) \in \mathbb{R}^d$. There are s^d points that we evaluate in this manner, as t iterates from 1 through s^d , each leading to one of the s^d right-hand side elements. It is easy to see that when $t = s^d$, we have $X(s^d)$ containing an entry at least as large as s^d raised to the s^{d-1} power. These numbers quickly become too large to permit solving the linear system with reasonable accuracy, even with very high precision. Thus, we scale our Vandermonde system to keep the size of the values in $X(t)$ bounded by unity.

8 Parallel implementation

In this section we will consider how the algorithm described above is currently implemented as it targets the Blue Gene/L computer at IBM's T.J. Watson Research Center. The initial implementation uses a one-dimensional data decomposition and a few algorithmic wrinkles that allow the code to run with a data footprint that might be smaller than expected. Additionally, we delineate a series of slight changes to the implementation that will increase the size of the largest problems that can be solved with this technique (without the use of out-of-core extensions).

First, let us consider the generation of the right-hand side (or right-hand sides) for the systems $V^T g = b$ that we wish to solve. As a brief aside, it is important to remember that we will not be performing a backsolve, but an explicit $V^{-T}b$ calculation; the distinction will be an important one.

In Section 7, the mathematical definition of the right-hand side's components was given. From the point of view of a parallel implementation, this is a simple process. First, A is generated (identically) on all nodes in the processor grid. Next, the $X(t)$ are generated based on the processor id. For example, if s^d is 100 and there are 10 processors, $X(1)$ might be generated on processor #1, along with $X(11)$, $X(21)$, etc. Then for each t , the matrix $AX(t)A^T$ is computed, its Cholesky factor derived, and the diagonal elements of the Cholesky factor squared and multiplied in order to calculate the right-hand side entry $\det(AX(t)A^T)$.

As has been stated, the initial implementation of the algorithm was done with a one-dimensional data decomposition in mind. Because we are, essentially, doing an explicit formation of V^{-T}, it is logical (and efficient) to form the i-th element of the right-hand side on the processor that will contain the i-th column of the matrix V^{-T}. Thus, at the end of this entirely parallel phase, the right-hand side is formed, distributed (and nowhere duplicated) over the entire processing grid.

The next step was the formation of the list of the Stirling numbers of the first kind. In order to form row $N+1$ of the Stirling numbers, that is $\begin{bmatrix} N+1 \\ j+1 \end{bmatrix}$ for $-1 \leq j \leq N$, we have taken advantage of the fact that these values can be computed with a Pascal's Triangle approach. Thus, in the first step only one processor is busy, in the second step two processors are calculating, etc., up to the number of processors in the machine or the Stirling row of interest, whichever is less. In a one-dimensional setting, this sub-problem can be viewed as a simple systolic shift. In step r, processor p, computes the Stirling number $\begin{bmatrix} r \\ p \end{bmatrix}$, having received $\begin{bmatrix} r-1 \\ p-1 \end{bmatrix}$ from processor $p-1$ and having sent $\begin{bmatrix} r-1 \\ p \end{bmatrix}$ to processor $p+1$ in the previous round. The performance-degrading "ripple-effect" is easily avoided by a two stage send/receive (processors of even rank), receive/send (processors of odd rank) process.

The manner in which this step of the implementation is performed is independent of the other steps in the process as, even in the most naive implementation, an Allgather (collect) phase can provide all processors with the entire list of Stirling numbers. This is of interest because, as we will later describe, memory parsimony may become a concern as the size of the problems increase. During the Stirling computation, which can be performed even before the formation of the right-hand sides, the only memory required is that which holds the Stirling vector. We will address this issue in Section 8.2 when we consider some alternate design choices. For now, let us suppose that the Stirling numbers are generated by mapping a one-dimensional, self-avoiding walk (placing processors 0 and $P-1$ next to each other) onto the three-dimensional Blue Gene machine, and let us further suppose, for the sake of specificity, that every processor has a copy of the entire list of Stirling values.

