High-Performance Linpack Optimization

Gurusiddesha Chandrasekhara
Department of Informatics
Technische Universität München
guru.rvcs@gmail.com

Madhukar Sollepura Prabhu Shankar
Department of Informatics
Technische Universität München
madhutime@gmail.com

Madhura Kumaraswamy
Department of Informatics
Technische Universität München
madhura.kswamy@gmail.com

Lukas Grillmayer
Department of Informatics
Technische Universität München
lukas.grillmayer@in.tum.de

February 22, 2015

Abstract

The purpose of this project is to automate the process of tuning parameters for the Linpack benchmark to obtain the optimal configuration. A Persicope plugin was created to perform this task. The plugin was then tested on consumer grade machines and built on the SuperMUC supercomputer. Results show the best scenario for manipulation of a set of parameters.

1 Introduction

Supercomputers embody the climax of high-performance computing (HPC) systems. They provide computation power for governments, industries, and researchers. Applications include the general areas of cryptanalysis, simulation, and modeling. With advancing technology the need for more computation power increases constantly in order to solve more complex problems. Demand-driven, the computation capabilities of supercomputers increase rapidly and many institutions compete for the most powerful system. The Top500 project biannually updates a list of the 500 best supercomputers based on the LINPACK benchmark, which measures the system's floating-point rate of execution. At the time of writing this paper, the best supercomputer is "Tianhe-2" at the National Super Computer Center in Guangzhou, China scoring 33.86 petaflops/s [1].

LINPACK is a library which solves a dense system of linear equation in double precision on distributed-memory systems. The theoretical peak performance of a system is generally not achieved. For different problem sizes the actual performance is measured and maximized. Along with the problem size, the benchmark result is also influenced by other parameters which are discussed later.

1.1 Project Description

The High Performance Linpack (HPL) benchmark has a configurable file called HPL.dat, which contains several parameters that may possibly impact the performance of a supercomputer. The goal of this project is to firstly, build HPL on the SuperMUC petascale system, and then, to identify which parameters impact the performance, with special regard to block sizes, type of process mapping (row-major or column-major) and type of swapping algorithm, and to determine how the problem size impacts scalability. Finally, to create a Periscope Tuning Framework (PTF) plugin that automates a new HPL.dat creation using ”sed” (stream editor) and tunes the parameters in the SuperMUC system.

1.2 Outline

In section 2 tools are introduced which were used in the course of this project. Secondly, we discuss the tuning process and available benchmark parameters in detail before determining an initial configuration for a consumer grade laptop and SuperMUC. The implementation and functioning of the periscope plugin is described in section 4. Next, the benchmark results are presented, highlighting the impact of the automated tuning process. In conclusion we summarize our findings and discuss topics for future work.

2 Tools

In this section, numerous tools are presented, which were used in the process of this project. This is list is non-exhaustive, so the scope was set to the most important programs.

2.1 Periscope

Periscope is an online distributed performance analysis tool[10] developed at the Technische Universität München that is used for dynamic tuning, which is a technique to automatically analyze performance issues of parallel MPI applications to evaluate single node performance[10]. Periscope comprises an analysis agent network, consisting of three agents, namely, the master agent, communication agents and analysis agents, which search for performance problems in one or more experiments, called iterations and use a monitoring interface to congure the measurements. After the end of the search, the detected problems are communicated to the master agent through the communication agents and the results of the performance problem detection are displayed on a user-interface.

The Periscope Tuning Framework (PTF) was developed under the AutoTune project [11] to combine and automate analysis and optimization of aspects such as energy consumption, inter-process communication and load balancing. PTF identies tuning variants based on expert knowledge, evaluates the variants online and then, produces a report, which describes where and how to improve the code. PTF provides plugins for tuning high-level patterns for GPGPUs, HMPP codelets, MPI runtime, compiler flag selection and the energy consumption [11].

