Power Measurement and Modulization in the Network Protocol Independent Performance Evaluator (NetPIPE)

Rachael A. Mansbach
Office of Science, Science Undergraduate Laboratory Internship Program
Correspondence: rmansba1@swarthmore.edu

Swarthmore College, Swarthmore, Pennsylvania

Ames Laboratory
Ames, Iowa

July 29, 2009

Prepared in partial fulfillment of the requirement of the Office of Science, Department of Energy's Science Undergraduate Laboratory Internship under the direction of Troy Benjegerdes in the Scalable Computing Laboratory at Ames Laboratory.

Table of Contents

Table of Figures

Abstract
In order to provide a more user-friendly environment and a clearer benchmark for computational efficiency and to promote America's energy security through reliable, clean and affordable energy, a version of the Network Protocol Independent Performance Evaluator (NetPIPE) was created, which was written completely in the object-oriented Python language. NetPIPE performs simple ping-pong tests for increasing message sizes to determine network bandwidth and latency. The base code created last year by Science Undergraduate Laboratory Internship student Torrey Dupras, which implemented NetPIPE using Python and a Python module written in C, was modified to be purely Python, and its efficiency was compared with the previous version. The NetPIPE package was also modified to include code which documents power use during a NetPIPE experiment and outputs the results of a NetPIPE run using the Python matplotlib module to show graphs of various data. The power data was obtained by using a Watts up? PRO meter which registers the base power consumption of a device once per second. The results of the investigation revealed that, for both the implementations, there appeared to be a correlation between network bandwidth and rate of energy consumed. Further, the Python module had about one-half the peak bandwidth of the C module; however, it was much more portable to operating systems other than Linux. As energy rather than computing speed becomes the dominant factor in computer performance, these experiments could provide a base for efficiency measurements in the future and also a greater ease of access for those wishing to perform those measurements.

Introduction

One of the larger issues in the scientific community currently is that of energy. As we near the end of our nonrenewable resources, it is becoming increasingly important not only to seek out new sources of energy but to concentrate more and more heavily on preserving those that we already have; those in the computing community are not exempt. Previously, the limiting factor on computing power has been runtime, but in recent years, power consumption has begun to appear as a limiting factor in its own right. As recently as June 2008, the TOP500 supercomputing site announced its decision to begin to track power consumption in computers. [1] In 2007, Bob Wambach reported on the Computer Technology Review that energy costs "already account for the second largest line item associated with datacenter operations." [2] In 2001, it was noted that "five years ago there were only a handful of papers in [the area of energy

conservation], though today it is common to find mobile wireless conferences devoting 5–10 % of their papers to mechanisms for energy conservation." [3] In more and more papers of the past ten years, energy conservation is being cited as an important consideration. [4],[5]

In view of these facts, it seems important that some kind of benchmark for measuring power should be established and that it be made reasonably easy to access. Therefore, the Network Protocol Independent Performance Evaluator (NetPIPE), already used to benchmark computer network efficiency in terms of bandwidth and latency, was modified to include a power measurement feature. NetPIPE is a performance evaluator which uses simple ping-pong tests of bouncing data back and forth between two processors to determine the bandwidth and latency of a network. [6]

In summer 2008, a Python port of the original C code was developed in order to provide greater portability and ease of access, which still used a C module to implement the core NetPIPE algorithm. (See Figure (1)) [6],[7]

This summer, the Python code was supplemented by the addition of a Python port of the NetPIPE algorithm (to further enhance portability, since moving pure Python code from one operating system to another is simpler than moving C code or C code implemented in Python) and by the addition of programs to collect power usage data and analyze the results with respect to the other NetPIPE data.

```
/* First set T to a very large time. */
   T = MAXTIME
   For i = 1 to NTRIALS

   t0 = Time()
    For j = 1 to nrepeat
        if I am transmitter
            Send data block of size c
      Recv data block of size c
        else
      Recv data block of size c
      Send data block of size c
        endif
    endFor
    t1 = Time()
   /* Insure we keep the shortest trial time. */
  T = MIN(T, t1&shyp;t0)

   endFor
T = T/(2 * nrepeat)
```

Figure 1: The core NetPIPE algorithm as presented in the original NetPIPE paper

The goal of this work was to provide a more user-accessible NetPIPE and a basis for more understanding and development of a power benchmark associated with NetPIPE.

Materials and Methods

The core algorithm of the NetPIPE benchmark relies on passing messages of increasing size back and forth between two processes (which may or may not be on different computers). The message sizes are chosen generally at regular intervals with small perturbations. [8] NetPIPE is given a target run time and number of trials, and it passes a certain message size back and forth for a number of iterations which it calculates will make a certain trial last for approximately the target run time. This approximation is based on the following equation:

$$i = \frac{ro}{tn} \quad (1)$$

where i is the number of iterations, r is the target run time, o is the size of the message in the previous set of trials, t is the time it took on average per iteration in the previous set of trials, and n is the current message size.

Both the original NetPIPE program and the version created by Torrey Dupras followed the same format of data output, which was to report for each set of trials the message size in bytes, the maximum bandwidth of that set of trials in megabits/s, the minimum latency of that set of trials in seconds, the number of iterations for that trial and, for each trial, the overall time it took to complete those iterations. The core NetPIPE algorithm reports to the larger program the message size in bits, the number of iterations, the average time per iteration and the overall time for the trial, which are used to compute the given output.

The version of NetPIPE created by Torrey Dupras in summer 2008 was written in Python with the core NetPIPE algorithm written in C but implemented in Python. It was shown that this implementation was slightly more efficient than the original, fully C implementation of NetPIPE, but research into the cause of this somewhat surprising result remains to be carried out. This summer, the previous code was modified so that the core NetPIPE algorithm was likewise written in Python, using the Python sockets module to carry out the passing of information back and forth between processes. Despite the fact that a purely Python version requires more overhead than a Python version implementing a C module, the object-oriented nature of Python makes it of prime use when it comes to portability and user-friendliness. In the same way,

evaluating the efficiency of the pure Python module versus the C module may shed light on why the C Python module is more efficient than the pure C module.

The experiments performed using the programs discussed here consisted of running NetPIPE for approximately three to four hours on a single computer with two processors (Intel, 2.8 GHz, 2 GB of memory, linux kernel 2.6.28, Ubuntu).

In order to measure power consumption during NetPIPE use, a Watts Up? PRO power meter was used. The power meter works as follows: a device (in this case the computer) is plugged into it, and it reports the current power usage of the device once per second to a USB port. Because it only clocks the power once per second, it was necessary to have the power meter running in a separate process from NetPIPE, as otherwise it would cause the program to have comparatively long stretches of idle time in the middle of its trial runs, which would impact its own results.

