PetaKit Software Suite: Automated Performance Scaling Analysis

Samuel Leeman-Munk, Ivan Babic, Mobeen Ludin, Charles Peck

January 21, 2013

Abstract

A description of Earlham College's PetaKit software suite, with a guide for using PetaKit's central unit StatKit to gather and analyze performance data on various systems.

Contents

1 Background

High performance computing raises the bar for benchmarking. Existing benchmarking applications such as Linpack[2] measure the raw power of a computer in one dimension, but in the myriad architectures of high performance cluster computing an algorithm may show excellent performance on one cluster while performing poorly on another cluster. Petakit aims to improve this weakness of standard benchmarking by using multidimensional benchmarking technique that measure a cluster's abilities via multiple unique tests rather than just one. In its final form, PetaKit will support thirteen different tests - one for each of the the thirteen dwarfs of computing as published in Berkeley's parallel computing research paper[1].

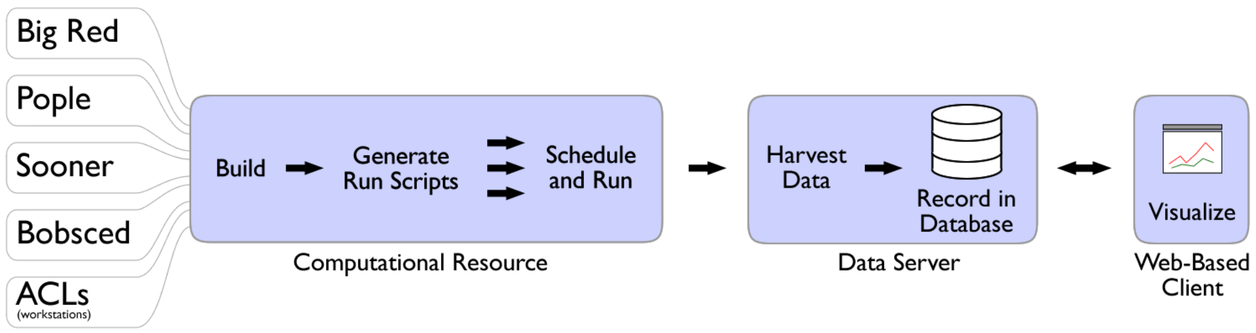


Figure 1: PetaKit's flow of data[4]

2 PetaKit Infrastructure

When a PetaKit tarball is decompressed on a supported cluster, first Autotools is run to build the files. Then stat.pl takes its parameters and generates a set of shell scripts, one for each combination. Stat.pl submits these scripts to the scheduler, resubmitting for each repetition, and when each is finished it pipes its output through ssh to the parser.pl script on a computer at Earlham College. Parser.pl parses the output into data which it sends to a PostgreSQL relational database. The data remains in the database where it can be accessed via a web browser application which allows the user to pick dependent and independent variables and compare the scaling of one setup to another (see Figure 1). The most common use of the graph is to compare OpenMP performance to MPI performance to hybrid performance by plotting the three threads vs. walltime.

3 Parallel and Distributed Computing

In parallel computing, a problem is divided up and each of multiple processors performs a part of the problem. The processors work at the same time, in parallel. There are three primary reasons to parallelize a problem.

The first is to achieve speedup, or to solve the problem in less time. As one adds more processors to solve the problem, one may find that the work is finished faster, although there is a limit. An observation known as Amdahls Law says that there is a theoretical limit to the amount of speedup that can be achieved from strong scaling. At some point, the number of processors exceeds the amount of work available to do in parallel, and adding processors results in no additional speedup. Worse, the time spent communicating between processors overwhelms the total time spent solving the problem, and it actually becomes slower to solve a problem with more processors than with fewer.

The second reason to parallelize is to solve a bigger problem. More processors may be able to solve a bigger problem in the same amount of time that fewer processors are able to solve a smaller problem. If the problem size increases as the number of processors increases, we call it weak scaling.

The third reason to parallelize is that it allows a problem that is too big to fit in the memory of one processor to be broken up such that it is able to fit in the memories of multiple processors.

