HPL Tuning

After having built the executable hpl/bin/<arch>/xhpl, one may want to modify the input data file HPL.dat. This file should reside in the same directory as the executable hpl/bin/<arch>/xhpl. An example HPL.dat file is provided by default. This file contains information about the problem sizes, machine configuration, and algorithm features to be used by the executable. It is 31 lines long. All the selected parameters will be printed in the output generated by the executable.

We first describe the meaning of each line of this input file below. Finally, a few useful experimental guide lines to set up the file are given at the end of this page.

Description of the HPL.dat File

Line 1: (unused) Typically one would use this line for its own good. For example, it could be used to summarize the content of the input file. By default this line reads:

```
HPL Linpack benchmark input file
```

Line 2: (unused) same as line 1. By default this line reads:

```
Innovative Computing Laboratory, University of Tennessee
```

Line 3: the user can choose where the output should be redirected to. In the case of a file, a name is necessary, and this is the line where one wants to specify it. Only the first name on this line is significant. By default, the line reads:

```
HPL.out  output file name (if any)
```

This means that if one chooses to redirect the output to a file, the file will be called "HPL.out". The rest of the line is unused, and this space to put some informative comment on the meaning of this line.

Line 4: This line specifies where the output should go. The line is formatted, it must begin with a positive integer, the rest is unsignificant. 3 choices are possible for the positive integer, 6 means that the output will go the standard output, 7 means that the output will go to the standard error. Any other integer means that the output should be redirected to a file, which name has been specified in the line above. This line by default reads:

```
6        device out (6=stdout,7=stderr,file)
```

which means that the output generated by the executable should be redirected to the standard output.

Line 5: This line specifies the number of problem sizes to be executed. This number should be less than or equal to 20. The first integer is significant, the rest is ignored. If the line reads:

```
3        # of problems sizes (N)
```

this means that the user is willing to run 3 problem sizes that will be specified in the next line.

Line 6: This line specifies the problem sizes one wants to run. Assuming the line above started with 3, the 3 first positive integers are significant, the rest is ignored. For example:

```
3000 6000 10000    Ns
```

means that one wants xhpl to run 3 (specified in line 5) problem sizes, namely 3000, 6000 and 10000.

Line 7: This line specifies the number of block sizes to be runned. This number should be less than or equal to 20. The first integer is significant, the rest is ignored. If the line reads:

```
5        # of NBs
```

this means that the user is willing to use 5 block sizes that will be specified in the next line.

Line 8: This line specifies the block sizes one wants to run. Assuming the line above started with 5, the 5 first positive integers are significant, the rest is ignored. For example:

```
80 100 120 140 160 NBs
```

means that one wants xhpl to use 5 (specified in line 7) block sizes, namely 80, 100, 120, 140 and 160.

Line 9: This line specifies how the MPI processes should be mapped onto the nodes of your platform. There are currently two possible mappings, namely row- and column-major. This feature is mainly useful when these nodes are themselves multi-processor computers. A row-major mapping is recommended.

Line 10: This line specifies the number of process grid to be runned. This number should be less than or equal to 20. The first integer is significant, the rest is ignored. If the line reads:

```
2        # of process grids (P x Q)
```

this means that you are willing to try 2 process grid sizes that will be specified in the next line.

Line 11-12: These two lines specify the number of process rows and columns of each grid you want to run on. Assuming the line above (10) started with 2, the 2 first positive integers of those two lines are significant, the rest is ignored. For example:

```
1 2          Ps
6 8          Qs
```

means that one wants to run xhpl on 2 process grids (line 10), namely 1-by-6 and 2-by-8. Note: In this example, it is required then to start xhpl on at least 16 nodes (max of Pi-by-Qi). The runs on the two grids will be consecutive. If one was starting xhpl on more than 16 nodes, say 52, only 6 would be used for the first grid (1x6) and then 16 (2x8) would be used for the second grid. The fact that you started the MPI job on 52 nodes, will not make HPL use all of them. In this example, only 16 would be used. If one wants to run xhpl with 52 processes one needs to specify a grid of 52 processes, for example the following lines would do the job:

```
4  2         Ps
13 8         Qs
```

Line 13: This line specifies the threshold to which the residuals should be compared with. The residuals should be or order 1, but are in practice slightly less than this, typically 0.001. This line is made of a real number, the rest is not significant. For example:

```
16.0         threshold
```

