Math. Prog. Comp. (2010) 2:103–124
DOI 10.1007/s12532-010-0014-4

FULL LENGTH PAPER

# Efficient high-precision matrix algebra on parallel architectures for nonlinear combinatorial optimization

**John Gunnels · Jon Lee · Susan Margulies**

Received: 21 August 2009 / Accepted: 2 June 2010 / Published online: 19 June 2010
© Springer and Mathematical Programming Society 2010

**Abstract** We provide a first demonstration of the idea that matrix-based algorithms for nonlinear combinatorial optimization problems can be efficiently implemented. Such algorithms were mainly conceived by theoretical computer scientists for proving efficiency. We are able to demonstrate the practicality of our approach by developing an implementation on a massively parallel architecture, and exploiting scalable and efficient parallel implementations of algorithms for ultra high-precision linear algebra. Additionally, we have delineated and implemented 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 efficiency, reliability, and interconnect perspectives.

**Keywords** Nonlinear combinatorial optimization · Matroid optimization · High-performance computing · High-precision linear algebra

**Mathematics Subject Classification (2000)** 90-08 · 90C27 · 90C26

## 1 Introduction

Our goal is to demonstrate that matrix-based algorithms for nonlinear combinatorial optimization problems, mainly conceived in the context of theoretical computer

J. Gunnels · J. Lee (✉)
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
e-mail: jonlee@us.ibm.com

J. Gunnels
e-mail: gunnels@us.ibm.com

S. Margulies
Department of Computational and Applied Mathematics, Rice University, Houston, TX, USA
e-mail: susan.margulies@rice.edu

science for proving (worst-case) computational efficiency, can be efficiently implemented on massively parallel architectures by exploiting efficient (and 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 nonlinear combinatorial 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., [6,7,16,22]), 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., [19,20]). 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 linear algebra.

In Sect. 2, we describe the nonlinear matroid-base optimization problem and give a few applications. In Sect. 3, we describe the algorithm of [6], which was designed to establish, under some technical assumptions, the polynomial-time worst-case complexity of the nonlinear vectorial matroid-base optimization problem. At the heart of the algorithm is the solution of an extremely large (dual) Vandermonde system. In Sect. 4, we describe a strategy for solving the (dual) Vandermonde system, using a closed form inverse. In Sect. 5, we demonstrate how the solution of a pair of simple linear-objective matroid-base optimization problems can be used to reduce to a smaller (dual) Vandermonde system. In Sect. 6, we describe how we generate the needed right-hand side for our (dual) Vandermonde system. In Sect. 7, we describe our parallel implementation on a Blue Gene/P supercomputer. In Sect. 8, we describe strategies to employ at extreme scale. In Sect. 9, we describe the results of our computational experiments. Finally, in Sect. 10, we make some brief conclusions.

## 2 Background

Our starting point is the paper [6] which presents a theoretically efficient algorithm for a broad class of multi-objective nonlinear combinatorial optimization problems. The type of problem addressed is of the form

$$\mathcal{P} : \min \{ f(W(B)) \ : \ B \in \mathcal{B} \},$$

where

(1) $W$ is a $d \times n$ matrix of integers, and $W(B)$, the $W$-image of $B$, is defined to be the $d$-vector with $i$th component $\sum_{j \in B} W_{i,j}$,

(2) $f : \mathbb{R}^d \to \mathbb{R}$ is arbitrary, and

(3) $\mathcal{B}$ is a set of subsets of the set $N := \{1, 2, \ldots, n\}$ satisfying the *matroid base properties*:

(**B1**) $\mathcal{B} \neq \emptyset$.

(**B2**) If $B_1, B_2 \in \mathcal{B}$ and $e_1 \in B_1 \backslash B_2$, then $\exists\ e_2 \in B_2 \backslash B_1$, such that $(B_1 \backslash e_1) \cup \{e_2\} \in \mathcal{B}$.

The system $(N, \mathcal{B})$ is a *matroid* with *ground set* $N$ and *set of bases* $\mathcal{B}$. Matroids are basic objects in combinatorial optimization, usually linked with linear objective functions. Without going into details concerning matroids (see [19,20,23]), we note that the basis-exchange property **(B2)** implies that all $B \in \mathcal{B}$ have the same number of elements, called the *rank* of the matroid, which we denote throughout this paper as $m$. A matroid is *vectorial* (also called *representable* or *matric*) if there is an $m$-row matrix $A$, with columns indexed by the ground set $N$ such that elements of $\mathcal{B}$ are in one-to-one correspondence with sets of columns of $A$ indexing $m \times m$ non-singular submatrices of $A$. Though the field used for the entries in $A$ can be arbitrary, for the purpose of the algorithm that we describe and our implementation, we restrict our attention to the rationals.

A main motivation for considering the problem at hand is multi-criteria optimization, where the function $f$ balances $d$ different linear functions specified by the rows of $W$. As we assume that the trade-off function $f$ is completely general, our goal is to efficiently enumerate the possible values of $W$-images $W(B) \in \mathbb{R}^d$, *without* enumerating all $B \in \mathcal{B}$. Then a decision maker can compare the different achievable $d$ vectors and choose the best according to their trade-off function $f$. In this spirit, we further assume that $d$ is small (e.g., $d = 2, 3, 4$ are already interesting) and that the entries in $W$ are also not too large. These are technical assumptions used in [6] to establish polynomial-time complexity.

