On the performance and energy efficiency of the PGAS programming model on multicore architectures

Jérémie Lagravière¹, Johannes Langguth¹, Phuong H. Ha², and Xing Cai¹

¹ Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway
jeremie@simula.no, langguth@simula.no, xingca@simula.no

² The Arctic University of Norway, NO-9037 Tromsø, Norway
phuong.hoai.ha@uit.no

Abstract. Using large-scale multicore systems to get the maximum performance and energy efficiency with manageable programmability is a major challenge. The partitioned global address space (PGAS) programming model enhances programmability by providing a global address space over large-scale computing systems. However, so far the performance and energy efficiency of the PGAS model on multicore-based parallel architectures have not been investigated thoroughly. In this paper we use a set of selected kernels from the well-known NAS Parallel Benchmarks to evaluate the performance and energy efficiency of the UPC programming language, which is a widely used implementation of the PGAS model. In addition, the MPI and OpenMP versions of the same parallel kernels are used for comparison with their UPC counterparts. The investigated hardware platforms are based on multicore CPUs, both within a single 16-core node and across multiple nodes involving up to 1024 physical cores. To provide insight into the observed performance differences, we use the Intel Performance Counter Monitor to quantify in detail the communication time, cache hit/miss ratio and memory usage. Our experiments show that UPC is competitive with OpenMP and MPI on single and multiple nodes, with respect to both the performance and energy efficiency.

1 Introduction & Motivation

The overarching complexity of parallel programming is one of the fundamental challenges that the HPC research community faces. In the last decade, the Partitioned Global Address Space model or PGAS has been established as one possible solution for this problem. It promises improved programmability while maintaining high performance, which is the primary goal in HPC. In recent years, energy efficiency has become an additional goal. Optimizing energy efficiency without sacrificing computational performance is the key challenge of energy-aware HPC, and an absolute requirement for attaining Exascale computing in the future. This requirement has been codified by the US Department of Energy, which imposes a limit of 20 MegaWatts for Exascale machines [24]. The reason for this is twofold:
reducing both the electricity cost and the environmental footprint of supercomputers.

In this study we investigate whether PGAS can meet the HPC goals. We focus on UPC, one of the most widely used PGAS implementations and compare it to the MPI, the standard for the explicit message passing model, and OpenMP used for shared-memory parallel programming. Both have been used for many years in parallel programming and HPC. OpenMP offers ease of programming for shared memory machines, while MPI offers high performance on distributed memory supercomputers.

UPC, and PGAS in general, aims at combining these advantages 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 data private or shared, PGAS languages ensure data consistency across the different memory regions. When necessary, shared data is transferred automatically between the nodes through a communication library such as GASnet [3].

Several recent studies advocate the use of PGAS as a promising solution for HPC [25, 28], many of which have focused on the evaluation of PGAS performance and UPC in particular [15–17, 20, 22, 27, 33]. However, the previous UPC studies haven’t taken energy efficiency into consideration. This motivates us to investigate UPC’s energy efficiency and performance using the latest CPU architecture with advanced support for energy and performance profiling.

For our evaluation we use the well-established NAS Benchmark provided by NASA. We use MPI, OpenMP [9], and UPC implementations [8] to compare the performance and energy efficiency of the different programing models. The energy measurements are done on a single-node machine using Intel PCM [18]. The multi-node performance measurements are obtained on an Intel Xeon based supercomputer for both UPC and MPI. We provide an analysis of the performed measurements in order to explain the difference in performance, by focusing on the cache performance and the memory traffic of MPI, OpenMP and UPC.

This paper improves upon previous works [15–17, 22, 27, 33] by: (1) providing measurements for a larger number of nodes and cores for MPI, OpenMP and UPC on recent single-node and multi-node systems; (2) including energy measurements of both UPC, OpenMP and MPI on a single node (up to 16 physical cores); (3) investigating the reasons for the difference in performance between UPC, OpenMP and MPI.

