Introduction to HPL

JUSTIN RAGATZ

1.0 Introduction

One of the major hurdles to getting started with HPL is finding easily digestible documentation and resources. The primary resource for this module is the official HPL website (netlib.org/benchmark/hpl). From this vast collection of information the most essential and digestible components were identified. The purpose of this module is to achieve a working proficiency level with HPL, culminating with a performance and efficiency exercise. This module is not intended for students who have prior experience using HPL. These students would be better served by directly viewing the referencing used in this module in their full detail.

2.0 The TOP500

Statistics on high-performance computers are of major interest to manufacturers, users, and potential users. These people wish to know not only the number of systems installed, but also the location of the various supercomputers within the high-performance computing community and the applications for which a computer system is being used.[1]

Since 1993 the TOP500 project has ranked and detailed the 500 most powerful computer systems in he world twice a year. The rankings are determined by a computer's performance on the LINPACK Benchmark.

The main objective of the TOP500 list is to provide a ranked list of general purpose systems that are in common use for high end applications. Any system designed specifically to solve the LINPACK Benchmark problem or have as its major purpose the goal of a high TOP500 ranking is not eligible.

3.0 The LINPACK Benchmarks

The Linpack package is a collection of Fortran subroutines for solving various systems of linear equations. In the 1979 User's Guide the first LINPACK Benchmark appeared. The original intention was nothing more than to give users an estimate for how long it would take to solve certain matrix problems. Although it was originally included only as a minor feature, the importance of the benchmark has grown every year since its introduction.[2]

Today there are four versions of the LINPACK Benchmark: LINPACK 100, LINPACK 1000, LINPACK Parallel, and HPLinpack. HPL is a portable implementation of the HPLinpack benchmark. Unlike the other versions, HPL allows the user to scale the problem size and optimize the software in order to achieve the best performance possible. Additionally, unlike LINPACK 100 or 1000, HPL is appropriate for distributed-memory computers.

4.0 Introduction to HPL

HPL is a software package that generates, solves, checks and times the solution process of a random dense linear system of equations on distributed-memory computers. The package uses 64-bit floating point arithmetic and portable routines for linear algebra operations and message passing. The outline of HPL's driver code is shown below.[3]

```
/* Generate and partition the matrix data among MPI computing
    nodes */

MPI_Barrier(...); /* Start all nodes at the same time. */

HPL_ptimer(...); /* Start wall-clock timer. */

HPL_pdgesv(...); /* Solve system of equations. */

HPL_ptimer(...); /* Stop wall-clock timer. */

MPI_Reduce(...); /* Obtain the maximum wall-clock time. */

/* Gather statistics about performance rate (base on the
    maximum wall-clock time) and accuracy of the solution. */
```

The goal of the benchmark is to completely fill RAM and to maximize processor use for the duration of the test. To achieve this goal, HPL includes a configuration file that the user may customize to find the ideal parameters for their specific platform.

5.0 HPL Algorithm

5.1 Main Algorithm

HPL generates and solves a linear system of equations of order n: Ax=b, by first computing the LU factorization with row partial pivoting of the n-by-n+1 coefficient matrix [A,b]=[[L,U],y]. Since the lower triangular factor

L is applied to b as the factorization progresses, the solution x is obtained by solving the upper triangular system Ux=y. The lower triangular matrix L is left unpivoted and the array of pivots is not returned.[3]

The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure "good" load balance as well as the scalability of the algorithm. The n-by-n+1 coefficient matrix is first logically partitioned into nb-by-nb blocks, that are cyclically "dealt" onto the P-by-Q process grid. This is done in both dimensions of the matrix.

The right-looking variant has been chosen for the main loop of the LU factorization. This means that at each iteration of the loop a panel of nb columns is factorized, and the trailing submatrix is updated. Note that this computation is thus logically partitioned with the same block size nb that was used for the data distribution.

5.2 Panel Factorization

At a given iteration of the main loop each panel factorization occurs in one column of processes. The user is offered the choice of three (Crout, left- and right-looking) matrix-multiply based recursive variants. The software also allows the user to choose in how many sub-panels the current panel should be divided into during the recursion. Furthermore, one can also select at run-time the recursion stopping criterium in terms of the number of columns left to factorize. When this threshold is reached, the sub-panel will then be factorized using one of the three matrix-vector based variant. Finally, for each panel column the pivot search, the associated swap, and the broadcast operation of the pivot row are all combined into one single communication step. A binary-exchange reduction performs these three operations at once.

5.3 Panel Broadcast

Once the panel factorization has been computed, this panel of columns is broadcast to the other process columns. There are many possible broadcast algorithms and the software currently offers 6 variants to choose from.

5.3.1 Increasing-ring

The classic algorithm. $0 \to 1; 1 \to 2; 2 \to 3 \ldots Q-1 \to Q$.

5.3.2 Increasing-ring (modified)

Process 0 sends two messages and process 1 only receives one message. $0 \to 1; 0 \to 2; 2 \to 3 \ldots Q-1 \to Q$.