2.2 BLAS

Basic Linear Algebra Subprograms (BLAS) are a collection of routines which provide basic vector and matrix operations. It is efficient and portable, which makes it suitable for usage in high performance calculations. It is required by HPL [4].

2.3 ATLAS

Automatically Tuned Linear Algebra Software (ATLAS) provides a complete BLAS API for C and Fortran77. For all operations it optimizes performance for a specific machine, allowing it to execute faster than plain BLAS. Since HPL requires BLAS it can be substituted by ATLAS [5].

2.4 Open MPI

OpenMPI is an open source implementation of MPI (Message Passing Interface), a message passing library specification commonly used in the HPC community. Maintained by numerous academic, research, and industry partners this platform independent interface became the industry-standard for parallel programs in the last decade and has reached version 3.0 in September 2012. As it is designed for portability, efficiency, and flexibility, it runs on virtually any distributed memory, shared memory, and hybrid systems. Among others, the MPI interface has been implemented for C, C++, Fortran, and Python [12, 13].

MPI uses Communicator and Group objects to form a communication domain for a set of processes. A Communicator assigns a unique identifier to each process, which can be used to specify source and destination of messages. By default the program aborts if MPI encounters and error.

2.5 High-Performance Linpack

LINPACK addresses scalability in distributed memory systems by providing additional testing rules and environment through the Highly-Parallel LINPACK (HPL) NxN benchmark, which is used for measuring supercomputer performance in the Top500 list. HPL is a portable implementation of the HPL NxN benchmark, and generates and solves a random dense linear system of equations using Gaussian Elimination with partial pivoting and double precision floating point arithmetic. The HPL package is written in C and requires the implementation of either the BLAS library, which can be automatically generated by the ATLAS tool, or the Vector Signal Image Processing Library (VSIPL). The HPL package provides testing rules to obtain the accuracy of the obtained solution, and timing rules to obtain the computation time to determine the best performance scenario.

The computational steps performed by HPL to obtain the benchmark rating are :

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

/* All the nodes start at the same
    time. */
MPI_Barrier(...);

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

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

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

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

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

2.5.1 Scalabilty

Scalability is an important aspect that must be considered while running problems on thousands of nodes. Strong scaling is the process of fixing a workload or the problem size and increasing the number of processors so that a program scales linearly when the speedup increases with the number of processing elements used. With respect to the total energy consumption, embarrassingly parallel codes achieve better energy efficiency with increasing system size[1]. Weak scaling is the process of increasing both the workload or the problem size and the number of processors so that the program scales linearly when the run-time stays constant while increasing the workload or problem size proportionally to the number of processors. HPL is a good weakly scalable application and gets more efficient with increasing memory [10].

2.6 Installation Script

We created a installation script (see appendix A) which automates the installation process of Periscope and HPL with all dependents. It was created for deployment on a clean installation of Ubuntu 14.04 LTS 64bit Desktop using VirtualBox. We suggest a hard disk size of 16GB or larger. The installation process also depends on the system properties, the installation script can serve as useful guideline.

3 Tuning

As discussed before HPL needs a set of parameters to derive a scenario for benchmarking. This is done via the HPL.dat file, which lists all available options. If multiple values are provided for individual parameters, HPL will execute all possible combinations in a separate scenario. In the following, parameters relevant for benchmarking are presented [6, 8].

3.1 Parameters

- Problem size (N): The problem should be as large as possible, whereas N denotes the size of the coefficient matrix. If the problem size exceeds the available memory, performance will decrease due to swapping. It is suggested to initially use about 80% of available memory.