*Example 1 Multi-criteria spanning-tree optimization.* Consider a connected graph $\mathcal{G}$ on $n$ edges. Let $\mathcal{B}$ be the set of edge-sets of spanning trees of $\mathcal{G}$. Then $\mathcal{B}$ is the set of bases of a (graphic, hence vectorial) matroid. We can imagine several linear criteria on the edges of $\mathcal{G}$. For example,

(1) the first row of $W$ may encode the *fixed installation cost* of each edge of $\mathcal{G}$;
(2) the second row of $W$ may encode the *monthly operating cost* of each edge of $\mathcal{G}$;
(3) assuming that each edge $j$ fails independently with probability $1 - p_j$, then by having the values $\log p_j$ as the third row of $W$ (scaled and rounded suitably), $\sum_{j \in B} \log p_j$ captures the *reliability* of the spanning tree $B$.

Clearly it can be difficult for a decision maker to balance these three competing objectives. There are many issues to consider, such as the time horizon, repairability, fault tolerance, etc. These issues can be built into a concrete $f$, for example a weighted norm, or can be simply thought of as determining a black-box $f$.

Note that a complete graph on $m + 1$ vertices has $n := m(m + 1)/2$ edges but $(m + 1)^{m-1}$ spanning trees. If $W$ has non-negative integer entries bounded by $\omega$ and we have $d$ criteria, then $W(B)$ lies in the set of lattice points $\{0, 1, \ldots, m\omega\}^d$. So, the number of possible $W(B)$ to enumerate is at most $(m\omega + 1)^d$, while brute-force enumeration would require looking at all $(m + 1)^{m-1}$ spanning trees. So that is our goal: to cleverly enumerate these $(m\omega + 1)^d$ points, checking somehow which of these really arise from a spanning tree, without directly enumerating the much larger number of $(m + 1)^{m-1}$ spanning trees. □

*Example 2 Minimum aberration model fitting.* The 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 [15,29]); for example, a simple polynomial model is typically defined to be one where the chosen monomials have low degrees. In particular, we may choose to minimize the norm of the total-degree vector, computed over monomials in the model.

In one application, 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.

We are given an $m \times d$ *design matrix* $P$ of floating point numbers and an $d \times n$ *monomial degree matrix* $W$ (of small non-negative integers). Connecting this with the description above:

$d$ = 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 column $W_{\cdot,i}$ of $W$ specifies the degrees of each factor in the monomial corresponding to that column. That is, $W_{\cdot,i}$ describes the monomial $\prod_{k=1}^{d} x_k^{W_{k,i}}$.

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

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

A polynomial model

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

is determined by a set $B \subset \{1, 2, \ldots, n\}$ indexing some of the monomials. Let $A_B$ be the submatrix of $A$ comprising its columns with indices from $B$. The set $B$ is *identifiable* if $A_B z = y$ has a unique solution for all ("response vectors") $y$ (see [24], for example). This is exactly the condition that is needed for there to be a unique choice

of the $c_i$ above so that $\pi(p_i.) = y_i$, for $i = 1, \ldots, m$, for every possible $y$. Typically $A$ has full row rank (and we do assume this for convenience), so $B$ is identifiable if and only if $A_B$ is square $(m \times m)$ and $\det(A_B) \neq 0$. Importantly, the set of identifiable $B$ is the set of bases of a (vectorial) matroid.

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

$$x(B) := \sum_{i \in B} W_{\cdot,i} \in \mathbb{Z}_+^d.$$

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 which we use in our computational experiments. For a positive integer $\omega$, we look at the matrix $W$ having all possible columns of non-negative integers satisfying $W_{k,i} \leq \omega$. Such a matrix $W$ has $n = (\omega + 1)^d$ columns. 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 $m = 28$ (design points), then the number of potential identifiable models is $\sim 5.0 \times 10^{24}$ while the number of potential total-degree vectors is only 64, 009 (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 $\sim 5.0 \times 10^{24}$ potential identifiable models. In this way, we can optimize any aberration function over these at most 64, 009 total-degree vectors of identifiable models. Note that we have in fact carried this out successfully (see Sect. 9). □

The authors of [6] developed an algorithm for $\mathcal{P}$ based on matrix methods. That algorithm was designed to be theoretically efficient, in the worst-case sense, when the number of criteria $d$ is fixed. Indeed, the problem is provably intractable when $d$ is not fixed. The dependence of their algorithm on $d$ is exponential, so it can only be practical for very small values, but this is often the case in applications.

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 criteria $d$. Moreover, the base form of 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 ill-conditioned, and the state-of-the-art for their solution on a serial machine in limited precision floating point is well below our needs; indeed, solution of even an order-30 Vandermonde system is a challenge, while we are considering systems that are 3–4 orders of magnitude larger.