5.3.3 Increasing-2-ring

The Q processes are divided into two parts and $0 \to 1$ and $0 \to Q/2$. Then processes 1 and $Q/2$ act as sources of two rings: $1 \to 2, Q/2 \to Q/2+1; 2 \to 3, Q/2+1 \to Q/2+2$ and so on. This algorithm has the advantage of reducing the time by which the last process will receive the panel at the cost of process 0 sending 2 messages.

5.3.4 Increasing-2-ring (modified)

First $0 \to 1$, then the Q-1 processes left are divided into two equal parts: $0 \to 2$ and $0 \to Q/2$; Processes 2 and $Q/2$ act then as sources of two rings: $2 \to 3$, $Q/2 \to Q/2+1; 3 \to 4, Q/2+1 \to Q/2+2$ and so on. This algorithm is probably the most serious competitor to the increasing ring modified variant.

5.3.5 Long (bandwidth reducing)

This algorithm synchronize all processes involved in the operation. The message is chopped into Q equal pieces that are scattered across the Q processes. The pieces are then rolled in Q-1 steps. The scatter phase uses a binary tree and the rolling phase exclusively uses mutual message exchanges. In odd steps $0 \leftrightarrow 1, 2 \leftrightarrow 3, 4 \leftrightarrow 5$ and so on; in even steps $Q-1 \leftrightarrow 0, 1 \leftrightarrow 2, 3 \leftrightarrow 4, 5 \leftrightarrow 6$ and so on. More messages are exchanged, however the total volume of communication is independent of Q, making this algorithm particularly suitable for large messages.

5.3.6 Long (bandwidth reducing modified)

This algorithm is the same as the non-modified version except the $0 \to 1$ first, and then Long variant is used on processes $0, 2, 3, \ldots, Q-1$.

5.4 Look-ahead

Once the panel has been broadcast, the trailing submatrix is updated using the last panel in the look-ahead pipe. This package allows to select various depths of look-ahead. By convention, a depth of zero corresponds to no lookahead, in which case the trailing submatrix is updated by the panel currently broadcasting. Look-ahead consumes some extra memory to essentially keep all the panels of columns currently in the look-ahead pipe. A look-ahead of depth 1 is likely to achieve the best performance gain.

5.5 Update

The update of the trailing submatrix by the last panel in the look-ahead pipe is made of two phases. First, the pivots must be applied to form the current row panel U. U should then be solved by the upper triangle of the column panel. U finally needs to be broadcast to each process row so that the local rank-nb update can take place. We choose to combine the swapping and broadcast of U at the cost of replicating the solve. Two algorithms are available for this communication operation: binary-exchange and long. The long algorithm is used to accomplish the same task with a lower bandwidth.

5.6 Backward Substitution

Once the factorization is complete, back-substitution must be performed. HPL uses look-ahead of depth one to do this. The right hand side is forwarded in process rows so that QN_B entries are solved at a time. At each step, this shrinking piece of the right-hand-side is updated. The process just above the one owning the current diagonal block of the matrix updates its last N_B entries of vector x, forwards it to the previous process column, and then broadcasts it in a decreasing-ring fashion. The solution is then updated and sent to the previous process column. The solution of the linear system is left replicated in every process row

5.7 Checking the Solution

The final step is to verify that the results obtained are correct. The solution is considered correct if the normwise backward error is is less than a threshold value of the order of 1.0

6.0 HPL Tuning

To achieve the best performance possible HPL allows users to edit the input file HPL.dat. The file is 31 lines long and contains the settings for all of the parameters that may affect the programs performance.

6.1 Default HPL.data File

HPL comes with a default input that is useful for a first time run, but not for acheiving optimal performance. The following section will refer to the default HPL.data file and line numbers shown below.[4]

```
01  HPLinpack benchmark input file
02  Innovative Computing Laboratory, University of Tennessee
03  HPL.out       output file name (if any)
04  6             device out (6=stdout,7=stderr,file)
05  4             # of problems sizes (N)
06  29 30 34 35   Ns
07  4             # of NBs
08  1 2 3 4       NBs
09  0             PMAP process mapping (0=Row-,1=Column-major)
10  3             # of process grids (P x Q)
11  2 1 4         Ps
12  2 4 1         Qs
13  16.0          threshold
14  3             # of panel fact
15  0 1 2         PFACTs (0=left, 1=Crout, 2=Right)
16  2             # of recursive stopping criterium
17  2 4           NBMINs (>= 1)
18  1             # of panels in recursion
19  2             NDIVs
20  3             # of recursive panel fact.
21  0 1 2         RFACTs (0=left, 1=Crout, 2=Right)
22  1             # of broadcast
23  0             BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
24  1             # of lookahead depth
25  0             DEPTHs (>=0)
26  2             SWAP (0=bin-exch,1=long,2=mix)
27  64            swapping threshold
28  0             L1 in (0=transposed,1=no-transposed) form
29  0             U in (0=transposed,1=no-transposed) form
30  1             Equilibration (0=no,1=yes)
31  8             memory alignment in double (> 0)
```

6.2 Understanding the HPL.data File

Lines 1 and 2 can be used to summarize the contents of the input file. The text on these lines is ignored.