In parallel processing, rather than having a single program execute tasks in a sequence, the program is split among multiple execution flows executing tasks in parallel, i.e. at the same time. The term execution flow refers to a discrete computational entity that performs processes autonomously. A common synonym is execution context; flow is chosen here because it evokes the stream of instructions that each entity processes.

Execution flows have more specific names depending on the flavor of parallelism being utilized. In distributed memory parallelism, in which execution flows keep their own private memories (separate from the memories of other execution flows), execution flows are known as processes. In order for one process to access the memory of another process, the data must be communicated, commonly by a technique known as message passing. The standard for this is the Message Passing Interface (MPI), which defines a set of primitives for packaging up data and sending them between processes.

In another flavor of parallelism known as shared memory, in which execution flows share a memory space among them, the execution flows are known as threads. Threads are able to read and write to and from memory without having to send messages. The standard for shared memory considered in the BCCD modules is OpenMP, which uses a series of directives for specifying parallel regions of code to be executed by threads.

4 Supported Dwarfs

The combination of the BCCD and PetaKit support representatives of many of the thirteen parallel dwarfs[1] outlined by the group at UC Berkeley. The parallel architecture of each is described below.

4.1 Embarrassingly Parallel - Area Under a Curve

Calculating the area under a curve on graph is an important task in science. There are many applications of the area under a curve in many fields of science, including pharmacokinetics and clinical pharmacology, machine learning, medicine, neuroscience, psychiatry and psychology, chemistry, environmental science, fisheries and aquatic sciences, economics, and many others.

The calculus developed by Isaac Newton and Gottfried Leibniz in the 17th century allows for the exact calculation of the area of simple curves through integration, but for many functions integrals do not exist for finding the area under their curves, or these integrals cannot be used to find the area in a reasonable number of steps. To compensate for this, other techniques can be used that provide acceptable approximations for the area under a curve.

This module considers the Riemann method of integration, developed by Bernhard Riemann in the 19th century for approximating the area under a curve. There are four methods of Riemann sum for approximating the area under curves on a graph. Right and left methods make the approximation using the right and left endpoints of each subinterval. Maximum and minimum methods make the approximation using the largest and smallest endpoint values of each subinterval. The values of the sums converge as the subintervals halve from top-left to bottom-right. The specific Riemann method explored in this module involves dividing the domain over which we are integrating into segments of equal width that serve as the bases of rectangles. The heights of the rectangles correspond to the y-value of the function for an x-value found somewhere within the rectangles widths. The sum of the areas of the rectangles formed by this method is the approximation of the area under the curve. This module considers the Left Riemann sum, in which each rectangle has the height of the left-most point of its width.

The complete background, science and operating instructions for the GalaxSee module can be found at `http://www.shodor.org/petascale/materials/UPModules/AreaUnderCurve/`

4.2 N-Body - GalaxSee

One of the grand challenge problems in astronomy is the evolution and structure of the universe and galaxies. The universe is seen to have a structure of sheets and voids on a large scale. Galaxies are seen to often have a spiral structure that is difficult to explain. Space is not occupied by a homogeneous fluid, but by discrete particles that interact through gravity over long ranges. The N-body problem is the problem of predicting the motion of a group of celestial objects that interact with each other with respect to gravity.

The N-Body problem is a problem in which more than 2 particles interact in such a way that every particle has the potential to interact with every other particle in a meaningful way. Typically this is defined to be any problem where you expect to see forces acting at a distance, such as gravitational interactions on astronomical spatial scales, or electroweak interactions on molecular spatial scales. The characteristics of the N-Body problem are that it has historically pushed the boundaries of our ability to handle both large computational problems and chaotic computational problems. BCCD Comes with two flavor or N-body problems. GalaxSee and GalaxSee-v2. GalaxSee:

The GalaxSee code is a simple implementation of parallelism. Since most of the time in a given N-Body model is spent calculating the forces, we only parallelize that part of the code. Client programs that just calculate accelerations are fed every particles information, and a list of which particles that client should compute. A server runs the main program, and sends out requests and collects results during the force calculation. You could think of the total running time in the following way.