The remainder of this paper is organized as follows: Section 2 briefly presents the UPC framework and the reasons why have chosen this programming language. Section 3 describes the benchmark chosen for this study. Section 4 explains the hardware and software set-up used for running our experiments, the results of which are presented in Section 5 and discussed in Section 6. Section 7 concludes the paper.
2  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 [14, 31]. Figure 1 shows a view of the communication model of the PGAS paradigm [23]. In this model, each node ($C_0$ or $C_1$) has access to both private and shared memory. Accessing the shared memory to either read or write data can imply inter-node communication which is handled automatically by the runtime. The blue arrow in Figure 1 represents remote access to the shared memory. Conceptually, such distant accesses work in an RDMA (Remote Direct Memory Access) fashion, providing one-sided communication that is not visible to the remote process. However, most PGAS languages are built over a low-level communication layer which limits their physical capabilities. Thus, actual RDMA is available only if the underlying hardware and software support it.

Fig. 1. PGAS Communication Model [23] - Figure courtesy of Marc Tajchman

In recent years several languages implementing the PGAS model have been proposed. UPC, which is essentially an extension of the C language, was one of the first among them and also one of the most stable ones [23]. Other members of the PGAS family of languages include the Fortran counterpart, Coarray Fortran [1], X10 [6], and Cray Chapel [2]. In addition, libraries such as Global Arrays [4] and SHMEM/OpenSHMEM [5] which implement PGAS functionality are available.
The key characteristics of UPC are:

- A parallel execution model of Single Program Multiple Data (SPMD);
- Distributed data structures with a global addressing scheme, with static or dynamic allocation;
- Operators on these structures, with affinity control;
- Copy operators between private, local shared, and distant shared memories.

Additionally, multiple open-source implementations of the UPC compiler and runtime environment are available, in particular Berkeley UPC [13], GCC/UPC [19] and CLANG/UPC [7].

3 The NAS Benchmark

The NAS Benchmark [11] consists of a set of kernels that each provides a different way of testing the capabilities of a supercomputer. The NAS Benchmark was originally implemented in Fortran and C. We use the C implementation for OpenMP and MPI, as well as the UPC version of the benchmark [8]. For our study, we select four kernels: Integer Sort (IS), Conjugate Gradient (CG), Multi-Grid (MG), and Fourier Transformation (FT).

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

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.

CG, IS, MG and FT are selected because they are the most relevant ones: stressing memory, communication and computation. They involve very different communications patterns, which is important for evaluating the performance of the selected languages (see Section 5). The other kernels in the NAS Benchmark are of limited relevance to this study. See [11] for their descriptions.
4 Experimental Setup

In this section we describe the software and hardware solutions that we used to carry out our experiments. We ran the NAS kernels both on a single-node machine and on Abel [30], a supercomputer operated by the University of Oslo, using a varying number of cores and nodes.

4.1 Hardware

Table 1 shows the specifications of the systems used in our experiments. Both architectures are equipped with Sandy Bridge processors, with the multi-node machine having a higher CPU frequency.

<table>
<thead>
<tr>
<th>Type of architecture</th>
<th>Single Node</th>
<th>Multi Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>2</td>
<td>2 (per node)</td>
</tr>
<tr>
<td>Number of cores</td>
<td>16</td>
<td>16 (per node)</td>
</tr>
<tr>
<td>HyperThreading</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CPU model</td>
<td>Intel Xeon E5-2650 @ 2.00GHz</td>
<td>Intel Xeon E5-2670 @ 2.60GHz</td>
</tr>
<tr>
<td>Interconnect</td>
<td>N/A</td>
<td>Infiniband: 6.8 GB/s</td>
</tr>
</tbody>
</table>

4.2 Software

On both the single-node machine and the multi-node supercomputer we used UPC version 2.22.0, the Intel Compiler version 15.0.1, and MPI Library 5.0 Update 2 for Linux. The single-node machine runs Ubuntu 12.04.5 LTS. The multi-node supercomputer runs CentOS release 6.7.

Benchmarking

For each measurement we executed three separate runs and reported the best result. Doing so filters out OS interference since the algorithms are deterministic. Furthermore, variations in the running time were generally very small.