In order to correlate the power measurement trial with the NetPIPE trial, timestamps were added to both the NetPIPE data and the power data, using the time function in the Python time module. It was assumed that the time function takes negligible time to run and therefore does not impact the collection of results; validating this assumption will be a necessary piece of future work. The program for power collection simply connects to the USB device to which the power meter is attached and continuously reads what it has written to that device for a certain number of iterations. In order to analyze the results of the NetPIPE and power experiments, another program was written in Python.

This NetPIPE analysis program uses the total length of each trial in seconds and the message size of each trial in bytes to calculate the bandwidth and its uncertainty. The equation used to find the bandwidth is

$$b = \frac{2 * i * m}{t} \quad (2)$$

where b is bandwidth, m is message size in bits (found by multiplying the given message size in bytes by eight), i is iteration number and t is the total time taken to perform those iterations. The numerator of this equation is equal to the total number of bits transferred over the course of the trial, since in each iteration a certain number of bits is passed first one way and then the other, i.e. the two represents the round-trip, which is necessary since the time returned by NetPIPE is the round-trip time.

There is no uncertainty in iteration number, which is given by the program. There is likewise assumed to be no uncertainty in message size, because the messages are passed back and forth using TCP/IP sockets, which are known for their reliability in not losing data, and because there is a check in NetPIPE itself so that if the data sent and received is not the same, it will exit with an error message. Therefore the uncertainty in the bandwidth is wholly governed by the uncertainty in the time it takes to run a trial. The uncertainty in the time is taken to be the standard deviation of the mean, and the uncertainty in the bandwidth is then found by assuming the equality of the percent error in the time and the percent error in the bandwidth, i.e.

$$\sigma_b = \frac{b}{\sigma_t * t} \quad (3)$$

where σ_b is the percent error in the bandwidth and σ_t is the percent error in the time.

One difficulty with the analysis was the issue of whether there was a delay from when a power reading occurred and when it was recorded. This was investigated by starting a program which, at regular intervals, performed an operation (in this case a large matrix multiplication) geared to cause a steep power increase. The program then recorded the time at which the operation occurred. Simultaneously, a program was running which noted the times at which the power meter showed a sudden increase, which indicated the occurrence of the matrix multiplication. The multiplication was performed at ten second intervals to increase the resolution of the power spikes, and from this, a certain amount of delay in the power meter was found (see Results section)

The NetPIPE analysis program, taking into account the delay calibration of the power meter, matches the power readings with their corresponding message sizes by using the timestamps placed on both of them. It further calculates the rate of energy consumption in bits/joule (or bits/s/watt) by dividing the bandwidth by the power for a given message size. In this case, since the uncertainty in the bandwidth and the uncertainty in the power are assumed to be less than the actual bandwidth and power, the maximum energy consumption rate occurs at

$$\frac{b+\sigma_b}{p-\sigma_p} \quad (4)$$

where b is the bandwidth, σ_b is the uncertainty in the bandwidth, p is the power, and σ_p the uncertainty in the power. The minimum energy consumption rate occurs (likewise) at

$$\frac{b-\sigma_b}{p+\sigma_p} \quad (5).$$

The uncertainty in the energy consumption rate is found by adding the percent error of the bandwidth and the power in quadrature, i.e.

$$\sigma_r = r * \sqrt{\left[\frac{\sigma_b}{b}\right]^2 \square \left[\frac{\sigma_p}{p}\right]^2} \quad (6)$$

where σ_r is the uncertainty in the energy consumption rate, and r is the rate itself. The uncertainty in the mean of the power is taken to be its standard deviation.

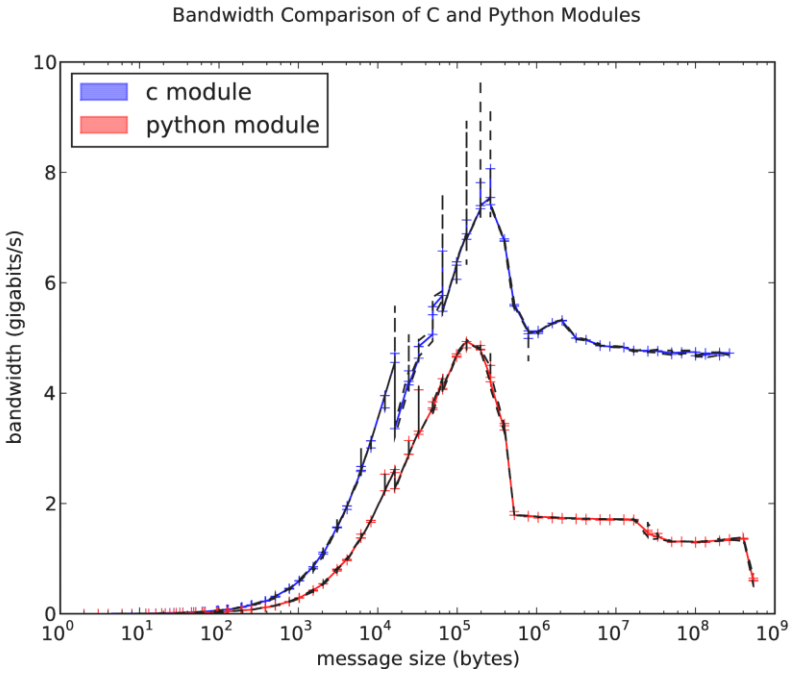


Figure 2: Comparison of Bandwidth for C and Python Modules

Results

A comparison of the efficiency of the C module implementation and the pure Python implementation in terms of bandwidth shows that the C module implementation reached a

significantly higher peak bandwidth but had a larger spread of maximum and minimum bandwidths, whereas the Python module had a lower peak bandwidth but also a lower spread of maximum and minimum bandwidths. (see Figure (2))