$$N = sqrt((\#nodes * \#memoryPerNode * 0.8)/sizeOf(double))$$

- Block size (NB): It is used for data distribution and computational granularity. Small values are desired as data distribution and load balancing improves. If chosen too small, the message overhead will increase and the available cache hierarchy might not be used properly. A value between 32 and 256 is suggested for modern systems.
- Process grid size (PxQ): The number of process rows (P) and columns (Q) depends on the number of available nodes and the physical network connection. It defines the number of nodes onto which HPL data should be distributed and executed, the process grid. Q must be slightly larger or equal to P. For simple Ethernet networks P should be very small since network scaling is limited. $P*Q$ must be equal to the number of nodes in the system. If surpassed, additional nodes will not be utilized. The suggested range is:

$$P \leq Q \leq 3*P$$

- Residual threshold: The residuals of the main algorithm will be compared to this value. The threshold is a real number of magnitude 1, whereas for negative values this threshold will not be checked. A suggested value is 16.0.
- Recursive factorization method (RFACT): Panel factorization is a matrix-matrix operation, which divides a large matrix into several submatrices. This is also known as LU decomposition. The left-looking and right-looking variants of this algorithm comply to the Doolittle method, dividing the matrix into lower and upper triangular unit matrices. The Crout variant is similar, but the returned lower matrix is not a unit matrix.
- Panels in recursion (NDIV): During each iteration of panel factorization the current matrix is divided into NDIV submatrices.
- Recursion stopping criteria (NBMIN): The panel factorization recursion stops when the matrix is is composed of NBMIN columns or less.
- Panel factorization method (PFACT): After the factorization recursion stops, the algorithm continues with LU decomposition on a matrix-vector basis using a left-looking, Crout, or right-looking approach defined by RFACT.
- Lookahead depth (DEPTH): The lookahead depth defines the number of next panels which should be factorized immediately after being updated. For large problem sizes a value of 1 is suggested.
- Panel broadcasting method (BCAST): After panel factorization is complete, the resulting values are sent to other process columns using messages. Using different approaches leads to changes in execution time and throughput. There are multiple broadcast algorithms available:
  - Increased-ring: Every process has to send a message the next process consequently.
  - Increased-ring (modified): Process 0 sends a message to process 1 and 2 each.
  - Increasing-2-ring: Processes are split into two groups, whereas process 0 sends a message to the second group and acts as the first element of the first group.
  - Increasing-2-ring (modified): The modification to increasing-2-ring is to additionally apply the increasing-ring algorithm to the first group.
  - Long (bandwidth-reducing): The message is divided into Q equal pieces to be distributed to Q different processes in a synchronized fashion. The submessages are

then exchanged between sets of two processes until every process received all parts of the original message.

  – Long (bandwidth-reducing modified): The modification to long (bandwidth reducing) is to additionally apply the increasing-ring algorithm.

  The best method depends on the problem size and hardware performance. If the platform nodes are expected to outperform the network, "Long (bandwidth reducing)" (Lng) or "Long (bandwith reducing modified)" (LnM) can be recommended.

- Process mapping(PMAP): This parameter specifies how the MPI processes are mapped to the nodes. It is recommended to use row-major mapping for systems with multi-processor computer nodes.

- Swap algorithm (SWAP): This parameter specifies the swapping algorithm used during all tests. Available algorithms are based on binary-exchange (bin) and spread-roll (long). For mix, the binary-exchange will be used for a number of columns less than the swapping threshold before switching to spread-roll. The swapping threshold is only used for mix. For bin and long, the equilibrium phase can be enabled / disabled.

- Storage of panel triangle (L1, U): These parameters specify whether the lower (L1) and upper (U) triangle matrix should be stored in no-transposed or transposed form.

- Memory alignment: This parameters specifies the memory alignment HPL uses to allocate memory. The value is measured in double.

For the purpose of this project following parameters were considered for optimization:

- Problem size (N)
- Block size (NB)
- Process grid size (PxQ)
- Process mapping (PMAP)

Appendix B shows the optimal configuration for a Lenovo G510 laptop.

4 Implementation

A Periscope plugin must implement the "Tuning Plugin Interface (TPI)". This interface requires developers to follow the generic flow displayed in figure 1. As some steps are optional only the steps relevant to the implementation of this project's plugin are discussed in this section.