Finally, we give one more example. The example does not strictly fit our framework. Rather it fits in the broader framework of (vectorial) *matroid intersection*. Such problems greatly broaden the scope of applications, including statics, electrical networks, etc. (see [20,25]). Nonetheless, a similar algorithm to the one we describe exists (see [7]), and we plan to eventually implement it.

*Example 3 Multi-criteria assignment problem.* We are given $m$ jobs that should be assigned to $m$ processors. Feasible assignments are encoded as the perfect matchings in a bipartite graph. As is well known, the perfect matchings of a bipartite graph can be viewed as the *intersection* of the sets of bases of a *pair* of (graphic, hence vectorial) matroids.

## 3 The algorithm

We will not give a detailed justification for the basic algorithm in this paper—this can be found in [6]—but we do give all details to fully specify it.

Let $\omega := \max W_{h,j}$ (the maximum weight), let $s-1 := m\omega$ (the maximum possible entry in the $W$-image of a base), and let $Z := \{0, 1, \ldots, m\omega\}^d$.

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

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

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: $\sum_{u \in Z} t^{\sum_{k=1}^{d} u_k s^{k-1}} g_u = \det(AX(t)A^T), \quad t = 1, 2, \ldots, s^d$ ;

**Algorithm 1**: The interpolation algorithm

For each potential $W$-image $u$, we have $g_u \geq 0$. In fact, $g_u = \sum \det^2(A_S)$, where the sum is over the bases $S$ having $W$-image $u$ (see [6]). Therefore, $g_u > 0$ if and only

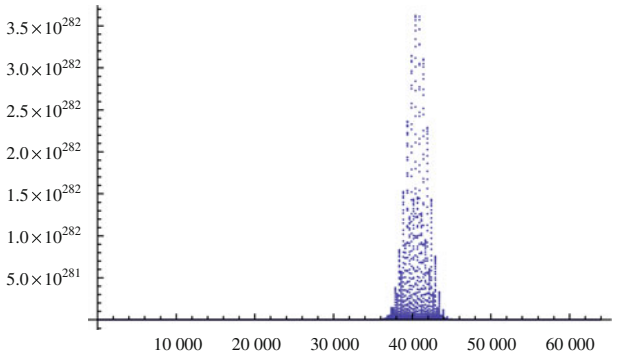


**Fig. 1** Solution values

if $u$ is the $W$-image of a base $S$. The trick is to compute each $g_u$ without explicitly carrying out the 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 *nonlinear combinatorial optimization problem* is just

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

and for any given $f : \mathbb{Z}^d :\to \mathbb{R}$, 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 $W$-images of the bases.

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. We fabricated our own utilities, employing ARPREC (a C++/Fortran-90 arbitrary precision package; see [3]) in concert with MPI. We note that working in very high precision is a growing trend in scientific computation (see [4]).

In Fig. 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 $u \in Z := \{0, 1, \ldots, mw\}^d$, 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 $u = (0, 0, \ldots, 0)$ gets numbered 1, and the point $u = (mw, mw, \ldots, mw)$ gets numbered by $(mw + 1)^d = s^d$. So, in Fig. 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 this fact and show how it can be exploited 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 [17, Chap. 22], for example, which contains an excellent survey of numerical techniques for solving Vandermonde systems). 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, we selected our particular Vandermonde system to have this relatively simple form because it enables an appealing parallelization. Several extensions and enhancements to the basic 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 in the typical sense—we have a particular polynomial implicitly defined, and we are seeking an efficient manner to determine its nonzero coefficients.

## 4 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 [13,14], 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 [26]). 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 [28]), and a C code to calculate Stirling numbers of the first kind is available as mStirling.c (see [21]). Of course, we could not work with the Stirling numbers in double precision or long ints—even for the smallest problem that we consider ($N = 1{,}369$), we encounter Stirling numbers of magnitude around $10^{3{,}700}$.

## 5 Limiting the range

From a completely naïve standpoint, it might appear that every variable $g_u$, $u \in Z := \{0, 1, \ldots, m\omega\}^d$, could be positive (i.e., nonzero) in the solution. But the plot of Fig. 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 $1, 2, \ldots, (m\omega + 1)^d = s^d$, according to the map $n(u) := 1 + \sum_{k=1}^{d} u_k s^{k-1}$, for $u \in Z$.

Notice for example that $u = (0, 0, \ldots, 0)$ and $u = (m\omega, m\omega, \ldots, m\omega)$ are typically not achievable by adding up $m > 1$ distinct columns of $W$ (after all, $W$ itself often has distinct columns). 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

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

where $c := (s^0, s^1, \ldots, s^{d-1})$, 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-base optimization problem. As such, it is exactly solvable by, for example, the well-known *greedy algorithm* (see [19] for example). We simply select variables to include into the solution, in a greedy manner, starting from the minimum objective-coefficient value $(cW)_i$, working up through the greater values. We only include the $i$th column $A_{\cdot,i}$ of $A$ in the solution (i.e., set $y_i = 1$) if the columns from $A$ already selected, together with $A_{\cdot,i}$, 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 $(cW)_i$, working *down* through the *lesser* values.

