Performance Evaluation of Intel and Portland Compilers Using Intel Westmere Processor

M. Al-Mulhem, S. Sait
Department of Computer Science, KFUPM
{mulhem, sait}@kfupm.edu.sa

R. Al-Shaikh
EXPEC Computer Center, Saudi Aramco
raed.shaikh@aramco.com

Abstract - **In recent years, we have witnessed a growing interest in optimizing the parallel and distributed computing solutions using scaled-out hardware designs and scalable parallel programming paradigms. This interest is driven by the fact that the microchip technology is gradually reaching its physical limitations in terms of heat dissipation and power consumption. Therefore and as an extension to Moore's law, recent trends in high performance and grid computing have shown that future increases in performance can only be reached through increases in systems scale using a larger number of components, supported by scalable parallel programming models. In this paper, we evaluate the performance of two commonly used parallel compilers, Intel and Portland's PGI, using a state-of-the-art Intel Westmere-based HPC cluster. The performance evaluation is based on two sets of experiments, once evaluating the compilers' performance using an MPI-based code, and another using OpenMP. Our results show that, for scientific applications that are matrices-dependant, the MPI and OpenMP features of the Intel compiler supersede PGI when using the defined HPC cluster.**

Index Terms— *HPC, Intel, PGI, compilers, Infiniband.*

I. INTRODUCTION

In recent years, we have witnessed a growing interest in optimizing the parallel and distributed computing solutions using scaled-out hardware designs and scalable parallel programming paradigms. This interest is driven by the fact that single CPU-chips are reaching their physical limits in terms of heat dissipation and power consumption. Therefore and as a continuation to Moore's law, recent trends in high performance and grid computing have shown that future increases in performance can only be achieved through increases in systems scale using a larger number of components, which are supported by scalable parallel programming models. Accordingly, scaled-out computing is clearly becoming the trend.

In terms of the underlying hardware, multi-cores CPUs and ultra-fast interconnects are today's ingredients for the High Performance Computing systems. Intel and AMD are still the leaders in the CPU industry, dominating the top500.org list of the most powerful supercomputers worldwide, and taking over 80% of HPC as of 2010 [9]. Nowadays, most of the high performance clusters use multi-core CPUs in their compute nodes, ranging from 2 to 4 cores per nodes, while 6-cores sockets will become more common on clusters as Intel and AMD released their Westmere and Phenom II multi-core CPUs, respectively [7]. On the HPC interconnects side, there are several network interconnects that provide ultra-low latency (less than 1 microsecond) and high bandwidth (several gigabytes per second). Some of these interconnects may even provide flexibility by permitting user-level access to the network interface cards for performing communication, and also supporting access to remote processes' memory address spaces [1]. Examples of these interconnects are Myrinet from Myricom, Quadrics and Infiniband [1]. The experiments in this paper are done on the Infiniband architecture, which is one of the latest industry standards, offering low latency and high bandwidth as well as many advanced features such as Remote Direct Memory Access (RDMA), atomic operations, multicast and QoS [2]. Currently, available Infiniband products can achieve latency of 200 nanoseconds for small messages and a bandwidth of up to 3-4 GB/s [1]. As a result, it is becoming increasingly popular as a high-speed interconnect technology option for building high performance clusters.

On the parallel programming level, MPI and OpenMP have become the de facto standard to express parallelism in a program. OpenMP provides a fork-and-join execution model, in which a program begins execution as a single process or thread. This thread executes sequentially until a parallelization directive for a parallel region is found. At this time, the thread creates a team of threads and becomes the master thread of the new team. All threads execute the statements until the end of the parallel region. Work-sharing directives are provided to divide the execution of the enclosed code region among the threads. The advantage of OpenMP is that an existing code can be easily parallelized by placing OpenMP directives around time consuming loops which do not contain data dependences, leaving the source code unchanged. The disadvantage is that it is a big challenge to scale OpenMP codes to tens or hundreds of processors. One of the difficulties is a result of limited parallelism that can be exploited on a single level of loop nest.

Another program parallelization can be achieved through the message passing programming paradigm, which can be employed within and across several nodes. The Message Passing Interface (MPI) [4] is a widely accepted standard for writing message passing programs. MPI provides the user with a programming model where processes communicate with other processes by calling library routines to send and receive messages. The advantage of the

MPI programming model is that the user has complete control over data distribution and process synchronization, permitting the optimization of data locality and workflow. The disadvantage is that existing sequential applications require a fair amount of restructuring for parallelization based on MPI.