Given that the Stirling numbers are locally available, computing V^{-T}, by column, is quite straightforward. Again, let us view the columns in a one-dimensional cyclic distribution over the processor grid (array). For any given column we only need to compute (see the equations in Section 5) $V^{-1}_{i,N}$ then, using this value and the list of Stirling numbers, we sequentially compute the inverse of the values for the remainder of the column.

Once the inverse has been computed for all columns held by a processor, a simple scalar $\times$ vector calculation, using the corresponding right-hand side entry (local) yields a given processor's contribution to the solution vector. At this point, a simple reduction (add) operation yields the overall solution vector.

8.1 Why ARPREC?

Initially, our goal was to use the rigorously tested and widely available LAPACK and ScaLAPACK libraries to solve for our generated right-hand sides. However, it turns out that for matrices of the type under consideration here, even a 9×9 matrix solve was not possible using LAPACK and standard double-precision arithmetic. We considered using iterative refinement methods[7] and, perhaps quad-precision arithmetic, but the 25×25 matrix (our next larger example) did not appear likely to yield to that approach. Because our goal was to work with instances of the problem that are larger by several orders of magnitude, we chose not to use Gaussian elimination and iterative refinement, but the explicit formation of the inverse of the matrix using very high precision arithmetic. The ARPREC package met our needs exactly, as it was written in C++, well-documented, and heavily tested.

Because we are using an unusual approach (and not the one we first intended to pursue), the next subsection is included to (partially) justify the path we chose.

8.1.1 The intractable numerics of small problems

We performed some experiments with Vandermonde matrices coming from our application, but of very modest size. Our goal was to try to see the numerical limitations of even small examples. In Table 1, "N" indicates the order of the Vandermonde matrix, and "prec" indicates the number of digits of precision.

We took the design matrices P in these examples to consist of very small integer entries. With this, it is easy to argue that the solution of our Vandermonde system should be all integer.

The "y" and "n" entries in the table signify either *yes*, the solution was probably valid (meaning all solution values were exact integers and no solution value was negative) at that particular numerical precision, or *no*, the solution was numerically unstable. The sample zero is an example of the kind of "zero" seen at that particular numerical precision. The table increases by order of precision, and displays the lowest digits of precision for which the solution was numerically stable. For example, on the 25×25 example, the "no scaling" option did *not* work at 29 digits of precision, but it *did* work at 30 digits of precision. One further side comment: although we indicate that the 25×25 example works with the "scaling" option (see last line of Section 7) at 16 digits of precision, one of the entries was 1.00372, which we accept as numerically stable. This entry remained at this accuracy until the 30 digits of precision test, at which point it became exactly 1. As mentioned before, all other solutions contained exact integers.

Table 1
Intractable numerics.

N	prec	scaling on	sample zero	scaling off	sample zero
9	1	y	6.20126×10^{-11}	y	-3.50394×10^{-7}
25	1	n	n/a	n	n/a
25	10	n	n/a	n	n/a
25	15	n	n/a	n	n/a
25	16	y	-2.05918×10^{-18}	n	n/a
25	25	y	-2.05918×10^{-18}	n	n/a
25	30	y	-2.05918×10^{-18}	y	3.16422×10^{-12}
32	15	n	n/a	n	n/a
32	16	y	-3.47723×10^{-14}	n	n/a
32	30	y	-5.398×10^{-27}	y	-6.71336×10^{-12}

In general, it has been our observation that the number of digits of precision required for an acceptable answer is roughly the same as (somewhat higher for small systems and only slightly lower for larger instances) the dimension of the matrix being implicitly inverted.

8.2 Implementation challenges and solutions

While the initial implementation allows us to handle matrices of unprecedented size (in this sub-field), there are some limitations to the implementation described above. These issues stem from three areas: memory consumed, time-to-solution, and scale-down to smaller processor configurations. In this section we will discuss the next steps in refining our implementation and how we are addressing each of these issues. When a concrete example is required, we will consistently use the original, 360 TF, 128K processor, Blue Gene/L supercomputer installed at Lawrence Livermore National Laboratories along with a problem size of $N = 128K$ [1] (in order to simplify the mathematics; there will be nothing that requires powers-of-two in our implementation). The authors realize that the LLNL system is not for public use, and we reference it for illustrative purposes only.