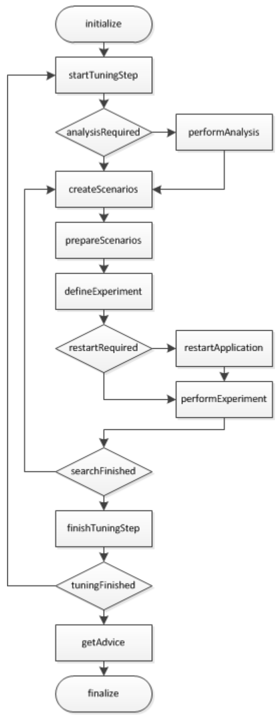

Figure 1: PTF plugin flow [14]

- Initialize: First set the context and pool_set references, passed by the runtime system to the plugin, define the search algorithm and set the tuning parameters in the following way.

```
//Problem Size ranging from 6000
    to 10000 [Increasing in steps
    of 1000]
TuningParameter *inputSizeTP = new
     TuningParameter();
inputSizeTP->setId(0);
inputSizeTP->setName("PROBLEMSIZE
    ");
inputSizeTP->setRange(6000, 10000,
     1000); // (start, max, step)
tuningParameters.push_back(
    inputSizeTP);
```

- Start Tuning: Create instances of Variant space and Search space. Add each tuning parameter to variant space and set variant space to search space. Finally add search space to search algorithm.
- Analysis Required: Perform analysis of the application in this step. If no analysis is not required, then return false.
- Create Scenario: Create the scenario for the application and place it in create scenario pool.
- Prepare scenario: Retrieve the scenario from create scenario pool. Extract the integral value from it. Use these values to update the application specific file. Example: For HPL application, we are updating HPL.dat file with necessary value using 'sed' command. It's as shown below. Later push the scenario to prepared scenario pool.

```
const Variant* v = scenario->
    getTuningSpecifications()->
    front()->getVariant();
map<TuningParameter*,int> tpValues
     = v->getValue();
long problemSize = tpValues[
    tuningParameters[0]];
long blockSize = tpValues[
    tuningParameters[1]];
long pMapping = tpValues[
    tuningParameters[2]];
long qMapping = tpValues[
    tuningParameters[3]];
long procMapping = tpValues[
    tuningParameters[4]];

sprintf(cmnd,"sed -e '6s/.*/%ld/'
    -e '8s/.*/%ld/' -e '11s/.*/%ld
    /' -e '12s/.*/%ld/' -e '9s/.*/%
    ld/' -e '3s/.*/HPL%d.out/' <HPL
    .dat.sed >HPL.dat ",
    problemSize, blockSize,
    pMapping, qMapping, procMapping
    , hplcounter++);
```

- Define experiment: Get scenario from prepared scenario pool and push it to experiment scenario pool. After this operation, the runtime system performs the experiment to acquire the specified properties. In our case it acquired execution time.
- Restart Required: Specify restart is required for next experiment by returning value as true, otherwise false.
- Search Finished: Return true if search is finished, otherwise false.
- Finish Tuning: The post processing operation are performed here, before entering to next tuning iteration.
- Tuning finished: Return true, if plugin has finished tuning process, otherwise false.
- Get Advice: Provide resulting information to the user in a file. In our case, we extracted the Gflops value from a temporary result file using a user defined function 'extract'. Following command was used to extract time and Gflops information from output file.

```
sprintf(cmnd,"awk '/WR/{print %d
    \"\\t\"$6 \"\\t\"$7}' HPL%d.out
     >> TimeAndGflops", (hplcounter
     - 1),(hplcounter - 1));
```

- Finalize: All created objects are freed in this method.

5 Evaluation

The plugin was executed on a Lenovo G510 laptop (Intel i7-4700MQ @ 2.4 GHz, 8.0GB PC3-12800 DDR3 SDRAM @ 1.6Ghz) conducting 270 experiments with various parameter configurations. It was determined that the configuration shown in appendix B yielded the best results, with an actual performance of 30.13 Gflops. The following subsections show the results of this benchmark in detail.