Our objective in this paper is to evaluate the performance of two commonly used parallel compilers, Intel and Portland's PGI, using a state-of-the-art HPC cluster. As described in the evaluation section, the performance evaluation is based on two sets of experiments, once evaluating the compilers' performance using an MPI-based code (between cluster nodes), and another using OpenMP-based code (using a single cluster node with dual hexa-cores Westmere sockets). To the best of our knowledge, this is the first paper that discusses Intel and PGI compilers' performance based on the latest Intel's Westmere technology and Infiniband QDR interconnect.

The rest of the paper is organized as follows: In section 2, we briefly shed some light on the compilers, the Infiniband interconnect technology, the Intel Westmere CPU architecture, and the MPI implementations used to benchmark our compilers, while in section 3 we describe our experimental evaluation and interprets the benchmark results. We state our conclusion and future work in the last section.

II. BACKGROUND

In this section, we briefly describe the characteristics of both the Intel and PGI compilers. Also, we will shed light on the technologies used to benchmark the two compilers. These are: the Quad Data Rate (QDR) Infiniband interconnect technology, the Intel Westmere architecture, and the MPI implementations.

A. Intel and PGI Compilers

Both Intel C and Fortran compilers support compilation for IA-32, Intel 64, Itanium 2, processors and certain non-Intel but compatible processors, such as certain AMD processors [7]. The Intel compiler further supports both OpenMP 3.0 and automatic parallelization for SMP. With the add-on capability Cluster OpenMP, the compiler can also automatically generate MPI calls for distributed memory multiprocessing from OpenMP directives.

Similar to the Intel compilers, PGI C/C++ includes native parallelizing/optimizing OpenMP C++ and ANSI C compilers. In addition, PGI's server version includes the OpenMP and MPI parallel graphical debugger (PGDBG) and the OpenMP and MPI parallel graphical performance profiler (PGPROF) that can debug and profile up to 16 local MPI processes. PGI Server also includes a precompiled MPICH message passing library.

Both Intel and Portland Group Inc. (PGI) continuously tune their compilers to optimize for hardware platforms to minimize stalls and to produce code that executes in the fewest number of cycles. Both compilers share many technical features and high-level optimizations, such as: interprocedural optimization (IPO), profile-guided optimization (PGO), and high-level optimizations (HLO) [7, 8]. High-level optimizations are optimizations performed on a version of the program that more closely represents the source code, such as loop interchange, loop unrolling, loop distribution and data-prefetch. These optimizations are usually very expensive and may take considerable compilation time.

Interprocedural optimization applies typical compiler optimization that may affect multiple procedures, multiple files, or the entire program. IPO aims to reduce or eliminate duplicate calculations, inefficient use of memory, and to simplify iterations such as loops. In addition, IPO reorders the procedures for better memory utilization and locality. IPO also incorporates typical compiler optimizations on the entire program, for example, removing codes that are never executed in a program.

Profile-guided optimization, on the other hand, refers to a mode of optimization where the compiler performs a sample run of the program across a representative input set. The data would then indicate which sections of the program are executed more frequently, and which areas are accessed less frequently. All optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions.

B. Infiniband Architecture

Infiniband is a technology that provides a high bandwidth I/O communication over a high speed serial data bus. It uses a switched fabric topology, as opposed to a hierarchical switched network like Ethernet [2]. It is designed to directly route data from one point to another point through a switch, where all transmissions begin or end at a channel adapter (HCA). Each Infiniband processor contains a host channel adapter (HCA) and each peripheral has a target channel adapter (TCA).[3] The Infiniband serial connection signaling rate is 2.5 Gbit/s in single data rate (SDR) technology, 5.0 Gbit/s in double data rate (DDR) technology or 10 Gbit/s in quad data rate (QDR), in each direction per connection. Moreover, the links can be aggregated in units of 4 or 12, designated as 4X and 12X. However, Infiniband uses 8B/10B encoding, which implies four fifths of the traffic is useful, therefore DDR 4X link curries 20 Gbit/s raw, or 16 Gbit/s of useful data. Table-1 summarizes the different Infiniband technologies with their associated theoretical performance numbers.

Table 1: Performance numbers of different Infiniband technologies

IB technology	SD IB Data Rate	DD IB Date Rate	QDR IB Data Rate
1x	2Gbps	4Gbps	8Gbps
4x	8Gbps	16Gbps	32Gbps
12x	24Gbps	48Gbps	96Gpbs

Infiniband uses a hardware-offload protocol stack [3]. Extra memory copies that are sent from the application to an adapter can be avoided by the zero copy mechanism that optimizes the message transfer time. Moreover, Infiniband allows moving data from local memory to remote memory using RDMA (Remote Direct Memory Access), which allows the zero copy mechanism without involving the receiver host processor [2]. The number of user-kernel context switching and memory copies can be reduced by the direct access to the Infiniband HCA. Obviously, enabling communication between devices and hosts, without the traditional system resource overhead associated with network protocols, off-loads data movement from the server CPUs to the Infiniband HCA. Through virtual lanes (VLs), Infiniband offers traffic management, creating multiple virtual links within a single physical link that allows a pair of linked devices to isolate communication interference from other connected devices.