Recall that there are five phases to this computation.

[1] For example, we can produce a problem of almost this size having $d = 2$, $\omega = 9$, $n = 100$ and $m = 39$. For such parameters, the direct enumeration of all $\sim 9 \times 10^{28}$ potential identifiable models is vastly beyond any possibility.

(1) The generation of the right-hand sides (this could also take place after step (2) or (3)).
(2) Computation of the Stirling values of the first kind.
(3) Computation of V^{-T} .
(4) Computation of $V^{-T} \times b$ locally (where b is a single right-hand side).
(5) Reduction of the local contributions computed in (4) for the solution vector.

While we indicate (above) that some of these steps can be interchanged, we will address them in the indicated order. Note that we are driving towards an implementation that will achieve a relatively high percentage of peak machine utilization while being parsimonious enough with memory to handle very large problems.

8.2.1 Generation of the right-hand sides: revisited

As has been mentioned, the generation of the right-hand sides was viewed as embarrassingly parallel. However, when we consider the scaling challenge, we need to consider that a problem size of 128K can be achieved with an A matrix of dimensions 40×100. This does not, at first glance, appear to be a concern, but if we use 128K digits of precision, each element of the matrix will consume approximately 72KB of memory and this "small" matrix will overflow the 256MB of memory available when running on the original (512MB/node) Blue Gene/L series machines in "virtual node mode" in order to transparently take advantage of both cores on the machine (i.e. to almost double the performance). For the rest of the discussion, we restrict ourselves to a target memory footprint of 100MB per processor. This may seem an overly restrictive approach, but in view of scaling to even larger problems (or smaller configurations) as well as future processing systems with fewer (main memory) bytes per flop, it does not seem ill-advised.

For the target problem size, the 40×100 matrix A and the 40×40 output matrix $AX(t)A^T$ could co-reside if they are stored across a 2×2 subgrid of processors in cyclic fashion. In this manner, the right-hand sides can be generated on the subgrids and moved within those subgrids to the appropriate location. Because the right-hand side points generated will "stack up" (i.e. consume memory) this can be extended to 4×4 grids at a slight cost to generation time. After the right-hand sides are generated, these matrices are no longer required and the memory can be reclaimed.

It is important to note that the calculation of the right-hand side involves roughly 220K expops (extended-precision operations) per value. Thus, the formation of right-hand sides is as expensive as the formation of the inverse of the large matrix. The formation of the right-hand sides is a matter of matrix-

vector and matrix-matrix multiplication (albeit in extremely high-precision), followed by a high-precision Cholesky factorization. The techniques required to get superb scaling in this arena are well-understood [9,22], and the underlying data distribution (block-cyclic) allows us to deal with memory limitations quite easily. The only potential issue is the multiple ongoing subcommunicator collectives, but this avenue has been exercised in the target architecture before, and performance has been admirable. Because we are not restricting ourselves to Blue Gene, however, it is possible that we will have to address this issue more thoroughly in the future.

While we have not tested the component pieces of the code on the target problem size/architecture, we are confident that these routines will scale as required.

8.2.2 Generating Stirling numbers of the first kind: revisited

Generating Stirling numbers, in the manner of forming Pascal's triangle, is a well-understood process. Every processor receives one value, computes one value (an extended precision multiply and add), and sends one value at every step of the algorithm in which they are active. In the example of interest here, every processor is responsible for a single "column" of the vector of Stirling numbers. The triangular nature of the computation implies that half of the processors are busy on average in any given step (because the order of the matrix is the same as the processor count in this instance). This computation can use a mapping completely separate from that used by the other stages in the process. On Blue Gene/L, to avoid interference, one simply embeds a one-dimensional self-avoiding walk on the three-dimensional machine. This results in communication that is nearest-neighbor (taking advantage of torus links) as well as interference free. Note that the algorithm does not depend upon the number of processors and the number of Stirling values needed being equal, but we do not address the load imbalance issues that naturally arise here.