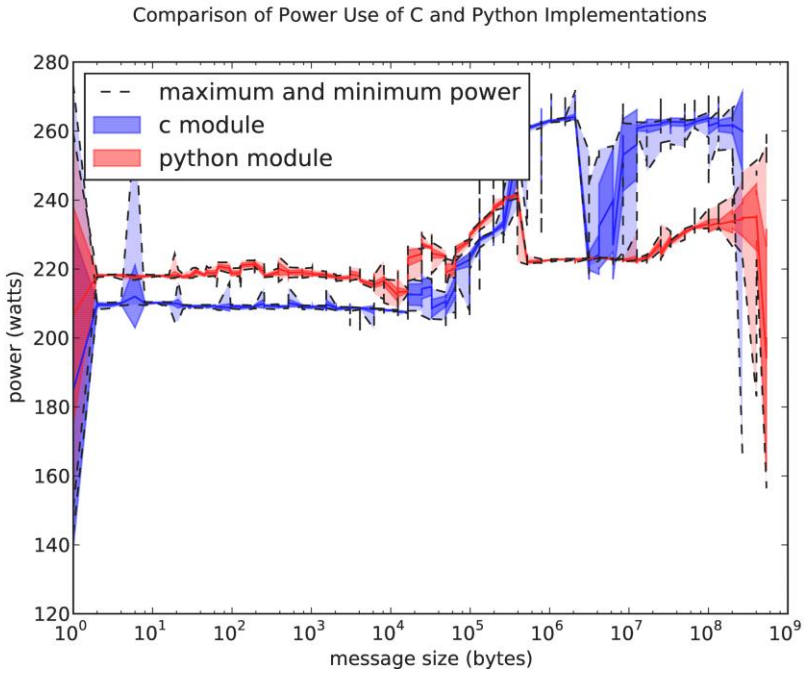


Figure 3: Comparison of Power for C and Python Modules

A comparison of the efficiency of the C module implementation and the pure Python implementation in terms of power is less straightforward. The C module reached a higher peak power later than the Python module, which peaked and dipped back down, but the Python module seemed to consume more power at lower message sizes. The C module also appeared to peak later than the Python, and though both of them dropped after peaking, the C module grew again almost immediately, whereas the Python module dropped and remained constant for some time before beginning to grow again. (see Figure (3))

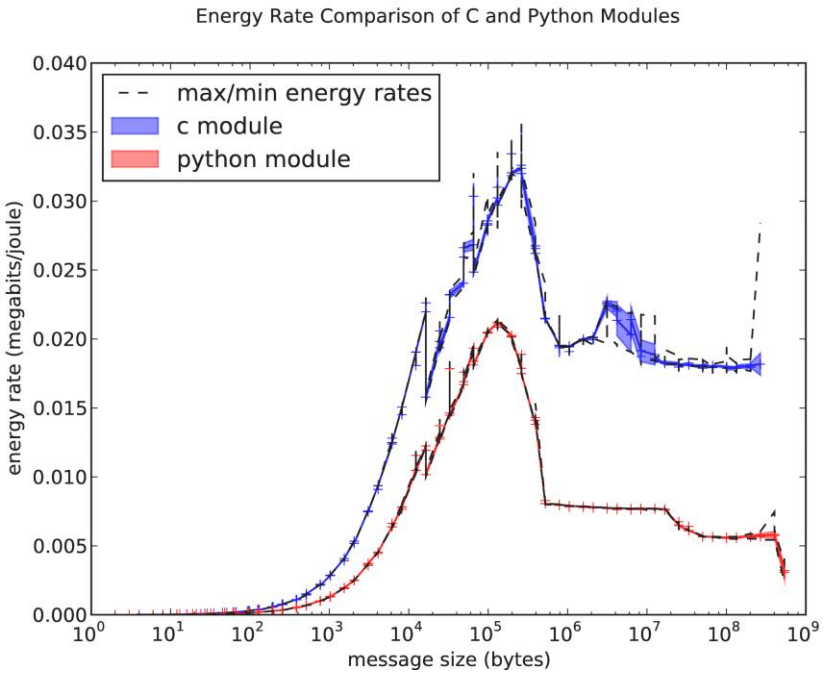
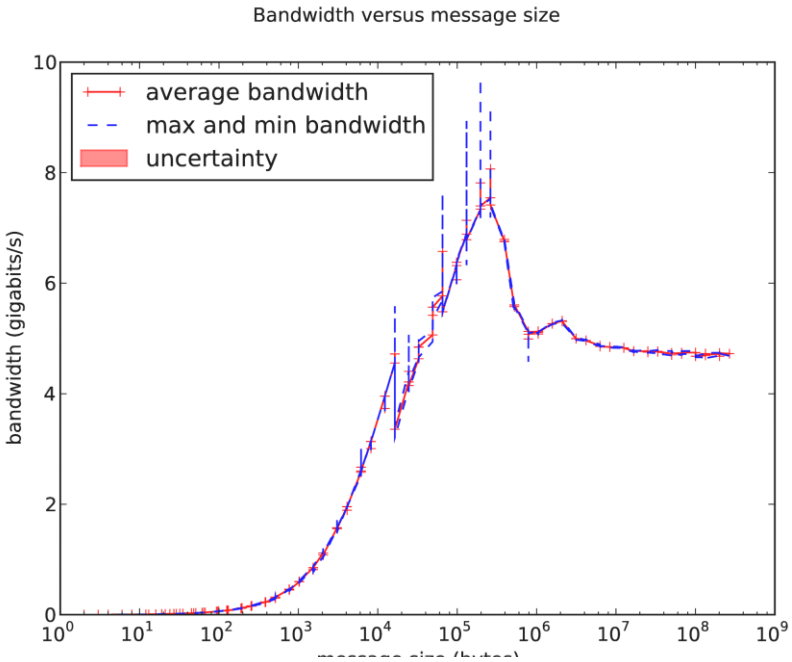
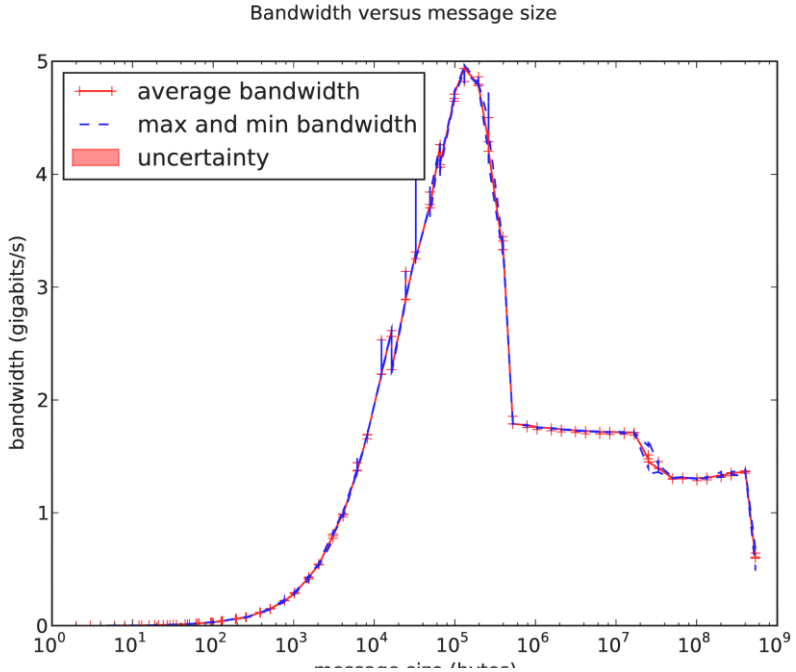


