Ref: IFI-UIT Technical Report 2015-76 ——

Technical Report: Evaluation of the power efficiency of UPC, OpenMP and MPI

Is PGAS ready for the challenge of energy efficiency? A study with the NAS benchmark.

Jérémie Lagravière · Phuong H. Ha · Xing Cai

-

Abstract In this study we compare the performance and power efficiency of Unified Parallel C (UPC), MPI and OpenMP by running a set of kernels from the NAS Benchmark. One of the goals of this study is to focus on the Partitioned Global Address Space (PGAS) model, in order to describe it and compare it to MPI and OpenMP. In particular we consider the power efficiency expressed in millions operations per second per watt as a criterion to evaluate the suitability of PGAS compared to MPI and OpenMP. Based on these measurements, we provide an analysis to explain the difference of performance between UPC, MPI, and OpenMP.

Keywords PGAS · power efficiency · performance evaluation · UPC · MPI · OpenMP · NAS Benchmark

1 Introduction & Motivations

The High Performance Computing research community is strongly interested in reducing energy consumption and optimizing power efficiency of the programs. Even though computation speed is still the number one objective, understanding the impact of hardware and software on energy consumption is equally important.

Jérémie Lagravière
Simula Research Laboratory, Martin Linges vei 25 1364 Fornebu, Norway
E-mail: jeremie@simula.no

Phuong H. Ha
UiT The Arctic University of Norway, NO-9037 Tromsø, Norway
E-mail: phuong.hoai.ha@uit.no

Xing Cai
Simula Research Laboratory, Martin Linges vei 25 1364 Fornebu, Norway
E-mail: xingca@simula.no

Optimizing energy efficiency, without sacrificing computation performance is the key challenge of energy-aware HPC, which is an absolute requirement of Exascale computing in future. Particularly for the race to Exascale, staying under 20 MegaWatts is one of the objectives proposed by the US Department of Energy [26]. The goal of energy efficiency is twofold: reducing both the cost related to energy consumption and the environmental footprint of supercomputers.

In this study we focus on three parallel programming languages and models: UPC (UPC is based on Partitioned Global Address Space model or PGAS), MPI (message passing) and OpenMP (shared memory multiprocessing). MPI and OpenMP are well-known and have been used for years in parallel programming and High Performance Computing. UPC (PGAS) provides ease of programming through a simple and unified memory model. On a supercomputer, this means that the programmer can access the entire memory space as if it is a single memory space that encompasses all the nodes. Through a set of functions that makes the data *private* or *shared*, PGAS languages ensure data consistency across the different memory regions.

We will evaluate performance and measure energy consumption of a well-established parallel benchmark suite implemented in all the models mentioned above. We compare the behaviors of the three different implementations based on MPI and OpenMP and UPC. We run the Numerical AmeS Parallel Benchmark (NAS Benchmark or NPB) provided by NASA and implemented in each of these languages.

Our motivation for this report is based on recent studies that advocate the use of PGAS as a promising solution for High Performance Computing [20,24]. Previous studies have focused on the evaluation of PGAS performance and UPC in particular[13,18]. Thus, we are warranted to re-investigate using the latest CPU architecture and focusing on energy efficiency.

This report is organized as follows: Section 2 outlines the related work and background to this study. Section 3 briefly presents the UPC framework and the reasons why we chose this programming language. Section 4 describes the benchmark chosen for this study. Section 5 contains the description of the application used to measure energy consumption for our experiments. Section 6 explains in detail the hardware and software set-up used for running our experiments. Section 7 presents the result obtained through the experiments.

In the appendix, we provide a technical description on how to perform the experiments used in this paper. In particular, we explain how to handle Intel PCM in order to get energy measurement and other metrics concerning communication behavior and memory traffic.

2 Previous Work

Previous studies [10,13,23,29] focused on the NAS Benchmark and considered only the performance evaluation of this benchmark. While part of our work is based on the results of these studies, we also use the NAS Benchmark as an evaluation basis to measure energy consumption.

The NAS Benchmark was released in 1991 [5] and has been maintained by the NASA. Since then, the NAS Benchmark has been used as the basis of many studies in High Performance Computing. Even in recent years, the NAS Benchmark has been considered a valid base to analyze computation performance of a chosen technology. In [23], published in 2011, the NAS Benchmark is used in cooperation with OpenCL. In [29], the performance of the NAS Benchmark is characterized to evaluate a hybrid version combining MPI and OpenMP. In [10,13], UPC performance are evaluated by using NPB.

The need for energy awareness in High Performance Computing has been increasingly covered over the last 10 years: Two major trends can be identified, energy estimation or profiling as in [9,16] and energy measurement or monitoring as in [15,22,25]. In [6], several approaches are described to achieve energy measurement for HPC architectures: software approach, hardware approach and hybrid approach. In our study, we use the software approach to obtain energy measurements. In particular, Intel PCM (Intel Performance Counter Monitor) is chosen to perform these measurements. We consider Intel PCM to be a valid choice because many studies have selected this tool for their energy measurements such as in [6,14].

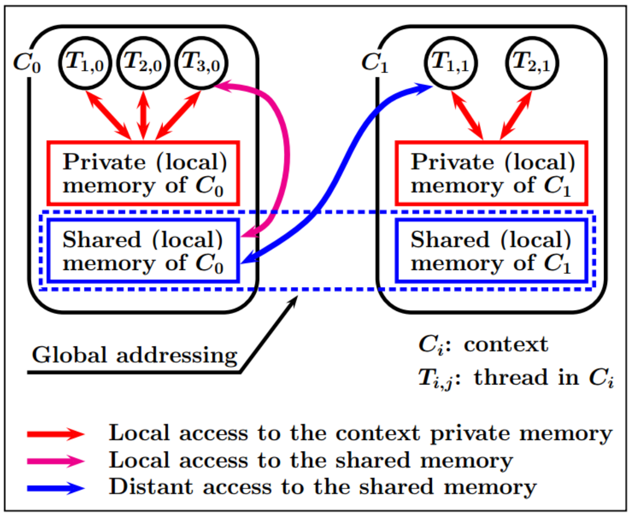
C_0
$T_{1,0}$
$T_{2,0}$
$T_{3,0}$
C_1
$T_{1,1}$
$T_{2,1}$
Private (local) memory of C_0
Private (local) memory of C_1
Shared (local) memory of C_0
Shared (local) memory of C_1
Global addressing
C_i: context
$T_{i,j}$: thread in C_i
Local access to the context private memory
Local access to the shared memory
Distant access to the shared memory

Fig. 1 PGAS Communication Model [19] - Figure used with the courtesy of Marc Tajchman

In the past decade, many studies have chosen UPC as a central topic in High Performance Computing. In [8,13,18] UPC performance is compared to MPI and OpenMP. The goal of this technical report is to provide a continuation to these studies, and additionally deliver a power efficiency analysis of UPC, MPI and OpenMP.

