Practical HPC benchmarking

Overview

Benchmarking can be a critical part of system design. Often a system must meet certain benchmarks to meet "acceptance". We will be focused on ***synthetic*** benchmarks in this guide. A real benchmarking suite should also include some ***real*** benchmarks, i.e. runs of real programs of interested on well-understood problem sets.

We will go through a handful of common benchmarking techniques through the steps of this guide. We will spend most of the steps working on different micro-benchmarks focused on the performance of sub-components. We will end by running a suite of HPL/Linpack benchmarks that we will analyze.

Step 1: Memory performance benchmarking with Stream

Building and running Stream

Stream is a straight forward and easy to build benchmark. We will use it to get a sense of the memory performance of our nodes.

You can get the Stream benchmark from: https://github.com/jeffhammond/STREAM .

Remember, your nodes probably aren't configured to get directly to the internet, so you should work on your master. It might make sense to set up all of the benchmarking software in your `/home/share/group$i` directory.

```
[lowell@te-master ~]$ git clone https://github.com/jeffhammond/STREAM
Cloning into 'STREAM'...
remote: Enumerating objects: 54, done.
remote: Total 54 (delta 0), reused 0 (delta 0), pack-reused 54
Unpacking objects: 100% (54/54), done.
[lowell@te-master ~]$ cd STREAM/
```

```
[lowell@te-master STREAM]$ ls
HISTORY.txt  LICENSE.txt  Makefile  README  mysecond.c  stream.c  stream.f
```

If you look at the `Makefile` you'll see it specifically references `gcc-4.9` and `gfortran-4.9`. You can simply change these to `gcc` and `gfortran`.

You should be using `gcc` versio `8.3.0`:

```
[lowell@te-master STREAM]$ gcc --version
gcc (GCC) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

This is provided by `lmod`:

```
[lowell@te-master STREAM]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
```

Once the `Makefile` is modified, you should be able to build without trouble:

```
[lowell@te-master STREAM]$ make
gcc -O2 -fopenmp   -c -o mysecond.o mysecond.c
gcc -O2 -fopenmp -c mysecond.c
gfortran -O2 -fopenmp -c stream.f
gfortran -O2 -fopenmp stream.o mysecond.o -o stream_f.exe
gcc -O2 -fopenmp stream.c -o stream_c.exe
```

This makes two executables: `stream_c.exe` (C-based) and `stream_f.exe` (Fortran-based). When I built this, my `stream_f` was unusable, but `stream_c` works fine.

Go ahead and run `stream_c` to test it out:

```
[lowell@te-master STREAM]$ ./stream_c.exe
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
```

```
Total memory required = 228.9 MiB (= 0.2 GiB).
...
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           16103.3     0.010317     0.009936     0.010613
Scale:          14988.0     0.011003     0.010675     0.011660
Add:            17613.6     0.014080     0.013626     0.014446
Triad:          18186.0     0.013529     0.013197     0.013859
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

You'll notice there are results for four tests:

- "Copy" copies an element of memory: `a[i] = b[i]`
- "Scale" multiplies an element of memory by a scalar: `a[i] = s * b[i]`
- "Add" adds two different elements in memory: `a[i] = b[i] + c[i]`
- "Triad" adds two elements while scaling one: `a[i] = b[i] + s * c[i]`

Together, these four numbers should help determine the overall memory bandwidth of the system.

Stream is not very run-time configurable. Generally, configuring parameters involves modifying the code.

