Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling

Jack Dongarra* and Piotr Luszczek§
*Oak Ridge National Laboratory †Rice University ‡University of Manchester §University of Tennessee

***Abstract*—We present a modelling framework to accurately predict time to run dense linear algebra calculation. We report the framework's accuracy in a number of varied computational environments such as shared memory multicore systems, clusters, and large supercomputing installations with tens of thousands of cores. We also test the accuracy for various algorithms, each of which having a different scaling properties and tolerance to low-bandwidth/high-latency interconnects. The predictive accuracy is very good and on the order of measurement accuracy which makes the method suitable for both dedicated and non-dedicated environments. We also present a practical application of our model to reduce the time required to tune and optimize large parallel runs whose time is dominated by linear algebra computations. We show practical examples of how to apply the methodology to avoid common pitfalls and reduce the influence of measurement errors and the inherent performance variability.**

***Index Terms*—Linear systems, parallel algorithms, modelling techniques**

I. Introduction

Dense systems of linear equations are found in numerous applications, including:

- airplane wing design;
- radar cross-section studies;
- flow around ships and other off-shore constructions;
- diffusion of solid bodies in a liquid;
- noise reduction; and
- diffusion of light through small particles.

The electromagnetics community is a major user of dense linear systems solvers. Of particular interest to this community is the solution of the so-called radar cross-section problem. In this problem, a signal of fixed frequency bounces off an object; the goal is to determine the intensity of the reflected signal in all possible directions. The underlying differential equation may vary, depending on the specific problem. In the design of stealth aircraft, the principal equation is the Helmholtz equation. To solve this equation, researchers use the method of moments [18], [31]. In the case of fluid flow, the problem often involves solving the Laplace or Poisson equation. Here, the boundary integral solution is known as the panel methods [19], [20], so named from the quadrilaterals that discretize and approximate a structure such as an airplane. Generally, these methods are called boundary element methods. Use of these methods produces a dense linear system of size $O(N)$ by $O(N)$, where N is the number of boundary points (or panels) being used. It is not unusual to see size $3N$ by $3N$, because of three physical quantities of interest at every boundary element. A typical approach to solving such systems is to use LU factorization. Each entry of the matrix is computed as an interaction of two boundary elements. Often, many integrals must be computed. In many instances, the time required to compute the matrix is considerably larger than the time for solution. The builders of stealth technology who are interested in radar cross-sections are using direct Gaussian elimination methods for solving dense linear systems. These systems are always symmetric and complex, but not Hermitian.

A major source of large dense linear systems is problems involving the solution of boundary integral equations [14]. These are integral equations defined on the boundary of a region of interest. All examples of practical interest compute some intermediate quantity on a two-dimensional boundary and then use this information to compute the final desired quantity in three-dimensional space. The price one pays for replacing three dimensions with two is that what started as a sparse problem in $O(n^3)$ variables is replaced by a dense problem in $O(n^2)$.

A recent example of the use of dense linear algebra at a very large scale is physics plasma calculation in double-precision complex arithmetic based on Helmholtz equations [2].

Finally, virtually all large supercomputer sites do run the High Performance LINPACK (HPL) benchmark [13] which is primarily based on dense linear algebra. The reduction of time to run the benchmark is of paramount importance. The first machine on the 34th TOP500 list took over 20 hours to complete the HPL run [17]. And this is only a single run not counting the time spent in optimizing the parameters for the run. This puts a tremendous stress on every component of the machine and brings to the fore MTBF and MTBI metrics. Addition, the power use during the run is very high due to the very high utilization of all the cores of the system. And in the end, only a single performance number is reported without any public account of the days worth of computing devoted to tuning.

II. Performance Prediction by Correlation

One of the building blocks of dense linear algebra solvers and, by far, the main source if their high performance is a

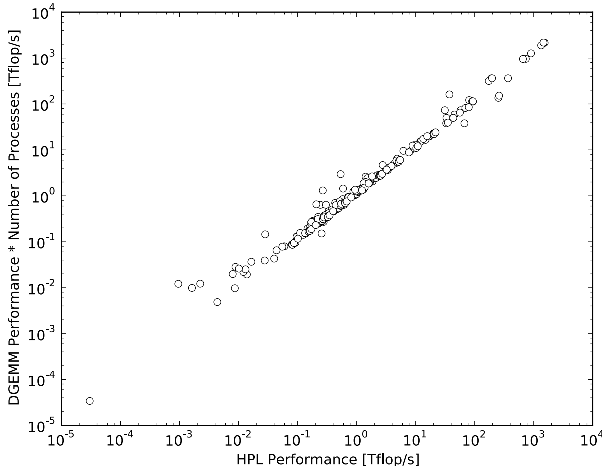

Fig. 1. Correlation between HPL and DGEMM as reported by HPCC and available at the HPCC website.

dense matrix-matrix multiply routine – a Level 3 BLAS [9], [8] called DGEMM. Naturally then, the sequential version of the routine may be used to estimate the time to run as well as the performance of a parallel dense solve. By utilizing the information from the publicly available[1] results of the HPC Challenge benchmark [22]: during a single execution both matrix-matrix multiplication and HPL are benchmarked which should provided a consistent experimental setup. Figure 1 gives a visual indication that DGEMM and HPL are indeed correlated based on over 250 entries in the HPCC database. In fact, the Pearson product-moment correlation coefficient [26] exceeds 99%. This is a somewhat deceptive achievement though. If we use DGEMM as a predictor for HPL then the median relative prediction error will be just over 15% and the smallest one will be 1.4%. Even if we generously dismiss all the results with greater-than-median error then we are still left with 1% to 15% variability in prediction accuracy. It is beyond the scope of this paper to fully explain such variability. We can mention briefly, however, that the HPCC results use different BLAS implementations, MPI implementations, and vastly varying hardware. Each of these components contributes its share of uncertainty. Hence we proceed to develop a more accurate prediction framework.

III. Execution Model of HPL

Aside from TOP500, HPL is an important tool in science. Boundary Element Methods, physics plasma calculation based on Helmoltz equations, and multipol methods for anthenae design studies result in very large dense systems of equations. Accurate model for HPL is essential in planning the use of computational resources for these scientific disciplines.

