Slow Nodes Cost Your Users Valuable Resources. Can You Find Them?

Ricky A. Kendall & Don Maxwell
National Center for Computational Sciences
Oak Ridge National Laboratory

Jeff Becklehimer & Cathy Willis
Cray Inc.

May 1, 2009

Abstract

Many High Performance Computing applications have a static load balance which is easy and cheap to implement. When one or a few nodes are not performing properly this makes the whole code slow down to the rate limiting performance of the slowest node. We describe the utilization of a coded called Bugget which has been used on Catamount and the Cray Linux Environment to quickly identify these nodes so they can be removed from the user pool until the next appropriate maintenance period.

KEYWORDS: System Diagnostics, Parallel Matrix Multiply

1 Introduction

High Performance Computing (HPC) algorithms are inherently parallel today. In fact, most leadership computational resources have multiple levels of concurrency that must be managed by the programming model used to implement those algorithms. With the size and complexity of these systems having bad hardware or hardware performing in less than an optimal fashion is a constant issue that must be mitigated. The hardware issues coupled with the software complexity of implemented parallel algorithms offers a significant challenge to the computational science community for the development and utilization of scientific applications.

Most software developers make a few assumptions to make progress toward their scientific goals for simulation. These include:

- hardware gives correct results
- hardware is performing optimally
- compilers produce correct runnable binaries
- the computer is doing what I think it is doing.

We all know that these assumptions are a base for making progress but that sometimes there are failures in each. For long running simulations we know that round-off errors creep in and that the algorithms we use have to be tolerant to those errors or we have to correct for them. We expect that the hardware we are using is performing as best as it can given our description of our algorithm in a high level programming language. Most computational scientists have filed a compiler bug where incorrect code is generated. Often half way through the process of developing the reproducer for this bug we realize we have made the mistake and the compiler did do what we asked it to, but not always. Furthermore, optimal performance is a nebulous and difficult dance with the compiler and the myriad of compiler options as well as the necessary reformatting of the high level language that we are using to implement our algorithms.

In this work we attempt to verify the first two assumptions above but based on the second two assumptions. This is how most computational scien-

tists interact with their computer resources. We use the Fox and Hey algorithm [1] also known as the broadcast multiply and roll algorithm for matrix multiplication to determine a computed matrix multiply based on defined matrices that allow us to compare that result with the analytical result. This gives us some assurance that the hardware is giving us the correct result. The same instance of matrix multiplication is replicated across the various nodes of the system and the timing information is collected. Statistically we determine which nodes are "bad" and identify them. Once nodes are identified we can either not use them in a batch job or have the systems staff remove them from the available pool of nodes.

2 Bugget

Bugget was originally developed as a simple platform for understanding programming model choices for various parallel computing systems. It is based on matrix multiply which is considered the "hello world" program for parallel programs. The surface to volume ratio is well suited for this in that there is O(N^3) work and O(N^2) data movement. There are roughly a 1/2 a mole of different parallel matrix multiplication algorithms available [1-12] and there are many implementation details that make one algorithm more suitable than another. If you are using a parallel system or programming model that won't allow you to scale a matrix multiplication algorithm then there is a problem with that system or model.

2.1 What does Bugget do?

Since the main goal for Bugget was to explore characteristics of a parallel computational system, the algorithmic choice was not driven by the best performance. We chose to use the Fox and Hey algorithm [1] for the parallel matrix multiply. There is a main driver that handles breaking up the processes assigned to a given job into appropriate groups of processors. In each group of processes the algorithm is separately applied to arbitrarily defined matrices of known quantities. The simple operation is:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \tag{1}$$

where A and B are defined as:

$$A_{i,j} = (a * i + b * j + c)/rank_A \tag{2}$$

$$B_{k,j} = (d * k + e * j + f)/rank_B \tag{3}$$

Where a, b, c, d, e, and f are arbitrary constants. Because the quantities are known it is easy to compute what the result should be for $C_{i,j}$ via equation 1. This "analytical result" can be compared to the computed result via a parallel matrix multiply algorithm. The difference between the computed C and the analytical C should be zero:

$$\sum_i \sum_j C_{i,j}^{analytical} - C_{i,j}^{computed} = 0 \tag{4}$$

This difference norm gives a single number to check the result of the parallel matrix multiply algorithm. The generation of A, B and $C^{analytical}$ are all O(N^2) and are very cheap to compute. With this we now have a handle on the first assumption listed above in section 1.