Line 3 is used to name the output file. If output is set to be redirected to a file on line 4, then a name must be specified here. Text after the first string is ignored and can be used for comments.

Line 4 specifies where the output should go. Set to 6 for standard output and 7 for standard error. Any integer other than 6 or 7 will redirect output to the file specified in line 3.

Line 5 specifies the number of problem sizes that will be executed. This number should be less than or equal to 20. All text after the first integer is ignored.

Line 6 specifies the problem sizes to be run. Any text after the number of integers specified in line 5 will be ignored.

Line 7 specifies the number of block sizes that will be executed. This number should be less than or equal to 20.

Line 8 specifies the block size to run. Any text after the number of integers specified in line 7 will be ignored.

Line 9 specifies how MPI processes will be mapped onto the nodes. If the nodes are single processor, then the mapping should not matter. If the nodes are themselves multi-processor computers, then row-major mapping is recommended.

Line 10 specifies the number of process grids that will be executed. This number should be less that or equal to 20.

Lines 11 and 12 specify the number of process rows and columns of each grid to be run. Any text after the number of integers specified in line 10 is ignored. P refers to the number of rows and Q refers to the number of columns. A square or slightly flat process grid is recommended. Note: there must be at least PQ nodes available for the program to run.

Line 13 specifies the threshold that the residuals will be compared with. A value of 16.0 is recommended and covers almost all cases. If the threshold is negative, then the checks will be bypassed. This may save time during the tuning phase.

Lines 14 to 21 are the panel factorization parameters. The panel factorization is matrix-matrix operation based and recursive, dividing the panel into $NDIV$ subpanels at each step. The recursive part of the panel factorization is denoted by *RFACTs*. The recursion stops when the current column is less than or equal to $NBMIN$ columns. Once this threshold is reached a matrix-vector operation based factorization (*PFACTs*) is used. A LU factorization algorithm variant must be chosen for both $RFACT$ and $PFACT$. The left-looking variant ($left$) computes a column at a time using previously computer columns. The right-looking variant (*Right*) computes a block row and column at each step and uses them to update the trailing submatrix. The *Crout* variant is a hybrid algorithm in which a block row and column is computed to each step using previously computed rows and previously computed columns.

Lines 22 to 23 specify the virtual ring topology that will be used to broadcast the current panel. See section 4.3 for detailed explanations of these algorithms.

Lines 24 to 25 specify the look-ahead depth used by HPL. A depth of k means that the k next panels are factorized immediately after being updated. The update by the current panel is then completely finished. A depth of 1 is recommended for almost all cases.

Lines 26 to 27 specify the swapping algorithm used by HPL for all tests. The three options are binary exchange ($bin - exch$), spread-roll ($long$), and a hybrid algorithm (mix). See section 4.5 for further explanation of this step.

Line 28 specifies whether or not the upper triangle of the panel of columns should be stored in a transposed form.

Line 29 specifies whether or not the panel of rows U should be stored in a transposed form.

Line 30 enables or disables the equilibrium phase.

Line 31 specifies the alignment in memory for the memory space allocated by HPL.

6.3 Getting Started

Tuning HPL to achieve peak performance can be a tedious and time intensive task. One of the best ways to get started is to use one of the well respected

HPL.dat file calculators available online (www.advancedclustering.com/act-kb/tune-hpl-dat-file). The calculator performs the tedious calculations used to approximate the best parameters for a given architecture and allows the user to skip to the fine tuning stage.

7.0 HPL Output

HPL outputs data in a table with the following headers.

T/V	N	NB	P	Q	Time	Gflops

All of these fields are self explanatory except for T/V, which is the encoded variant. An example encoded variant is WR11C2R4. Each character after W is used to encode a different parameter of the variant that was used in a particular run. The meaning of the seven characters is shown below.

PMAP	DEPTH	BCASTS	RFACTS	NDIV	PFACTS	NBMIN

So the example encoded variant WR11C2R4 has the following meaning.

R	$\rightarrow$	Use row-major process mapping.
1	$\rightarrow$	Use a look-ahead depth of 1.
1	$\rightarrow$	Use the increasing-ring broadcasting algorithm.
C	$\rightarrow$	Use the Crout factorization algorithm.
2	$\rightarrow$	Divide into 2 panels at each step.
R	$\rightarrow$	Use right-looking factorization after reaching the threshold.
4	$\rightarrow$	Stop recursion when current column is less than or equal to 4.

Correctly identifying variants is vital when multiple trials with different parameter settings are used during a single run of HPL.

References

1. top500.org/project/introduction.

2. netlib.org/utk/people/JackDongarra/faq-linpack.html.

3. netlib.org/utk/people/JackDongarra/PAPERS/hplpaper.pdf.

4. netlib.org/benchmark/hpl/tuning.html.