For the simplicity of exposition, we only show the derivation of the performance model for HPL that uses LU factorization with the theoretical operation count

[1] See http://icl.cs.utk.edu/hpcc/

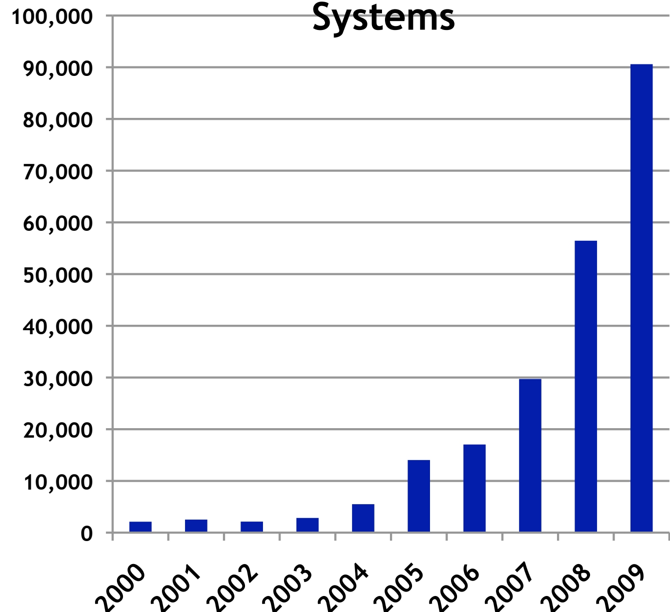

Fig. 2. Average number of cores for the top 20 entries on the recent TOP500 editions.

$2/3n^3 + O(n^2)$. But a similar argument easily applies to other one-sided factorizations such as Cholesky factorization (with operation count $1/3n^3$) and QR ($4/3n^3$). Later on, we show the results for all three factorization that uses the model developed below.

Execution time of HPL will grow with the total number of cores in the system provided that the local memory available per core remains constant. This is because the best performance is achieved when the problem size (the size of the linear system matrix) is large – preferably it should fill-up the usable portion of the main memory (excluding the memory needed by the operating system and the software libraries). Large problem sizes allow the software to hide high latencies and low throughputs of the memory hierarchy behind sufficient amount of very fast floating-point operations that operate directly on the registers and there can be many of such operations per clock cycle. Hence the time to run HPL is given by the formula:

$$t_{\text{Total}} \propto n^3 = \sqrt{M_{\text{Total}}}^3 = N_{\text{cores}}^{3/2} \quad (1)$$

In practice, this results in a rather drastic increase in running time as the number of cores of recent supercomputers started to rapidly increase as indicated on Figure 2.

Our goal is to come up with a comprehensive model for HPL without resorting to counting complexities of each and every routine involved in the factorization and back-solve as was done by Cuenca et al. [15] and Emmanuel Jeannot and Julien Langou [10]. The model for the HPL's floating-point execution rate is influenced by the operation count and the time to perform the solve. The operation count is fixed

regardless of the underlying algorithm to facilitate performance comparisons:

$$\mathrm{op}_{\mathrm{count}} = \frac{2}{3}n^3 + \frac{3}{2}n^2 \quad (2)$$

The time to do the solve has three components:

$$t = Fn^3 + Bn^2 + Ln + C \quad (3)$$

where F represents the inverse of the actual floating-point rate of the update phase of the LU factorization commonly refered to as the Schur's complement [16]. Values of F differ with the algorithm of choice: left-looking, right-looking, top-looking, Crout, recursive, etc. [12]. The B term corresponds to $O(n^2)$ floating-point operations (primarily panel factorizations) and various bandwidth levels such as between the cache and the main memory as well as the interconnect bandwidth. As B embodies execution rate of second-order terms, its value changes with the peak performance and bandwidth imbalance: the execution rate included in B may be an order of magnitude lower than the one represented by F. L mainly corresponds to both the memory hierarchy as well as the interconnect latency. Finally, C represents constant overheads such as populating caches with initial values and initializing network's communication layer. The floating-point rate is obtained as a ratio of operation count and the time to solution:

$$r_{\mathrm{fp}} = \frac{\mathrm{op}_{\mathrm{count}}}{t} \quad (4)$$

To an extent, the above model takes into account non-linear dependence of the running time on some algorithmic constants such as blocking factor NB. These constants may be hidden inside F, B, L, and C as long as they don't change with n. The algorithmic parameters[2] that may be hidden in this manner include:

1) blocking factor NB
2) logical process grid order
3) recursive panel factorization type RFACT
4) recursive sub-panel count NDIV
5) recursion stopping criterion NBMIN
6) matrix-vector panel factorization type PFACT
7) panel broadcast algorithm BCAST
8) look-ahead depth DEPTH
9) row swapping algorithm SWAP
10) row swapping threshold
11) L_1 panel columns transpose
12) U panel columns transpose
13) presence of equilibration
14) memory alignment

The reason why these parameters may be accounted for by properly selecting F, B, L, and C, is that they can be fixed and don't change with number of cores or the problem size N. However, logical process grid changes with the number of cores. If the ratio of logical process rows to logical process columns remains almost the same then the algorithm behaves similarly regardless of the actual logical process grid shape. However, as the number of cores changes, the parameters of the Equation (3) are not uniformly affected even if the aspect ratio of the process grid remains the same. Hence, only the runs with the same core count are modelled together.