5.1 Problem Size

The problem size for a system should be as large as possible (see section 3.1). Figure 2 shows that a problem size of 10,000 yielded the best results. It depicts the impact of the problem size ordered by performance:

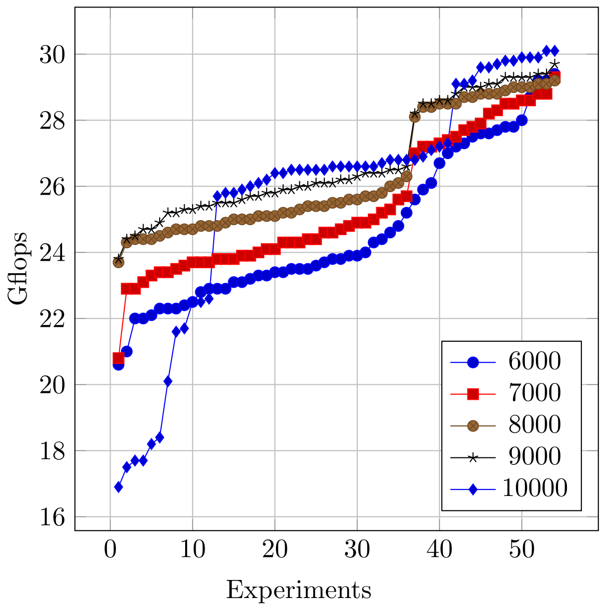

Figure 2: Performance impact of problem size

5.2 Block Size

Figure 2 indicates that block size has only a minor impact on performance readings. The best performance was reached with a block size of 160.

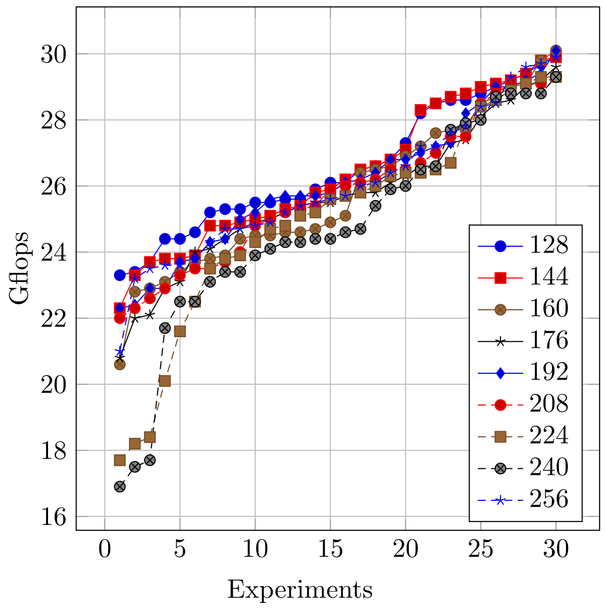

Figure 3: Performance impact of block size

5.3 Process Grid Size

The process grid should be equal to the number of cores per node (see section 3.1). A common misconception is the assumption that the cores presented in monitoring programs like Microsoft Windows Task Manager are real cores. By utilizing Hyperthreading, a single hardware core can be split in two threads. As a result, the performance measurements were significantly lower on experiments which utilized more than the physically available four cores. Figure 4 indicates that a process grid size of 2x2 yields the best results.