Note that this can be done with no numerical difficulties. We only use the objective vector $cW$ 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 $cW$; we just observe that this vector lexically orders the columns of $W$, 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 $W_{\cdot,i}$ is lexically less (respectively, greater) than $W_{\cdot,i'}$.

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 $I_{\min}$.

We can do better in regard to exploiting the calculation of $I_{\min}$ and $I_{\max}$. We can reduce our matrix work to an equivalent dual Vandermonde system, albeit with a relatively larger right-hand side, for which $N$ is only $\tilde{N} := I_{\max} - I_{\min} + 1$. We simply view our dual Vandermonde system $V^T g = b$ in block form:

$$\left[\begin{array}{c|c|c} \diagdown & \tilde{V}^T & \diagdown \\ \hline & \diagdown & \end{array}\right] \left[\begin{array}{c} \mathbf{0} \\ \hline \tilde{g} \\ \hline \mathbf{0} \end{array}\right] = \left[\begin{array}{c} \tilde{b} \\ \hline \diagdown \end{array}\right],$$

and in this form the non-zero part of $g$, namely $\tilde{g}$, satisfies $\tilde{V}^T \tilde{g} = \tilde{b}$. This system has order $\tilde{N}$. Moreover, it is very close to being in the form of the special dual Vandermonde systems that we have been considering.

We can see that $V^T_{i,j} = i^{j-1}$, for $1 \leq i, j \leq N$, implies that

$$\tilde{V}^T_{i,j} = V^T_{i,j+I_{\min}-1} = i^{j+I_{\min}-2}.$$

Let

$$D := \operatorname{diag}_{j=1,\ldots,\tilde{N}}\left(j^{-I_{\min}+1}\right).$$

Then

$$\left(D\tilde{V}^T\right)_{i,j} = i^{j-1}, \quad \text{for } 1 \leq i, j \leq \tilde{N}.$$

That is, $D\tilde{V}^T$ is an order-$\tilde{N}$ transposed Vandermonde matrix of the type that we have been looking at, and so the system that we now need to solve, namely

$$\left(D\tilde{V}^T\right)\tilde{g} = \left(D\tilde{b}\right),$$

is just an order-$\tilde{N}$ version of what we have been looking at. The only difference is that we scale the right-hand side elements appropriately.

Note that because $D\tilde{V}^T$ is a transposed Vandermonde matrix, the associated system has a unique solution, and so the choice of which $\tilde{N}$ rows we work with (we chose the first $\tilde{N}$ rows) is irrelevant. However, by working with the first $\tilde{N}$ rows, $D\tilde{V}^T$ is a transposed Vandermonde matrix of the special form that we have been considering (i.e., the columns are the successive powers of the natural numbers $1, \ldots, \tilde{N}$), and

so we have the nice form and methodology for working with its inverse that we have discussed.

## 6 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 (Sect. 3), 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.

## 7 Parallel implementation

Now we will consider how the algorithm described above is currently implemented as it targets the IBM Shaheen Blue Gene/P supercomputer at IBM's T.J. Watson Research Center. The current 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. Here, let us consider the most straightforward implementation and motivate some of our implementation decisions.

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 factorization and backsolve, but an explicit $V^{-T} b$ calculation; the distinction will be an important one.

In Sect. 6, the mathematical definition of the right-hand side components was given. From the point of view of a parallel implementation, generating them 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 number. For example, if $s^d$ is 10,000 and there are 1,000 processors, $X(1)$ might be generated on processor #1, along with $X(1001)$, $X(2001)$, 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 $\det(AX(t)A^T)$.

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 $V^{-T}$. At the end of this embarrassingly parallel phase, the right-hand side is formed, distributed (and nowhere duplicated) over the entire processing grid. Thus, we form the contributions to the matrix vector product without any further communication of the components of the right-hand side.

The next step is 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 $\left[\begin{matrix} N+1 \\ j+1 \end{matrix}\right]$ 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 $\left[\begin{matrix} r \\ p \end{matrix}\right]$, having received $\left[\begin{matrix} r-1 \\ p-1 \end{matrix}\right]$ from processor $p-1$ and having sent $\left[\begin{matrix} r-1 \\ p \end{matrix}\right]$ 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 which is only slightly complicated by the fact that we have to consider the instance when the cardinality of the Stirling numbers exceed the number of processors.

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 naïve 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 and corresponding precision required increases. During the Stirling computation, which can be performed even before the formation of the right-hand sides (or off-line and loaded from disk), the only memory required is that which holds the Stirling vector. We will address this issue in Sect. 7.2 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 Sect. 4) $V_{i,N}^{-1}$ 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 reduction

(add) operation yields the overall solution vector. While we do not address the use of multiple right-hand-sides in any depth in this paper, it should be apparent to the reader that the extension is straightforward, if slightly more memory-hungry, and we present a topical treatment of the issues involved in that domain.

## 7.1 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 $9 \times 9$ matrix solve was not possible using LAPACK and standard double-precision arithmetic. We considered using iterative refinement methods [8] and, perhaps quad-precision arithmetic, but the $25 \times 25$ 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. Since we began this project, there have been a number of studies regarding the possibility of bringing ARPREC-style functionality to LAPACK/ScaLAPACK (see [12] for instance). However, at the limits of memory consumption, it seems that it would not be easy to implement our methodology in the ScaLAPACK framework.