On the single-node machine we used size Class C [9, 10] for each kernel:

- IS Class C: Number of keys is $2^{27}$
- 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 $512 \times 512 \times 512$

There are 16 physical cores available on the test system. Results for 32 threads used hyperthreading.
On the multi-node supercomputer we used for each kernel Class D, except for IS which is not available in Class D in the UPC implementation [8]:

- IS Class C: Number of keys is $2^{27}$
- FT Class D: Grid size is $2048 \times 1024 \times 1024$
- CG Class D: Number of rows is 1500000
- MG Class D: Grid size is $1024 \times 1024 \times 1024$

Each kernel was run using up to 1024 CPU cores and thus 64 nodes of the supercomputer. However, for CG and IS limitations in the UPC implementation prevent us from using more than 256 threads, see Figures 2 and 5.

Sizes C and D provide data sets that are sufficiently large to exceed the cache size of the test systems [32] [10].

**Thread binding**

Thread binding or thread pinning is an approach that associates each thread with a specific processing element. In our experiments we applied thread/process binding to the physical cores. Scattering, i.e. dividing the threads evenly between the sockets of each node gave the best results in terms of both execution time and energy consumption. Consequently, scattering was used in all our experiments.

### 4.3 Energy Measurements

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 energy efficiency experiments on the single-node platform [18].

Intel PCM requires both root access and activation of counters in the BIOS or UEFI. It uses the Machine Specific Registers (MSR) and RAPL counters to disclose the energy consumption details of an application [12]. This is a major constraint when running experiments on supercomputers. For this reason we report energy measurements for the single-node system only.

Intel PCM is able to identify the energy consumption of the CPU(s), the RAM and the Quick Path Interconnect (QPI) between the sockets. However, in this study we focus only on the total energy consumption of the different programming models, and QPI energy consumption is not taken into account because Intel PCM was unable to provide measurement on the chosen hardware platform.

To better highlight the difference in energy efficiency among the three programming models, the reported energy consumption values have been corrected to subtract the idle energy consumption due to the operating system and the different services/daemons. To do so, we measured the average energy consumption of both CPU and RAM during one hour in which the system was idle. The idle energy consumption for the single-node machine is 19.5 watts from the CPU and RAM.
5 Results

In this paper, we consider two metrics to measure the performance and energy efficiency. To evaluate the performance we use Million Operations Per Second (MOPS or MOP/s). This metric is used for both the multi-node measurements and single-node measurements. To evaluate the energy efficiency, Millions Operations Per Seconds over Watts (MOPS / Watt) are used to measure the performance per power unit. The 500 Green - Energy Efficient High Performance Computing Power Measurement Methodology [29] advises this measurement methodology.

We use the following notation: [kernel name]-[number of threads/processes] to refer to the different experiments. For instance, CG-32 stands for the Conjugate Gradient kernel running on 32 threads (or processes for MPI).

5.1 Performance measurement on multi-node architecture

Figures 2, 3, 4, and 5 show the performance expressed in MOPS for the four kernels: CG, MG, FT and IS. Each kernel ran on up to 1024 threads (or MPI processes), except for IS and CG where the UPC implementation cannot run on more than 256 threads. Each of these figures shows the results for both UPC and MPI.

The performance results show that the kernels scale over the number of cores independently of the language. MPI is a clear winner when running on more than 32 cores, however UPC achieves a performance that is close to that of MPI. These results match those of previous studies, in particular [27].

![Fig. 2. Multi-node performance of the CG kernel - Class D](image-url)
Fig. 3. Multi-node performance of the MG kernel - Class D

Fig. 4. Multi-node performance of the FT kernel - Class D
Fig. 5. Multi-node performance of the IS kernel - Class C

UPC delivers slightly better performance than MPI for MG-16, CG-16, FT-16, and IS-16, with a performance advantage of 7.69%, 5.46%, 7.58%, and 1.96%, respectively.

On this cluster using modern CPU architectures, UPC and MPI implementations scale well and their performance is comparable: the UPC implementations outperform MPI by at most 8% while the MPI implementations outperform UPC by at most 12%.