5.4 Process Mapping

The available options for process mapping are row major and column major (see section 3.1). Since

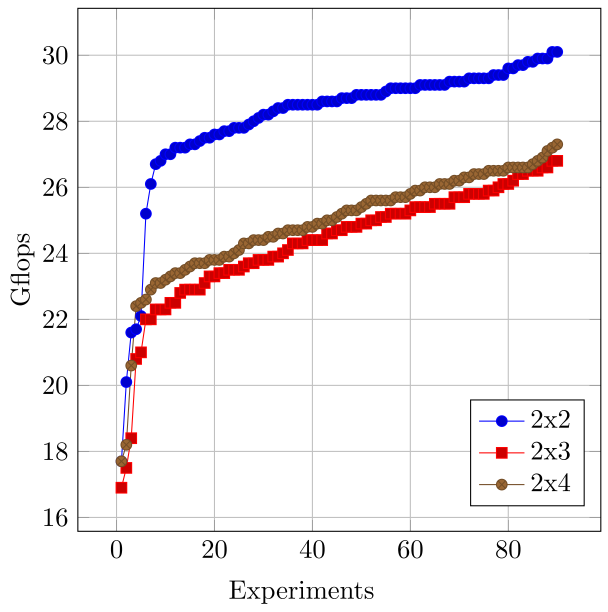

Figure 4: Performance impact of process grid size

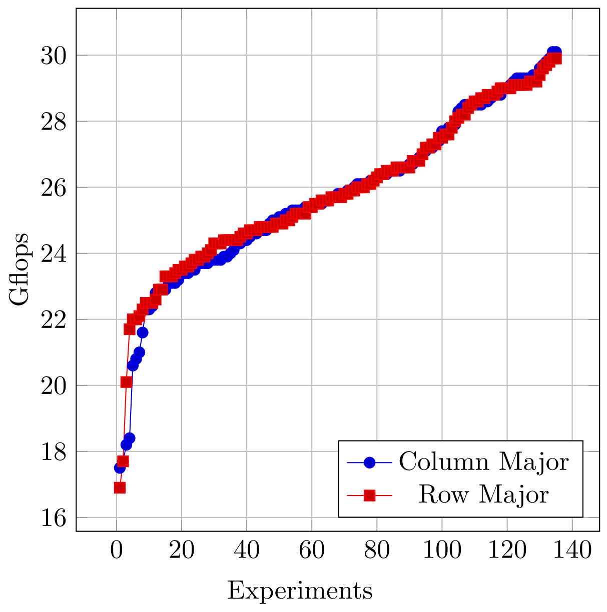

Figure 5: Performance impact of process mapping

the benchmarked system is a single processor system both options should perform similarly, which is supported by figure 5. The best result was reached with column major.

6 Challenges

7 Conclusion

Our Framework supports optimization of execution time and flops by changing one flag.

8 Future Work

Currently the plugin tries to brute force an optimal solution by trying every possible configuration. As this process is very expensive concerning computation time for the allocated resources, an intelligent approach to limit the search space should be applied in addition to common sense. For example the 270 experiments mentioned in this paper took several hours to complete. One might improve the search techniques and utilize online manipulation of benchmark parameters based on the latest benchmark results. Another approach is predicting the system performance by modeling it based on the system's architecture. By not applying brute force, the optimum solution might not be found as the risk of finding a local performance maximum exists.

In the course of this project ”flops per second” was used as benchmark metric. Although widely used to classify supercomputers the resulting value can only serve as a baseline for comparison. In general, users of HPC systems desire a minimal execution time and/or low cost. For example ”Tianhe-2” has a lower power to performance ratio than ”Titan”, second place in TOP500, which result in higher usage costs [1]. Thus, the fastest TOP500 system not necessarily yields the best results for every stakeholder. To optimize by execution time, the presented periscope plugin can be adapted to determine the average execution time over several runs and use it as benchmark metric. Likewise, the energy consumption of a system can be monitored using likwid-powermeter [2, 3].

Further, HPL solves dense matrix-matrix multiplication problems, but real-life HPC applications also contain differential equations, which need high bandwidth,low latency and irregular data access patterns, which HPL cannot execute efficiently. A new benchmark, called the High Performance Conjugate Gradient (HPCG) can deal better with lower computation-to-data-access ratios[9]. Moreover, LINPACK doesn't address issues like system latency and memory hierarchy.