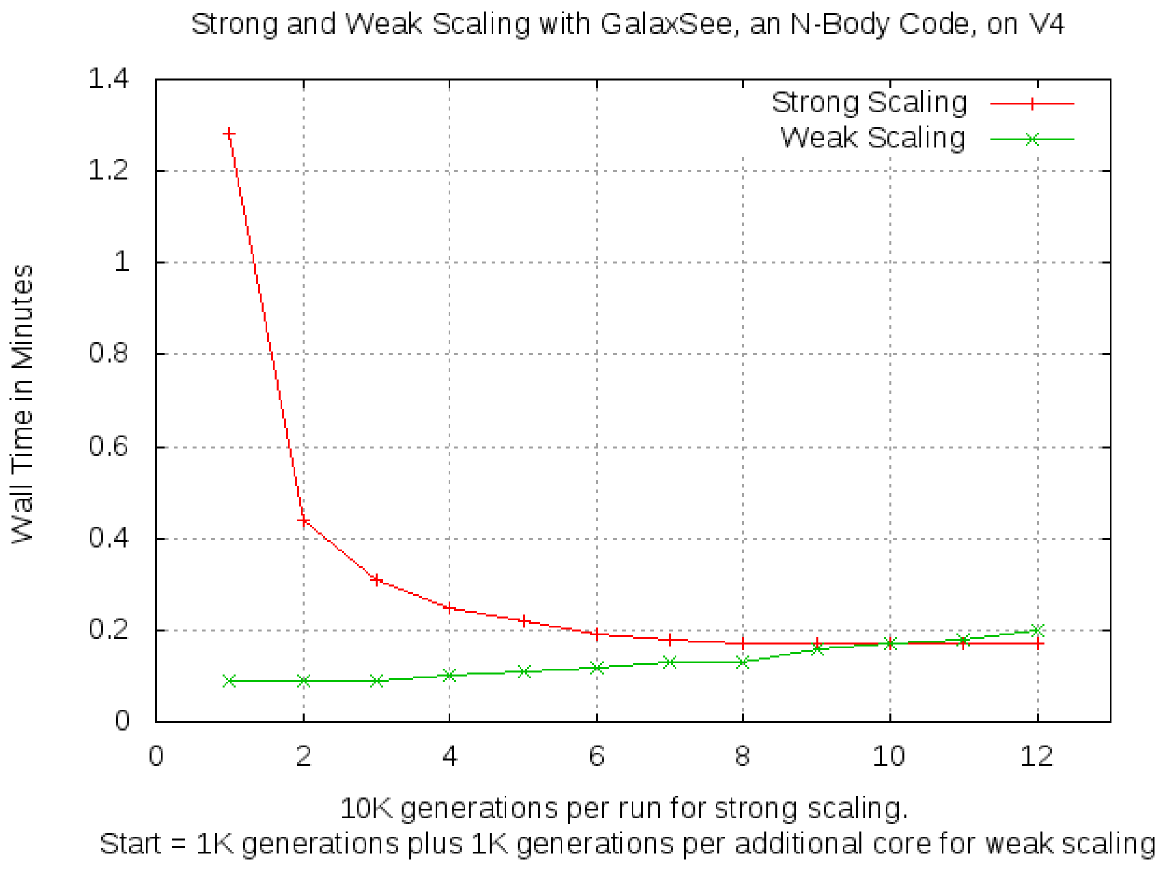
A) It takes some time to send out information on N particles to $P-1$ processors each of S time steps.

B) It takes some time for each processor to calculate $N/P * N$ interactions each of every S time steps.

As long as you run the model for enough time steps that not much time is spent setting-up the program, a reasonable model for how long the GalaxSee program will take to run on a given cluster is $time = A * N * (P - 1) + B * N * N/P$.