In practice, a value of 16.0 will cover most cases. For various reasons, it is possible that some of the residuals become slightly larger, say for example 35.6. xhpl will flag those runs as failed, however they can be considered as correct. A run should be considered as failed if the residual is a few order of magnitude bigger than 1 for

example 10^6 or more. Note: if one was to specify a threshold of 0.0, all tests would be flagged as failed, even though the answer is likely to be correct. It is allowed to specify a negative value for this threshold, in which case the checks will be by-passed, no matter what the threshold value is, as soon as it is negative. This feature allows to save time when performing a lot of experiments, say for instance during the tuning phase. Example:

```
-16.0        threshold
```

The remaning lines allow to specifies algorithmic features. xhpl will run all possible combinations of those for each problem size, block size, process grid combination. This is handy when one looks for an "optimal" set of parameters. To understand a little bit better, let say first a few words about the algorithm implemented in HPL. Basically this is a right-looking version with row-partial pivoting. The panel factorization is matrix-matrix operation based and recursive, dividing the panel into NDIV subpanels at each step. This part of the panel factorization is denoted below by "recursive panel fact. (RFACT)". The recursion stops when the current panel is made of less than or equal to NBMIN columns. At that point, xhpl uses a matrix-vector operation based factorization denoted below by "PFACTs". Classic recursion would then use NDIV=2, NBMIN=1. There are essentially 3 numerically equivalent LU factorization algorithm variants (left-looking, Crout and right-looking). In HPL, one can choose every one of those for the RFACT, as well as the PFACT. The following lines of HPL.dat allows you to set those parameters.

Lines 14-21: (Example 1)

```
3       # of panel fact
0 1 2   PFACTs (0=left, 1=Crout, 2=Right)
4       # of recursive stopping criterium
1 2 4 8 NBMINs (>= 1)
3       # of panels in recursion
2 3 4   NDIVs
3       # of recursive panel fact.
0 1 2   RFACTs (0=left, 1=Crout, 2=Right)
```

This example would try all variants of PFACT, 4 values for NBMIN, namely 1, 2, 4 and 8, 3 values for NDIV namely 2, 3 and 4, and all variants for RFACT.

Lines 14-21: (Example 2)

```
2       # of panel fact
2 0     PFACTs (0=left, 1=Crout, 2=Right)
2       # of recursive stopping criterium
4 8     NBMINs (>= 1)
1       # of panels in recursion
2       NDIVs
1       # of recursive panel fact.
2       RFACTs (0=left, 1=Crout, 2=Right)
```

This example would try 2 variants of PFACT namely right looking and left looking, 2 values for NBMIN, namely 4 and 8, 1 value for NDIV namely 2, and one variant for RFACT.

In the main loop of the algorithm, the current panel of column is broadcast in process rows using a virtual ring topology. HPL offers various choices and one most likely want to use the increasing ring modified encoded as 1. 3 and 4 are also good choices.

Lines 22-23: (Example 1)

```
1       # of broadcast
1       BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
```

This will cause HPL to broadcast the current panel using the increasing ring modified topology.

Lines 22-23: (Example 2)

```
2       # of broadcast
0 4     BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
```

This will cause HPL to broadcast the current panel using the increasing ring virtual topology and the long message algorithm.

Lines 24-25 allow to specify the look-ahead depth used by HPL. A depth of 0 means that the next panel is factorized after the update by the current panel is completely finished. A depth of 1 means that the next panel is immediately factorized after being updated. The update by the current panel is then finished. A depth of k means that the k next panels are factorized immediately after being updated. The update by the current panel is then finished. It turns out that a depth of 1 seems to give the best results, but may need a large problem size before one can see the performance gain. So use 1, if you do not know better, otherwise you may want to try 0. Look-ahead of depths 3 and larger will probably not give you better results.

Lines 24-25: (Example 1):

```
1       # of lookahead depth
1       DEPTHs (>=0)
```

This will cause HPL to use a look-ahead of depth 1.

Lines 24-25: (Example 2):

```
2       # of lookahead depth
0 1     DEPTHs (>=0)
```

This will cause HPL to use a look-ahead of depths 0 and 1.

Lines 26-27 allow to specify the swapping algorithm used by HPL for all tests. There are currently two swapping algorithms available, one based on "binary exchange" and the other one based on a "spread-roll" procedure (also called "long" below). For large problem sizes, this last one is likely to be more efficient. The user can also choose to mix both variants, that is "binary-exchange" for a number of columns less than a threshold value, and then the "spread-roll" algorithm. This threshold value is then specified on Line 27.

Lines 26-27: (Example 1):

```
1       SWAP (0=bin-exch,1=long,2=mix)
60      swapping threshold
```

This will cause HPL to use the "long" or "spread-roll" swapping algorithm. Note that a threshold is specified in that example but not used by HPL.