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.

### *7.1.1 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 integer entries, 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 $25 \times 25$ 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 $25 \times 25$ example works with the "scaling" option (see last line of Sect. 6) 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

**Table 1** Intractable numerics

| $N$ | Prec | Scaling on | Sample zero | Scaling off | Sample zero |
|---|---|---|---|---|---|
| 25 | 10 | n | n/a | n | n/a |
| 25 | 15 | n | n/a | n | n/a |
| 25 | 16 | y | $-2.05918 \times 10^{-18}$ | n | n/a |
| 25 | 25 | y | $-2.05918 \times 10^{-18}$ | n | n/a |
| 25 | 30 | y | $-2.05918 \times 10^{-18}$ | y | $3.16422 \times 10^{-12}$ |
| 32 | 15 | n | n/a | n | n/a |
| 32 | 16 | y | $-3.47723 \times 10^{-14}$ | n | n/a |
| 32 | 30 | y | $-5.398 \times 10^{-27}$ | y | $-6.71336 \times 10^{-12}$ |

**Table 2** Precision required to obtain a largest negative value less than 0.001 in magnitude

| $N$ | Precision | Sample zero | LSF (N, Precision) |
|---|---|---|---|
| 361 | 319 | $-1.70998 \times 10^{-33}$ | 0.883657 |
| 1,369 | 1,215 | $-2.8173 \times 10^{-6}$ | 0.887259 |
| 2,116 | 1,894 | $-1.28999 \times 10^{-11}$ | 0.892665 |
| 3,025 | 2,704 | $-1.00963 \times 10^{-4}$ | 0.893379 |
| 4,096 | 3,672 | $-2.52884 \times 10^{-9}$ | 0.894986 |
| 5,329 | 4,755 | $-4.66493 \times 10^{-6}$ | 0.893726 |
| 6,724 | 5,882 | $-1.03724 \times 10^{-13}$ | 0.885654 |
| 8,281 | 7,255 | $-1.04402 \times 10^{-11}$ | 0.881897 |

which point it became exactly 1. As mentioned before, all other solutions contained exact integers.

In general, it has been our observation that the number of digits of precision required for an acceptable answer has a linear relationship with the dimension of the matrix being implicitly inverted. We have come to this conclusion experimentally (see Table 2) where the least squares fit in the fourth column shows a great deal of stability as larger values of $N$ are evaluated. Because the solution of our Vandermonde system is non-negative (see the discussion after Algorithm 3), any solution that has negative entries clearly indicates numerical difficulties and a need to increase the precision. Furthermore, we have cross-validated our results against a completely different effective heuristic approach to the basic multi-objective nonlinear combinatorial optimization problem (see [11]), so we have further evidence to check that we have used sufficient precision. Probably the numbers of digits that we find suffice is well below what theoretical estimates would require. We leave a detailed analytical error analysis to others as future work.

### 7.2 Implementation challenges and solutions

While the implementation outlined in Sect. 7 allows us to handle matrices of unprecedented size (in this sub-field), there are some limitations to that approach. 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.

Recall that there are five phases to this computation.

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

While some of these steps can be interchanged, we will address them in the indicated order.

### *7.2.1 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 confront the scaling challenge, we need to consider issues related to space requirements. As the problem size grows, a single processor's memory will prove insufficient to the task of storing the matrices required for the creation of the right-hand sides. As that occurs, it is possible to store these matrices across a $2 \times 2$ subgrid of processors in a (small) block-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 $4 \times 4$ grids at a slight cost in generation time. After the right-hand sides are generated, these matrices are no longer required and the memory can be reclaimed.

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 [10,27], and the underlying data distribution (block-cyclic) allows us to deal with memory limitations and load imbalances 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.

### *7.2.2 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 formula in Sect. 4 makes it apparent that only two values are needed to compute the "next" Stirling number and we will leverage this. As we describe the later steps, we will address how these values need to be communicated across the machine.

### 7.3 Computing $V^{-T}b$: revisited (steps 3–5)

As we determine how to approach truly large problems in this domain, we must consider different data distributions and methods for staging the computations. In this section, we will list the alternatives that are typically used in traditional high-performance computing and give a topical treatment of some of the problems with these approaches. In the next section we will give a more detailed explanation of the approach selected. A somewhat similar design space was examined in-depth by Bailey and Borwein [5] in the context of high-precision numerical integration.

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: Staggered.
(3) Two-dimensional (blocked) distribution: "Row-twisted." Unstaggered.

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 an impractical amount of storage space.

This difficulty may not seem amenable to any solutions involving data distribution. The problem is that no processor has enough memory to hold its own contribution to the global 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, 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. Assume that $N = P \times Q$. In the $P \times Q$ processor configuration, every processor in a given column of the processor mesh must hold not $N$ elements of a given column, but $N/P$ ($R$) elements of that column and is responsible for computing $Q$ (sub-)columns. The distinction is important because in this instance the $Q$ sub-columns do not have to co-reside in memory. 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 $R$ elements of the first column (in their respective solution spaces) before the second row of processors is active, etc.