3 PGAS Paradigm and UPC

PGAS is a parallel programming model that has a logically partitioned global memory address space, where a portion of it is local to each process or thread. A special feature of PGAS is that the portions of the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference [8] [27].

Figure 1 shows a view of the communication model of the PGAS paradigm [19]. In this model, each node (C_0 or C_1) has access to a private memory and a shared memory. Accessing the shared memory to either read or write data can imply inter-node communication. The blue arrow in Figure 1 represents distant access to shared memory. This kind of distant access is of type RDMA (Remote Direct Memory Access) and is handled by one-sided communication functions.

PGAS is also a family of languages, among which UPC. UPC is an extension of the C language. The key characteristics of UPC are:

– A parallel execution model of Single Program Multiple Data (SPMD) type;

– Distributed data structures with a global addressing scheme, and static or dynamic allocation;
– Operators on these structures, with affinity control;
– Copy operators between private, local shared, and distant shared memories and
– Two levels of memory coherence checking (strict for computation safety and relaxed for performance).

Additionally, multiple open-source implementations of the UPC compiler and runtime environment are available, in particular Berkeley UPC [7] and GCC/UPC [11].

4 The NAS Benchmark

The NAS Benchmark [5], is a set of kernels that provides different ways to stress a supercomputer. The NAS Benchmark is originally implemented in Fortran and C, we also use the UPC version of the NAS Benchmark [8] [2]. In our study, we have selected four kernels: Integer Sort (IS), Conjugate Gradient (CG), Multi-Grid (MG) and Fourier Transformation (FT).

CG refers to a *conjugate gradient* method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular all-to-all communication through sparse matrix-vector multiplication.

MG is a simplified multigrid kernel. Multigrid (MG) methods in numerical analysis solve differential equations using a hierarchy of discretizations. For example, a class of techniques called multiresolution methods, are very useful in (but not limited to) problems exhibiting multiple scales of behavior. MG tests both short-and all-to-all data as well as local memory accesses.

FT is a three-dimensional partial differential equation solver using Fast Fourier Transformations. This kernel performs the essence of many spectral codes. It is a rigorous test of all-to-all communication performance. [5]

IS represents a large *integer sort*. This kernel performs a sorting operation that is important in *particle method codes*. It evaluates both integer computation speed and communication performance. [5]

Among the benchmarks available in NPB we selected CG, IS, MG and FT because they are the most relevant ones: stressing memory, communication and computation. The other benchmarks in NPB are of limited relevance to this study.

The above four benchmark kernels involve very different communications schemes, which is important in order to evaluate the performance of the selected languages (see Section 7).

5 Energy Measurement

We have chosen a software based solution in order to measure the CPU and RAM energy consumption. Intel Performance Monitor (Intel PCM) is used for the experiments of this study [12].

Intel PCM provides a set of ready-made tools that can output information about CPU energy, Dynamic Random Access Memory (DRAM) energy, NUMA details, performance flaws, and so forth in various formats (joules, watts). Intel PCM works on compatible processors, such as Intel Xeon, Sandybridge, Ivy Bridge, Haswell, Broadwell, or Skylake processors [6].

Intel PCM requires both root access and activation of counters in the BIOS/UEFI. Intel PCM uses the Machine Specific Registers (MSR) and RAPL counters to disclose the energy consumption details of the application [6]. This is a major constraint when running our experiments. We solved this limitation by running all the experiments on a machine equipped with two Intel Xeon E5-2650 on which we have root access.

Intel PCM is able to identify the energy consumption of the CPU(s), the RAM and the Quick Path Interconnect (QPI) between sockets. In this study, the energy measurement is considered as a whole, which means that we aggregate the energy consumption of the CPU and RAM. QPI energy consumption was not taken in account in this study because Intel PCM was unable to provide measurement on the chosen hardware platform.

In the appendix, we will describe how to use Intel PCM in order to perform measurements such as energy consumption, power efficiency, communication pattern and memory traffic.

6 Hardware Testbench

For our experiments, we used a computer equipped with two processors of type Intel Xeon E5-2650, and 32GB of RAM DDR3-1600MHz. This computer has 16 physical cores, and 16 additional logical cores, due to Hyper-Threading.

This machine runs Ubuntu 12.04.5 LTS with the Linux kernel 3.13.0-35-x64. We used Berkeley UPC implementation [7] in version 2.2.0 compiled with GCC 4.6.4. The NAS Benchmark is implemented in C and Fortran version 3.3 [3], to compile NPB we used GCC version 4.6.4 and gfortran version 4.8 OpenMPI version 1.8.4 was used for MPI and OpenMP was used with version 3.0. For energy measurement we used Intel PCM in version 2.8 [12].

Each measurement made use of three runs; the run with the best value values was chosen.

Table 1 Cases where thread-binding was activated

-	OpenMP	MPI	UPC
CG	16 threads	2 and 4 processes	16 threads
MG	8, 16 threads	-	-
FT	16 threads	16 processes	-
IS	-	2,4 and 8 processes	-

For each kernel Class "C" was chosen [3] [1]:

- IS Class C: Number of keys is 2^{16}
- FT Class C: Grid size is $512 \times 512 \times 512$
- CG Class C: Number of rows is 150000
- MG Class C: Grid size is $512 \times 512 \times 512$

Size C provides data sets that are sufficiently large to exceed the cache capacity of the chosen hardware architecture [28] [1]. Each benchmark is measured for execution time and energy consumption for 2, 4, 8, 16, 32 threads.