2.2 Basic overview of the Fox and Hey algorithm

The nature of the algorithm requires that you have square matrices and a perfect square of processes involved. Furthermore the matrix rank must be evenly divisible by the square root of the process count. With these two constraints the matrices involved can be evenly distributed in "patches" across all processes involved. The overall matrix computation is divided into stages. Assuming there are 9 processes involved we have a $3x3$ process grid and evenly distributed patches of matrices A and B.

The first stage involves broadcasting the diagonal "patch" of matrix A in each row. You thus need a row communicator for each of the 3 rows in our $3x3$ grid. The first row will broadcast the $A_{1,1}$ patch, the second row will broadcast the $A_{2,2}$ patch, and the third row will broadcast the $A_{3,3}$ patch. On each process the broadcasted patch of A is multiplied by the patch of B currently held on that process. Once the multiplication is done then the B patch is rolled up in the column of the process grid. You need a column communicator for each column of the process grid. The rolling is accomplished by using a call to `MPI_sendrecv`.

The next stage broadcasts the patch of A to the right of the last patch sent and that is multiplied by the rolled B matrix now on that process. When you look to the right for A and roll B you have to wrap in the processor grid space. In this specific example, there are 3 stages so that all patches of A are multiplied by all patches of B which are summed into

a patch of $C^{computed}$. This is why the Fox and Hey algorithm is also known as the Broadcast, Multiply, and Roll algorithm. In general if q is the square root of the number of processes then there are q stages and one patch per process or q^2 patches for each matrix involved. In checking the difference norm it is straightforward to generate the patch you are supposed to have in each process which should be zero (c.f., Equation 4). Furthermore you can sum the difference norm for each patch to a single number which should also be zero. It is zero when everything works and in accounting for round off error it is usually 10^{10} or less.

2.3 The single process multiply component

In each stage the matrix multiplication that occurs concurrently in each process can utilize any arbitrary "serial" matrix multiply. Bugget allows the choice of:

- simple 3 loops of equation 1 in either C or Fortran
- the daxpy rearrangement in either C or Fortran
- a cache blocked version of the two above in C or Fortran
- OpenMP parallelized versions of each of the above.
- a *dgemm* call provided by any optimized library.

any one choice is made at run time. There are a total of 17 different choices for the serial matrix multiply algorithm.

The diversity of the implementation allows us to explore the programming model space we targeted for using both C and Fortran as well as for hybrid mixed MPI and OpenMP codes. The analysis computes the standard deviation of the time for each instance of the fox and hey algorithm and identifies which processes are out of the specified range of the average time. This tolerance can be set at run time. The percentage of peak performance is computed as well. Some observations of running the various algorithms are evident from utilizing the code on Jaguar. The library call is always faster than any of the source code implementations and produces greater than 80% of peak performance. The blocked source code implementation is slightly faster than the daxpy rearrangement for smaller matrices and moderately faster for larger matrices. The simple three loops is always the slowest implementation. Fortran and C implementations of the same algorithm perform the same as long as you let the C compiler know that there are no dependencies among the pointers to the matrices passed into the serial matrix multiply routine.

3 Utilization

The use of Bugget for finding slow nodes was a side effect of exploring programming models on the XT system. Under catamount users who had statically balanced code noticed a run time variability that could not be explained by time propagation in the collective communication space. Depending on which set of nodes they were allocated the code consistently ran 10% to 15% slower. We noticed the same behavior in some of the Bugget runs. Since we could tune the time for Bugget runs trivially, we decided to use it as the diagnostic for finding slow nodes instead of using HPL or a real user application.

By exploring this further the nodes that we identified were those that had many single bit memory errors on a memory chip on a given socket. Once we converted to the Cray Linux Environment (CLE) we noticed a similar behavior but the slowness could not always be attributed to single bit memory errors. After further analysis, it was determined that we had bad memory controllers not running at the right frequency.