A memory performance consistency study

We don't have a good comparison point for Stream. What we can do is run stream across all of our nodes to make sure we get consistent performance:

We can do this with a one-liner `salloc`:

```
[lowell@te-master STREAM]$ salloc -N10 -n10 -c16 --tasks-per-node=1 srun --mpi=none -o stream.out ./stream_c.exe
salloc: Granted job allocation 107
salloc: Relinquishing job allocation 107
```

Note that we redirected output to `stream.out`. We can now compare performance across our nodes:

```
[lowell@te-master STREAM]$ grep Copy stream.out
Copy:           17370.9     0.009223     0.009211     0.009237
Copy:           17605.6     0.009109     0.009088     0.009147
Copy:           17348.0     0.009261     0.009223     0.009279
Copy:           17265.8     0.009291     0.009267     0.009333
Copy:           17617.6     0.009098     0.009082     0.009125
Copy:           17397.0     0.009248     0.009197     0.009286
Copy:           17526.5     0.009145     0.009129     0.009165
Copy:           17357.4     0.009247     0.009218     0.009297
Copy:           17338.6     0.009240     0.009228     0.009253
Copy:           17329.1     0.009269     0.009233     0.009315
```

```
[lowell@te-master STREAM]$ grep Scale stream.out
Scale:          17131.0     0.009357     0.009340     0.009383
Scale:          17215.3     0.009325     0.009294     0.009352
Scale:          16812.1     0.009541     0.009517     0.009582
Scale:          17028.4     0.009408     0.009396     0.009430
Scale:          17258.3     0.009293     0.009271     0.009323
Scale:          17254.3     0.009308     0.009273     0.009334
Scale:          17023.2     0.009425     0.009399     0.009445
Scale:          16890.0     0.009507     0.009473     0.009539
Scale:          16884.9     0.009530     0.009476     0.009589
Scale:          17254.3     0.009299     0.009273     0.009322
[lowell@te-master STREAM]$ grep Add stream.out
Add:            19174.0     0.012545     0.012517     0.012594
Add:            19526.6     0.012318     0.012291     0.012341
Add:            18970.9     0.012675     0.012651     0.012711
Add:            19136.1     0.012573     0.012542     0.012596
Add:            19500.8     0.012333     0.012307     0.012350
Add:            19209.1     0.012528     0.012494     0.012560
Add:            19304.9     0.012454     0.012432     0.012474
Add:            18704.7     0.012854     0.012831     0.012889
Add:            19178.7     0.012555     0.012514     0.012607
Add:            19131.0     0.012574     0.012545     0.012604
[lowell@te-master STREAM]$ grep Triad stream.out
Triad:          19772.8     0.012149     0.012138     0.012174
Triad:          19498.2     0.012334     0.012309     0.012343
Triad:          19663.9     0.012225     0.012205     0.012239
Triad:          19431.6     0.012392     0.012351     0.012431
Triad:          19381.4     0.012399     0.012383     0.012434
Triad:          19646.6     0.012237     0.012216     0.012256
Triad:          19638.6     0.012257     0.012221     0.012283
Triad:          19340.8     0.012431     0.012409     0.012448
Triad:          19550.4     0.012321     0.012276     0.012362
Triad:          19216.8     0.012517     0.012489     0.012546
```

We can see that there is some variance in the results, but nothing that would indicate that any single node performs especially worse than any other.

Try plugging this data into a spreadsheet and computing the `=STDEV()`. In my case, `Add` had a good deal more variance than other operations. The 8th entry is especially low. It might be worth re-running to see if that is consistent.

Step 2: Network performance benchmarking

We will look at two different network performance tools. One (Netperf) will measure performance at the TCP layer, the other will measure low-level Infiniband performance.

Netperf TCP benchmarks

Building Netperf & running

Netperf can be found at: https://github.com/HewlettPackard/netperf

It's a straight-forward build:

```
[lowell@te-master ~]$ git clone https://github.com/HewlettPackard/netperf
Cloning into 'netperf'...
remote: Enumerating objects: 4928, done.
remote: Total 4928 (delta 0), reused 0 (delta 0), pack-reused 4928
Receiving objects: 100% (4928/4928), 15.21 MiB | 11.78 MiB/s, done.
Resolving deltas: 100% (3696/3696), done.
[lowell@te-master ~]$ cd netperf
[lowell@te-master netperf]$ ./autogen.sh
configure.ac:28: installing './compile'
[lowell@te-master netperf]$ mkdir build
[lowell@te-master netperf]$ cd build
[lowell@te-master build]$ ../configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
...
```

Note: `./autogen.sh` will create the `./configure` script for us.