6.1 Thread binding

Thread binding or thread pinning is an approach that associates each thread with a specific processing element.

During our experiment, we realized that binding the threads to physical cores often gave the best results in terms of execution time and energy consumption. More specifically, thread binding means that we divided the threads evenly between the two sockets (see Table 1). We only applied thread binding if it resulted in the fastest computation and highest energy-efficiency. When not activated, the threads were managed automatically by the operating system. Concerning MPI, instead of *threads binding* the correct terminology is *process binding*.

Table 1 shows to which kernels thread binding has been applied.

7 Results

In this report, we selected two metrics to measure performance and energy efficiency. To evaluate performance we chose time in seconds for measuring the executing time. To evaluate energy efficiency Millions Operations Per Seconds over Watts (MOPS / Watt) were chosen to measure the performance per power unit. The *500 Green - Energy Efficient High Performance Computing Power Measurement Methodology* [4] advises this measurement methodology.

For convenience we use the following notation: [Benchmark name]-[number of threads/processes]. For instance, *CG-32* stands for Conjugate Gradient running on 32 threads (or processes for MPI).

Figures 2, 4, 6 and 8 show the execution time for the four benchmarks: CG, MG, FT and IS. Each benchmark ran for different thread numbers (process numbers for MPI): 2, 4, 8, 16 and 32. Each of these figures shows the results for all the selected languages: UPC, OpenMP, MPI.

Figures 3, 5, 7 and 9 show the performance per watt expressed in MOPS per watt. Thread numbers, colors and language orders are the same as in the previous figures.

The time measurement results show that the kernels scale over the number of thread/cores independently of the language. There is no clear winner in the sense that none of the chosen languages is better than the other two counterparts for all the benchmarks. We have the same observation for energy efficiency; there is an improvement in the efficiency by dividing the work over more threads.

Using 32 threads implies using the HyperThreading capability of the CPU. In most cases, only OpenMP took advantage of the HyperThreading. In CG-32, MG-32 and FT-32, UPC and MPI achieved lower performance than the same kernel running on 16 threads (processes).

Even though there is no global *winner* in the achieved measurements, we are interested in knowing whether UPC performs well in terms of computation performance and energy efficiency. Concerning the execution time, UPC is the best in CG-4, CG-8, CG-16, MG-2, MG-4, MG-8, MG-16 and MG-32. For FT and IS, UPC is not the winner, however it competes well with OpenMP and MPI.

UPC's energy efficiency is directly connected to its performance result. Therefore, the best results in energy efficiency are achieved, in most cases, for the same benchmarks and thread-count as mentioned above. UPC competes well with OpenMP and MPI in this aspect. UPC is the best in MG-2 to MG-32,CG-8, CG-16, and FT-8.

8 Discussion

In this section, we give an explanation of the differences in performance that were observed in the previous section.

Intel PCM provides access to metrics such as: L2 cache hit and miss rates, L3 cache hit and miss rates, Memory accesses (Local (same socket) and remote (other socket) etc. In this section we will use these measurements to find the explanation of differences in performance obtained with OpenMP, MPI and UPC. In the appendix, the procedure on how to obtain those metrics is explained.

For instance, in Figure 3, that represents the Conjugate Gradient power efficiency, there is a noticeable difference in the results obtained using UPC and OpenMP: in this case, UPC wins. By looking at the behavior of the OpenMP implementation compared to the UPC implementation, we can see that the level of L2 cache hit for UPC is higher than for OpenMP: UPC L2 cache hit ratio is 56% and OpenMP L2 cache hit ratio 14%. As a direct consequence the OpenMP implementation of CG running over 16 threads uses more CPU cycles to miss data in L2 cache and then fetching data in L3 cache than the UPC implementation. L2 cache hit rate is the cause for the difference in performance, in this case, as it is the only metric that differs significantly between the performance of the OpenMP implementation and that of UPC implementation.

In Figure 5, representing the Multigrid power efficiency, UPC performs better than OpenMP over 16 threads. In this case, the analysis of the memory traffic (amount of data that is read and written in/to RAM) is the useful metric that determines the cause of the lower performance of OpenMP. In total the OpenMP reads and writes 500GB of data into memory while the UPC only reads and writes 350GB, this difference of 150GB is the main cause for the difference in performance as all the other metrics indicate similar or comparable values. Another way to look at this difference of performance is by analyzing the use of memory bandwidth done by the OpenMP and UPC implementations: the OpenMP implementation reaches 60GB/s of memory bandwidth (cumulated bandwidth: read + write) while the UPC implementation reaches 47GB/s. In this case, only the memory related values (read and write) differ significantly between the performance of the OpenMP implementation and that of UPC implementation.

In Figure 7, representing the 3D-Fourier Transform power efficiency, OpenMP performs better than UPC. In this case, it is both the L3 cache miss and the amount of memory traffic (read+write) of the UPC implementation that cause poor performance compared to the OpenMP implementation. The UPC implementation reads and writes 950GB of data from/to memory and has a L3 cache hit ratio of 15% while the OpenMP implementation reads and writes only 500GB of data from/to memory and has a L3 cache hit ratio of 62%. For FT only the hit ratio and the memory related values (read and write) differ significantly between the measurements performed on both OpenMP implementation and UPC implementation.

9 Conclusion & Future Work

In this study, we have measured the energy efficiency and the computation performance of four kernels from the NAS Benchmark using three different programming languages: UPC, MPI and OpenMP. From the measurements presented in Figures 2 to 9, we observe that by scaling over more thread/cores the performance and the energy efficiency both increase for the four selected kernels on the chosen hardware platform.

As PGAS is our focus, it is important as a conclusion to highlight the fact that UPC can compete with MPI and OpenMP in terms of both computation speed and energy efficiency.

Another hardware solution that we would like to explore are accelerators, such as Many Integrated Cores (MIC) and GPUs. These kinds of accelerators are well-known for being more energy efficient than CPUs. This requires both the benchmark and UPC to support these accelerators.