4.2.1 Scaling

In order to measure parallel scaling performance of BCCD module Galaxy, we used weak and strong scaling. The point of the strong scaling is to keep the same amount for the workload (in our case 10.000 generations for N-Body problem) and run it against different number of cores. Since LittleFe V4 units have 6 dual core nodes, we run the same amount of load on 1 to 12 cores.



It obvious in the graph that we get speedup every time we increase the number of cores. Furthermore, the speedup is getting smaller and smaller every time we increment the number of cores, which provides a great way for teaching overload. In the graph, strong scaling is represented with the red line

In the case of the weak scaling (represented by the green line in this graph), the amount of the workload is proportionally increased every time we add another core. For example, when we were testing for one core the workload was 1.000 generations of N-Body problem, for two cores the workload was 2.000 generations etc. In the perfect and symmetric world, this should generate a straight line. Of course, in the world of HPC this is impossible and graph confirms that.

The complete background, science and operating instructions for the GalaxSee module can be found at `http://shodor.org/petascale/materials/UPModules/NBody/`

4.3 Monte Carlo - Paramater Space

The parameter space model is a study of the dart game 301. In the game of 301, you start at 301 points and work your way down to zero. The board is divided into 20 wedges, with the bull's-eye in the middle worth 25 points. The ring around the outer edge is a double score, the inner ring is a triple score, and the exact center of the bull's-eye is a double score. You get three darts per turn, and except for the beginning and end of the game (which require hitting the double ring) the goal is to score as high as possible.

The origin of this model began in the dart league at Eammon's in Loudonville, New York. It was a friendly league, with lots of beginners, and lots of free advice for beginners. One piece of advice often given to players was that if you missed a lot, then you should aim at the 1, so that when you miss you will hit either the 20 or the 18. This advice seems suspicious, so we need to make the following assumption before going further: Typical dart throws will have a random direction from some aim point and a normal distance from some aim point.

We set up a Monte Carlo model to test each spot on the board to determine the average score of a three dart throw if we aimed at that point, and we ran it with different accuracy levels, where the accuracy level was defined as the standard deviation of the normal distribution in distance from the aim point. And then we wait. Monte Carlo models work by running random events multiple times and averaging the results. If you want high accuracy, you may have to run the model many times. The code here is a reproduction of that model, designed to allow the user to visualize the solution as it progresses. It is also designed to break the model up into pieces that can be solved in parallel by different computers.

The complete background, science and operating instructions for the Parameter Space module can be found at `http://bccd.net/wiki/index.php/Parameter_Space`

4.4 Structured Grid - Conway's Game of Life

This module teaches: 1) Conways Game of Life as an example of a cellular automaton, 2) how cellular automata are used in solutions to scientific problems, 3) how to implement parallel code for Conways Game of Life (including versions that use shared memory via OpenMP, distributed memory via the Message Passing Interface (MPI), and hybrid via a combination of OpenMP and MPI), 4) how to measure the performance and scaling of a parallel application in multicore and manycore environments, and 5) how cellular automata fall into the Structured Grid dwarf (a class of algorithms that have similar communication and computation patterns).

The cellular automaton is an important tool in science that can be used to model a variety of natural phenomena. Cellular automata can be used to simulate brain tumor growth by generating a 3-dimensional map of the brain and advancing cellular growth over time [1]. In ecology, cellular automata can be used to model the interactions of species competing for environmental resources. These are just two examples of the many applications of cellular automata in many fields of science, including biology, ecology, cognitive science, hydrodynamics, dermatology, chemistry, environmental science, agriculture, operational research, and many others.

Cellular automata are so named because they perform functions automatically on a grid of individual units called cells. One of the most significant and important examples of the cellular automaton is John Conways Game of Life, which first appeared in [15]. Conway wanted to design his automaton such that emergent behavior would occur, in which patterns that are created initially grow and evolve into other, usually unexpected, patterns. He also wanted to ensure that individual patterns within the automaton could dissipate, stabilize, or oscillate. Conways automaton is capable of producing patterns that can move across the grid (gliders or spaceships), oscillate in place (flip-flops), stand motionless on the grid (still lifes), and generate other patterns (guns).