At its heart, performance modelling is deeply related to experimental science in that it relies on an actual experiment to verify the findings of the proposed model. The first issue to resolve is what to model: the perfromance rate or the time to run the code. From the data analysis standpoint, performance is the inverse of time multiplied by the matrix size cubed: this translates to amplification of the experimental and measurement errors by a large quantity. Naturally then, time to run HPL is less sensitive to the errors and outlying data points. The time is then chosen for modelling. By making this choice, we alleviate the influence of outliers on the data and thus we avoid the necessity of using non-linear statistical methods that involve medians. The model fitting may be performed with the standard least-squares formulation rather than its non-linear counterparts that are less developed and suffer from the inaccuracies of non-smooth optimization [3]. As described above, the time to run HPL can be written as:

$$t = f_3n^3 + f_2n^2 + f_1n + f_0 \quad (5)$$

Given a k number of experiments with varying problem sizes $n_1, n_2, n_3, \ldots, n_k$ we obtain the actual running times $t_1, t_2, t_3, \ldots, t_k$. Using the results of these experiments, we formulate the problem as a linear least-squares problem:

$$\begin{bmatrix} n_1^3 & n_1^2 & n_1 & 1 \\ n_2^3 & n_2^2 & n_2 & 1 \\ & & \ldots & \\ n_k^3 & n_k^2 & n_k & 1 \end{bmatrix} \begin{bmatrix} f_3 \\ f_2 \\ f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \ldots \\ t_k \end{bmatrix} \quad (6)$$

or more compactly:

$$Af = t \quad (7)$$

with $A \in \mathbb{R}^{k\times 4}$, $f \in \mathbb{R}^4$, and $t \in \mathbb{R}^k$. The absolute modelling error can then be defined as

$$M_{\mathrm{err}} = \|Af - t\|_\infty \quad (8)$$

with the assumption that the entries of t are relatively accurate. Such is the case when we use the median of time measurements.

The system matrix A from equation (7) is a Vandermode matrix and tends to be badly conditioned [16]. Matrices from typical experiments can have a norm-2 condition number as high as 10^{11} which means that the model fitting needs to be performed in double-precision arithmetic even though the data itself has only a handful of significant digits worth of accuracy. Equilibration [1] may reduce the condition number by nearly ten orders of magnitude but the resulting modelling error gets reduced only slightly. A similar reduction of the modelling error may be achieved with a simple row scaling that results from dividing each row of the linear system (6) by the respective problem size:

$$a_3n_i^2 + a_2n_i + a_1 + a_0/n_i = t_i/n_i \quad (9)$$

The reduction of error may be as high 20 percentage points.

[2] For more details see http://www.netlib.org/benchmark/hpl/.

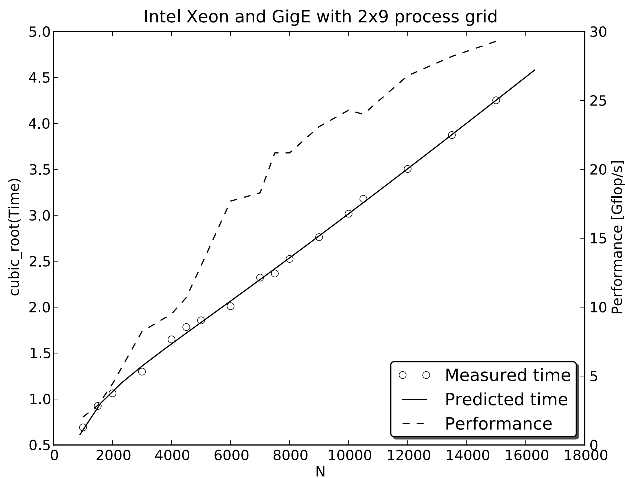

Fig. 3. Modeled versus measured time to run HPL on a common cluster. The cubic root of time is plotted to increase graph's clarity. Performance numbers are shown for reference only.

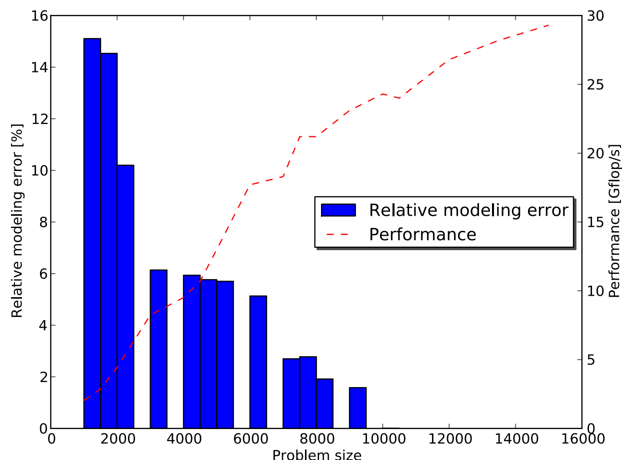

Fig. 4. Modeled vs measured time to run HPL on a common cluster. Performance numbers are shown for reference only.

Figure 3 shows how the model performs on a non-dedicated cluster[3] comprised of commodity hardware components. The modelling error is 14% when all the data points are accounted for. If a handful of initial data points is removed, the modelling error drops to just over 2% which is within the noise levels of a non-dedicated system. Thus, the method is sensitive to the measuring error but (in numerical sense) is stable because it delivers the answer whose quality is close to the quality of the input data – a property that is an established standard for properly implemenented numerical libraries [33], [34].
Figure 4 shows how the error gets reduced (the farther to the right the shorter are the bars) as the left-most points are eliminated. The explanation is that the leftmost data points do not represent the asymptotic performance rate of HPL: they cannot be modeled with Equation (5) because the attained performance varies significantly with the problem size (coefficients a_i no longer are constants but instead they are a function of n).

[3]Dual-core 1.6 GHz Intel Core 2 with Gigabit Ethernet interconnect.

Virtual process grid rows	Virtual process grid columns	Modeling error [%] [%]
1	30	0.7
2	15	1.0
3	10	0.8
5	6	2.2
6	10	1.6
1	60	0.5
3	20	0.7
2	30	0.5
4	15	2.2
5	12	2.7

TABLE I
VARIOUS LOGICAL PROCESS GRIDS AND THE CORRESPONDING MODELLING ERROR FOR LU FACTORIZATION AS IMPLEMENTED IN SCALAPACK.