Some of the UPC/PGAS libraries and runtime environment, provide support for both GPU and Intel Phi (Many Integrated Cores). To compete with more recent PGAS languages such as X10, UPC has to adapt: the MVAPICH initiative provides support of both MIC and GPU for UPC [17] [21]. One of our next objectives is to cover the energy consumption on accelerators by using this runtime environment.

The next stage in our study of PGAS, is to study UPC's performance for computation speed and energy efficiency on a supercomputer compared to MPI and OpenMP.

To go further in the energy measurements and to have a fine grained approached of the location of the energy consumption: it is interesting to study separately the energy cost of computation, communication between nodes, and memory. This would allow improvement to achieve both in the code and in the UPC compiler and runtime environment.

The need for precision in the energy measurement is a strong requirement. Thus validating the quality of the software-based energy measurement with hardware-based energy measurement is one of our objectives for future work.

10 Acknowledgments

We would like to thank Saeed Shariati for helping us in the early phase of this project.

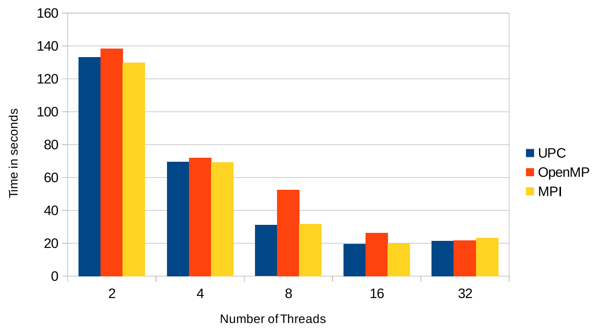

Fig. 2 CG Size C execution time in seconds. The lower, the better.

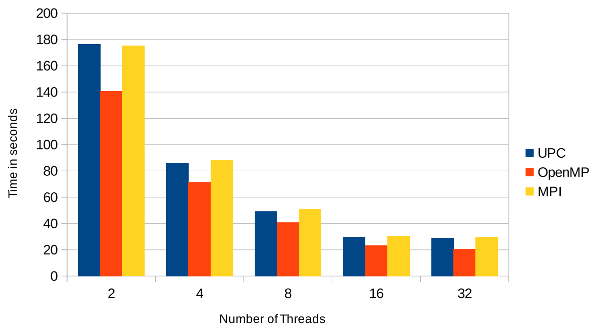

Fig. 6 FT Size C execution time in seconds. The lower, the better.

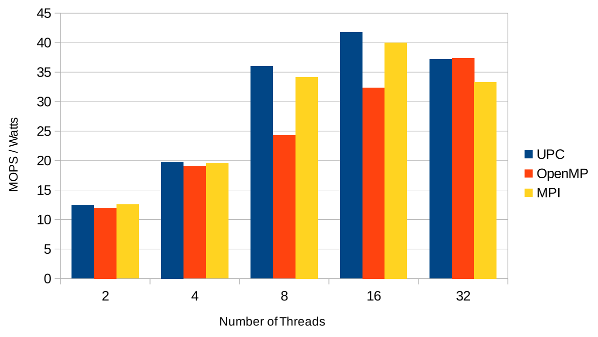

Fig. 3 CG Size C Performance per watt: MOPS / Watt. The higher, the better.

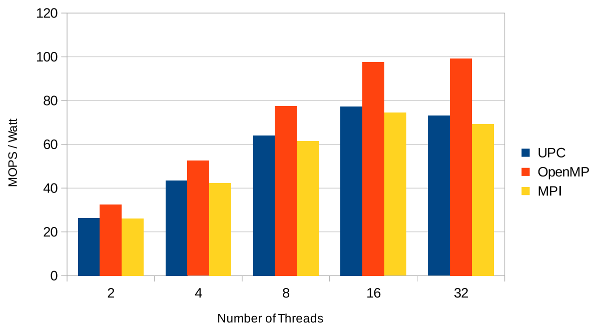

Fig. 7 FT Size C Performance per watt: MOPS / Watt. The higher, the better.

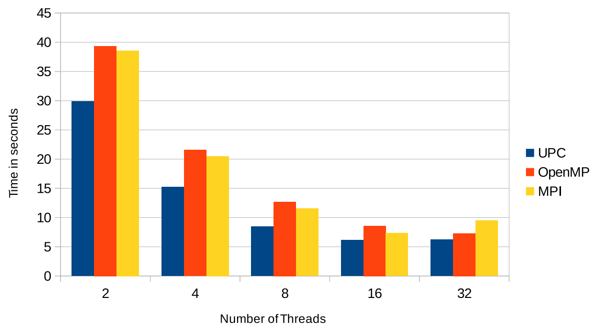

Fig. 4 MG Size C execution time in seconds. The lower, the better.

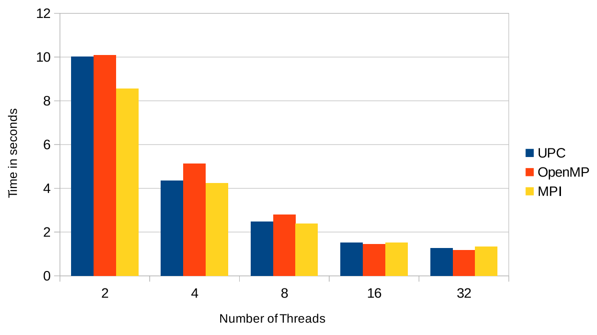

Fig. 8 IS Size C execution time in seconds. The lower, the better.

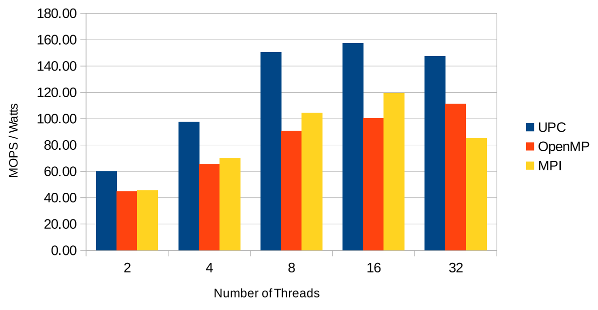

Fig. 5 MG Size C Performance per watt: MOPS / Watt. The higher, the better.