Conway established four simple rules that describe the behavior of cells in the grid. At each time step, every cell in the grid has one of two particular states: ALIVE or DEAD. The rules of the automaton govern what the state of a cell will be in the next time step.

The complete background, science and operating instructions for the Life module can be found at `http://www.shodor.org/petascale/materials/UPModules/GameOfLife/`

5 Using PetaKit to Collect Performance Data

Results collected and visualized from the full PetaKit infrastructure are available online at http://cluster.earlham.edu/ carrick/dbvis/petakit.php. The central unit of PetaKit, StatKit, and a plotting utility PlotKit are included in this module. StatKit will collect performance statistics for a command-line program that can be run non-interactively and place them in a text file that can be run through PlotKit or graphed in Microsoft Excel.

The following is an example of a command line for collecting performance data from an area-under-curve program over various process counts.

```
perl stat.pl --cl ‘mpirun --byslot -np $processes ../area $problem_size’
--t sooner.lsf
--problem_size 50000000 --processes 1,2,3,4
--database text
```

Arguments:

```
--cl ‘mpirun --byslot -np $processes ../area $problem_size’
```

This is the command line template. Everything in the command line template is read verbatim except for variables, which are preceded by dollar signs. The variables are populated with the values given to them in the other commands, and PetaKit automatically cycles through every possible combination of the sets of values given to these variables.

Currently, the Petakit command line template interface supports the following five variables:

- “function”
- “problem_size”
- “threads”
- “steps”
- “style”

While their names suggest very specific usages, only threads has a predefined meaning, defining, along with processes-per-node, how many nodes the scheduler will assign. The other four are actually just variables that you can use to represent anything. Their meaning depends entirely upon their placement within the command line template.

```
-t sooner.lsf
```

Although PetaKit’s developers seek to make their software work on all machines, many clusters, and especially their schedulers, have requirements that are difficult to predict. For this reason, StatKit uses what are called ”script templates” for telling PetaKit how to communicate with a given system’s scheduler. script templates use the template system just like the command line template, but are slightly more complicated. Run

```
perl stat.pl --help
```

for more information.

```
--database text
```

This tells PetaKit to dump its data into a text file to be read by PlotKit or copied into a Microsoft Excel sheet. This is good for running PetaKit on a local machine where one can easily view the data produced.

```
--proxy-output
```

The above code is designed for collecting data from a program that is equipped to provide the relevant output. Programs that are not so designed can still be evaluated via PetaKit using the –proxy-output command. Be aware that data proxy-output cannot figure out itself will be returned as dummy values. For example, cputime is always 0 when proxy-output is enabled.

6 Plotting PetaKit output with PlotKit

PlotKit aims to achieve for graphing PetaKit data what PetaKit does for collecting it. Assuming your output was sent to the default file in your stats folder, output.txt, to graph it give PlotKit the following command.

```
perl PlotKit.pl \
--independent threads --dependent walltime \
--datafile stats/output.txt \
myRun
```

Where independent and dependent indicate the columns from which to take the independent and dependent values. Take a look at your datafile to identify what the names of the columns are.

The last argument is the tag that you gave your data.

7 Outfitting C Programs for PetaKit

Although with the latest version of PetaKit it is not strictly necessary, there are a number of reasons one might want to outfit one's code for use with PetaKit.

- If you outfit your code, you can have the program automatically tell StatKit what its name is, its version number, and more. Normally it must be supplied on the command line.
- You can calculate CPU time by calculating it within the program and outputting it. Proxy-output just gives a CPU time of zero because it can't track all the processes of a parallel program from the outside.
- You get more precise control of what you choose to time. rather than being limited to recording the runtime of your entire program, you can time any particular part of your code. This can be useful if you want to isolate your biggest time hogs.

Modifying a program to supply PetaKit-compatible output is simple. All you need is your program's source code and access to the PetaKit C library, included in this module.