Again, we must consider the memory footprint issue. For clarity of exposition, it is easiest for us to figuratively leave the Stirling values where they were created, one on every processor in the machine. As we describe the later steps, we will address how these values need to be communicated across the machine.

8.3 Inverting the matrix : revisited

As we consider how to approach truly large problems in this domain, we must naturally consider different data distributions and methods for staging the computations. In this section, we will list the alternatives that readily suggest themselves and step through a bit of how one might perform these final three

steps (matrix inversion, local matrix-vector computation, and reduction of matrix-vector results for the final answer) separately. It appears that separating these steps is difficult and/or highly inefficient for the motivating example under consideration and, in later sections, we will necessarily be interleaving these steps.

First, a partial list of the potential approaches that one might take in this domain.

(1) One-dimensional column-based distribution. Unstaggered or Staggered.
(2) Two-dimensional (blocked) distribution. Unstaggered or Staggered.
(3) Two-dimensional (blocked) distribution. "Row-twisted." Unstaggered or Staggered.

Because our choices are something of a Cartesian product, some discussion of the separate features appears warranted.

The one-dimensional, column-based distribution is the simple one previously described. An entire column of V^{-T} will "live" on a given processor. This is obviously impractical if we separate steps (3)-(5) of the process described above as it would require 128K elements, each of size 72K to simultaneously exist on a processor. This would consume approximately 9 GB of storage, which is 90 times what we are allowing ourselves. Because this is considered impractical, we will delay the discussion of staggering the computation until the next subsection, wherein we will evaluate the utility of that technique in somewhat altered context.

The difficulty mentioned in the previous paragraph hardly seems amenable to any solutions involving data distribution. The problem is that no processor has enough memory to hold its own solution vector, and because the global size of the matrix does not change with data distribution, this path might appear that we are at an impasse.

However, if we consider a 128×1024 ($P \times Q$) processor mesh, imposed in block fashion on our 128K $\times$ 128K ($N \times N$) matrix and consider that while every processor computes the same number of values as in the one-dimensional approach, they do not compute the same contribution to a given column. Recall that elements of V^{-T} must be computed serially within a given column, but every column is independent of the other columns being computed. In the 128×1024 configuration every processor in a given column of the processing mesh must hold not 128K elements of a given column, but 1K elements of that column and is responsible for computing 128 (sub-)columns. The distinction is important because in this instance the sub-columns do not have to co-reside in memory.

At this point, let us take a half-step towards the interleaving more fully ex-

plored in the next subsection. Once we compute a partial column (1K elements) of V^{-T}, one could multiply the result by the corresponding right-hand side (element) and reduce the result. Thus it would appear that what is needed in this case is far less memory per processor at any one time:

(1) 1K elements of the V^{-T} matrix (N/P),
(2) 1K elements of the Stirling vector,
(3) 1K elements of the local contribution to the right-hand side (which can be almost entirely space-shared with (1)), and
(4) At least one element of the overall (reduced) solution (1K elements of the reduced solution distributed over Q, 1K, processors).

The cost of this solution is one of lag time. The computation is, as has been mentioned, serial in a column. Thus, the first processor row must complete (in parallel) computation of the first 1K elements of the first column (in their respective solution spaces) before the second row of processors is active, etc. On our motivating example (where the number of cores is the same as the matrix dimension), this imposes a 50% penalty. The other difficulty with this scheme has to do with our self-imposed 100 MB limit. However, 1K was only selected as a round number and we could simply cut that in half to remedy this apparent issue.