Now build:

```
[lowell@te-master build]$ make -j32
make  all-recursive
make[1]: Entering directory `/home/lowell/netperf/build'
...
make[2]: Leaving directory `/home/lowell/netperf/build'
make[1]: Leaving directory `/home/lowell/netperf/build'
```

We can just use the executables in our `build` directory without installing them:

```
[lowell@te-master build]$ ls src
Makefile          netlib.o          netserver.o     nettest_sctp.o
dscp.o            netperf           netsh.o         nettest_sdp.o
missing           netperf.o         nettest_bsd.o   nettest_unix.o
net_uuid.o        netperf_version.h nettest_dlpi.o  nettest_xti.o
netcpu_procstat.o netserver         nettest_omni.o
```

Netperf works by running `netserver` on one host, and connecting to it with `netperf` on another.

Let's open two terminals to two different nodes.

To start the server on the first node:

```
[lowell@te01 src]$ ./netserver -D -4
Starting netserver with host 'IN(6)ADDR_ANY' port '12865' and family
```

```
AF_INET
```

On the other node we can run our tests:

```
[lowell@te02 src]$ ./netperf -H te01
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to te01 ()
port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.02     941.42
```

By default, with no other options, netperf will perform a bandwidth stream test. As we can see here, I'm getting about `0.941 Gbps` over this connection. That's not surprising because this is a gigabit link.

This can be a good troubleshooting tool when we suspect network issues.

Let's run another test, this time focused on capturing latency:

```
[lowell@te02 src]$ ./netperf -H te01 -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET
to te01 () port 0 AF_INET : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       10.00    11496.13
```

We can put this in the more usual terms of latency with: `10/11496.13 * 10^6 = 869 ms`.

Netperf has quite a few other tests available.

Let's see how the Infiniband performance compares. Keep in mind, this will be the performance of the IPoIB IP emulation layer for Infiniband, not low-level performance:

```
[lowell@te02 src]$ ./netperf -H 192.168.0.1
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
192.168.0.1 () port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.00    15468.81
```

```
[lowell@te02 src]$ ./netperf -H 192.168.0.1 -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET
to 192.168.0.1 () port 0 AF_INET : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       10.00    46968.78
```

So, we're getting about `15.4 Gbps` bandwidth, and `213 ms` latency. This is nowhere near advertised IB specs, but clearly much better than the ethernet. The low performance is due to the emulation layer.

Infiniband bandwidth & latency benchmarks

The OFED stack we installed for InfiniBand came with a couple of tools that can help us get these numbers without the emulation layer. Let's use them to compare.

Let's do bandwidth tests first:

On one node we run `ib_write_bw` with no host specified, on the other `ib_write_bw <host>`:

```
[root@te01 ~]# ib_write_bw -F

************************************
* Waiting for client to connect... *
************************************
```

On the client side:

```
[root@te02 ~]# ib_write_bw -F 192.168.0.1
---------------------------------------------------------------------------
---
                    RDMA_Write BW Test
 Dual-port       : OFF          Device         : mlx5_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 TX depth        : 128
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 0[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------
---
 local address: LID 0x03 QPN 0x3b51 PSN 0x196d8d RKey 0x5bf0b9 VAddr
0x007f1675c70000
 remote address: LID 0x01 QPN 0x680b PSN 0x3d5056 RKey 0x904008 VAddr
```

```
0x007f3fc2170000
---------------------------------------------------------------------------
---
 #bytes     #iterations    BW peak[MB/sec]    BW average[MB/sec]
MsgRate[Mpps]
 65536      5000             11812.36            11811.81
0.188989
---------------------------------------------------------------------------
---
```

That's about `88 Gbps`. The theoretical peak is `25 * 4x Gbps ~= 100 Gbps`. Not too bad.