### 7.4 (Further) Conflating procedural steps

When we interleave steps (3–5), we will see that the problems associated with memory issues, mentioned above, largely disappear. Here we describe how these issues are dealt with, additional penalties incurred (if any), 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/P's fast collective networks). Fortunately, in this instance, the solution is not overly complex. The data distribution is simple, the added coordination cost is not onerous, and the solution scales (both up and down) quite well. Memory parsimony can be traded for algorithmic efficiency in a straightforward manner.

At this point, let us take a half-step towards the interleaving more fully explored in the next subsection. Once we compute a partial column (R 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. $R$ elements of the $V^{-T}$ matrix,
2. $R$ elements of the Stirling vector,
3. $R$ elements of the local contribution to the right-hand side, and
4. At least one element of the overall (reduced) solution

As shown in Sect. 4, 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. It is this property of the Vandermonde system that makes both the one-dimensional and the "block-skewed" algorithms possibilities, as, if the entire column were required to compute the next value of the inverse, only the two-dimensional approach (or a one-dimensional approach where a single *column* resides on the processor set) would be practical.

The astute reader will no doubt have noticed that the staggered, "block-skewed" algorithm, has no real advantage over a one-dimensional solution, save for purposes of exposition. In fact, the two-dimensional distribution requires one additional communication to one's neighbor (so that the neighbor can continue computation on the column of $V^{-T}$) per $R$ computations. Thus, the simplest approach may well be the best here. One simply views the 2D mesh as one dimensional, and whereas, previously, processor #1 would have had the first $R$ elements for matrix column #1, the second $R$ elements of matrix column #2, etc., we can simplify things. In the one-dimensional setting, one can compute in $R$ blocks and save the value needed to continue with the next $R$ elements while performing a distributed reduce, always working on the same column of the matrix. Thus, we will restrict our focus to the one-dimensional scheme wherein each processor is responsible for a single column in the motivating example. As has been mentioned, column formation is independent and having a single processor compute several columns presents no additional challenges. This also adds flexibility in choosing $R$, the "chunk size". In the one-dimensional setting, the previous definition of $R$ ($=P/Q$) is no longer motivated (in the two-dimensional setting, it was a natural level of granularity), and we can choose $R$ so as to optimize the efficiency of the algorithm (as large as possible, without overflowing available memory).

The apparent problem is that one cannot duplicate the Stirling vector on all processors (it is prohibitively large). The simplest (but quite inefficient) solution to this problem is to stagger the computations (intersperse Stirling computation with inversion). However, because every processor requires the Stirling number $\left[ \begin{matrix} N \\ j \end{matrix} \right]$ to compute the $N$th entry in the column, one could compute the entire Stirling vector, leave the 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 and a similar shift operation for the distributed summation of the matrix–vector product. 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 on the Blue Gene/P system. The punctuated broadcast having the advantage of achieving greater bandwidth (link utilization) and avoiding the start-up latency associated with the systolic shift (the calculation cannot begin until the first Stirling value arrives, requiring a number of shifts equal to the processor count in the worst case).

When considering the 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. This value does not have to be the same as the number of contributions to the matrix–vector product that are collected before a distributed reduce; the two can be adjusted independently. This allows an easily tuned trade between communication efficiency and memory consumed.

## 8 Extreme scale

Unsurprisingly, as we consider what is required 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 (wherein each processor can only hold the seven requisite values: the three values to compute the next element of the inverse, a Stirling value for global consumption, a single entry of the reduced matrix–vector product, the element of the right-hand-side used by this column of the matrix to produce the matrix–vector product, and the current contribution to the matrix–vector product), the operation is appreciably more expensive in terms of communications overhead incurred.

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. We are examining how this might be used for other applications in this space. We have also designed a simple checkpointing infrastructure for the high-precision values so as to facilitate solution of a very large problem on limited resources. Because the execution of this algorithm might take several days on a single rack (1,024 nodes) of Blue Gene/P, 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 faults in the same manner (where more frequent saves to non-volatile memory would be needed). If one carefully times when the checkpointing occurs (for example, immediately after a reduction), very little data per processor needs to be stored.

### 8.1 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, while we can store them in a distributed fashion, we have to hold them all in the system simultaneously (else we must recompute the columns of $V^{-T}$). The problem here is not only the right-hand sides, but the buffers for the summand $V^{-T}$ components which are each $R$ elements in size. While it is straightforward to shrink them (reduce $R$), this will cause inefficiency in the communications of the reduction (as soon as the size of that array drops below the number of processors). The simplest solution, one that costs little in terms of performance, is to simply reduce one right-hand side at a time and reuse the storage. This, of course, requires us to store the column of $V^{-T}$ as well as the vector to be reduced instead of reusing this storage space. Still, because this cost is only twice the storage for one component, no matter how many right-hand-sides are involved, the solution is practical for some problem instances.