Figure 4: Comparison of Energy Rate for C and Python Modules

The analysis of the power signatures was complicated slightly by the delay between a power occurring and the power being read by the power meter. Repeated runs of the program described in the previous section suggested that for the desktop computer, the time delay was 2 ± 0.3 seconds, and therefore a calibration of two seconds was introduced to the power when it was processed by the analysis program. However, the results of the delay program were different when run on a laptop computer with a single processor (Intel, 1.0 GHz, 2 GB of memory, Windows Vista 6.0.6000) , giving 1.7 ± 0.1 second. The code used to measure the delay was very experimental in nature and required a fair amount of manual tweaking, so one necessary objective of any further study would be to find a way to automate this experiment.


Bandwidth versus message size
average bandwidth
max and min bandwidth
uncertainty
bandwidth (gigabits/s)
0
2
4
6
8
10
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9


Bandwidth versus message size
average bandwidth
max and min bandwidth
uncertainty
bandwidth (gigabits/s)
0
1
2
3
4
5
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

A comparison of the efficiency of the C module implementation and the pure Python implementation in terms of energy rate, or bandwidth per watt, shows results very similar to the results of the bandwidth comparison. The C module had a higher peak rate but much greater spread of values in certain places than the Python module. Both seem to peak around the same message size, the C module peaking perhaps slightly later than the Python module. (see Figure (4))

An investigation of the Python and C module separately shows that in general their features are similar, except for the power signatures. Both show bandwidths which grew to a peak and then fell off, as expected from the results of runs of previous versions of NetPIPE. [8]-[9] (see Figures (5) and (6)) In both the C and Python modules (see Figures (7)–(10)), the power peaked after the peak bandwidth, but in the C module, the power peaked much longer after the peak bandwidth than in the Python module. Both the C and Python modules show a plateau of power before the peak, and both show a dip afterward, which in the case of the C module was significantly after the sharp drop in bandwidth, but which was almost at the same place as the drop in bandwidth in the Python module. Both the C module and the Python module began to rise again after the drop in bandwidth, but the Python module did not do so until the point at which it reached its last set of trials, during which the runs became so long that the uncertainty in most of the measurables shows that they are unreliable.