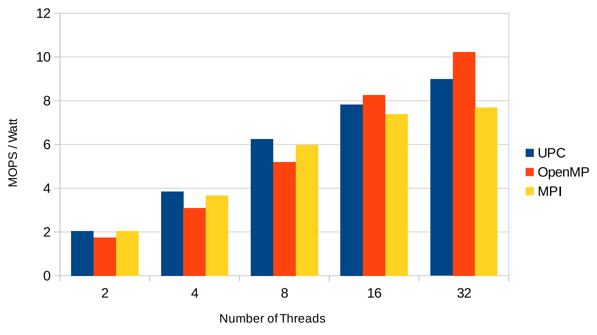

Fig. 9 IS Size C Performance per watt: MOPS / Watt. The higher, the better.

References

1. NAS Benhcmark Official Webpage - Problem Size (last accessed on 11/04/2015). URL www.nas.nasa.gov/publications/npb_problem_sizes.html
2. NAS Benchmark implemented in UPC (last accessed on 28/032015). URL https://threads.hpcl.gwu.edu/sites/npb-upc
3. NAS Benhcmark Official Webpage (last accessed on 28/032015). URL http://www.nas.nasa.gov/publications/npb.html
4. 500, T.G.: The 500 Green - Energy Efficient High Performance Computing Power Measurement Methodology (last accessed on 28/032015)
5. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The NAS Parallel Benchmarks. International Journal of High Performance Computing Applications **5**(3), 63–73 (1991)
6. Benedict, S.: Energy-aware performance analysis methodologies for hpc architectures—an exploratory study. Journal of Network and Computer Applications **35**(6), 1709–1719 (2012)
7. Berkeley: UPC Implementation From Berkeley (last accessed on 28/03/2015). URL http://upc.lbl.gov
8. Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y., Chavarría-Miranda, D.: An evaluation of global address space languages: co-array fortran and unified parallel C. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 36–47. ACM (2005)
9. Cupertino, L., Da Costa, G., Pierson, J.M.: Towards a generic power estimator. Computer Science - Research and Development pp. 1–9 (2014). DOI 10.1007/s00450-014-0264-x. URL http://dx.doi.org/10.1007/s00450-014-0264-x
10. El-Ghazawi, T., Cantonnet, F.: UPC performance and potential: A NPB experimental study. In: Supercomputing, ACM/IEEE 2002 Conference, pp. 17–17. IEEE (2002)
11. Inc., I.T.: UPC Implementation on GCC (last accessed on 28/03/2015). URL http://www.gccupc.org/
12. Intel: Intel PCM Official Webpage (last accessed on 28/032015). URL https://software.intel.com/en-us/articles/intel-performance-counter-monitor
13. Jose, J., Luo, M., Sur, S., Panda, D.K.: Unifying UPC and MPI runtimes: experience with MVAPICH. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, p. 5. ACM (2010)
14. Karpov, V., Kryukov, Y., Savransky, S., Karpov, I.: Nucleation switching in phase change memory. Applied physics letters p. 123504 (2007)
15. Kunkel, J., Aguilera, A., Hübbe, N., Wiedemann, M., Zimmer, M.: Monitoring energy consumption with SIOX. Computer Science - Research and Development pp. 1–9 (2014). DOI 10.1007/s00450-014-0271-y. URL http://dx.doi.org/10.1007/s00450-014-0271-y
16. Lively, C., Taylor, V., Wu, X., Chang, H.C., Su, C.Y., Cameron, K., Moore, S., Terpstra, D.: E-AMOM: an energy-aware modeling and optimization methodology for scientific applications. Computer Science - Research and Development **29**(3-4), 197–210 (2014). DOI 10.1007/s00450-013-0239-3. URL http://dx.doi.org/10.1007/s00450-013-0239-3
17. Luo, M., Wang, H., Panda, D.K.: Multi-Threaded UPC Runtime for GPU to GPU communication over InfiniBand. In: Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS'12) (2012)
18. Mallón, D.A., Taboada, G.L., Teijeiro, C., Touriño, J., Fraguela, B.B., Gómez, A., Doallo, R., Mouriño, J.C.: Performance evaluation of MPI, UPC and OpenMP on multicore architectures. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 174–184. Springer (2009)
19. Marc Tajchman, C.: Programming paradigms using PGAS-based languages. Figure used with the courtesy of Marc Tajchman (2015). URL http://www-sop.inria.fr/manifestations/cea-edf-inria-2011/slides/tajchman.pdf
20. Milthorpe, J., Rendell, A.P., Huber, T.: Pgas-fmm: Implementing a distributed fast multipole method using the x10 programming language. Concurrency and Computation: Practice and Experience **26**(3), 712–727 (2014)
21. MVAPICH, O.S.U.: MVAPICH Official Webpage (last accessed on 28/032015). URL http://mvapich.cse.ohio-state.edu/overview/
22. Scogland, T.R., Steffen, C.P., Wilde, T., Parent, F., Coghlan, S., Bates, N., Feng, W.c., Strohmaier, E.: A Power-measurement Methodology for Large-scale, High-performance Computing. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14, pp. 149–159. ACM, New York, NY, USA (2014). DOI 10.1145/2568088.2576795. URL http://doi.acm.org/10.1145/2568088.2576795
23. Seo, S., Jo, G., Lee, J.: Performance characterization of the NAS Parallel Benchmarks in OpenCL. In: Workload Characterization (IISWC), 2011 IEEE International Symposium on, pp. 137–148. IEEE (2011)
24. Shan, H., Wright, N.J., Shalf, J., Yelick, K., Wagner, M., Wichmann, N.: A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. SIGMETRICS Perform. Eval. Rev. **40**(2), 92–98 (2012). DOI 10.1145/2381056.2381077. URL http://doi.acm.org/10.1145/2381056.2381077
25. Shoukourian, H., Wilde, T., Auweter, A., Bode, A.: Monitoring power data: A first step towards a unified energy efficiency evaluation toolset for HPC data centers. Environmental Modelling & Software **56**, 13–26 (2014)
26. Tolentino, M., Cameron, K.W.: The Optimist, the Pessimist, and the Global Race to Exascale in 20 Megawatts. Computer **45**(1), 95–97 (2012). DOI http://doi.ieeecomputersociety.org/10.1109/MC.2012.34
27. Wikipedia: Wikipedia Definition of PGAS - Last accessed on 27/03/2015 (2015)
28. Wong, F.C., Martin, R.P., Arpaci-Dusseau, R.H., Culler, D.E.: Architectural requirements and scalability of the NAS parallel benchmarks. In: Supercomputing, ACM/IEEE 1999 Conference, pp. 41–41. IEEE (1999)
29. Wu, X., Taylor, V.: Performance characteristics of hybrid MPI/OpenMP implementations of NAS parallel benchmarks SP and BT on large-scale multicore supercomputers. ACM SIGMETRICS Performance Evaluation Review **38**(4), 56–62 (2011)