Let's test latency:

```
[root@te02 ~]# ib_write_lat -F 192.168.0.1
---------------------------------------------------------------------------
---
                    RDMA_Write Latency Test
 Dual-port       : OFF          Device         : mlx5_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 TX depth        : 1
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 220[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------
---
 local address: LID 0x03 QPN 0x3b52 PSN 0xedf8 RKey 0x2f95f9 VAddr
0x007f4a191fd000
 remote address: LID 0x01 QPN 0x680c PSN 0x1bd375 RKey 0x2e95ce VAddr
0x007f887710d000
---------------------------------------------------------------------------
---
 #bytes #iterations    t_min[usec]    t_max[usec]  t_typical[usec]
t_avg[usec]    t_stdev[usec]   99% percentile[usec]   99.9%
percentile[usec]
 2       1000          0.79           7.98         0.80
0.81            0.13           0.82       7.98
---------------------------------------------------------------------------
---
```

That's about `0.8 usec` latency. Theoretical is `0.61 usec` for EDR InfiniBand.

Step 3: MPI benchmarks (IMB)

The Intel MPI Benchmarks measure over-all MPI performance. This is largely a test of the network fabric but also tests MPI library speed.

We can get the IMB software from: https://github.com/intel/mpi-benchmarks

```
[lowell@te-master ~]$ git clone https://github.com/intel/mpi-benchmarks
Cloning into 'mpi-benchmarks'...
...
```

We will only build the C benchmarks, not the C++.

```
[lowell@te-master ~]$ cd mpi-benchmarks/src_c
[lowell@te-master src_c]$ make -j32
...
mpicc -DMPIIO -DIO -DIMB2018 -c IMB_init_transfer.c -o
build_IO/IMB_init_transfer.o
mpicc -DMPIIO -DIO -DIMB2018 -c IMB_chk_diff.c -o build_IO/IMB_chk_diff.o
mpicc  build_IO/IMB.o build_IO/IMB_utils.o build_IO/IMB_declare.o
build_IO/IMB_init.o build_IO/IMB_mem_manager.o build_IO/IMB_init_file.o
build_IO/IMB_user_set_info.o build_IO/IMB_benchlist.o
build_IO/IMB_parse_name_io.o build_IO/IMB_strgs.o
build_IO/IMB_err_handler.o build_IO/IMB_g_info.o build_IO/IMB_warm_up.o
build_IO/IMB_output.o build_IO/IMB_cpu_exploit.o build_IO/IMB_open_close.o
build_IO/IMB_write.o build_IO/IMB_read.o build_IO/IMB_init_transfer.o
build_IO/IMB_chk_diff.o -o IMB-IO
make[1]: Leaving directory `/home/lowell/mpi-benchmarks/src_c'
```

Once completed, we should submit these jobs using Slurm:

```
[lowell@te-master src_c]$ salloc -N2 -n2 srun --mpi=pmix ./IMB-MPI1
AllReduce
...
# Allreduce

#----------------------------------------------------------------
# Benchmarking Allreduce
# #processes = 2
#----------------------------------------------------------------
       #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
            0         1000         0.04         0.04         0.04
            4         1000         1.20         1.76         1.48
            8         1000         0.84         2.20         1.52
           16         1000         0.94         2.11         1.52
           32         1000         1.18         1.88         1.53
           64         1000         1.36         1.81         1.58
          128         1000         1.88         2.43         2.15
          256         1000         2.27         2.32         2.30
          512         1000         2.04         2.92         2.48
         1024         1000         2.13         3.49         2.81
         2048         1000         2.59         4.35         3.47
         4096         1000         4.08         6.11         5.09
```

```
          8192         1000         6.74         8.44         7.59
         16384         1000        13.21        13.34        13.28
         32768         1000        21.74        22.03        21.88
         65536          640        31.02        32.32        31.67
        131072          320        61.93        63.27        62.60
        262144          160       116.15       117.10       116.63
        524288           80       209.87       211.98       210.93
       1048576           40       391.60       393.57       392.59
       2097152           20       762.43       764.56       763.49
       4194304           10      1574.83      1577.18      1576.00
```

There are many different MPI tests available in IMB, one for each type of MPI operation. We can run a whole set up tests with:

```
[lowell@te-master src_c]$ salloc -N10 -n10 srun --mpi=pmix ./IMB-MPI1 -
multi 0
...(huge amount of output)
```

As we can see from both of these results, the latency increases a bit in a realistic MPI operation.