References

[1] TOP500 Supercomputer Site, *November 2014*. http://www.top500.org/lists/2014/11/

[2] Leibnitz Supercomputing Centre, *likwid - light weight performance tools*. https://www.lrz.de/services/software/programmierung/likwid/

[3] Liwkid, *LikwidPowermeter*. https://code.google.com/p/likwid/wiki/Likwid-Powermeter

[4] BLAS, *Basic Linear Algebra Subroutines*. http://www.netlib.org/blas/

[5] ATLAS, *Automatically Tuned Linear Algebra Software*. http://math-atlas.sourceforge.net/

[6] HPL, *HPL Tuning*. http://www.netlib.org/benchmark/hpl/tuning.html

[7] Luszczek and A. Petitet, *The LINPACK Benchmark: Past, Present and Future*, 2001, University of Tennessee, USA

[8] Netlib, *HPL Algorithm*. http://www.netlib.org/benchmark/hpl_oldest/algorithm.html

[9] Jack Dongarra and Michael Michael Heroux, *Toward a new metric for ranking high performance computing systems*, Sandia Report, SAND2013-4744, Volume 312, 2013

[10] Tibidabo, *Making the case for an ARM-based HPC system*, Future Generation Computer Systems, Elsevier, 2013

[11] Miceli, Civario, Sikora, Csar, Gerndt, Haitof, Navarrete, Benkner, Sandrieser, Morin, and Bodin, *AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications*, Lecture Notes in Computer Science, p. 328-342, 2013, Springer Berlin Heidelberg, Germany

[12] Blaise Barney, *Message Passing Interface (MPI)*, Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/mpi/

[13] The Open MPI Project, *A High Performance Message Passing Library*, 2015. http://www.open-mpi.org/

[14] Periscope, *Periscope Tuning Framework Tutorials: Tuning Plugin Interface*. http://periscope.in.tum.de/releases/latest/doxygen/a00004.html

A Installation Script

```
#!/usr/bin/env bash
cd ~


### PERISCOPE ###

# install dependencies
apt-get install -y bison flex build-essential gfortran libace-dev libxerces-c-dev libboost-all-dev doxygen automake autoconf libtool m4 libswitch-perl git mpich2
# clone repo
git clone http://periscope.in.tum.de/git-releases/PTF-PPE-Lecture.git
# prepare build
cd PTF-PPE-Lecture
./bootstrap
cd ..
mkdir build
cd build
# configure
../PTF-PPE-Lecture/configure --prefix=$HOME/install/periscope --enable-developer-mode
# find out number of cores
NP=`cat /proc/cpuinfo | grep processor | wc -l`
# build framework
make -j $NP
make install
export PATH=$HOME/install/periscope/bin:$PATH
echo "export PATH=$HOME/install/periscope/bin:$PATH" >> ~/.bashrc


### BLAS ###

# For more information see following resources:
# http://linuxtoolkit.blogspot.de/2013/03/running-linpack-hpl-test-on-linux.html

mkdir -p ~/src/
cd ~/src/
wget http://www.netlib.org/blas/blas.tgz
tar -zxvf blas.tgz
cd BLAS
gfortran -O3 -std=legacy -m64 -fno-second-underscore -fPIC -c *.f
ar r libfblas.a *.o
```

```
ranlib libfblas.a
rm -rf *.o
export BLAS=~/src/BLAS/libfblas.a
ln -s libfblas.a libblas.a
mv ~/src/BLAS /usr/local/


### ATLAS ###

# For more information see following resources:
# http://math-atlas.sourceforge.net/atlas_install/node6.html

cd ~
# disbale cpu throtteling
# Note: CPU throtteling might not be neccessary for virtual machines
apt-get install -y gnome-applets
apt-get install -y gnome-applets-data
#UNABLE TO LOCATE PACKAGE# apt-get install -y libpanel-applets2-0
/usr/bin/cpufreq-selector -g performance &

# download latest ATLAS library
wget http://www.netlib.org/lapack/lapack-3.5.0.tgz
wget http://sourceforge.net/projects/math-atlas/files/Stable/3.10.2/atlas3.10.2.tar.bz2
tar -xvf atlas3.10.2.tar.bz2
mv ATLAS ATLAS3.10.x                        # get unique dir name
cd ATLAS3.10.x                              # enter SRCdir
mkdir Linux_C2D64SSE3                       # create BLDdir
cd Linux_C2D64SSE3                          # enter BLDdir

../configure -b 64 --shared --with-netlib-lapack-tarfile=/home/$USER/lapack-3.5.0.tgz
# Note: These next steps might take very long...
make build                                  # tune & build lib
make check                                  # sanity check correct answer
make ptcheck                                # sanity check parallel
make time                                   # check if lib is fast
make install                                # copy libs to install dir


### OpenMPI ###

# For more information see following resources:
# http://www.sysads.co.uk/2014/05/install-open-mpi-1-8-ubuntu-14-04-13-10/