1. Get the pkit.h and pkit.c files and place them with your program's source.
2. Include pkit.h in your source, and make sure your makefile makes an object file (pkit.o) of pkit.c and pkit.h
3. At the beginning of your main function, type startTimer();[1]
4. At the point in your code where all the most important pieces have finished running, place the expression time = stopTimer() .
5. After that, include the printStats function, explained in the following section

7.1 printStats

```
printStats(program name,threads,style of parallelism,problem size,version number, time,
cputime, number of additional variables to be printed ...)
```

1. The first seven arguments to printStats() are required output that will be expected by the PetaKit data harvester.
2. Next is the count of whatever other values you would like your program to print - most likely for debug purposes.
3. For each additional printout, specify two arguments:
   (a) The label, which includes the type of the variable
   (b) The variable itself.

[1]This starts the timer for accurate wall time

7.1.1 Extra variables for printStats

Three general classes of variable are supported -

- s: string (stored as char*)
- i: integer (stored as long long int)
- d: double (stored as long double)

The first letter of the label is stripped and read as the variable type, so, say number of timesteps would be input as:

```
iTIMESTEPS, (long long int) num_timesteps
```

This prints as:

```
TIMESTEPS              : <number of timesteps>
```

7.1.2 Example

Here's an example of a call to printStats in an instance of John Conway's Game of Life written in C.

```
printStats("Life",life.size,"mpi",life.ncols * life.nrows, "1.3", time, 0, 3, "iCOLUMNS",
(long long int) life.ncols, "iROWS",(long long int)life.nrows, "iGENERATIONS",
(long long int)life.generations);
```

And the output:

```
!~~~#**BEGIN RESULTS**#~~~!
PROGRAM             : Life
HOSTNAME            : Sam's Computer
THREADS             : 1
ARCH                : mpi
PROBLEM_SIZE        : 11025
VERSION             : 1.3
CPUTIME             : 0
TIME                : 3.979
COLUMNS             : 105
ROWS                : 105
GENERATIONS         : 1000
!~~~#**END RESULTS**#~~~!
```

8 Laboratory - How to use StatKit

8.1 Prerequisites

- Prior experience using UNIX command-line interface.
- Knowledge of how to use the parallel processing environment (cluster and scheduler or personal computer)

8.2 Materials

- Computer (1 per student)
- Parallel Processing Environment[2]
- UPEP PetaKit Module

8.3 Procedure

Instruct students to take their own parallel programs and, using PetaKit and the included documentation, observe how well they scale up to sixteen processors[3]. A simple Reimann sum program (area-mpi.c) is included for this purpose, should your students have no parallel programs of their own. The included program requires some variant of the message passing interface (MPI), preferably openMPI, in order to run. A README file in the sample program folder walks the user through the build process.

8.4 Questions

1. How does your program scale? Is it how you expected it to scale?
2. What's the name of the curve that your program's scaling takes?
3. If you were to use, instead of sixteen, say, one hundred processors, how much less time would your program take? How about one thousand processors? one million? (See Amdahl's Law[4])
4. What could you do with your program to take better advantage of one million processors? (See Gustafson's Law[5])

8.5 Assessing Student Understanding

After completing this assignment, your student should have an understanding of how to use StatKit and the concepts behind both Amdahl's and Gustafson's laws. He or she should be able to explain to you

[2]A cluster is ideal for this purpose, but if one is not available, nearly all modern computers have at least two cores, which can serve as a very small-scale parallelization environment

[3]or more or fewer depending on hardware availability and teacher preference

[4]Amdahl's Law: http://en.wikipedia.org/wiki/Amdahl's_law

[5]Gustafson's Law: http://en.wikipedia.org/wiki/Gustafson's_law

in what situations it would and would not be helpful to assign additional processors to running his or her program.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[3] J. J. Dongarra, H. W. Meuer, E. Strohmaier, J. J. Dongarra, H. W. Meuer, and E. Strohmaier. Top500 supercomputer sites. Technical report, Supercomputer, 1997.

[4] S. Leeman-Munk, A. Weeden, A. F. Gibbon, B. Johnson-Stalhut, M. Edlefsen, G. Schuerger, D. Joiner, and C. Peck. SIGCSE Milwaukee, 2010.