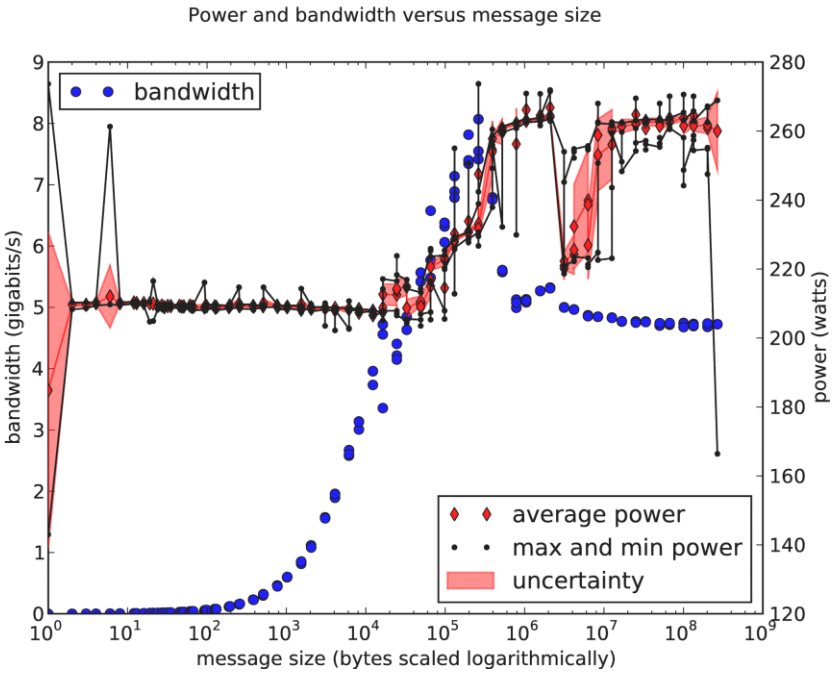


Figure 7: C Module Power and Bandwidth

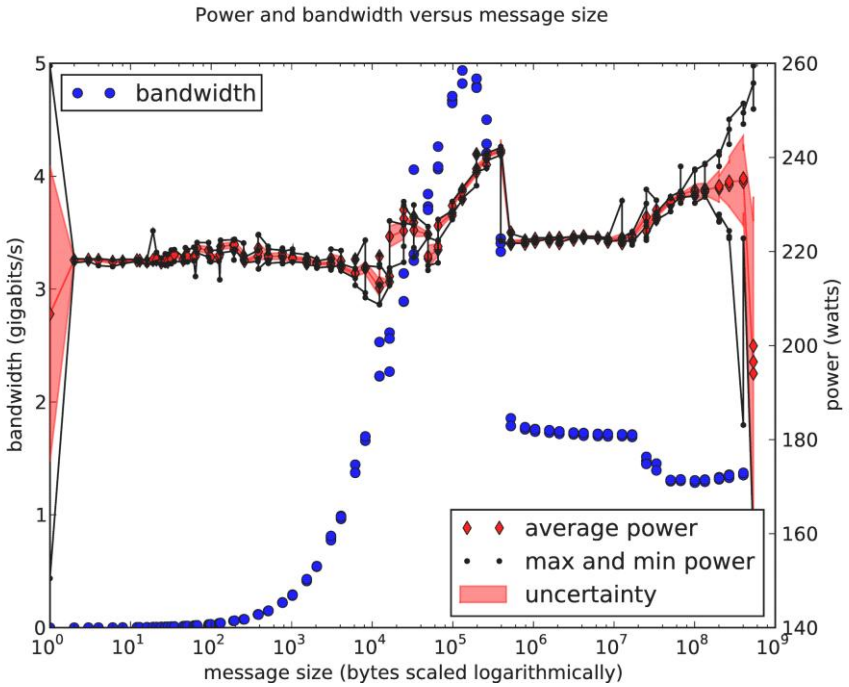


Figure 8: Python Module Power and Bandwidth

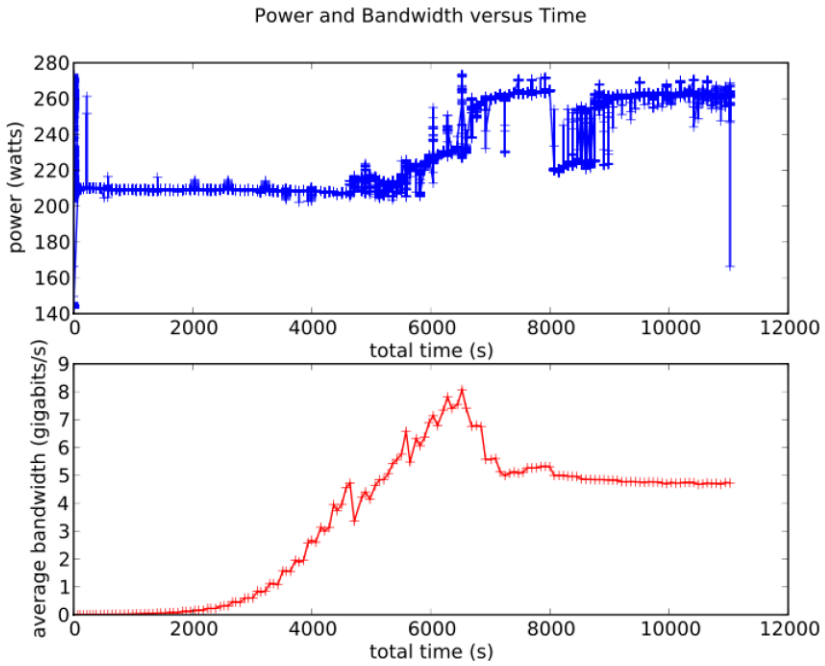


Figure 9: C Module Average Power and Bandwidth

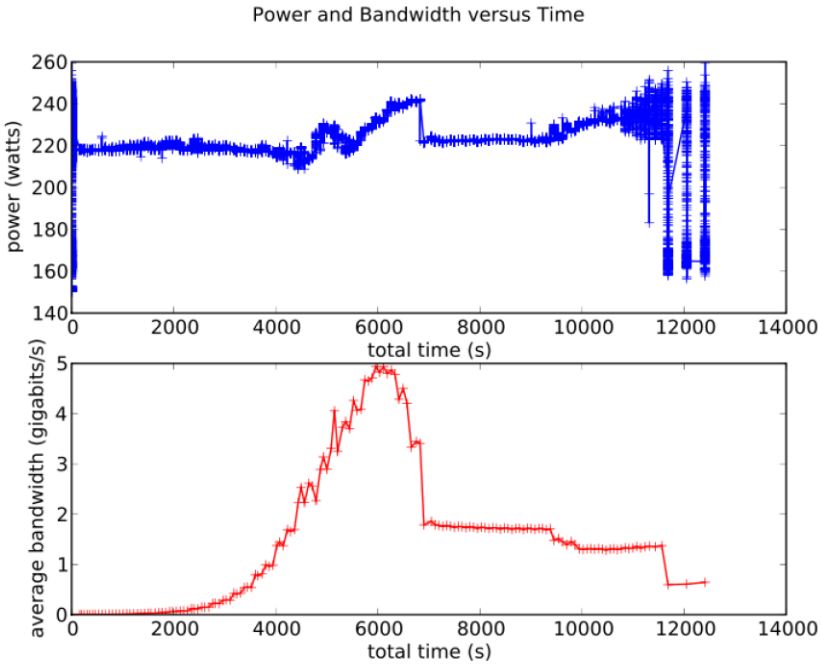


Figure 10: Python Module Average Power and Bandwidth

Inspecting the energy rates in light of the bandwidths of the C and Python modules (see Figures (11)–(14)) shows that there was a close correlation between bandwidth and energy rate. In fact, a calculation of the correlation coefficients of the C and Python modules, using the corrcoef function in the Numpy module, returns, respectively, 0.989079909642 and 0.998770039869, showing an almost surprisingly close correlation in each case between the bandwidth and the energy rate. Further, in both modules, the peak bandwidth and the peak rate occurred at the same time and had the same shape. The C module, which showed a secondary bandwidth peak, also showed a secondary energy rate peak, although that peak began as the bandwidth peak ended. It is also worth noting how small the uncertainty, and therefore the range of values at each point was, for the energy rate of both C and Python modules.

The pattern of iterations and of latency appeared to be the same for both the C and the Python modules (see Figures (15)–(18)).

Discussion and Conclusion

The results of these experiments indicate that it is more performance-efficient to run the C module version of NetPIPE than the pure Python version in terms of both bandwidth and bandwidth/power. In terms of pure power, the efficiency is difficult to calculate, given the way the power signatures shift as bandwidth increases. It seems that the Python version uses a larger amount of base power, while the C module version requires more power at the peak bandwidth. It appears that when running on a single computer, the rate of bandwidth/power has a signature