C. Intel Westmere Specifications

Westmere is the code name for the latest in the series of multi-core processors by Intel. This is Intel's true hexa-core processor with L2 cache sharing and utilizing the revolutionary Quick Path Interconnect (QPI) architecture [7] that provides two separate lanes for the communication between the CPU and the chipset. The QPI technology allows the CPU to transmit and receive I/O data in parallel, as opposed to the traditional architecture using a single external bus where the external bus is used for both input and output operations reads and writes cannot be done at the same time. The latest version of the QPI works with a clock rate of 3.2 GHz, transferring two data per clock cycle (Double Data Rate), making the bus to work as if it was using a 6.4 GHz clock rate.

Further, Intel Westmere generation is equipped with Turbo Boost Technology [7] that automatically allows processor cores to run faster than the base operating frequency if it's operating below power, current, and temperature specification limits. This frequency change is dependent on the number of active cores, estimated current consumption, estimated power consumption and processor temperature. When the processor is operating below these limits and the user's workload demands additional performance, the processor frequency will dynamically increase by 133 MHz on short and regular intervals until the upper limit is met or the maximum possible upside for the number of active cores is reached.

D. MVAPICH MPI Implementation

The Message Passing Interface (MPI) is the dominant programming model for parallel scientific applications. Given the role of the MPI library as the communication substrate for application communication, the library must ensure to provide scalability both in performance and in resource usage. In our experiments, we used MVAPICH, one of the most commonly used MPI implementations in the HPC industry. MVAPICH [12] implementation is mainly known for its support for Infiniband interconnect technologies as well as having high performance scalability support for clusters running thousands of cores. As for the Intel MPI, MVAPICH also supports various runtime environments such as SLURM and PBS.

III. PERFORMANCE EVALUATION AND RESULTS

To perform benchmark evaluation, a DELL cluster of PowerEdge M610 Blade Servers was used. The cluster consisted of 32 nodes with dual sockets and Intel hexa-Core x5670 (Westmere) 2.93GHz processors. The operating system running on the nodes was RedHat Enterprise Linux Server 5.3 with the 2.6.18-128.el5 kernel. Each node was equipped with an Infiniband Host Channel Adapter (HCA) supporting 4x Quad Data Rate (QDR) connections with the speed of 32Gbps. Each node also had 24 GB (6 x 4GB) DDR3 1333Mhz of memory, thus the total amount of memory the system had was around 786 GB.

The physical layout of the cluster consisted of two chassis, and each chassis hosts up to 16 blade nodes. From each node we had a 4x-QDR Infiniband connection going to a central 32-port Qlogic Infiniband switch. Figure 1 shows the Infiniband interconnection design as described. It is important to mention that this design is considered non-blocking as each node guarantees to have the full 4x QDR 32Gbps interconnect speed. This fast interconnect would drive the cluster to a higher utilization, which in theory, may affect the diskless concept.

Our Infiniband interconnect topology uses three switches: A top-level switch and other two leaf switches. Under this configuration, IPC communication among nodes of 12 sub-clusters is localized to one leaf switch, but for the cluster of 16 nodes, the top-level switch is involved to support more nodes.

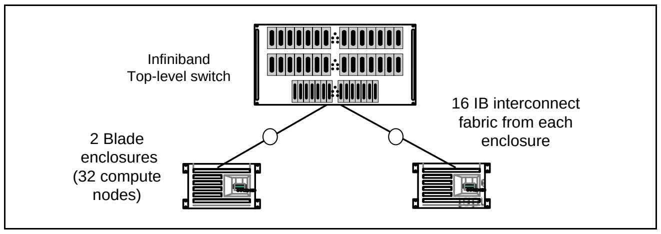

Figure 1: The DDR Infiniband interconnect for a 32 nodes cluster

In order to evaluate the performance of the two compilers, the benchmarks were run on the cluster nodes starting with one thread and scaling up to 12 threads for the OpenMP tests, and ranging from one node and up to 12 nodes for the MPI experiments.