## 9 Experiments

The Blue Gene/P supercomputer [18] is the second-generation incarnation of the Blue Gene computer series. Several papers detailing the architecture have been published (for example [1,2,9]). Only a few of the many features of this architecture are central to our algorithmic embodiment however. Blue Gene/P contains, among other networks, a three-dimensional torus interconnect. Each link is capable of sending or receiving data at 0.44 Bytes/cycle per link (thus, higher-dimensional broadcasts and one-to-many communications 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. 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 Blue Gene/P system that we used had 4 GB of memory shared between four SIMD-FPU-enhanced 450 PPC cores. While these cores each run at 850 MHz, they are capable of SIMD FMAs (Floating point Multiply-Adds), so that each is in fact able to reach 3.4 GF/core or 13.6 GF/node.

The largest problem that we computed with has $N = s^d = 64{,}009$; a more detailed description of that instance is in Example 2 of Sect. 2. We were able to solve this instance in about 15 h using just 2 Blue Gene/P racks (2,048 nodes). We estimate that brute-force enumeration of all potential bases, together with checking which are in fact bases, would require over 75,000 years using petaflop-class machines such as the ORNL-Jaguar and LANL-Roadrunner supercomputers.

We present additional computational results in Tables 3, 4. Table 3 indicates that we can scale up to large problems rather efficiently. Table 4 indicates that we do not lose substantial machine utilization as we go to higher precisions.

Furthermore, we solved a $N \approx 32$K problem on 8 Blue Gene/P racks (8,192 nodes), using a version of the code that set the collection vector sizes quite conservatively (which degrades performance slightly) and achieved 5.3 TF. Stepping to an order-280K matrix on a 1.0-PF 72-rack BG/P system, should yield 45–55 TF. (Note that such a problem corresponds, for example, to a an instance of Example 2 having $d = 2, \omega = 9, n = 100$ and $m = 59$).

**Table 3** Performance on 4,096 cores of the Blue Gene/P Supercomputer for various matrix sizes at a fixed level of precision

| $d$ | $\omega$ | $n$ | $m$ | $\binom{n}{m}$ | $N$ | Prec | Time |
|---|---|---|---|---|---|---|---|
| 2 | 9 | 100 | 4 | $3.92123 \times 10^6$ | 1,369 | 32,000 | 106.239 |
| 2 | 9 | 100 | 5 | $7.52875 \times 10^7$ | 2,116 | 32,000 | 139.232 |
| 2 | 9 | 100 | 6 | $1.19205 \times 10^9$ | 3,025 | 32,000 | 178.511 |
| 2 | 9 | 100 | 7 | $1.60076 \times 10^{10}$ | 4,096 | 32,000 | 224.556 |
| 2 | 9 | 100 | 8 | $1.86088 \times 10^{11}$ | 5,329 | 32,000 | 550.641 |
| 2 | 9 | 100 | 9 | $1.90223 \times 10^{12}$ | 6,724 | 32,000 | 673.805 |
| 2 | 9 | 100 | 10 | $1.73103 \times 10^{13}$ | 8,281 | 32,000 | 1203.76 |
| 2 | 9 | 100 | 11 | $1.41630 \times 10^{14}$ | 10,000 | 32,000 | 1446.43 |
| 2 | 9 | 100 | 12 | $1.05042 \times 10^{15}$ | 11,881 | 32,000 | 1690.77 |
| 2 | 9 | 100 | 13 | $7.11054 \times 10^{15}$ | 13,924 | 32,000 | 2628.13 |
| 2 | 9 | 100 | 14 | $4.41869 \times 10^{16}$ | 16,129 | 32,000 | 3021.44 |

Time to solution is essentially a constant $\times$ the number of columns solved $\times$ the length of the columns

**Table 4** Performance on 4,096 cores of the Blue Gene/P Supercomputer for a fixed matrix size at different levels of precision

The percentage of peak achieved does not remain constant, but the variation appears to dampen at very high precision levels

| $N$ | Prec | % Peak | GF | Time (s) |
|---|---|---|---|---|
| 4,096 | 2,000 | 4.262 | 593.477 | 12.171 |
| 4,096 | 4,000 | 4.849 | 675.270 | 26.102 |
| 4,096 | 8,000 | 5.071 | 706.846 | 52.043 |
| 4096 | 16,000 | 5.361 | 746.635 | 100.702 |
| 4,096 | 32,000 | 5.037 | 701.416 | 225.110 |
| 4,096 | 64,000 | 4.904 | 682.891 | 489.081 |
| 4,096 | 128,000 | 5.153 | 717.579 | 972.900 |

## 10 Conclusions

We have demonstrated that efficient high-performance linear-algebra based algorithms, implemented on high-performance supercomputers, can be successfully applied to a domain (nonlinear combinatorial optimization) where such algorithms looked to be completely impractical. We hoped that this is just a first step in seeing more impact of matrix methods and supercomputing in discrete optimization.