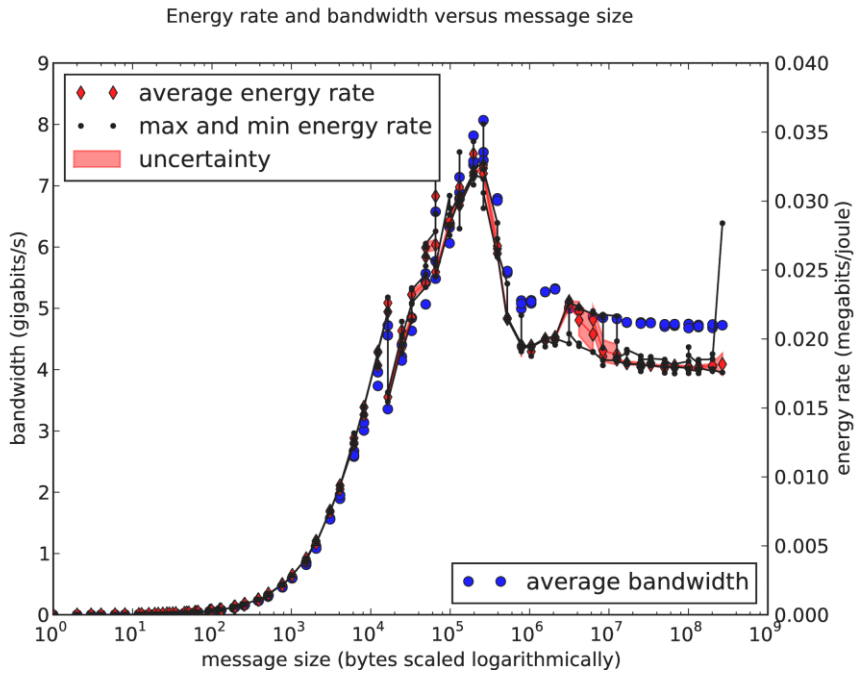


Figure 11: C Module Energy Rate and Bandwidth

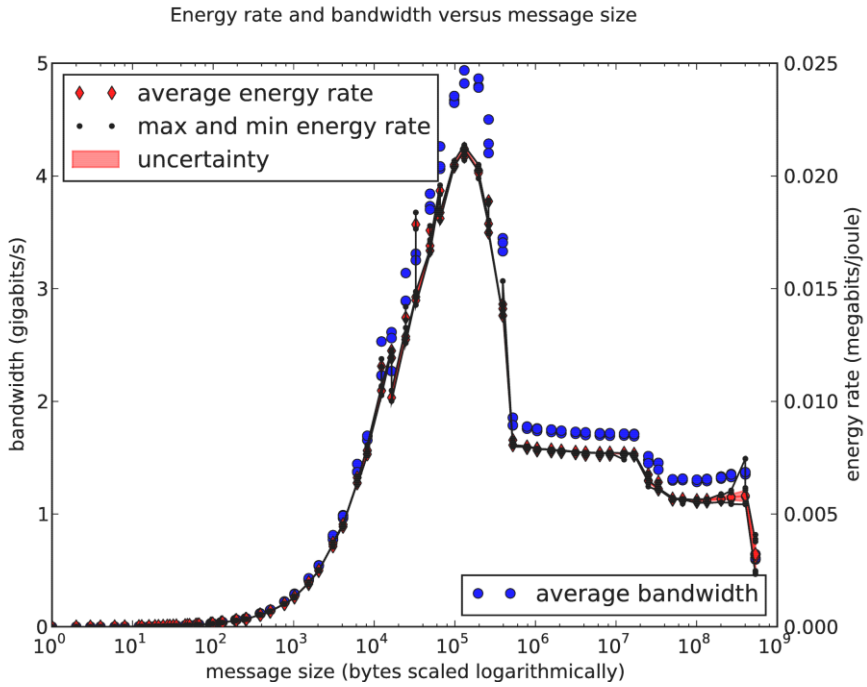


Figure 12: Python Module Energy Rate and Bandwidth

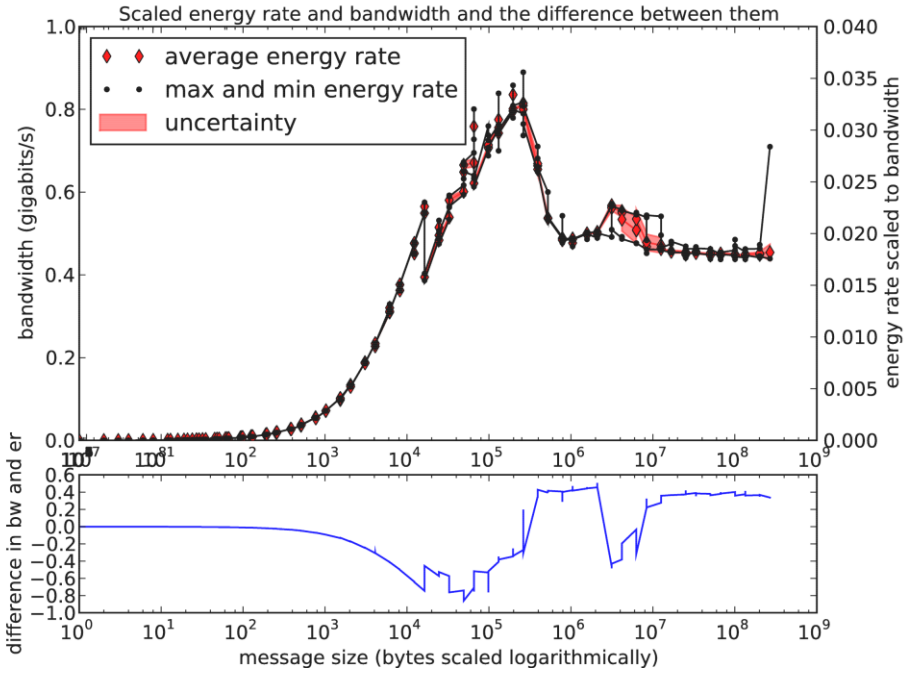


Figure 13: C Module Scaled Energy Rate, Bandwidth, and Difference Between Them

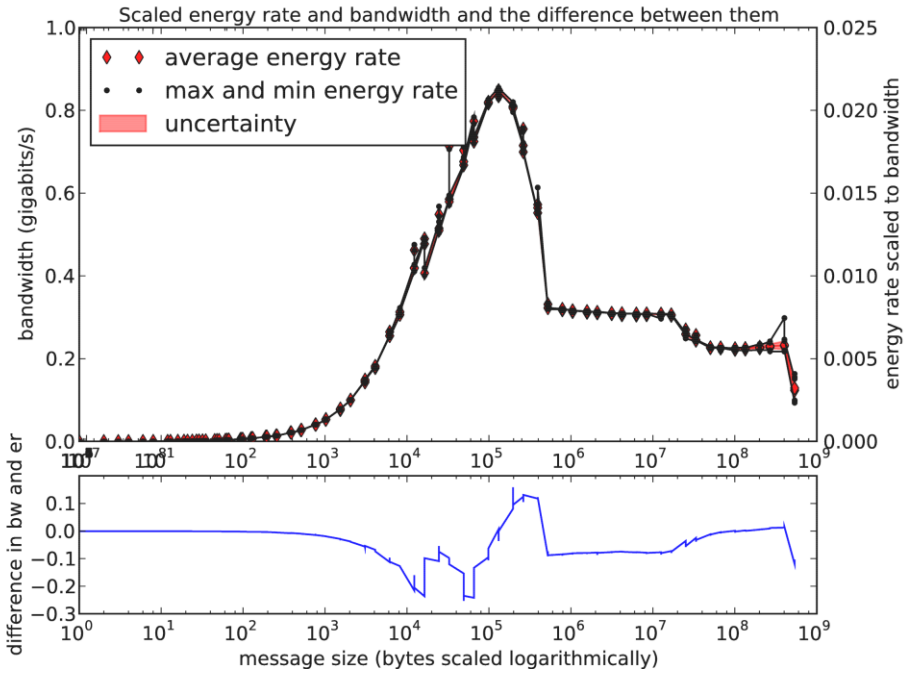


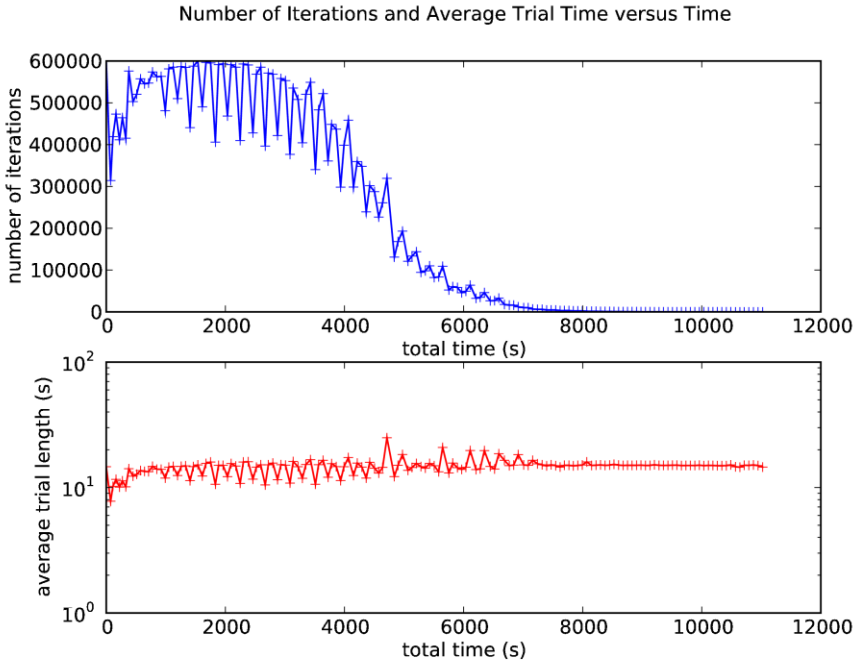
Figure 14: Python Module Scaled Energy Rate, Bandwidth, and Difference Between Them

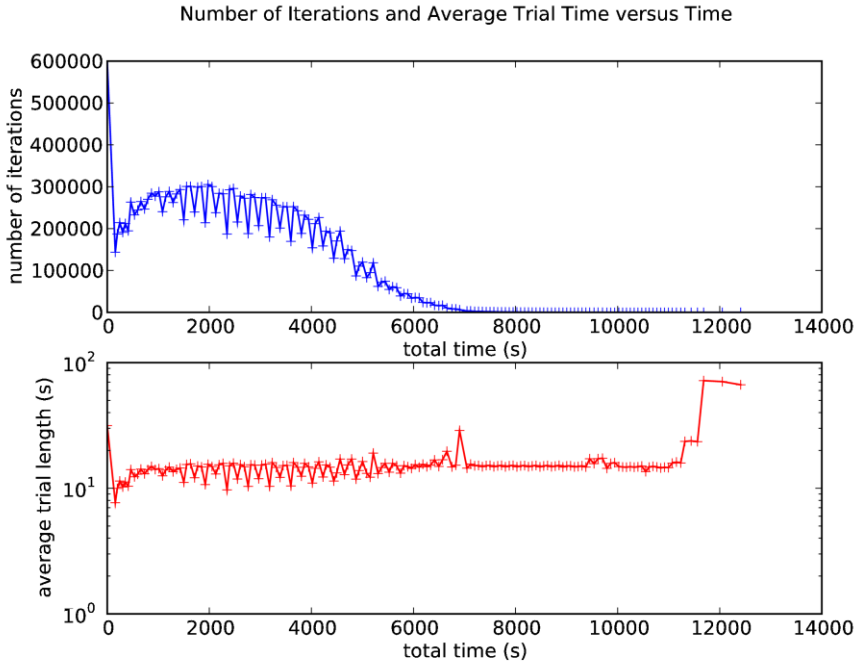
correlated very closely with the bandwidth signature, which would not necessarily be true were the experiment performed on two or more computers.

Therefore, the next experiment to be performed would be to run NetPIPE on multiple computers and try to produce a coherent bandwidth/power graph. This might be more difficult, since there would be two different power-sources involved. It might be possible to connect both computers to the same power source and measure the fluctuations of that source, or it might be preferable to measure the power signature of each computer separately and compare the resulting graphs of power and of bandwidth/power for each.

The close correlation of bandwidth/power to bandwidth for a single computer suggests that the bandwidth/power shape is mostly determined by the bandwidth shape, which would lead to the conclusion that power use is approximately constant; in light of the fact that it is not, this might merely reflect that its percent changes are much smaller than the percent changes in the bandwidth.

Another point of investigation might be why the C module shows such large spikes in bandwidth at certain message sizes. It is interesting that rather than showing a slowly increasing bandwidth over a large range, it shows a large jump in bandwidth at certain distinct points, suggesting that there is something special about those particular message sizes that causes the movement of packets to become much faster. Another experiment might be performed to isolate those message


Number of Iterations and Average Trial Time versus Time
number of iterations
600000
500000
400000
300000
200000
100000
0
0
2000
4000
6000
8000
10000
12000
total time (s)
average trial length (s)
10^2
10^1
10^0
0
2000
4000
6000
8000
10000
12000
total time (s)


Number of Iterations and Average Trial Time versus Time
number of iterations
600000
500000
400000
300000
200000
100000
0
0
2000
4000
6000
8000
10000
12000
14000
total time (s)
average trial length (s)
10^2
10^1
10^0
0
2000
4000
6000
8000
10000
12000
14000
total time (s)