A good comparison (***Challenge***) would be to build IMB using Intel MPI instead of OpenMPI and compare the results.

Step 4: HPL benchmarks

Undoubtedly the most ubiquitous benchmark in HPC is the HPL ("High Performance Linpack") benchmark. This is the benchmark that top500.org uses to rank systems. It's long been considered the gold standard for macro system performance, but that has become more debated in recent users, and benchmarks like the HPCG have become popular.

Building HPL

HPL can be a bit tricky to compile. It can be even trickier if you are trying to tweak every bit of performance out of it. You can grab the latest copy of HPL at: https://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz.

```
[lowell@te-master ~]$ wget https://www.netlib.org/benchmark/hpl/hpl-
2.3.tar.gz
--2019-06-17 00:51:00--  https://www.netlib.org/benchmark/hpl/hpl-
2.3.tar.gz
Resolving www.netlib.org (www.netlib.org)... 160.36.131.221
Connecting to www.netlib.org (www.netlib.org)|160.36.131.221|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 660871 (645K) [application/x-gzip]
Saving to: ‘hpl-2.3.tar.gz.1’

100%
[======================================================================
```

```
=>] 660,871     1.46MB/s   in 0.4s

2019-06-17 00:51:01 (1.46 MB/s) - ‘hpl-2.3.tar.gz’ saved [660871/660871]
```

Extract it:

```
[lowell@te-master ~]$ tar zxvf hpl-2.3.tar.gz
...
hpl-2.3/www/spreadM.jpg
hpl-2.3/www/tuning.html
```

I have provided a `Make.intel64` file in `/data/` on `te-cm`. You will need to place this file in the root of the `hpl-2.3` directory to build HPL.

Once the file is in place, we need to make sure that `openblas` is loaded in our `lmod` environment:

```
[lowell@te-master hpl-2.3]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc

[lowell@te-master hpl-2.3]$ module load openblas
[lowell@te-master hpl-2.3]$ module list
Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
6) openblas/0.3.5
```

Now we can build HPL with:

```
[lowell@te-master hpl-2.3]$ make arch=intel64
...
make[3]: Leaving directory `/home/lowell/hpl-2.3/testing/ptest/intel64'
touch dexe.grd
make[2]: Leaving directory `/home/lowell/hpl-2.3/testing/ptest/intel64'
make[1]: Leaving directory `/home/lowell/hpl-2.3'
```

Note: using the `-j <num>` option with make seems to break the HPL build process.

If anything goes wrong with the build, you'll want to clean with:

```
[lowell@te-master hpl-2.3]$ make arch=intel64 clean_arch_all
...
```

Once HPL is built, you can find it under `bin/intel64`:

```
[lowell@te-master hpl-2.3]$ cd bin/intel64/
[lowell@te-master intel64]$ ls
HPL.dat  xhpl
```

The executable is `xhpl`. It is an MPI executable.