IV. MODELING A GENERAL PURPOSE DENSE LINEARY ALGEBRA LIBRARY: SCALAPACK

As mentioned earlier, our modelling methodology is robust enough to be applicable to more than just HPL. For example, Table I shows the modelling errors achieved on a dedicated cluster[4] running LU factorization available in ScaLAPACK [4], [6]. The table indicates very high accuracy (mostly around 1% and not exceeding 3%) provided that the measurements with different virtual process grids are not modelled together but rather treated separately. Just like HPL, ScaLAPACK is written in terms of a number of tunable parameters one of which is the shape of the virtual process grid. However, each process grid shape is more than just an input parameter to the factorization routine: it acts more as an algorithm selector. For example, tall-and-skinny (the number of process rows is far greater than the number of process columns) and short-and-wide (the number of process columns is far greater than the number of process rows) process grids exhibit poor scaling while the square-like process grids have proven scalability properties [25], [5]. Therefore, as alluded before, it is important to consider the use of each grid shape to be an instance of a different algorithm and hence it should be modelled separately. Similarily, the size of the matrix blocking factor influences the tradeoff between the local performance and the ability to tolerate low-bandwidth/high-latency at the interconnect level [24]. For comparison, in Table II, we show results of modelling error for ScaLAPACK's main three one-sided factorizations on a shared-memory multicores machine[5]. The error is larger then in previous dedicated runs. This experiment is to show how our modelling framework performs in an environment with noise in collected data. In this case, the noise comes from a stock Linux kernel installation without necessary

[4]Intel 2.4 GHz Pentium 4 cluster with Gigabit Ethernet interconnect.
[5]The machine has 8 NUMA nodes with an AMD Instanbul 2.8 GHz 6-core processor for each node and a stock Linux kernel installation.

Factorization	Modelling error [%]	Standard deviation of measurement [% of median]
Cholesky	14	15
LU	14	20
QR	16	7

TABLE II
VARIOUS ONE-SIDED FACTORIZATIONS FROM SCALAPACK ON A SHARED-MEMORY NUMA MACHINE AND THEIR CORRESPONDING MODELLING ERROR AND STANDARD DEVIATION OF TIME MEASUREMENT.

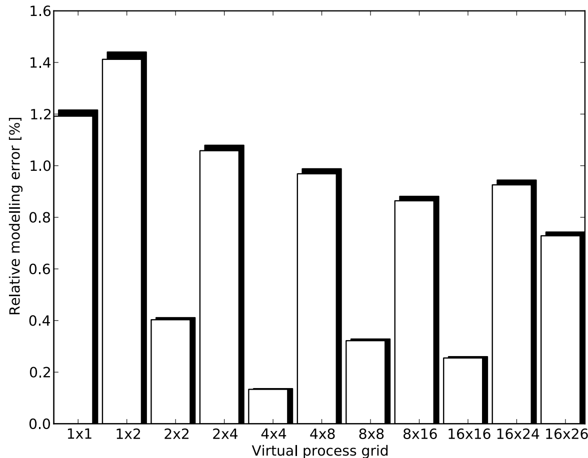

Fig. 5. Relative modelling error for a single-core dual-processor Intel Xeon 3.2 GHz cluster with 416 processors connected with InfiniBand interconnect.

optimizations for large shared memory installation. The last column of Table II indicates the standard deviation relative to the median for 16 time measurements for a relatively small problem size ($n = 2000$, median time under half a second). As shown previously, the modelling error is on the order of the standard deviation of time measurement which indicates to us that the method is as accurate as the quality of its input data.

V. QUALITY OF PREDICTION FOR EXTRAPOLATION

Arguably, the most useful aspect of modelling is extrapolation of acquired data. High quality models are able to provide information about runs with larger data sets based on the results from runs with smaller data sets. To evaluate our approach we chose a publicly availble data set of HPC Challenge (HPCC) benchmark results from runs performed in 2007[6]. Figure 5 shows a relative modelling error for all the available results sorted by the virtual process grids from 1-by-1 to 16-by-26. As before, the model is very accurate and the relative error stays around 1% and often drops to a fraction of a percent. This is an indication the runs were made in a dedicated mode because our model is deeply rooted in the knowledge of the algorithm and is not intented to model external influences such as operating system noise in a multi-user environment.

[6]The data set and more information on the hardware used is avilable at http://icl.cs.utk.edu/hpcc/custom/index.html?lid=111&slid=218.

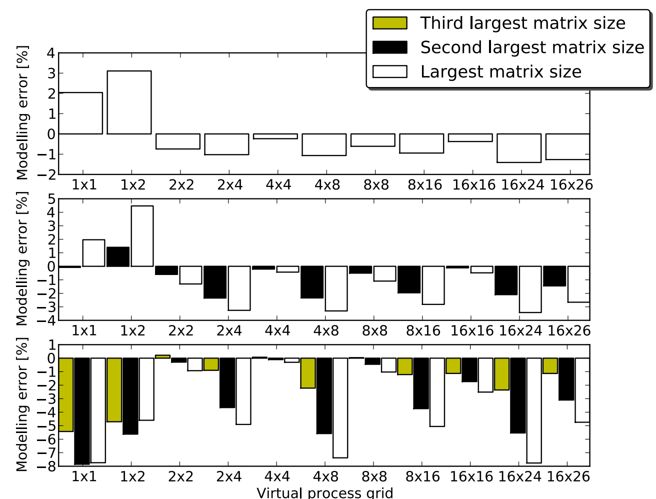

Fig. 6. Relative prediction error for the largest matrix size (top), the largest and the second largest matrix size (middle), as well as the largest, secondon, and third largest matrix size (bottom) on a single-core dual-processor Intel Xeon 3.2 GHz cluster with 416 processors connected with InfiBand interconnect.