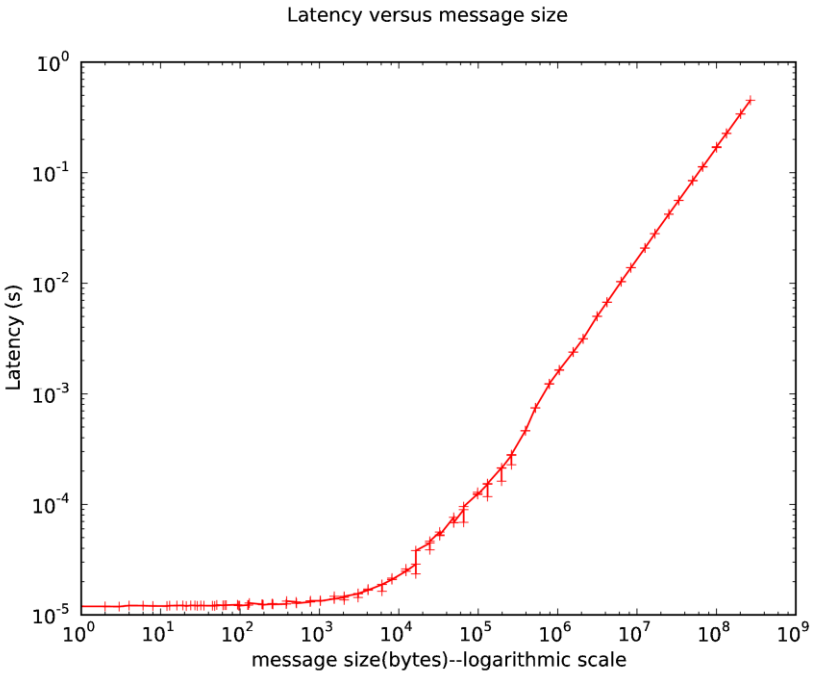


Figure 17: C Module Latency

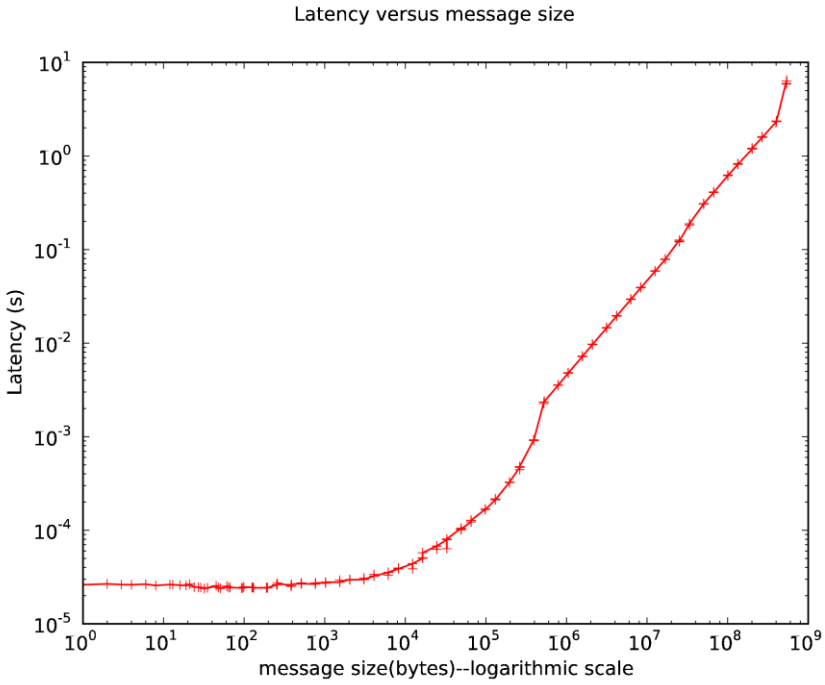


Figure 18: Python Module Latency

sizes and discover if there is anything special about them, or alternatively, to run NetPIPE on a different computer to see if it is a feature of the data unique to the computer on which it was being run, though the lack of presence of the spikes in the Python module suggests that it is in fact a feature of the way the C module sends data, or perhaps the way it interacts with its base Python code.

In terms of investigating NetPIPE behavior on other computers, the pure Python module is a valuable aid. It would be difficult and involved to move the C module to a different operating system, whereas to move the Python module all that was necessary was one tweak in the way that the program created two processes to pass the data between, which would not be necessary in an implementation running NetPIPE on two separate machines. A slightly more difficult problem is to run NetPIPE on two computers not in the same network, and a possible way to solve this would be to exploit the secure shell (SSH) protocol.

The experiments documented in the paper are an intriguing first step. It will be useful to have a NetPIPE not bound by the restrictions of the C module to a single operating system; also a more user-friendly NetPIPE may spread NetPIPE to more users, allowing it to aid non-computer-scientists, such as chemists who want to know the most efficient way to send data for simulations. So too, the power data and program provide a building block for a future power benchmark feature of NetPIPE, which in the long run may be beneficial in creating more energy-efficient computers and computer networks

Acknowledgments

This research was conducted in summer 2009 as part of the Student Undergraduate Laboratory Internship (SULI) program at the Scalable Computing Laboratory, a division of the Ames Laboratory at Iowa State University in Ames, Iowa. Thanks to Troy Benjegerdes for his help in conducting the research and writing the paper, and thanks to the U.S. Department of Energy and Ames Laboratory for creating, organizing and funding the program.

Bibliography

[1] Top 500 Supercomputer Sties. “Power consumption of supercomputers”, June 2008. http://www.top500.org/lists/2008/06/highlights/power.

[2] Wambach, Bob. “Green Storage: Strategies for Enhancing Energy Efficiency”. *Computer Technology Review*, April 2007. http://www.wwpi.com/ctr/spring-2007/5835-green-storage-strategies-for-enhancing-energy-efficiency.

[3] Petrioli, Chiara, Ramesh R. Rao and Jason Redi. “Guest editorial: energy concserving protocols”. *Mobile Networks and Applications*, volume 6, no. 3: pp. 207-209, June 2001.

[4] Higgs, Tim. “Energy Efficient Computing”. In *Proceedings of the 2007 IEEE Symposium on Electronics and the Environment*, pp.210-215, May 2007.

[5] Deris, Kaveh Jokar and Amirali Baniasadi. “Investigating cache energy and latency break-even points in high performance processors”. *Memory Performance: Dealing With Applications, Systems and Architecture*, volume 260, pp.13-20, 2006.

[6] Snell, Quinn O. , Armin R. Mikler and John L. Gustafson. “NetPIPE: A Network Protocol Independent Performance Evaluator”. In *IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[7] Dupras, Torrey and Benjegerdes, Troy. “NetPIPE: Network Protocol Independent Performance Evaluator, Version 5.0”. *Journal of Undergraduate Research*, volume IX, 2009.

[8] Turner, Dave, Adam Oline, Xuehua Chen and Troy Benjegerdes. “Integrating new capabilities into NetPIPE.” In *Lecture Notes in Computer Science*, pp. 37-44, Spring, 2003.

[9] Turner, Dave and Xuehua Chen. “Protocol Dependent Message-Passing Performance on Linux Clusters”. In *Fourth IEEE International Conference on Cluster Computing*, p. 187, 2002.