Because it conceptually interesting, if not easy to motivate from a practical standpoint (as in the problem at hand, using a one-dimensional approach with a bit of added processor coordination will prove sufficient on the target architecture), we will briefly describe row twisting or skewing. In the preceding paragraph, the 50% penalty was mentioned. However, the astute reader has probably noticed that these machine cycles are needlessly wasted. In row-skewing, one should consider a single column of processors and the columns of the matrix that will be formed there. Instead of having all processors "start" (have their 1-st element) on the same row of processors, this can be skewed so that every column is rotated around the processor ring, in the case of interest, by 1K elements. Thus, all processors can begin computing at the same time. The apparent cost to this added efficiency is that every processor in that column must now store all 128K Stirling numbers. This is, of course, illusory. The most straightforward solution to this is to have every processor contain a vector for 1K Stirling numbers (those currently being used) and its "shared store" Stirling vector. Put most simply, every 1K subset of the overall 128K Stirling vector has a "home" processor. Because all processors in a column are using the same 1K subset at the same time, broadcasts of shared stores would punctuate the calculations. These shared stores could, if need be, be distributed over processor rows (thus only one extra value is stored per processor). The cost here is the additional broadcasts and distributed collects that would have to punctuate computation and, while these are not really an issue on the Blue Gene system, they may be problematic on other

architectures.

8.4 (Further) Conflating procedural steps

When we couple steps (3)-(5), the problems mentioned above largely disappear. Here we describe how these issues are dealt with, additional penalties incurred (if any), a straightforward algorithm that would likely exceed 10 TF on the motivating system, and additional algorithmic variants that would allow for further scaling (of problem and machine size) and would be performance portable to other systems (systems without Blue Gene's fast collective networks). Fortunately, in this instance, the solution is not overly complex. The data distribution is simpler than that described above, the added coordination cost is not onerous, the solution scales (both up and down) quite well, and memory parsimony can be traded for algorithmic efficiency in a straightforward manner.

The problem with the one-dimensional data distribution is easily seen as not a matter of matrix memory consumption. In order to compute any element of V^{-T} , one requires only the value $V_{i,N}^{-T}$, the previous entry in that column, and the corresponding Stirling value. Thus, only three elements of a given columns of V^{-T} are required at any point in time on a given processor. The apparent problem is that one cannot duplicate the Stirling vector on all processors (it too is 9 GB in size). The general solution to this problem is to stagger the computations. Because every processor requires the Stirling number $\begin{bmatrix} N \\ j \end{bmatrix}$ to compute the N-th entry in the column, one could simply leave the Stirling vector distributed and use a systolic shift to move the Stirling vector around the (embedded) ring of processors. This precisely coincides with the bubbling up (or down, depending on your view) of the V^{-T} values. The penalty for this is that it yields a 50% utilization in the motivating example (the first processor is finishing when the initial Stirling entry arrives at the last processor and that processor begins its computation). Instead of a systolic shift, the stagger could be eliminated via a broadcast at each step of the process. Both of these do incur a communications overhead, but it appears to be minimal. While the systolic shift may well be superior, the punctuated broadcast is easier to unambiguously analyze. The broadcast of 128K repeats of 72KB of data across Blue Gene would take only 37 seconds on the torus network. Here, we assume 0.40 Bytes/cycle communicated and an MPI overhead of 15000 cycles per iteration[1]. This seems to compare favorably to the extrapolated (computation-only) execution time of 75 minutes for a problem of this size on the 128K processor system at 128K digits of precision. Further, on the system that we will likely use, the next-generation Blue Gene/P system, we expect a smaller drop-off in performance (when considered in percent of realizable peak). The drop-off will likely be compounded by the shift to the use of SIMD

arithmetic (see section 9.1) which results in a higher percentage of peak (because communication time is viewed, for simplicity, as non-overlapped, the effect is greater as serial efficiency is enhanced).

When considering the two-dimensional distribution scheme solutions discussed in the previous section through the lens of memory parsimony there is little to add. We require only a few (three) values from the V^{-T} matrix at any given time. We can tune the number of Stirling values required on any process by distributing the collection across every processor row and doing periodic collects. For example, collecting 128 Stirling numbers to every processor in the row every 128 computations. This allows an easily tuned trade between processor efficiency and memory consumed. This same "periodic" approach carries over to the "twisted case" where the only difference is that below a certain threshold of problem size all Stirling numbers (not just $1/P$ of them) can exist somewhere (for collection) in all processor rows. Above that threshold, where there is not enough space to hold all such values, the values can be distributed across the entire grid and multiple collections are required. In this domain, however, communication begins to become a large component of overall runtime and a variation on the one-dimensional approach (with periodic reductions of the subsections of the computed $V^{-T} \times b$ values) seems preferable. In conclusion, the more complex two-dimensional distribution is advantageous because of architectural considerations (on some architectures), not for algorithmic reasons.