A more interesting application of our model is to compute the time to run without actually running the code. Figure 6 illustrates this kind of experiment using the same data as was modelled in Figure 5. There are eleven process grid in the modelled data set and each grid was used seven times to execute the HPCC benchmark with increasing problem sizes. However, for any given process grid, the first problem size corresponded to the same percentage of the memory or (in other words) the same amount of data per process was used. Then, the second process size corresponded to twice as large amount of data per process and so on. If we look at this data set from the perspective of our model, we may treat first six data points as measurements and use them for obtaining the coefficients of the model and then we can predict the runtime for the remaining seventh data point. The modelling error for each process grid is presented in the top graph of Figure 6. The middle graph shows the scenario where 5 data points establish the coefficients and the remaining two data points are predicted. Finally, the bottom of the figure shows the situation where 4 data points are used to compute the coefficients and the remaining 3 data points are being predicted. Since our model has 4 coefficients we need at least 4 data points to calculate them. By looking at Figure 6 we see that the modelling error is small, always below 8% and often within or below 1%. What the figure doesn't show is the reduction of time that a prediction would afford. With 6 data points for modelling and 1 data point predicted (top of the figure) there are no savings in time, in fact the time to run the first 6 experiments is about 20% longer then the time it takes to run the remaining seventh experiment. In the case of 5 data points for modelling (middle of the figure), there is already a substantial reduction of time: the last two experiments take about 70% of the total time. And finally, with 4 experiments predicting the 3 longest runs, the savings in time exceed 90%.

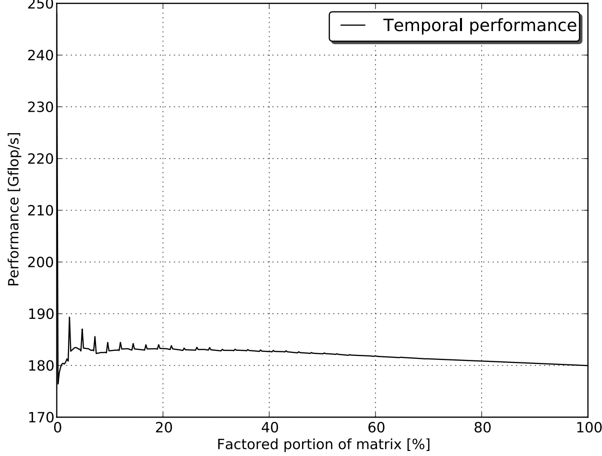

Fig. 7. Temporal performance (in Gflop/s) achieved in parallel HPL on an Intel Xeon 3 GHz cluster with 100 cores.

VI. SAMPLED EXECUTION

In this section we test our modelling framework in yet another context where we attempt to reduce the time to run the HPL benchmark by performing only some part of the computation.

Figure 7 shows the temporal performance observed from process 0 during a parallel HPL run: various portions of the run contribute differently to the final performance number of 182.135 Gflop/s. Note the strongly nonlinear behavior of performance across time.

We first present a theoretical framework that allows selective execution and verification of the standard factorization and then we show performance results at larger scale and give a justification of the selected portion of the standard factorization.

A. Sampled Factorization without Pivoting

First, for simplicity, consider LU factorization without any pivoting

$$A = LU. \tag{10}$$

We'd like to perform *partial execution*: only some parts of the matrix will be factored and subsequently updated. For this, let's assume the following structure of A

$$A = \begin{bmatrix} I & A_{12} & 0 & A_{14} & 0 \\ 0 & A_{22} & 0 & A_{24} & 0 \\ 0 & A_{32} & I & A_{34} & 0 \\ 0 & A_{42} & 0 & A_{44} & 0 \\ 0 & A_{52} & 0 & A_{54} & I \end{bmatrix}. \tag{11}$$

To take advantage of this structure, it's only necessary to factor columns 2 and 4. The only updates that need to be performed are applied from column 2 to column 4. This will be sufficient to have a correct LU factorization and can be used to subsequently solve a linear system and compute a residual to check the answer.

The goal, however, is to perform all the updates "to the right" (assuming that the right-looking algorithm is used): it is the updates that afford high performance numbers for HPL runs as evident in Figure 7. That's why we use the following matrix instead:

$$A = \begin{bmatrix} I & A_{12} & A_{13} & A_{14} & A_{15} \\ 0 & A_{22} & A_{23} & A_{24} & A_{25} \\ 0 & A_{32} & A_{33} & A_{34} & A_{35} \\ 0 & A_{42} & A_{43} & A_{44} & A_{45} \\ 0 & A_{52} & A_{53} & A_{54} & A_{55} \end{bmatrix}, \tag{12}$$

but still only require factorization of columns 2 and 4. The updates have now to be applied to columns 3 and 5 (in addition to updates from column 2 to column 4) because they contain non-zero entries. The result of a sampled factorization should be as follows:

$$A = \begin{bmatrix} I & A_{12} & \tilde{A}_{13} & A_{14} & \tilde{A}_{15} \\ 0 & D_{22} & \tilde{A}_{23} & U_{24} & \tilde{A}_{25} \\ 0 & L_{32} & \tilde{A}_{33} & U_{34} & \tilde{A}_{35} \\ 0 & L_{42} & \tilde{A}_{43} & D_{44} & \tilde{A}_{45} \\ 0 & L_{52} & \tilde{A}_{53} & L_{54} & \tilde{A}_{55} \end{bmatrix}. \tag{13}$$

Where D_{ii} are diagonal submatrices with embedded lower and upper portions of L and U factors:

$$D_{ii} = L_{ii} + U_{ii} - I \tag{14}$$

Verifying correctness of the factorization can be done using two steps:

1) Check the columns 2 and 4 – the fully factored columns – using the structure from equation (11).
2) Check the columns 3 and 5 – the columns with updates only – using the structure from equation (11) and multiple right-hand sides as shown below.

The second step involves the following system:

$$\begin{bmatrix} I & A_{12} & A_{13} & A_{14} & A_{15} \\ 0 & A_{22} & 0 & A_{24} & 0 \\ 0 & A_{32} & I & A_{34} & 0 \\ 0 & A_{42} & 0 & A_{44} & 0 \\ 0 & A_{52} & 0 & A_{54} & I \end{bmatrix} X = \begin{bmatrix} 0 & 0 \\ A_{23} & A_{25} \\ A_{33} & A_{35} \\ A_{43} & A_{45} \\ A_{53} & A_{55} \end{bmatrix}. \tag{15}$$

If the cost of fully solving the system in Equation (15) is prohibitive in terms of time then it is possible to select only a few columns for the final solution and the residual calculation. The choice of columns can be made randomly which will eliminate a potential use of the knowledge of which columns are to be selected. This in effect will require proper updates to the full set of columns 3 and 5. Such randomizition is a typical method of preventing unwanted optimization [22].