5.2 Performance and energy efficiency on single-node architecture

To the best of our knowledge, this is the first investigation of UPC’s energy efficiency. Our experimental results show that the energy efficiency of UPC, MPI, and OpenMP implementations scale over the number of cores and are comparable to each other. In this section we give more details about these results and in Section 6 we provide an analysis of the difference in performance between UPC, OpenMP, and MPI.

Figures 6, 7, 8, and 9 show the performance expressed in MOPS for the four kernels. Each kernel ran on up to 32 threads/processes. Each of these figure shows the results for all three programing models.

Figures 10, 11, 12, and 13 show the energy efficiency expressed in MOPS per watt for all the four kernels and all three programing models. The energy efficiency results show that the kernels scale over the number of threads/cores independently of the language. There is no clear winner since none of the chosen languages is better than the other two competitors for all the kernels.
Fig. 6. Single-node performance of the CG kernel - Class C

Fig. 7. Single-node performance of the MG kernel - Class C
Fig. 8. Single-node performance of the FT kernel - Class C

Fig. 9. Single-node performance of the IS kernel - Class C
Using 32 threads implies using the HyperThreading capability of the CPU. In most cases, only OpenMP benefited from HyperThreading. In CG-32, MG-32, FT-32, and IS-32, UPC and MPI achieved lower performance than the same kernel running on 16 threads (processes). When using 16 threads or less, each thread runs on a physical core.

Even though there is no global winner in the obtained single-node measurements, we are interested in knowing whether UPC performs well in terms of the computation performance and energy efficiency. Concerning the execution time, UPC is the best in CG-4, CG-8, CG-16, and in all MG kernels. 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 kernels and thread-counts mentioned above. UPC is the best in MG-2 to MG-32, CG-8, CG-16, and FT-8. It also competes well with OpenMP and MPI in this aspect.

As described in [27], on a single-node platform, UPC scales well over more CPU cores and competes well with OpenMP and MPI. However, the performance of OpenMP is better in many cases.

![Fig. 10. Single-node energy efficiency of the CG kernel - Class C](image-url)
In this section, we give an explanation of the differences in performance and energy efficiency that were observed in the previous section.

Intel PCM provides access to metrics such as memory traffic and hit and miss rates for L2 and L3 cache. In this section we will use measurements of these metrics to explain the differences in performance among OpenMP, MPI, and UPC. Table 2 shows the measurements obtained via Intel PCM.

For instance, in Figure 6 and Figure 10, which show performance and power efficiency for the Conjugate Gradient kernel, there is a noticeable difference between the results obtained using UPC and OpenMP: in CG-8 and CG-16, UPC wins. By looking at the behavior of the OpenMP implementation compared to the UPC implementation, we can see that the L2 cache hit ratio for UPC is higher than that for OpenMP: UPC L2 cache hit ratio is 56% while the OpenMP L2 cache hit ratio is only 14%. As a direct consequence the OpenMP implementation of CG running over 16 threads uses more CPU cycles due to L2 cache misses and then fetching data from L3 cache. 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 OpenMP implementation and the UPC implementation. In CG-16, UPC performs better than MPI. In this situation MPI has slightly lower performance in L2 cache hit ratio than UPC: 56% for UPC and 53% for MPI.

In Figure 11, representing the Multigrid power efficiency, and Figure 7 representing the Multigrid performance in MOPS, UPC performs better than OpenMP on 16 threads. In this case, the analysis of the memory traffic (amount of data
that is read and written from/to RAM) is the relevant metric that determines the cause of the lower performance of OpenMP. In total OpenMP reads and writes 520 GB of data into memory while the UPC only reads and writes 419 GB. This difference of 101 GB is the main cause for the difference in performance as all the other metrics indicate similar or comparable values.

Figure 12 shows the power efficiency of the 3D-Fourier Transform, while Figure 8 shows the performance in MOPS. Clearly OpenMP performs better than UPC. Here it is both the L3 cache miss ratio 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 950 GB of data from/to memory and has a L3 cache hit ratio of 15%, while the OpenMP implementation reads and writes only 505 GB of data from/to memory and has a L3 cache hit ratio of 62%.