```

```
cd ~
# download latest package
wget https://www.open-mpi.org/software/ompi/v1.8/downloads/openmpi-1.8.4.
    tar.gz
apt-get install -y libibnetdisc-dev
tar -xvf openmpi-1.8.4.tar.gz
cd openmpi-1.8.4.tar.gz
./configure --prefix="/home/$USER/.openmpi"
make
make install
export PATH=$HOME/.openmpi/bin:$PATH
echo "PATH=$HOME/.openmpi/bin:$PATH" >> ~/.bashrc
export LD_LIBRARY_PATH=$HOME/.openmpi/lib/:$LD_LIBRARY_PATH
echo "LD_LIBRARY_PATH=$HOME/.openmpi/lib/:$LD_LIBRARY_PATH" >> ~/.bashrc


### HPL ###

cd ~
# download latest package
wget http://www.netlib.org/benchmark/hpl/hpl.tgz
tar -xvf hpl.tgz
cd hpl
# configure
ARCH="Linux_PII_CBLAS"
cp setup/Make.$ARCH .

# change parameters and practice sed
sed -i '/^ARCH*/ c\ ARCH = $(ARCH)' Make.$ARCH
sed -i '/^TOPdir*/ c\ TOPdir = $(HOME)/hpl' Make.$ARCH
sed -i '/^MPdir*/ c\ MPdir = /usr/lib/openmpi' Make.$ARCH
sed -i '/^MPinc*/ c\ MPinc = -I$(MPdir)/include' Make.$ARCH
sed -i '/^MPlib*/ c\ MPlib = $(MPdir)/lib/libmpi.so' Make.$ARCH
sed -i '/^LAdir*/ c\ LAdir = /usr/local/atlas/lib' Make.$ARCH
sed -i '/^CC*/ c\ CC = /usr/bin/mpicc' Make.$ARCH
sed -i '/^LINKER*/ c\ LINKER = /usr/bin/mpicc' Make.$ARCH

make arch=$ARCH
cd bin/$ARCH

# run HPL
mpirun -np 4 ./xhpl
```

B HPL.dat for Laptop

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL230.out      output file name (if any)
6               device out (6=stdout,7=stderr,file)
1               # of problems sizes (N)
10000               # N -- fantastically important; see ahead
1               # Number of block sizes
160                     # Block sizes
1               PMAP process mapping (0=Row-,1=Column-major)
1               # of process grids (P x Q)
2               # Ps
2               # Qs
16.0            threshold
1               # of panel fact<
0               PFACTs (0=left, 1=Crout, 2=Right)
1               # of recursive stopping criterium
4               NBMINs (>= 1)
1               # of panels in recursion
2               NDIVs
1               # of recursive panel fact.
1               RFACTs (0=left, 1=Crout, 2=Right)
1               # of broadcast
3               BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1               # of lookahead depth
1               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)
```