PetaKit Software Suite:

Automated Performance Scaling Analysis

Samuel Leeman-Munk

April 14, 2011

Abstract

A brief explanation 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

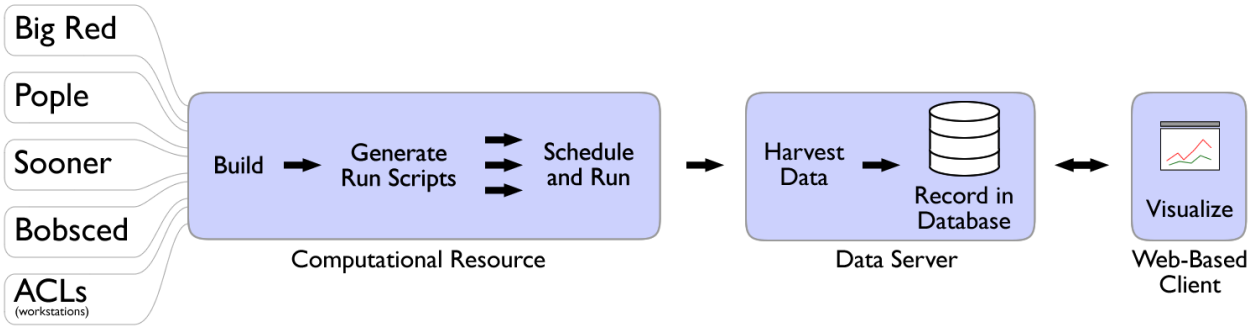

Figure 1: PetaKit's flow of data[4]

1 Background

High performance computing raises the bar for benchmarking. Existing benchmarking applications such as Linpack[2] measure 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 on another cluster of the same benchmark it performs poorly. Petakit aims to improve this weakness of standard benchmarking by using a multidimensional benchmarking technique that would 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].

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 Supported Dwarfs

PetaKit currently supports representatives of two of the thirteen dwarfs[1]. These are an embarrassingly parallel Riemann sum algorithm, area-under-curve (AUC) – an example of map-reduce, and GalaxSee – a galaxy simulation that serves as an n-body problem. AUC gives each thread a fraction of the area under the curve to estimate, they work, and then they sum all the areas they each found. Computation increases linearly over problem size while communication remains constant. GalaxSee, on the other hand, splits the stars among the threads and each calculates the new positions based on the positions of all the other stars. Then the new positions are collected and redistributed and the new positions are calculated again. Communication and computation both increase at a rate of $O(n^2)$ where n is the problem size(number of stars).

4 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 in 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 StatKit automatically cycles through every possible combination of the sets of values given to these variables.

Currently, the StatKit command line template interface supports iteration over 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[1]. 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 StatKit 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 StatKit to dump its data into a text file to be read by PlotKit or copied into a Microsoft Excel sheet. This is good for an individual running StatKit for his or her personal benefit because the resulting data file can easily be graphed with PlotKit or copied into Microsoft Excel.

```
--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 StatKit 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.

[1]This is actually handled in another template for working with schedulers that we will describe later

5 Plotting StatKit output with PlotKit

PlotKit aims to achieve for graphing StatKit data what StatKit 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. This argument is required.

6 Outfitting C Programs for PetaKit

Although with the latest version of PetaKit it is not strictly necessary, Modifying a program to supply output compatible with the StatKit program is simple. All you need is your program's source code and access to the PetaKit C library, included in this module[2].

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();[3]
4. At the point in your code where all the most important pieces have finished running, place the expression time = stopTimer() .

[2]Currently a PetaKit output library exists only for the C and C++ programming languages, but any program can create the necessary output - just take a look at the output in the "Example" section, and make your own output code!

[3]This starts the timer for accurate wall time

5. After that, include the printStats function, explained in the following section

6.1 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 StatKit 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.

6.1.1 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>
```

6.1.2 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**#~~~!
```

Note that the CPUTIME is hard-coded as zero. Because PetaKit expects a certain set of values, some values are just set as dummies when for whatever reason their real values cannot be calculated .

7 Laboratory - How to use StatKit

This lab is generalized to be usable with whatever program the student wishes to benchmark. For a step-by-step lab with included programs, look in the Area-Under-Curve module. `http://shodor.org/petascale/materials/UPModules/Area_Under_Curve/`

7.1 Prerequisites

- Prior experience using UNIX command-line interface.
- Knowledge of how to use an available parallel processing environment

7.2 Materials

- Computer (1 per student)
- Parallel Processing Environment[4]
- UPEP PetaKit Module

7.3 Procedure

Instruct students to take their own parallel programs and, using StatKit and the included documentation, observe how well they scale up to sixteen processors[5]. A simple Riemann 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.

7.4 Questions

1. How does your program scale? Is it how you expected it to scale?

[4]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

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

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[6])

4. What could you do with your program to take better advantage of one million processors? (See Gustafson's Law[7])

7.5 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 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.

[6] Amdahl's Law: http://en.wikipedia.org/wiki/Amdahl's_law
[7] Gustafson's Law: http://en.wikipedia.org/wiki/Gustafson's_law

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