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** We gratefully acknowledge the use of the IBM Shaheen (which at the time of our experiments was an 8-rack Blue Gene/P supercomputer housed at the IBM T.J. Watson Research Center). The IBM Shaheen is now owned and operated by the King Abdullah University of Science and Technology (KAUST). We would like to thank Bob Walkup at IBM Research for his help in many aspects of this work, including the use of his performance-counter library for performance evaluation and Fred Mintzer at IBM

Research for arranging for our use of the Blue Gene/P Supercomputer. We are deeply indebted to David Bailey and his team for the ARPREC software package and documentation. Our work was partially carried out, while S. Margulies was a graduate student at U.C. Davis, under an Open Collaborative Research agreement between IBM and U.C. Davis.

## References

1. Alam, S., Barrett, R., Bast, M., Fahey, M.R., Kuehn, J., McCurdy, C., Rogers, J., Roth, P., Sankaran, R., Vetter, J.S., Worley, P., Yu, W.: Early evaluation of IBM BlueGene/P, SC '08. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp 1–12 (2008)
2. 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. J. Res. Dev. **49**(2–3), 393–406 (2005)
3. ARPREC. http://crd.lbl.gov/~dhbailey/mpdist/mpdist.html
4. Bailey, D.H.: High-precision arithmetic in scientific computation. Comput. Sci. Eng. **7**(3), 54–61 (2005)
5. Bailey, D.H., Borwein, J.M.: Highly Parallel, High-Precision Numerical Integration. LBNL-57491 (2005)
6. Berstein, Y., Lee, J., Maruri-Aguilar, H., Onn, S., Riccomagno, E., Weismantel, R., Wynn, H.: Nonlinear matroid optimization and experimental design. SIAM J. Discret. Math. **22**(3), 901–919 (2008)
7. Berstein, Y., Lee, J., Onn, S., Weismantel, R.: Parametric nonlinear discrete optimization over well-described sets and matroid intersections. Math. Program. (to appear)
8. Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., Kurzak, J.: Mixed precision iterative refinement techniques for the solution of dense linear systems. Int. J. High. Perform. Comput. Appl. **21**(4), 457–466 (2007)
9. Chiu, G.L.-T., Gupta, M., Royyuru, A.K. (guest eds.): Blue Gene. IBM J. Res. Dev. **49**(2/3) (2005)
10. Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., Whaley, C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Sci. Program. **5**(3), 173–184 (1996)
11. De Loera, J., Haws, D.C., Lee, J., O'Hair, A.: Computation in multicriteria matroid optimization. ACM J. Exp. Algorithmics **14**, Article No. 8, (2009)
12. Demmel, J.: The future of LAPACK and ScaLAPACK, PARA 06: workshop on State-of-the-Art. In: Scientific and Parallel Computing, Umea, Sweden, 18–21 June 2006. http://www.cs.berkeley.edu/~demmel/cs267_Spr07/future_sca-lapack_CS267_Spr07.ppt
13. Eisinberg, A., Fedele, G., Imbrogno, C.: Vandermonde systems on equidistant nodes in [0, 1]: accurate computation. Appl. Math. Comput. **172**, 971–984 (2006)
14. Eisinberg, A., Franzé, G., Pugliese, P.: Vandermonde matrices on integer nodes. Numerische Mathematik **80**(1), 75–85, (1998)
15. Fries, A., Hunter, W.G.: Minimum aberration $2^{k-p}$ designs. Technometrics **22**, 601–608 (1980)
16. Harvey, N.: Algebraic algorithms for matching and matroid problems. SIAM J. Comput. (2010, to appear)
17. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2002)
18. IBM Blue Gene Team: Overview of the IBM Blue Gene/P project, IBM J. Res. Dev., v52, (2008), 199–220
19. Lee, J.: A First Course in Combinatorial Optimization. Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge (2004)
20. Lee, J., Ryan, J.: Matroid applications and algorithms. INFORMS (Formerly ORSA) J. Comput. **4**, 70–98 (1992)
21. mStirling.c. http://ftp.bioinformatics.org/pub/pgetoolbox/addins/src/mStirling.c
22. Mulmuley, K., Vazirani, U.V., Vazirani, V.V.: Matching is as easy as matrix inversion. Combinatorica **7**, 105–113 (1987)
23. Oxley, J.G.: Matroid Theory. Oxford Science Publications, The Clarendon Press, Oxford University Press, New York (1992)
24. Pistone, G., Riccomagno, E., Wynn, H.P.: Algebraic Statistics. Monographs on Statistics and Applied Probability, vol. 89. Chapman & Hall/CRC, Boca Raton (2001)

25. Recski, A.: Matroid theory and its applications in electric network theory and in statics. Springer, Berlin (1989)
26. Tweddle, I.: James Stirling's methodus differentialis: an annotated translation of Stirling's text. In: Sources and Studies in the History of Mathematics and Physical Sciences, Springer (2003)
27. van de Geijn, R.A., Watts, J.: SUMMA: scalable universal matrix multiplication algorithm. Concurr. Pract. Experience **9**(4), 255–274 (1997)
28. VanInverse.m. http://www.mathworks.com/matlabcentral/files/8048/VanInverse.m
29. Wu, H., Wu, C.F.J.: Clear two-factor interactions and minimum aberration. Ann. Stat. **30**, 1496–1511 (2002)