Appendix

Measuring energy consumption of parallel applications

Experiments based on measuring NAS Benchmarks implemented in OpenMP, MPI and UPC

Author
Jérémie Lagravière – PhD candidate – Simula Research – www.simula.no – jeremie@simula.no

Table of Contents

Introduction....3
Technical information about our experiments....3
Hardware architecture....3
Selected benchmark....3
Programming languages....4
Intel Performance Counter Monitor....5
Installation....5
Download....5
Instructions....5
Compiling the tools....5
Compiling the library....6
How to use....6
Global Information....7
Intel PCM in the code....8
Intel PCM from the command line....9
Measuring energy....9
Monitoring Communications....10
Tracking memory traffic....12
Tracking memory bandwidth....13
Turbostat: another way to measure energy consumption....14
How to install....14
How to use....14
Bibliography....15

Introduction

In this document we present a set of practical solutions to measure the energy consumption of parallel programs on systems based on Intel Xeon processors.

The energy measurements are performed on the software side, which is a necessity when direct access to the hardware is not possible.

We have used the NAS Benchmark (see next chapter for a description) and the Intel Performance Counter Monitor[1] to evaluate the energy consumption.

Technical information about our experiments

Hardware architecture

To run our experiments we have used Intel PCM v2.8 on a two socket machine equipped with two Intel(R) Xeon(R) CPU E5-2650 and 32 GB RAM.

Selected benchmark

The NAS Benchmark, is a set of kernels that provides different ways to stress a supercomputer. The NAS Benchmark is implemented in Fortran and C, we also use the UPC version of the NAS Benchmark. In our study, we have selected four kernels: Integer Sort (IS), Conjugate Gradient (CG), Multi-Grid (MG) and Fourier Transformation (FT).

IS represents a large integer sort. This kernel performs a sorting operation that is important in particle method codes. It evaluates both integer computation speed and communication performance.

MG is a simplified multi-grid kernel. Multigrid (MG) methods in numerical analysis are a group of algorithms for solving differential equations using a hierarchy of discretizations. They are an example of a class of techniques called multiresolution methods, very useful in (but not limited to) problems exhibiting multiple scales of behavior.

CG is a conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular all-to-all communication, using unstructured matrix-vector multiplication.

FT is a three-dimensional partial differential equation solution using Fast Fourier Transformations. This kernel performs the essence of many spectral codes.

1 https://software.intel.com/en-us/articles/intel-performance-counter-monitor

The NAS benchmark provides very different communications schemes, which is important in order to stress and evaluate the performance of the selected languages.

Programming languages

We used the OpenMP and MPI version of NAS Bencharmk available at:
http://www.nas.nasa.gov/publications/npb.html

We also used the UPC version of NAS Benchmark, available at:
https://threads.hpcl.gwu.edu/sites/npb-upc

Compiled with the Berkeley UPC Compiler, available at:
http://upc.lbl.gov/

Intel Performance Counter Monitor

Intel Performance Counter Monitor is a practical tool to measure energy consumption, memory usage and cache behavior on a compatible computer (Intel Xeon Processor):

The complexity of computing systems has tremendously increased over the last decades. Hierarchical cache subsystems, non-uniform memory, simultaneous multithreading and out-of-order execution have a huge impact on the performance and compute capacity of modern processors.

Software that understands and dynamically adjusts to resource utilization of modern processors has performance and power advantages. The Intel® Performance Counter Monitor provides sample C++ routines and utilities to estimate the internal resource utilization of the latest Intel® Xeon® and Core™ processors and gain a significant performance boost.

Installation

Download

Intel PCM can be downloaded at this URLs:

- https://software.intel.com/en-us/articles/intel-performance-counter-monitor

  or

- https://software.intel.com/protected-download/328018/326559

From this link you obtain a zip file, such as: IntelPerformanceCounterMonitorV2.8.zip

Instructions

Unzip the file that you downloaded in a directory intelPCM/

Compiling the tools

Intel PCM provides some ready made tools (described in How to use Intel PCM).

To compile these tools, run this command:

```
IntelPCM $ make
```

All you need is a working C++ compiler configured in environment variable CXX. If the compilation process ran correctly you should obtain this set of executable:

- ./pcm-pcie.x
- ./pcm-numa.x
- ./pcm-memory.x
- ./pcm.x
- ./pcm-sensor.x
- ./pcm-power.x
- ./pcm-msr.x
- ./pcm-tsx.x

Compiling the library

In the directory IntelPCM/intelpcm.so/ you can compile Intel PCM as a shared library in order to use Intel PCM API directly in your code.

```
IntelPCM/intelpcm.so $ make
```

All you need is a working C++ compiler configured in environment variable CXX.

If the compiling process ran correctly you should obtain these files:

- client_bw.o
- cpucounters.o
- libintelpcm.so
- msr.o
- pci.o

How to use

There are three different ways to use Intel PCM:

- In the code of your application, you add some Intel PCM function calls to measure acurately any kind of metrics related to energy consumption, memory, bandwidth etc.;
- From the command line, by specifying an external program to make measurements from;
- From the command line by without specifying an external program. This gives general information and measurements on the whole computer.

Global Information

When measuring energy on any system, root access is required. This is due to the fact that Intel PCM has to read "MSR" registers (not write though) only accessible by the root.

Intel PCM requires a compatible processors: for all of our experiments we have used a dual socket Intel(R) Xeon CPU E5-2650 @ 2.00GHz, delivering 16 physical cores, and 32 cores with HyperThreading.