B. Sampled Factorization with Partial Pivoting

For numerical stability, HPL performs LU factorization with partial pivoting:

$$PA = LU, \tag{16}$$

where P is a permutation matrix (it is a symmetric matrix with mostly zero entries except for a single one in each column and row).

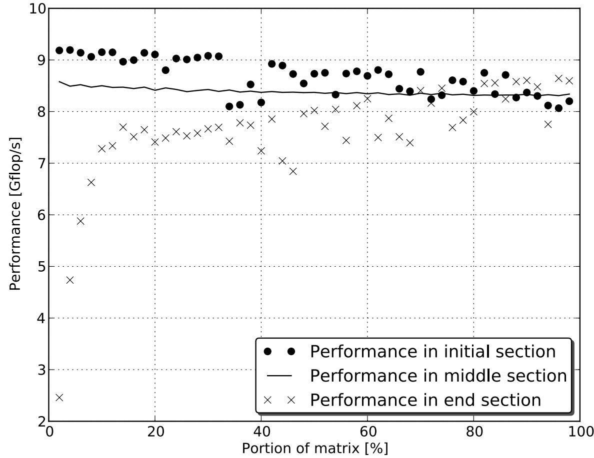

Fig. 8. Performance (in Gflop/s) achieved while sequentially performing the sampled factorization of three sections of a matrix on a dual-core Intel Core2 Duo 2.53 GHz computer.

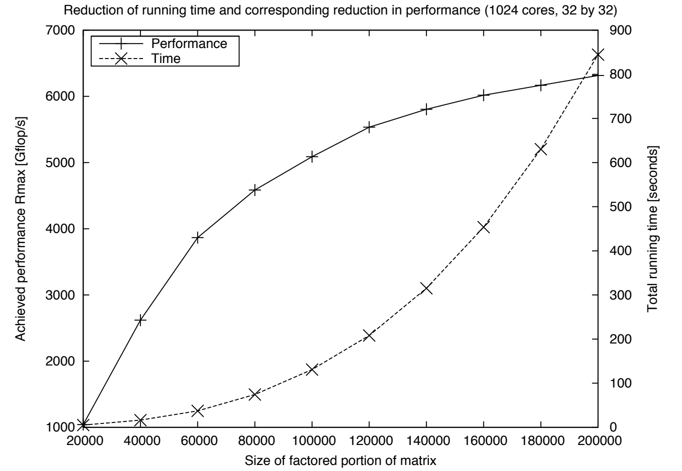

Fig. 9. Performance of end section factorization of matrix of size 0.2×10^6 on Cray XT 4 using 1024 cores.

Introduction of the pivoting matrix slightly modifies the derivation shown above – the partitioning of the permutation matrix becomes:

$$P = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & P_{22} & P_{23} & P_{24} & P_{25} \\ 0 & P_{32} & I & 0 & 0 \\ 0 & P_{42} & 0 & P_{44} & P_{45} \\ 0 & P_{52} & 0 & P_{54} & I \end{bmatrix}. \quad (17)$$

Note that the first column is not altered by pivoting because it is already in factored form (it is a submatrix of an identity matrix). The facatorization routine now has to return a pivoting matrix (usually in a form of a vector of integer values). This matrix (or its inverse) is then used to solve the system (15) in a standard way [29], [28].

C. Performance Aspects of Sampled Factorization

The previous sections showed a theoretical famework for performing partial execution to achieve sampled factorization and subsequently verify the result. They did not provide any guidance how the portions of factorization contribute to the overall performance. This is the subject of this section. Let's first divide the system matrix into three parts:

$$A = \begin{bmatrix} A_{11} & 0 & 0 & 0 & 0 \\ A_{21} & 0 & 0 & 0 & 0 \\ A_{31} & 0 & A_{33} & 0 & 0 \\ A_{41} & 0 & A_{43} & 0 & 0 \\ A_{51} & 0 & A_{53} & 0 & A_{55} \end{bmatrix}. \quad (18)$$

We will be referring to the first set of columns from the left ($[A_{11}, A_{21}, A_{31}, A_{41}, A_{51}]^T$) as the *initial section*. The second set of columns ($[A_{33}, A_{43} A_{53}]^T$) – the *middle section*; and the third (A_{55}) – the *end section*. The sampled factorization of the initial section factorizes the initial section set of columns and performs the Schur complement update of the rest of the matrix. Similarly, the middle section only factorizes the middle section's set of columns and performs an update to the right but leaves the initial section untouched. Finally, the end section of sampled factorization is a full factorization of the matrix A_{55} – initial and middle sections are not referenced at all. Figure 8 shows how the achieved performance of sampled factorization varies with the relative size of the portion of the matrix used for factorization. The results are from a sequential run but are representive of parallel runs as well. To the right of the figure all three data point series converge because each sampled factorization simply becomes a standard factorization of the whole matrix. On the left side of the graph in the Figure, the series are divergent: the initial section factorization largely overestimates the actual performance, the middle section factorization overestimates the performance only slightly, and the end section underestimates the peformance achieved while factoring the whole matrix with the standard method. These results can be looked from various perspectives. A system designer view is to maximize the achieved performance and therefore the initial section factorization is the most attractive. The middle section has a very desirable property of very quickly converging to the actual performance of the full matrix factorization. However, the middle section performance still overestimates the actual result which is is a drawback from the benchmarking perspective. Finally, only the end section performance can be modelled directly with the previously described performance model because it is a full factorization of the lower right portion of the matrix A_{55} just as the standard factorization operates on the whole matrix A. We therefore proceeed by using the end section in our subsequent experiments.