The fact that slow nodes exist and can be identified with Bugget is the goal of the work we have done with the software. In "slow node finding mode" (SNF mode) we simply run an individual serial matrix multiply on all processes involved. Through experimentation we learned that if a node was going to be slow it could be identified by running a single process on the socket. If a socket is "slow" it is slow whether you run 1, 2, 3, or 4 processes on the given socket. In the XT4 partition of Jaguar there is only one socket per node. On the XT5 partition there are two sockets per node. You only need to make sure you deploy one running Bugget process per socket to identify the slow components.

By varying the algorithm and matrix size default choices for SNF mode were determined they are matrices of rank 3565, using the C implementation of

the daxpy rearrangement with a $1x1$ process grid. Using a larger process grid would identify the group of nodes but by using the 1 process grid we can easily identify the failing socket or node. This runs in about 100 seconds and reliably finds the slow nodes if they exist. The mechanism required to find the slow nodes involves having a large number of nodes to run on and assumes that the majority of those nodes are okay. If *all* are slow then the code will not appropriately identify the slow nodes. The identification is based on the statistics collected over all the sockets involved in the computation.

To facilitate SNF mode work with Bugget the XT4 partition is divided into sets that are of the same kind of hardware. In the Jaguar XT4 partition we have all the same kind of opteron chips but we have some nodes with 800Mhz memory and some with 667 MHz memory which will lead to a wider distribution over the computed set of processes. We have made the sets such that they include only 800MHz memory or 667MHz memory. On the XT5 partition the nodes were identified in columns and rows. This grouping is important in that we can run Bugget jobs over sets or columns without having to pin a whole in our scheduling for full machine run.

We have to mitigate the dynamic nature of the machine. We currently submit jobs that run either daily or weekly on the machine depending on the real user load and the level of instability we are seeing on the system. We use either a C based code to monitor this or a series of c-shell scripts that take care of generating the batch script required. We gather information from the Tourque/Moab tools to know how many nodes are active in either a set or column on the machine. Scripts are generated based on this information. Between the time the Bugget script is submitted and when it gets actually released to run the number of nodes available in a given set or column may change. We have monitoring scripts that handle renewing a job that has a different node count for a set or column. This happens as nodes fall out as well as when nodes get recovered from a reboot or hardware maintenance period. These same monitoring mechanisms also keep track of the output of 100 different instances of these Bugget jobs. If needed we can track over time how things have changed from run to run.

In addition to the "scheduled" runs that we do to facilitate timely identification of bad nodes users have access to the Bugget software via a module. Once the module is loaded they may run the binary in any way they desire. There is a convenience script that will run Bugget on the allocated processes in SNF mode; the name of this script is `bugget_my_nodes`.

4 Results

The results are tabulated for each group and overall for all groups. As an example here is the output[1] for a job running 8 processes on each node with a total of 400 process using a group of 100 processes. There are 4 groups that will eventually average the statistics. The *dgemm* library call was the matrix multiply algorithm used in the serial but concurrent component of the Fox and Hey algorithm and this was run on the XT5 partition.

```
aprun -n 400 -N 8 bugget q 10 r 59000 a dgemm
 Bugget MPI Version 2

rank = 59000
algorithm is 9: DGEMM
group nproc = 100 (q=10)
 total number of processors allocated = 400
4 groups of size 100
Number of groups: 4
Per-processes over all groups of any size:
Minimum Memory:    1593.48 MB =       1.56 GB
Average Memory:    1593.48 MB =       1.56 GB
Maximum Memory:    1593.48 MB =       1.56 GB

Minimum Patch Rank:       5900
Maximum Patch Rank:       5900


Rank 0 is on nid00163:c0-1c2s0n3
             (core affinity = 0)
Rank 1 is on nid00163:c0-1c2s0n3
             (core affinity = 1)
Rank 2 is on nid00163:c0-1c2s0n3
             (core affinity = 2)
Rank 3 is on nid00163:c0-1c2s0n3
             (core affinity = 3)
For Groups that have more than one process

Group 0 Statistics (group size:100)
Min fox Time:  532.187  Min Norm: 4.489e-12
Ave fox Time:  535.956  Ave Norm: 9.382e-12
```