Intel PCM in the code

Here is an example of C++ code using Intel PCM to perform measurements

```
#include <stdlib.h>
#include <iostream>
#include <stdio.h>
#include "cpucounters.h"
int main(void)
{

       //Instantiate Intel PCM singleton
       PCM * m = PCM::getInstance();

       //Creation of a counter before the code to measure
       SystemCounterState before_sstate = getSystemCounterState();


       //Some code that you want to measure


       //Creation of a counter before the code to measure
       SystemCounterState after_sstate = getSystemCounterState();

       /*
       From the counters you can extract values such as:
       Instruction per clock, L3 Hit ratio, Memory transferred from the memory controller      to
L3 cache.
       And much more
       */

       cout << "Instructions per clock:" << getIPC(before_sstate,after_sstate) << endl;

       cout<<"L3 cache hit ratio:" << getL3CacheHitRatio(before_sstate,after_sstate) << endl;

       cout << "Bytes read from memory controller: ";

       cout << getBytesReadFromMC(before_sstate,after_sstate) / double(1024ULL * 1024ULL * 1024ULL)
<< endl;

       m->cleanup();

}
```

This code sample show the usage of a few available Intel PCM functionality, more at:
http://intel-pcm-api-documentation.github.io/annotated.html

Intel PCM from the command line

In this section we use Intel PCM from a terminal in order to get some measurements: energy, power, memory usage, bandwidth etc. while a given program is running.

All the Intel PCM program that we describe are used in this way:

```
intelProgram --external_program programToExecute <options for the program to
execute>
```

This command implies that the intelProgram will perform measurements as long as programToExecute is running.

Reminder: all the commands are supposed to be run as root

Measuring energy

To measure energy consumption with Intel PCM, we use the pcm-power.x program as follow

```
pcm-power.x --external-program programToExecute
```

For example with an MPI application

```
pcm-power.x --external-program mpirun --allow-run-as-root -n 32 ./cg.C.32
```

Pcm-power.x will produce this kind of output:

```
Time elapsed: 25757 ms
Called sleep function for 1000 ms
[...]
S0; Consumed energy units: 119574580; Consumed Joules: 1824.56; Watts: 70.84; Thermal
headroom below TjMax: 42
S0; Consumed DRAM energy units: 33159075; Consumed DRAM Joules: 505.97; DRAM Watts:
19.64
[...]
S1; Consumed energy units: 117557570; Consumed Joules: 1793.79; Watts: 69.64; Thermal
headroom below TjMax: 48
S1; Consumed DRAM energy units: 41041785; Consumed DRAM Joules: 626.25; DRAM Watts:
24.31
```

For each socket (S0, S1) we got measurements of the consumed Joules, and the related power in Watts (Joules / s).
Pcm-power.x also outputs the energy consumption, for each socket, of the memory controller.
This measurements DOES NOT include the energy consumed by the Quick Path Interconnects, because our hardware setup is not compatible for such measurements.

Monitoring Communications

Using pcm.numa.x we are able to have a view of the inter-socket communication scheme of a given program.

To do so, we run a command that looks like this:

```
pcm-numa.x --external_program programToExecute
```

For example we ran MG over 32 cores:

```
pcm-numa.x --external_program mpirun  --allow-run-as-root -n 32 ./mg.C.32
```

This gives an output that looks like this

```
Core | IPC  | Instructions | Cycles  |  Local DRAM accesses | Remote DRAM Accesses
   0   0.00         22 G     18446462 T       222 M                26 M
   1   0.00         22 G     18446462 T       221 M                25 M
[...]
  30   0.00         23 G     18446462 T       219 M                27 M
  31   0.00         23 G     18446462 T       218 M                28 M
-------------------------------------------------------------------------------------
   *   0.00        728 G     18437737 T      7028 M               869 M
```

We can specifically look at the local and remote accesses, that are given in million of cache lines.
The measurements are available both globally (whole computer) and by core (from 0 to 31).

If we run MG over 16 cores we will get a different communication scheme:

```
pcm-numa.x --external_program mpirun  --allow-run-as-root -n 16 ./mg.C.16
```

```
Time elapsed: 9666 ms
Core | IPC  | Instructions | Cycles  |  Local DRAM accesses | Remote DRAM Accesses
   0   2.04         28 G       14 G       254 M               28 M
   1   2.02         40 G       19 G       354 M               38 M
   2   2.02         40 G       19 G       355 M               38 M
   3   2.06         34 G       16 G       298 M               33 M
   4   2.06         29 G       14 G       257 M               28 M
   5   2.06         24 G       11 G       213 M               23 M
   6   1.98       9500 M     4793 M        84 M             9593 K
   7   2.05       9486 M     4621 M        84 M             9323 K
   8   1.99         29 G       14 G       259 M               29 M
   9   2.02         40 G       20 G       353 M               39 M
  10   2.07       9817 M     4734 M        85 M             9487 K
  11   2.03         40 G       19 G       351 M               40 M
  12   0.33         51 M      155 M       332 K              485 K
  13   2.02         40 G       19 G       353 M               40 M
  14   0.90         35 M       39 M       267 K               28 K
  15   1.52        120 M       79 M      1025 K               56 K
  16   1.88         11 G     5955 M       100 M               11 M
  17   0.27         12 M       47 M        40 K               13 K
  18   0.64         42 M       66 M        72 K               42 K
  19   1.76       6148 M     3502 M        56 M             6465 K
  20   1.89         11 G     5844 M        98 M               11 M
  21   1.93         15 G     8240 M       141 M               15 M
  22   2.00         30 G       15 G       270 M               30 M
  23   2.00         30 G       15 G       270 M               30 M
  24   2.06         10 G     5252 M        93 M               10 M
  25   0.74         64 M       87 M        59 K               90 K
  26   2.01         30 G       15 G       267 M               30 M
  27   0.38       3942 K       10 M        10 K             6369
  28   2.02         40 G       19 G       353 M               40 M
  29   0.44         32 M       73 M        85 K              155 K
  30   2.02         40 G       19 G       351 M               40 M
  31   2.02         40 G       20 G       350 M               40 M
-------------------------------------------------------------------------------------
*   2.01       645 G      320 G      5661 M             638 M
```

In the output above, the 32 cores are still monitored, because Intel PCM does not take in account the fact that we required only 16 cores in the mpi command.