VII. Combined Application of Modelling and Sampled Factorization

To test our modelling framework together with the sampled factorization approach, we performed a series of experiments at the Oak Ridge National Laboratory on a large scale supercomputer: Cray XT 4 with quad-core AMD Opteron 1345 processor clocked at 2.6 GHz. The whole installation

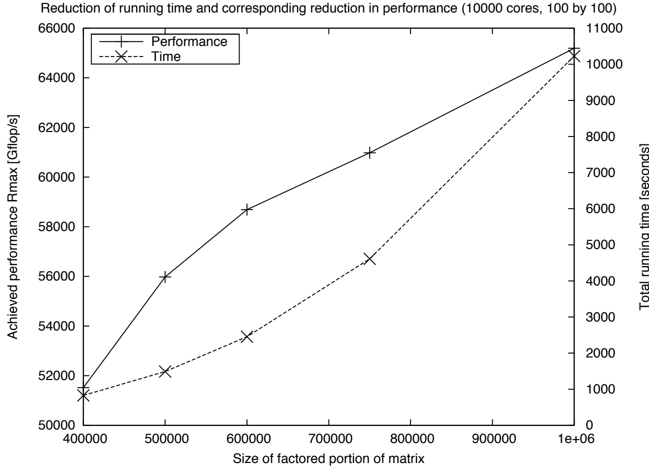

Fig. 10. Performance of end section factorization of matrix of size 1×10^6 on Cray XT 4 using 10000 cores.

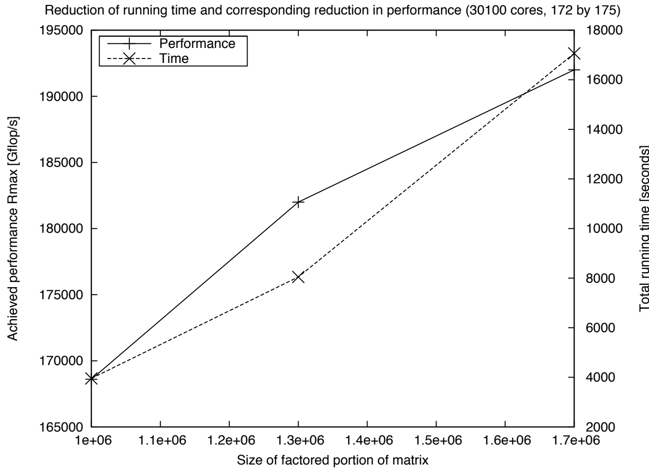

Fig. 11. Performance of end section factorization of matrix of size 1.7×10^6 on Cray XT 4 using 30100 cores.

consists of over 30 thousands cores (31328 to be exact). Figures 9, 10, and 11 show performance and running time for varying number of cores, matrix sizes, and portions of fully factored matrix. The code used for runs was the HPL code modified to allow for partial execution as described earlier. Despite the order of magnitude difference in core counts and matrix sizes, the model is accurate to about 1% or less. The *performance-time curves* from the figures show the practical consequences of Equation (1): in order to attain comparable fraction of the peak performance the time to run will increase faster than linearly with the number of cores. Figure 12 shows the performance-time curve from Figure 9 but this time it is recast in relative terms as a fraction of the maximum attained performance. In this manner both time and performance may coexist on the same Y-axis because they both vary between 0% and 100%. In this setting it is now easy to perform a what-if analysis of the data as it is indicated in the Figure 12 with arrows. The question being answered by the Figure is this: if the time to run the benchmark is reduced by 50% how much will the

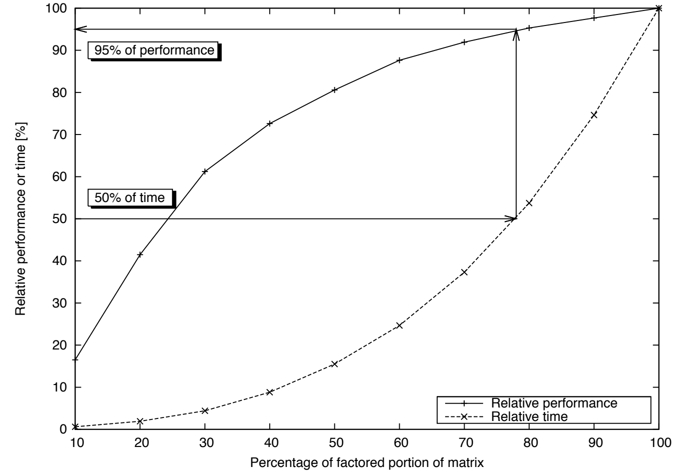

Fig. 12. Relative running time and achieved performance with relation to fraction of factored portion of matrix (1024 cores, with 32 by 32 virtual process grid.)

performance result be reduced. The answer is encouraging: the resulting performance drop will only be 5%. The making of such predictions is simplified with our model based on a handful of runs that accurately determine the modelling coefficients. Without a model, it would be necessary to make many more additional runs which would diminish the benefits of reduced running time. And even with the extra runs the exact point of 50% reduction of time is unlikely to be found experimentally as is the case in Figure 12 and a curve fitting or an approximation technique would be necessary to provide more refined guidance for finding the right problem size. These issues are already accounted for in our model. The sampled factorization provides the added benefit of stressing the entire system: the system matrix occupies the entire memory. Also, the result of the partial execution can be verfied as rigorously as is the case for the standard factorization.

VIII. Related Work

Execution model for HPL based on Self-Similarity Theory and the Π-Theorem models floating-point performance on a p by q process grid [23]:

$$r_{\text{fp}} = \gamma\, p^{\alpha} q^{\beta} \tag{19}$$

The values for α, β, and γ need to be determined experimentally. Because of the nonlinear relation between these coefficients and the performance is nonlinear, we need to use a nonlinear optimization to fit the model to the experimental data.

A more complex models for HPL and other parallel linear algebra routines resort to modelling each individual period of time spent in every non-trivial computational or communication routine involved in the factorization and back-solve [15], [10]. They tend to give accurate results provided that they are populated with accurate software and hardware parameters such as computational rate of execution as well as bandwidth and latency of both memory and the interconnect network.