[1] Modified to fit this paper format. The actual output is 90 characters wide.

```
Max fox Time:  539.536  Max Norm: 1.391e-11
STD DEV Time: 3.228e-01 STD DEV : 4.893e-13

Group 1 Statistics (group size:100)
Min fox Time:  533.196  Min Norm: 4.489e-12
Ave fox Time:  535.953  Ave Norm: 9.382e-12
Max fox Time:  538.411  Max Norm: 1.391e-11
STD DEV Time: 1.487e-01 STD DEV : 4.893e-13

Group 2 Statistics (group size:100)
Min fox Time:  533.072  Min Norm: 4.489e-12
Ave fox Time:  536.238  Ave Norm: 9.382e-12
Max fox Time:  538.365  Max Norm: 1.391e-11
STD DEV Time: 6.824e-02 STD DEV : 4.893e-13

Group 3 Statistics (group size:100)
Min fox Time:  532.661  Min Norm: 4.489e-12
Ave fox Time:  536.054  Ave Norm: 9.382e-12
Max fox Time:  537.984  Max Norm: 1.391e-11
STD DEV Time: 1.830e-01 STD DEV : 4.893e-13

Stats over all procs in a group of size: 100
             (100:400:r=59000:p=9.2GF:DGEMM)
Min fox Time: 532.19 (-2.6) GF(7.7) PP(83.9)
Ave fox Time: 536.05 ( 0.0) GF(7.7) PP(83.3)
Max fox Time: 539.54 ( 2.4) GF(7.6) PP(82.8)
Max-Min Time:   7.35 ( 5.0)
STD DEV Time: 1.460e+00

Min Norm: 4.489e-12 (-2.3)
Ave Norm: 9.382e-12 ( 0.0)
Max Norm: 1.391e-11 ( 2.1)
STD DEV : 2.150e-12

out 127: 273 of 400 w/in 1.0-sig ( 68.2%)
out  20: 380 of 400 w/in 2.0-sig ( 95.0%)
out   0: 400 of 400 w/in 3.0-sig (100.0%)
Sigma Threshold: 4.2 sec
Time  Threshold: 0.333 sec

Total reported bad sigma nodes: 0 of 400
                  nodes tested.

Global time: 545.683 seconds
```

All groups of size greater than 1 print individual group statistics and all groups of the same size provide final statistics for the job. Inspection of the output shows that the per process memory utilization (1.5 GBytes) is constant since all groups are of the same size. The core affinity is printed out for the first 4 processes showing that as expected from the `aprun` command the processes are fully packed. The individual group statistics show a normal timing spread for this type of calculation. The summary statistics show an even distribution around the average time. The minimum time is 2.6σ below the average and the maximum time is 2.4σ above the average. The global maximum/minimum time difference is 7.3σ. The percentage of peak performance is 82.8% for the maximum time of all 4 instances run in this job and the performance was 7.6 Gflops per process. This job identified no bad nodes that are greater than 4.2σ. The default thresholds have been set empirically but can be set by the user at run time. Comparing the overall maximum time for the algorithm and the global timing the overhead for setting up the computation and the analysis is 6 seconds or 1%.