Tracking memory traffic

Using pcm.x we are able to have a view on what happens at different memory and cache levels when running a given program.

To do so, we run a command that looks like this:

```
pcm.x --external_program programToExecute
```

We used this command to illustrate the use of pcm.x

```
pcm.x --external_program  mpirun --allow-run-as-root --cpu-set 0-15 -bind-to core -n
16 ./mg.C.16
```

We obtain this output:

```
 Core (SKT) | EXEC | IPC  | FREQ  || L3MISS | L2MISS | L3HIT | L2HIT |||  READ | WRITE | TEMP

   0    0     1.84   2.06   0.89      39 M     46 M    0.16    0.39       N/A     N/A     48
   1    0     1.58   2.06   0.77      34 M     40 M    0.15    0.39       N/A     N/A     50
[...]
  30    1     2.08   2.02   1.03      48 M     56 M    0.14    0.37       N/A     N/A     57
  31    1     2.08   2.00   1.04      48 M     57 M    0.15    0.36       N/A     N/A     51
-------------------------------------------------------------------------------------------------------------
 SKT    0     1.04   2.01   0.52     392 M    456 M    0.14    0.37      178.29    68.94     46
 SKT    1     1.04   2.01   0.52     391 M    454 M    0.14    0.36      178.35    69.42     50
-------------------------------------------------------------------------------------------------------------
 TOTAL  *     1.04   2.01   0.52     784 M    911 M    0.14    0.36      356.65    138.36     N
```

For memory traffic the values to consider are the collumns “READ” and “WRITE”.

In our case, to run MG in MPI over 16 cores 356GB of data were read from memory and 138GB were written to memory in total.

These values are also available by socket: SKT 0 and SKT 1.

Tracking memory bandwidth

By using pcm.memory.x it is possible to have a view of the bandwidth usage during the execution of a given program.

To do so, we run a command that looks like this:

```
pcm-memory.x --external_program programToExecute
```

We used this command to illustrate the use of pcm-memory.x:

```
pcm-memory.x --external_program  mpirun --allow-run-as-root --cpu-set 0-15 -bind-to core
-n 16 ./mg.C.16
```

```
---------------------------------------||---------------------------------------
--             Socket 0              --||--             Socket 1              --
---------------------------------------||---------------------------------------
---------------------------------------||---------------------------------------
---------------------------------------||---------------------------------------
--   Memory Performance Monitoring   --||--   Memory Performance Monitoring   --
---------------------------------------||---------------------------------------
--  Mem Ch 0: Reads (MB/s): 4953.48  --||--  Mem Ch 0: Reads (MB/s): 4944.73  --
--            Writes(MB/s): 1916.74  --||--            Writes(MB/s): 1914.39  --
--  Mem Ch 1: Reads (MB/s): 4950.63  --||--  Mem Ch 1: Reads (MB/s): 4936.47  --
--            Writes(MB/s): 1915.93  --||--            Writes(MB/s): 1907.49  --
--  Mem Ch 2: Reads (MB/s): 4952.48  --||--  Mem Ch 2: Reads (MB/s): 4932.35  --
--            Writes(MB/s): 1921.75  --||--            Writes(MB/s): 1907.39  --
--  Mem Ch 3: Reads (MB/s): 4949.20  --||--  Mem Ch 3: Reads (MB/s): 4934.67  --
--            Writes(MB/s): 1915.87  --||--            Writes(MB/s): 1907.20  --
-- NODE0 Mem Read (MB/s):  19805.79  --||-- NODE1 Mem Read (MB/s):  19748.21  --
-- NODE0 Mem Write (MB/s):  7670.29  --||-- NODE1 Mem Write (MB/s):  7636.47  --
-- NODE0 P. Write (T/s) :    180188  --||-- NODE1 P. Write (T/s):     180380  --
-- NODE0 Memory (MB/s):    27476.08  --||-- NODE1 Memory (MB/s):    27384.69  --
---------------------------------------||---------------------------------------
--                   System Read Throughput(MB/s):  39554.01                  --
--                  System Write Throughput(MB/s):  15306.76                  --
--                 System Memory Throughput(MB/s):  54860.77                  --
---------------------------------------||---------------------------------------
```

In this example we have access to the read and write performance for each memory channel for socket 0 and socket 1.

Global values are also given at the bottom of the generated table.

Turbostat: another way to measure energy consumption

How to install

Turbostat is a stand linux tools, it does not require any specific installation on most of the systems as long as you have this packages installed:

- linux-tools-common
- linux-tools-generic
- linux-cloud-tools-common

Turbostat

How to use

Turbostat is simple to use, it can be run with a command line that looks like this:

```
turbostat programToExecute <options for programToExecute>
```

We used this command to illustrate the use of turbostat:

```
turbostat  mpirun --allow-run-as-root --cpu-set 0-15 -bind-to core -n 16
./mg.C.16
```

This produces this output

```
pk cor CPU    %c0  GHz  Pkg_W  Cor_W RAM_W PKG_% RAM_%
            48.43 2.39 145.90 109.03 48.17  0.00  0.00
 0   0   0  71.90 2.40  73.50  55.13 21.04  0.00  0.00
 1   0   0  71.90 2.40  72.40  53.90 27.13  0.00  0.00
```

Values are given per socket and the first line is for global values (socket 0+socket 1).

Cor_W is the power consumption of the core itself, Pkg_W includes the Cor_W plus their dedicated caches, and the “uncore”, including their shared caches and the communication network between them, the PCI IO sub-system, and the memory controller.

Bibliography

UPC NAS Benchmarks
https://threads.hpcl.gwu.edu/sites/npb-upc

NAS Benchmarks
http://www.nas.nasa.gov/publications/npb.html

Intel PCM API
http://intel-pcm-api-documentation.github.io/annotated.html
https://software.intel.com/en-us/articles/intel-performance-counter-monitor

Intel PCM Download

https://software.intel.com/en-us/articles/intel-performance-counter-monitor

https://software.intel.com/protected-download/328018/326559

UPC Compiler

http://upc.lbl.gov/