An attempt to model HPL using memory traces [32] resulted in scoring HPL on par if not worse with an FFT implementation in terms of memory locality. This was due to the use of reference BLAS implementation and lack of accounting for register reuse. Modern BLAS implementations outperform the reference code by at least an order of magnitude. One of the leading sources of matrix-matrix multiply performance comes from register reuse as evindenced by the efficiency (percentenge of the peak floating point performance) of IBM POWER3 and POWER4: the former achieves nearly 90% of peak performance in matrix-matrix multiply BLAS kernel called DGEMM while the latter only about 70% [7]. With similar cache structure, these two microprocessors differ drastically in the number of registers. Register allocation, latency hinding, and bandwidth utilization is a key to efficient BLAS routines (such as dense matrix-matrix multiply): when done properly it is still possible to surpass the highly optimized vendor implementation [30]. Our model captures the efficiency of register reuse.

Partial execution was successfully used to perform cross-platform performance predictions of scientific simulations [35]. The study relied on the iterative nature of the simulated codes. Dense linear algebra computations, as opposed to iterative linear algebra methods [11], are not iterative in nature and commonly exhibit non-linear variation in performance as shown in Figure 7.

Performance prediction in the context of grid environments focuses work load characterization and its use in effective scheduling and meta-scheduling algorithms[27]. The techniques used in such characterization tend to have a higher error rates ("between 29 and 59 percent of mean application run times").

IX. Conclusions

We have presented a modelling framework and its applications to prediction of performance across very diverse hardware platforms ranging from commodity clusters to large scale supercomputer installations. The accuracy is as good as the quality of the input data which has been a golden standard for numerical linear algebra for years [33], [34]. Thus, we consider the model validated by experiments on hardware with varying measurement error due to both multi-user environment and operating system noise. We applied our methodology to provide a viable process of performing what-if analysis that can allows informed decision regarding the tradeoff between higher performance and shorter running time.

The obtained results encourage us to pursue a more detailed study of the model such as sensitivity analysis and the ability of the model to show measurement error, hardware problems, or software misconfiguration [21].

Acknowledgements

This research was supported by DARPA through ORNL subcontract 4000075916. We would like to thank Patrick Worley from ORNL for facilitating the large scale runs on Jaguar's Cray XT4 partition.

References

[1] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[2] R. F. Barrett, T. H. F. Chan, E. F. D'Azevedo, E. F. Jaeger, K. Wong, and R. Y. Wong. Complex version of high performance computing LINPACK benchmark (HPL). *Concurrency and Computation: Practice and Experience*, 22(5):573–587, 2010.

[3] Åke Björk. *Numerical methods for Least Squares Problems*. SIAM, 1996. ISBN 0-89871-360-9.

[4] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[5] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for Clusters. *Parallel Computing*, 29(11-12):1723–1743, November-December 2003.

[6] J. Choi, Jack J. Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clint Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[7] J. Dongarra. Performance of various computers using standard linear equations software. Computer science dept. technical report, University of Tennessee, Knoxville, TN, April 1996. up-to-date version available in http://www.netlib.org/benchmark/.

[8] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):18–28, March 1990.

[9] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[10] Jack Dongarra, Emmanuel Jeannot, and Julien Langou. Modeling the LU factorization for SMP clusters. In *Proceeedings of Parallel Matrix Algorithms and Applications (PMAA'06)*, IRISA, Rennes, France, September 7-9 2006.

[11] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[12] Jack J. Dongarra, Fred G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[13] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[14] Alan Edelman. Large dense numerical linear algebra in 1993: the parallel computing influence. *International Journal of High Performance Computing Applications*, 7(2):113–128, 1993.

[15] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Using experimental data to improve the performance modelling of parallel linear algebra routines. In *Lecture Notes in Computer Science*, volume 4967, pages 1150–1159. Springer Berlin / Heidelberg, 2008. ISSN 0302-9743 (Print) 1611-3349 (Online), DOI 10.1007/978-3-540-68111-3.

[16] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, third edition, 1996.

[17] Meuer H., Strohmaier E., Dongarra J., and Simon H. *TOP500 Supercomputer Sites*, 34th edition, November 2009. (The report can be downloaded from http://www.netlib.org/benchmark/top500.html and http://www.top500.org/.

[18] R. Harrington. Origin and development of the method of moments for field computation. *IEEE Antennas and Propagation Magazine*, June 1990.

[19] J. L. Hess. Panel methods in computational fluid dynamics. *Annual Reviews of Fluid Mechanics*, 22:255–274, 1990.

[20] L. Hess and M. O. Smith. Calculation of potential flows about arbitrary bodies. In D. Kuchemann, editor, *Progress in Aeronautical Sciences*, volume 8. Pergamon Press, 1967.

[21] Darren J. Kerbyson, Adolfy Hoisie, and Harvey J. Wasserman. Verifying Large-Scale System Performance During Installation using

Modeling. In *in High Performance Scientific and Engineering Computing, Hardware/Software Support*. Kluwer, October 2003.

[22] Piotr Luszczek, Jack Dongarra, and Jeremy Kepner. Design and implementation of the HPCC benchmark suite. *CT Watch Quarterly*, 2(4A), November 2006.

[23] Robert W. Numerich. Computational forces in the Linpack benchmark. *Concurrency Practice and Experience*, 2007.

[24] Andy Oram and Greg Wilson, editors. *Beautiful Code*. O'Reilly, 2007. Chapter 14: How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination.

[25] Kenneth J. Roche and Jack J. Dongarra. Deploying parallel numerical library routines to cluster computing in a self adapting fashion. In *Parallel Computing: Advances and Current Issues*. Imperial College Press, London, 2002.

[26] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.

[27] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application runt times with historical information. In *Proceedings of IPPS Workshop on Job Scheduling Strtegies for Parallel Processing*. Elsevier Inc., 1998. DOI: 10.1016/j.jpdc.2004.06.2008.

[28] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[29] G. W. Stewart. *Matrix algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[30] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008.

[31] J. J. H. Wang. *Generalized Moment Methods in Electromagnetics*. John Wiley & Sons, New York, 1991.

[32] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snavely. Quantifying locality in the memory access patterns of HPC applications. In *Proceedings of SC05*, Seattle, Washington, 2005. IEEE Computer Society Washington, DC, USA.

[33] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice Hall, Englewood Cliffs, 1963.

[34] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, 1965.

[35] Leo T. Yang, Xiasong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the ACM/IEEE SC—05 Conference (SC'05)*. IEEE, 2005.