9 At what cost performance?

Unsurprisingly, as we attempt to scale up to huge matrices, processor counts, and digits of precision (simultaneously), the algorithms we plan to use become more involved and, at extreme scale, appreciably more expensive in terms of communications overhead incurred. We remark that even at extreme scale, the newer Blue Gene/P's ability to deftly overlap computation and communication makes it an appealing architectural platform for our application.

In the previous sections we have described, at some length, how one can use the non-local memory of the machine as a shared cache of sorts and how Blue Gene's highly efficient collective operations, as embodied in MPI, make this an attractive option. Unfortunately, such techniques do take away the ability to model large scale problems on small scale systems (even when one is willing to pay the cost in terms of run-time). To that end, we have begun the design of an out-of-core implementation, but that algorithm and its expected performance is beyond the scope of this paper. We have also constructed a simple checkpointing infrastructure for the high-precision values so as to facilitate the execution of the 128K run on limited resources. Because the

execution of this algorithm would take several days on a single rack (1024 processors) of Blue Gene/L, checkpointing, of a very simple form, would have to be added to the code as we would likely not get such a dedicated resource for several consecutive days. On a less reliable system, the same checkpointing infrastructure could be used to address that issue in the same manner (where more frequent saves to non-volatile memory would be needed).

9.1 SIMD units

Anyone familiar with the Blue Gene system architecture is almost certainly aware of the SIMD floating-point units in the system. If we could effectively use the SIMD units in Blue Gene and tune the code a bit better for timing (rarely is anything in the L1D cache when we need it in the ARPREC calculations, though the much larger L3 cache can be so-exploited), we could well get a twofold improvement in compute speed. One way of achieving this SIMDization is to interleave the computations. For an x-way SIMD unit, one would interleave (in storage) x pieces of data/values. For example, for the two-way SIMD unit under consideration here, instead of storing two columns of data in column-major order, we would store the pair in contiguous storage, but in row-major order within this contiguous memory. A straightforward unrolling of sections of the ARPREC code could allow the x components of the computation to proceed in parallel. More ambitious techniques (rewriting crucial parts of the functionality ARPREC) are also possible. Not only would this allow the use of the SIMD units, but it would allow some re-use of data once loaded into the L1 cache.

9.2 Multiple right-hand sides

Multiple right-hand sides present some unusual difficulties in this context. Generally, additional right-hand sides are not heavily factored into storage considerations. Here, they are one of the few persistent pieces of storage that we have to work around. As the number of right-hand sides increases, not only do the right-hand sides themselves have to be held distributed (naively, the 2D scheme would have us storing 128 elements of the rhs on every processor in the column, duplicated). The real problem here is not the right-hand sides, but the buffers for the summand V^{-T} components which are each 1K elements in size must be distributed over processor rows and either rotated around the row or collected. Both of these solutions are expensive in terms of data movement, even with self-avoiding walks imposed on each 2D sub-plane (32x32, where the overall machine is typically viewed as 2x64x32x32) of the Blue Gene/L machine under consideration. The most straightforward solution is to generate

the right-hand sides "on-demand," and while this appears a sound approach, we have not yet implemented it.

10 Experiments

The Blue Gene/L supercomputer [8] is the first-generation incarnation of the Blue Gene computer series. Several papers detailing the architecture have been published (for example [13]). Only a few of the many features of this architecture are central to our algorithmic embodiment however.

Blue Gene/L contains, among other networks, a three-dimensional torus interconnect. Each link is capable of sending or receiving data at 0.22 Bytes/cycle per link (thus, higher-dimensional broadcasts and one-to-many can proceed at greater than single-link speed) and collectives, available for use through the pervasive MPI interface, are highly tuned to take advantage of Blue Gene's architectural capabilities[1]. In essence, this means that both nearest neighbor communications and one-to-many communications are highly efficient primitives upon which an application can be based.