The file `HPL.dat` is where you put configuration. The provided version will run a simple 4-processor example. Let's run it to make sure things are working:

```
[lowell@te-master intel64]$ mpirun -np 4 ./xhpl
...(dumps a huge amount of output)
```

The summary lines for each run start with `WR<xxxx>`. Let's get something more meaningful by grepping for that:

```
[lowell@te-master intel64]$ mpirun -np 4 ./xhpl | grep -E '^WR'
...
WR00C2L4          35     4     4     1               0.00
4.9069e-02
WR00C2C2          35     4     4     1               0.00
3.7408e-02
WR00C2C4          35     4     4     1               0.00
4.5927e-02
WR00C2R2          35     4     4     1               0.00
4.5384e-02
WR00C2R4          35     4     4     1               0.00
3.5792e-02
WR00R2L2          35     4     4     1               0.00
3.5779e-02
WR00R2L4          35     4     4     1               0.00
4.3305e-02
WR00R2C2          35     4     4     1               0.00
4.8408e-02
WR00R2C4          35     4     4     1               0.00
4.8844e-02
WR00R2R2          35     4     4     1               0.00
4.7717e-02
WR00R2R4          35     4     4     1               0.00
3.9889e-02
...
```

The last column here is a measurement of GFlops. We don't expect high numbers here, because we're running on only 4 cores.

The `HPL.dat` file

Take a look in the `HPL.dat` file:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
4            # of problems sizes (N)
29 30 34 35  Ns
4            # of NBs
1 2 3 4      NBs
0            PMAP process mapping (0=Row-,1=Column-major)
3            # of process grids (P x Q)
2 1 4        Ps
2 4 1        Qs
16.0         threshold
3            # of panel fact
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
2            # of recursive stopping criterium
2 4          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
3            # of recursive panel fact.
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

There are a lot of parameters in here, but only a few that we care about.

We won't use any of the output file parameters. In the end, we'll let Slurm take care of outputting results to a file for us.

The `N` field ("# of problem sizes") specifies how many of the following "NBs" we will be giving. In this case, we have 4. All of the `#` fields work like this, specifying how many of the next field type will be specified.

The `Ns` specifies the problem size. This ultimately controls how big the matrix we're going to work with is.

The next part that especially matters is the "process grids". This says how HPL will split up the problem across nodes and cores. So, if you want to run across a total of 10 nodes and 32 cores, that's 320 cores, and the process grid should multiply out to 320. The general rule of thumb is that you want to be as close to "square" as you can, i.e. `P` close to `Q`. So, for 320, 16 x 20 is likely the best choice. Usually, you want `P < Q`.

HPL has been studied for many years, and a lot of patterns have been found. If you go to https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/ there is a reasonably good calculator for estimating the best `HPL.dat` file.

Another good resource is: http://hpl-calculator.sourceforge.net. This calculator doesn't provide you with the `HPL.dat`, but it will tell you how close you get to the theoretical max.

Making a Slurm job for HPL

The real HPL jobs will run for a little while. We'll want to submit them as batch jobs. Let's put together an example batch job and try it out. This will also give us more experience with `sbatch` scripts, especially with `mpi`.

The first thing to know is that `srun` is MPI aware. When we tested out our build of HPL, we ran it inside of the `mpirun` command and gave it a number of processes. This is the traditional way to start an MPI process, along with a hostlist file that would indicate which hosts to use. This is how you would start an MPI job if we didn't have a WLM system.

`srun` handles all of this bring-up for us. We don't even have to say which hosts to run on, or how many processes; it will take care of all of that.

We *do* have to think carefully about how we want to arrange our processes. HPL uses MPI for ***all*** of its communication, unlike some programs that combine MPI with multi-threading for instance. This means that Slurm needs to start an HPL process for each core we want to run on.

To get started, we'll make a 2-node job. That will be 2-nodes, and 64 tasks. Make a directory called `2-node` under `bin/intel64` under the HPL directory. We will run in this directory:

```
[lowell@te-master ~]$ cd hpl-2.3/bin/intel64/
[lowell@te-master intel64]$ ls
HPL.dat  xhpl
[lowell@te-master intel64]$ mkdir 2-node
[lowell@te-master intel64]$ cd 2-node/
```

First, create a 2-node `HPL.dat` file. You can use the calculator from above. The system has 32 cores, 2 nodes, and 64GB RAM. You can leave the block size, `NB`, as the default. Write the result to `HPL.dat` in the `2-node` directory.