Lines 26-27: (Example 2):

```
2       SWAP (0=bin-exch,1=long,2=mix)
60      swapping threshold
```

This will cause HPL to use the "long" or "spread-roll" swapping algorithm as soon as there is more than 60 columns in the row panel. Otherwise, the "binary-exchange" algorithm will be used instead.

Line 28 allows to specify whether the upper triangle of the panel of columns should be stored in no-transposed or transposed form. Example:

```
0           L1 in (0=transposed,1=no-transposed) form
```

Line 29 allows to specify whether the panel of rows U should be stored in no-transposed or transposed form. Example:

```
0            U  in (0=transposed,1=no-transposed) form
```

Line 30 enables / disables the equilibration phase. This option will not be used unless you selected 1 or 2 in Line 26. Example:

```
1            Equilibration (0=no,1=yes)
```

Line 31 allows to specify the alignment in memory for the memory space allocated by HPL. On modern machines, one probably wants to use 4, 8 or 16. This may result in a tiny amount of memory wasted. Example:

```
8       memory alignment in double (> 0)
```

Guide Lines

1. Figure out a good block size for the matrix multiply routine. The best method is to try a few out. If you happen to know the block size used by the matrix-matrix multiply routine, a small multiple of that block size will do fine. This particular topic is discussed in the FAQs section.

2. The process mapping should not matter if the nodes of your platform are single processor computers. If these nodes are multi-processors, a row-major mapping is recommended.

3. HPL likes "square" or slightly flat process grids. Unless you are using a very small process grid, stay away from the 1-by-Q and P-by-1 process grids. This particular topic is also discussed in the FAQs section.

4. Panel factorization parameters: a good start are the following for the lines 14-21:

   ```
   1        # of panel fact
   1        PFACTs (0=left, 1=Crout, 2=Right)
   2        # of recursive stopping criterium
   4 8      NBMINs (>= 1)
   1        # of panels in recursion
   2        NDIVs
   1        # of recursive panel fact.
   2        RFACTs (0=left, 1=Crout, 2=Right)
   ```

5. Broadcast parameters: at this time it is far from obvious to me what the best setting is, so i would probably try them all. If I had to guess I would probably start with the following for the lines 22-23:

   ```
   2        # of broadcast
   1 3      BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
   ```

   The best broadcast depends on your problem size and harware performance. My take is that 4 or 5 may be competitive for machines featuring very fast nodes comparatively to the network.

6. Look-ahead depth: as mentioned above 0 or 1 are likely to be the best choices. This also depends on the problem size and machine configuration, so I would try "no look-ahead (0)" and "look-ahead of depth 1 (1)". That is for lines 24-25:

   ```
   2        # of lookahead depth
   0 1      DEPTHs (>=0)
   ```

7. Swapping: one can select only one of the three algorithm in the input file. Theoretically, mix (2) should win, however long (1) might just be good enough. The difference should be small between those two assuming a swapping threshold of the order of the block size (NB) selected. If this threshold is very large,

HPL will use bin_exch (0) most of the time and if it is very small (< NB) long (1) will always be used. In short and assuming the block size (NB) used is say 60, I would choose for the lines 26-27:

```
2        SWAP (0=bin-exch,1=long,2=mix)
60       swapping threshold
```

I would also try the long variant. For a very small number of processes in every column of the process grid (say < 4), very little performance difference should be observable.

8. Local storage: I do not think Line 28 matters. Pick 0 in doubt. Line 29 is more important. It controls how the panel of rows should be stored. No doubt 0 is better. The caveat is that in that case the matrix-multiply function is called with (Notrans, Trans, ...), that is C := C - A B^T. Unless the computational kernel you are using has a very poor (with respect to performance) implementation of that case, and is much more efficient with (Notrans, Notrans, ...) just pick 0 as well. So, my choice:

```
0        L1 in (0=transposed,1=no-transposed) form
0        U  in (0=transposed,1=no-transposed) form
```

9. Equilibration: It is hard to tell whether equilibration should always be performed or not. Not knowing much about the random matrix generated and because the overhead is so small compared to the possible gain, I turn it on all the time.

```
1        Equilibration (0=no,1=yes)
```

10. For alignment, 4 should be plenty, but just to be safe, one may want to pick 8 instead.

```
8        memory alignment in double (> 0)
```

[Home] [Copyright and Licensing Terms] [Algorithm] [Scalability] [Performance Results] [Documentation] [Software] [FAQs] [Tuning] [Errata-Bugs] [References] [Related Links]