At the individual node level, the first-generation Blue Gene/L machines contained 512MB of data shared between two SIMD-FPU-enhanced 440 PPC cores. While these cores each run at 700 MHz, they are capable of SIMD FMAs (Floating point Multiply-Adds), so that each is in fact able to reach 2.8 GF/core or 5.6 GF/node. Taking advantage of the SIMD units is a realizable goal for the next-generation of our implementation, but we did do so in this paper (when we report percentage of peak achieved, we are "honest," as we report the percentage of SIMD peak, though we are not using the SIMD facility).

The last salient feature of the architecture worthy of any mention in this paper is the fact that it can be run in co-processor mode (one processor for computation and one-processor as a communication engine) or virtual node mode (memory is split and all processors compute and communicate). In its present form, our application experiences virtually no slowdown (as measured by percentage of peak) in using virtual node mode. This may somewhat surprising because the algorithms employed by ARPREC lead to, unsurprisingly, 2 read/1 write behavior that could be bandwidth intensive, until one considers the percentages of peak currently seen. More relevantly, because virtual node mode has half of the memory per core, we will have to streamline our application (and perhaps the ARPREC library, if we can get permission to do that) to take advantage of both cores at scale.

We present our main computational results in Tables 2-3. Table 2 indicates

Table 2
Performance on 8192 cores of the Blue Gene/L Supercomputer for various matrix sizes. The decrease in time when going from 3,025 to 4,096 appears to be due to the simple bit representation of 4,096 and the manner in which ARPREC takes advantage of that representation. The largest run was measured at approximately 884 GF.

d	ω	n	m	$\binom{n}{m}$	N	prec	time
2	9	100	4	3.92123×10^6	1,369	10000	39.893
2	9	100	5	7.52875×10^7	2,116	10000	55.9402
2	9	100	6	1.19205×10^9	3,025	10000	76.6629
2	9	100	7	1.60076×10^{10}	4,096	10000	74.4021
2	9	100	8	1.86088×10^{11}	5,329	10000	128.941
2	9	100	9	1.90223×10^{12}	6,724	10000	160.191
2	9	100	10	1.73103×10^{13}	8,281	10000	372.479
2	9	100	11	1.41630×10^{14}	10,000	10000	451.132
2	9	100	12	1.05042×10^{15}	11,881	11000	545.739

Table 3
Performance on 1024 cores of the Blue Gene/L Supercomputer for a fixed matrix size at different levels of precision. The % of peak achieved gradually decreases, but at a decreasing rate.

N	prec	% Peak	GF	time (sec.)
4096	10000	4.499	128.993	399.82
4096	15000	4.363	125.081	559.78
4096	20000	4.060	116.419	875.88
4096	25000	4.030	115.431	912.53
4096	30000	4.002	114.738	1301.71

that we can scale up to large problems rather efficiently. A simple linear fit to the last three data points in Table 3 suggests that, unchanged, the current algorithm could achieve almost 3.5% of peak at the scale under discussion here. While the simple one-dimensional approach outlined above would run at, essentially, that same rate (3.49%), our plan is to only use that as a stepping stone to one of the more scalable two-dimensional solutions. In concert with the various opportunities for SIMDization we have outlined, we believe that the code will run at greater than 6% of peak on systems as large as (or larger than) the original Blue Gene/L system at LLNL.

11 Conclusions

We believe that we have made a convincing case that the application under consideration in this paper could achieve 7 TF on the Watson Blue Gene/L system (BGW) and scales almost perfectly to the largest configurations available today. As petascale and exascale systems are realized, we believe that methods such as those used here for memory conservation, in concert with high-precision arithmetic, will need to be explored. We hope that we have made some small contribution to such an effort.

Acknowledgments

Our work was carried out under an Open Collaborative Research agreement between IBM and UC Davis. We would like to thank Bob Walkup at IBM Research for his help in many aspects of this work, Fred Mintzer at IBM Research for the use of the Blue Gene/L Supercomputer, Jim Sexton, also of IBM Research, for his tutorial on how to use the hardware performance counters on Blue Gene/L. We are deeply indebted to David Bailey and his team for the ARPREC software package and documentation.