Now, create a script called `hpl.sbatch`. Given it the following contents:

```
#!/bin/bash
#SBATCH --job-name=hpl-32x2
#SBATCH --ntasks=64
#SBATCH --nodes=2
#SBATCH --output=hpl_%j.log

pwd; hostname; date

module load openmpi3 openblas

srun --mpi=pmix ../xhpl

date
```

The sbatch parameters we gave provide a unique job name, that we need 64 tasks across 2 nodes, and say that the output should go to `hpl_<jobid>.log`, where `<jobid>` is the numerical job id that slurm assigns when the job runs. Using the `%j` keeps us from over-writing the output file if we run multiple times.

The next line, `pwd; hostname; date` are just to provide some convenient bookkeeping for possible later troubleshooting.

The real command is `srun --mpi=pmix ../xhpl`. `xhpl` will use the `HPL.dat` that sits in the current working directory. The `--mpi=pmix` option tells Slurm to use `pmix` for process management (there are various options, but this is the one we're set up for).

We can now `sbatch` our job:

```
[lowell@te-master 2-node]$ sbatch hpl.sbatch
Submitted batch job 115
[lowell@te-master 2-node]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
               115   cluster hpl-32x2   lowell  R       0:03      2 te[01-
02]
[lowell@te-master 2-node]$ ls
hpl_115.log  HPL.dat  hpl.sbatch
```

Our job is running. We can tail its output:

```
[lowell@te-master 2-node]$ tail hpl_115.log

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be
1.110223e-16
- Computational tests pass if scaled residuals are less than
16.0

Column=000000192 Fraction= 0.2% Gflops=9.465e+03
Column=000000384 Fraction= 0.3% Gflops=1.402e+03
Column=000000576 Fraction= 0.5% Gflops=1.085e+03
```

This will take a while to run.

An HPL scaling study

Now that `2-node` is running, make a similar directory for `1-node`, `4-node`, `8-node`, and `10-node`.

If you use the calculator for the `10-node` it will give you an improper config. You can use what it gives you, but change the `Ns` line to be the same is what you have for `8-node`.

Once those are all set up, `sbatch` them.

I started all of mine at once like this:

```
[lowell@te-master intel64]$ for i in 1 2 4 8 10; do echo $i; ( cd $i-node
; sbatch hpl.sbatch ) ; done
1
Submitted batch job 116
2
Submitted batch job 117
4
Submitted batch job 118
8
Submitted batch job 119
10
Submitted batch job 120
[lowell@te-master intel64]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
               119   cluster hpl-32x8   lowell PD       0:00      8
(Resources)
               120   cluster hpl-32x1   lowell PD       0:00     10
(Priority)
               116   cluster hpl-32x1   lowell  R       0:02      1 te01
               117   cluster hpl-32x2   lowell  R       0:02      2 te[02-
03]
               118   cluster hpl-32x4   lowell  R       0:02      4 te[04-
07]
```

At this point, you should verify that all of the jobs are starting without error by checking the `hpl_*.log` files.

This will take a while, and we will likely work on something else for a bit while it runs.

Once it has finished, get all of the final values. We're going to want to plot these to see if we are scaling linearly. We can collect the values with:

```
[lowell@te-master intel64]$ grep WR */hpl_*.log
10-node/hpl_103.log:WR11C2R4      234240   192    16    20
2462.54             3.4795e+03
1-node/hpl_93.log:WR11C2R4       82560   192     4     8
1021.66             3.6722e+02
2-node/hpl_95.log:WR11C2R4      117120   192     8     8
1469.01             7.2910e+02
4-node/hpl_96.log:WR11C2R4      165504   192     8    16
2094.34             1.4431e+03
8-node/hpl_97.log:WR11C2R4      234240   192    16    16
3038.27             2.8201e+03
```

Now, put these values into `Microsoft Excel`. Make an X-Y scatter plot (will demo how to do this live). Do you get linear scaling? Are 10 nodes 10x as fast as 1?