The next output is run in the SNF mode (e.g., -f option) on column 10 of the Jaguar XT5 partition. This demonstrates what is seen when bad nodes are identified.

```
aprun -n 1408 -S 1 bugget -f
 Bugget MPI Version 2

rank = 3565
algorithm is 2: DAXPY
group nproc = 1 (q=1)
 total number of processors allocated = 1408
1408 groups of size 1
Number of groups: 1408
Per-processes over all groups of any size:
Minimum Memory:     581.78 MB =       0.57 GB
Average Memory:     581.78 MB =       0.57 GB
Maximum Memory:     581.78 MB =       0.57 GB

Minimum Patch Rank:       3565
Maximum Patch Rank:       3565

Rank 0 is on nid00032:c0-0c1s0n0
             (core affinity = 0)
Rank 1 is on nid00032:c0-0c1s0n0
             (core affinity = 4)
Rank 2 is on nid00033:c0-0c1s0n1
             (core affinity = 0)
Rank 3 is on nid00033:c0-0c1s0n1
             (core affinity = 4)

Stats over all procs in a group of size: 1
             (1:1408:r=3565:p=9.2GF:DAXPY)
Min fox Time: 102.173 (-0.8) GF(0.9) PP(9.6)
Ave fox Time: 103.424 ( 0.0) GF(0.9) PP(9.5)
```

```
Max fox Time: 120.982 (11.6) GF(0.7) PP(8.1)
Max-Min Time:  18.809 (12.4)
STD DEV Time: 1.516e+00

Min Norm: 1.052e-12 ( 0.0)
Ave Norm: 1.052e-12 ( 0.0)
Max Norm: 1.052e-12 ( 0.0)
STD DEV : 0.000e+00

out 20: 1388 of 1408 w/in  1.0-sig ( 98.6%)
out 20: 1388 of 1408 w/in  2.0-sig ( 98.6%)
out 10: 1398 of 1408 w/in  3.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in  4.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in  5.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in  6.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in  7.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in  8.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in  9.0-sig ( 99.3%)
out 10: 1398 of 1408 w/in 10.0-sig ( 99.3%)
out  8: 1400 of 1408 w/in 11.0-sig ( 99.4%)
out  0: 1408 of 1408 w/in 12.0-sig (100.0%)
Sigma Threshold: 4.2 sec
Time  Threshold: 0.333 sec

rank   762 time:    120.597
    Sigma: 11.329  Delta Time:  17.173 sec
rank   762 Node: nid00413:c0-4c0s7n1

rank   763 time:    107.718
    Sigma:  2.833  Delta Time:   4.294 sec
rank   763 Node: nid00413:c0-4c0s7n1

rank   764 time:    120.982
    Sigma: 11.583  Delta Time:  17.557 sec
rank   764 Node: nid00414:c0-4c0s7n2

rank   765 time:    107.609
    Sigma:  2.760  Delta Time:   4.184 sec
rank   765 Node: nid00414:c0-4c0s7n2

rank   766 time:    120.963
    Sigma: 11.571  Delta Time:  17.539 sec
rank   766 Node: nid00415:c0-4c0s7n3

rank   767 time:    107.577
    Sigma:  2.739  Delta Time:   4.152 sec
rank   767 Node: nid00415:c0-4c0s7n3

rank   768 time:    120.540
    Sigma: 11.291  Delta Time:  17.115 sec
rank   768 Node: nid00416:c0-4c1s0n0

rank   769 time:    107.369
    Sigma:  2.603  Delta Time:   3.945 sec
rank   769 Node: nid00416:c0-4c1s0n0

rank   770 time:    120.727
    Sigma: 11.415  Delta Time:  17.302 sec
rank   770 Node: nid00417:c0-4c1s0n1

rank   771 time:    107.699
    Sigma:  2.820  Delta Time:   4.275 sec
rank   771 Node: nid00417:c0-4c1s0n1

rank   772 time:    120.146
    Sigma: 11.032  Delta Time:  16.721 sec
rank   772 Node: nid00418:c0-4c1s0n2

rank   773 time:    107.145
    Sigma:  2.455  Delta Time:   3.721 sec
rank   773 Node: nid00418:c0-4c1s0n2

rank   774 time:    119.935
    Sigma: 10.893  Delta Time:  16.511 sec
rank   774 Node: nid00419:c0-4c1s0n3

rank   775 time:    107.275
    Sigma:  2.541  Delta Time:   3.851 sec
rank   775 Node: nid00419:c0-4c1s0n3

rank   776 time:    120.926
    Sigma: 11.546  Delta Time:  17.502 sec
rank   776 Node: nid00420:c0-4c1s1n0

rank   777 time:    107.679
    Sigma:  2.807  Delta Time:   4.254 sec
rank   777 Node: nid00420:c0-4c1s1n0

rank   778 time:    120.081
    Sigma: 10.989  Delta Time:  16.657 sec
rank   778 Node: nid00421:c0-4c1s1n1

rank   779 time:    107.173
    Sigma:  2.473  Delta Time:   3.748 sec
rank   779 Node: nid00421:c0-4c1s1n1

rank   780 time:    120.134
    Sigma: 11.024  Delta Time:  16.710 sec
rank   780 Node: nid00422:c0-4c1s1n2

rank   781 time:    107.081
    Sigma:  2.413  Delta Time:   3.657 sec
```

```
rank   781 Node: nid00422:c0-4c1s1n2

Total reported bad sigma nodes: 20 of 1408
                  nodes tested.

Global time: 125.376 seconds
```

Inspection of the output shows that defaults were used for the input parameters as outlined above. The per process memory requirements are modest. Core affinity of the first four process shows that 2 processes one on each socket are placed appropriately (using the -S 1 argument). The timing spread is 18.8 seconds and is an indicator that there is a problem. The spread shows that the maximum time is 11.6σ from the average. The specific rank output shows that rank 762 is 11.3σ from the average and the node id and cray node name are printed. Since rank 762 and rank 763 are in the same node Bugget will print out both socket's information if either socket is "bad." Note that both sockets are not "equally" bad. Again the overhead for setting up the calculation and doing the analysis is relatively small.