Finally, for the Integer Sort performance and power efficiency, shown in Figure 9 and 13, OpenMP performs better than MPI and UPC on both 16 and 32 threads.
For IS-16, the OpenMP L2 cache behavior is much better than that of UPC. Additionally, OpenMP performs fewer reads and writes than UPC: 32 GB against 59 GB. In this scenario (FT-16), MPI has lower L2 cache hit ratio than UPC and OpenMP, and MPI reads and writes more data from/to memory than UPC and OpenMP.

Globally the results presented show a correlation between the number of cores used to run a given kernel and the achieved power efficiency. However, the aim of this study is not to make a general rule from these observations. Nevertheless, we have hints to explain why the performance per watt is better when using all the physical cores on our single-node system: as the speedup of the performance, for example between CG-8 and CG-16 in OpenMP, is 1.75 and the Watts consumption between CG-8 and CG-16 in OpenMP increases by 1.58, the performance per watt increases despite the increase in power usage.
7 Conclusion

In this study, we have measured the energy efficiency and the computational performance of four kernels from the NAS Benchmark, both on a single-node system and on a multi-node supercomputer using three different programming models: UPC, MPI, and OpenMP. On the multi-node supercomputer we observed that UPC is almost always inferior to MPI in terms of performance, although UPC scales well to 1024 cores and 64 nodes, the maximum system size used in this study. From the measurements performed on the single-node system, we observe that by using more 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 computational speed and energy efficiency. We obtained this conclusion by analyzing the results produced on the single-node system in order to localize the origin of difference in performance. We found that data locality is the main reason for the difference in performance. From this analysis we concluded that UPC could be improved in this aspect.

Our conclusions about UPC are compatible with results obtained in previous studies, in particular [22, 27]. We confirm that UPC can compete with both MPI and OpenMP.

In future work, we will explore hardware accelerators such as Many Integrated Cores (MIC) and GPUs. These accelerators are well-known for being more energy efficient than CPUs for many applications. However, this requires both the benchmark and UPC to support these accelerators.

Furthermore, it would be interesting to investigate why UPC has less memory-L3 data traffic than OpenMP for MG, but more data transfer for FT and IS. This investigation would provide some insights into how to improve UPC for the computation/access patterns represented by FT and IS. Also, in order to enhance the energy measurements, we aim for a more fine grained approach of measuring the energy consumption. By studying the energy cost of computation, communication between nodes, and between CPU and memory separately, we can suggest improvements to energy consumption both in the user codes and in the UPC compiler and runtime environment.

Finally, an additional objective of our future work is to validate the quality of the software-based energy measurement with hardware-based solutions in order to improve precision in our energy measurements.
References

2. Cray - Chapel Official Webpage (last accessed on 17/12/2015), http://chapel.cray.com
3. GASNET Official Webpage (last accessed on 17/12/2015), http://gasnet.lbl.gov
5. OpenSHMEM Official Webpage (last accessed on 17/12/2015), http://openshmem.org/site/
6. X10 Official Webpage (last accessed on 17/12/2015), http://x10-lang.org
7. CLANG UPC (last accessed on 18/12/2015), https://clangupc.github.io
8. NAS Benchmark Implemented in UPC (last accessed on 18/12/2015), https://threads.hpcl.gwu.edu/sites/npb-upc
10. NAS Benchmark Official Webpage - Problem Size (last accessed on 18/12/2015), www.nas.nasa.gov/publications/npb_problem_sizes.html
19. Intrepid Technology Inc.: UPC Implementation on GCC (last accessed on 18/12/2015), http://www.gccupc.org/


23. Marc Tajchman, CEA: Programming paradigms using PGAS-based languages. Figure used with the courtesy of Marc Tajchman (2015), http://www-sop.inria.fr/manifestations/cea-edf-inria-2011/slides/tajchman.pdf


26. MVAPICH, O.S.U.: MVAPICH Official Webpage (last accessed on 18/12/2015), http://mvapich.cse.ohio-state.edu/overview/


30. UiO: Abel SuperComputer Official Webpage (last accessed on 15/12/2015), http://www.uio.no/english/services/it/research/hpc/abel/