References

[1] Almási, G., Archer, C., Castaños, J.G., Gunnels, J.A., Erway, C.C., Heidelberger, P., Martorell, X., Moreira, J.E., Pinnow, K., Ratterman, J., Steinmacher-Burow, B.D., Gropp, W., Toonen, B.: Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development* **49:2-3** (2005) 393–406.

[2] ARPREC, `http://crd.lbl.gov/~dhbailey/mpdist/mpdist.html`

[3] Bailey, D.H.: High-precision arithmetic in scientific computation. *Computing in Science and Engineering* **7:3** (2005) 54–61.

[4] Berstein, Y., Lee, J., Maruri-Aguilar, H., Onn, S., Riccomagno, E., Weismantel, R., Wynn, H.: Nonlinear matroid optimization and experimental design. *SIAM J. on Discrete Mathematics* **22(3)** (2008) 901–919.

[5] Berstein, Y., Lee, J., Onn, S., Weismantel, R.: Nonlinear optimization for matroid intersection and extensions, *IBM Research Report* RC24610, 07/2008.

[6] Björck, A., Pereyra, V.: Solution of Vandermonde systems of equations. *Mathematics of Computation* **24** (1970) 893–903.

[7] Buttari,A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., and Kurzak, J.: Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications*, **21:4** (2007) 457–466.

[8] Chiu, G.L.-T., Gupta, M., Royyuru, A.K. (guest editors): "Blue Gene." *IBM Journal of Research and Development* **49:2/3** (2005).

[9] Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., Whaley, C.: Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming* **5:3** (1996) 173–184.

[10] Eisinberg, A., Fedele, G., Imbrogno, C.: Vandermonde systems on equidistant nodes in $[0, 1]$: accurate computation *Applied Mathematics and Computation* **172** (2006) 971–984.

[11] Eisinberg, A., Franzé, G., Pugliese, P.: Vandermonde matrices on integer nodes. *Numerische Mathematik* **80:1** (1998) 75–85.

[12] Fries, A., Hunter, W.G.: Minimum aberration 2^{k-p} designs. *Technometrics* **22** (1980) 601–608.

[13] Gara, A., Blumrich, M.A., Chen, D., Chiu, G.L.-T., Coteus, P., Giampapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Liebsch, T.A., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vranas, P.: Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development* **49:2–3** (2005) 195–212.

[14] Harvey, N.: Algebraic algorithms for matching and matroid problems. To appear in: *SIAM Journal on Computing*.

[15] Lee, J.: "A First Course in Combinatorial Optimization." Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge (2004).

[16] Lee, J., Ryan, J.: Matroid applications and algorithms. *INFORMS (formerly ORSA) Journal on Computing* **4** (1992) 70–98.

[17] mStirling.c,
`http://ftp.bioinformatics.org/pub/pgetoolbox/addins/src/mStirling.c`

[18] Mulmuley, K., Vazirani, U.V., Vazirani, V.V.: Matching is as easy as matrix inversion. *Combinatorica* **7** (1987) 105–113.

[19] Pistone, G., Riccomagno, E., Wynn, H.P.: "Algebraic Statistics." *Monographs on Statistics and Applied Probability* **89**, Chapman & Hall/CRC, Boca Raton (2001).

[20] Tweddle, I.: "James Stirling's Methodus Differentialis: An Annotated Translation of Stirling's Text." Sources and Studies in the History of Mathematics and Physical Sciences, Springer (2003).

[21] Tang, W.P., Golub, G.H.: The block decomposition of a Vandermonde matrix and its applications. *BIT* **21** (1981) 505-517.

[22] van de Geijn, R.A., Watts, J.: SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency — Practice and Experience* **9:4** (1997) 255–274.

[23] VanInverse.m, `http://www.mathworks.com/matlabcentral/files/8048/VanInverse.m`

[24] Wu, H., Wu, C.F.J.: Clear two-factor interactions and minimum aberration. *Annals of Statistics* **30** (2002) 1496–1511.