5 Future Directions

The code has evolved over the last year to do better analysis and be more informative. There are a few things that need to be implemented to provide more information to users. Right now an error status is returned if there are any bad nodes which can be used to abort the job. What is really needed is a mechanism to generate a node list that excludes any identified nodes. This assumes that the application can run on an arbitrary number of processors. We also plan to augment the code to keep track of the compute and communication timings in the algorithm. A histogram based set of data for all aggregated timings will be introduced. Finally, for completeness Pthreads based implementations of the matrix multiply kernels should be developed as well as interfaces to thread parallel libraries.

Acknowledgments

This research was conducted in part under the auspices of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This research used resources of the Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

About the Authors

Ricky A. Kendall is the Group Leader for the Scientific Computing team in the Leadership Computing Facility at Oak Ridge National Laboratory. His team provides the liaison support for users to assist them in adapting applications for the Cray XT systems and to help the science teams to effectively utilize the resources provided. He can be reached at Oak Ridge National Laboratory, PO Box 2008 MS 6008, Oak Ridge, TN 37831-6008 or kendallra@ornl.gov.

Don Maxwell is a Senior HPC System Administrator at Oak Ridge National Laboratory primarily focused on the Cray XT series. He has been a key member of past teams in bringing up new supercomputers for the LCF. He can be reached at Oak Ridge National Laboratory, PO Box 2008 MS 6016, Oak Ridge, TN 37831-6016 or maxwellde@ornl.gov.

Jeff Becklehimer is a Principal Engineer with Cray Inc. He resides at Oak Ridge National Laboratory and is involved in all aspects of the Cray XT computers. He can be reached at jlbeck@cray.com

Cathy Willis is a Systems Engineer IV with Cray Inc. She is assigned as a site analyst at Oak Ridge National Laboratory and is involved in all aspects of support of the Cray XT computers. She can be reached at willis@cray.com.

References

[1] G. C. Fox, A. Hey, and S. Otto, "Matrix Algorithms on the Hypercube I: Matrix Multiplication," Parallel Computing, vol. 4, pp. 17 (1987).

[2] L. E. Cannon, "A cellular computer to implement the Kalman filter algorithm" Ph.D. dissertation, Montana State Univ., Bozeman, MT, (1969).

[3] K. K. Mathur and S. L. Johnsson, "Multiplication of matrices of arbitrary shape on a data parallel computer." Parallel Computing, vol. 20, pp. 919-951, (1994).

[4] E. E. Santos, "Parallel Complexity of Matrix Multiplication." J. Supercomp. vol. 25, pp. 155-175, (2003) .

[5] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage." ACM Trans. Math. Soft. vol. 5, pp. 308-323, (1979).

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh, "ALGORITHM 539, Basic Linear Algebra Subprograms for Fortran Usage." ACM Trans. Math. Soft. vol. 5, pp. 324-245, (1979).

[7] J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms." ACM Trans. Math. Soft. vol. 14, pp. 1-17, (1988).

[8] J. J. Dongarra, J. Du Croz, S. Hammarling and R. J. Hanson, "ALGORITHM 656, An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs." ACM Trans. Math. Soft. vol. 14, pp. 18-32, (1988).

[9] J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms." ACM Trans. Math. Soft. vol. 16, pp. 1-17, (1990).

[10] R. C. Whaley, A. Petitet and J. J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS project." Parallel Computing. vol. 27, pp. 3-35, (2001). Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (http://www.netlib.org/lapack/lawns/lawn147.ps).

[11] R. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm." University of Texas, Department of Computer Sciences, Tech. Rep. TR-95-13, April (1995).

[12] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication." IBM Journal of Research and Development, vol. 38, no. 6, pp. 673-681, (1994).