In our experiments, we used two versions of matrix multiplication algorithms [13, 14] to benchmark the two compilers. Beside it is computationally intensive with $O(n^3)$ iterations, we chose the matrix multiplication since it is a fundamental operation in many numerical linear algebra applications. Its efficient implementation on parallel computers is an issue of prime importance when providing such systems with scientific software libraries.

```
1.    #include <omp.h>
2.    #include <stdio.h>
3.    #include <stdlib.h>
4.    #define NRA 4000                 /* # rows in matrix A */
5.    #define NCA 4000                 /* # columns in matrix A */
6.    #define NCB 4000                  /* # columns in matrix B */
7.    int main (int argc, char *argv[])
8.    {
9.    int       tid, nthreads, i, j, k, chunk;
10.   double    a[NRA][NCA],           /* matrix A to be multiplied */
                b[NCA][NCB],           /* matrix B to be multiplied */
                c[NRA][NCB];           /* result matrix C */
11.   chunk = 10;                     /* set loop iteration chunk size */
12.   /*** Spawn a parallel region explicitly scoping all variables ***/
13.   #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
14.   {
15.   tid = omp_get_thread_num();
16.   if (tid == 0)
17.   {
18.   nthreads = omp_get_num_threads();
19.   printf("Starting matrix multiple example with %d threads\n",nthreads);
20.   printf("Initializing matrices...\n");
21.   }
22.   /*** Initialize matrices ***/
23.   #pragma omp for schedule (static, chunk)
24.   for (i=0; i<NRA; i++)
25.   for (j=0; j<NCA; j++)
26.   a[i][j]= i+j;
27.   #pragma omp for schedule (static, chunk)
28.   for (i=0; i<NCA; i++)
29.   for (j=0; j<NCB; j++)
30.   b[i][j]= i*j;
31.   #pragma omp for schedule (static, chunk)
32.   for (i=0; i<NRA; i++)
33.   for (j=0; j<NCB; j++)
34.   c[i][j]= 0;
35.   /*** Do matrix multiply sharing iterations on outer loop ***/
36.   /*** Display who does which iterations ***/
37.   printf("Thread %d starting matrix multiply...\n",tid);
38.   #pragma omp for schedule (static, chunk)
39.   for (i=0; i<NRA; i++)
40.   {
41.   printf("Thread=%d did row=%d\n",tid,i);
42.   for(j=0; j<NCB; j++)
43.   for (k=0; k<NCA; k++)
           c[i][j] += a[i][k] * b[k][j];
44.   }
45.   }   /*** End of parallel region ***/
46.   /*** Print results ***/
47.   printf("Result Matrix:\n");
48.   for (i=0; i<NRA; i++)
49.   {
50.   for (j=0; j<NCB; j++)
51.   printf("%6.2f   ", c[i][j]);
52.   printf("\n");
53.   }
54.   printf ("Done.\n"); }
```

Figure 2: Matrix multiplication in C with OpenMP directives

Figure 2 shows the OpenMP C code for matrix multiplication. The routine `omp_get_num_threads` in line 15 is responsible for returning the number of threads that are currently in the team executing the parallel region from which it is called, while the `omp for (static, chunk) schedule` directive divides the iterations in the loop into pieces of size “chunk” and then statically assigns them to threads.

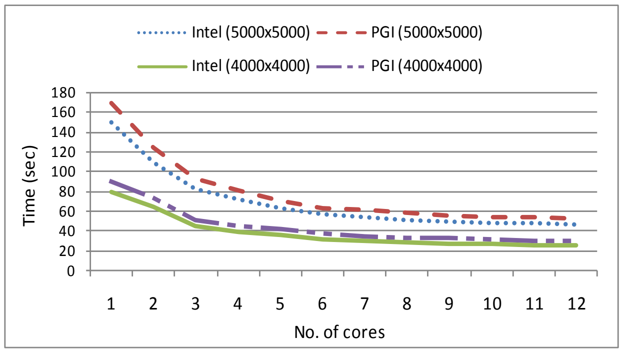

Figure 3: Intel vs. PGI using OpenMP directives in matrix multiplication

The OpenMP code was compiled using `-openmp` and `-mp` options for Intel and PGI compilers, respectively, while all other advance options were ignored to achieve a fair comparison. Figure 3 shows the performance benchmark of both Intel and PGI for multiplying 4000x4000 and 5000x5000 size matrices. Initially, all runs were significantly improved when adding more cores, while their improvement slowed down when reaching 6 cores. It was also observed that when the size of the matrices were increased from 4000 to 5000, the Intel-compiled code run time was increased by 79% in average, while PGI-compiled code was increased by 85%. In this OpenMP set of tests, the Intel compiler superseded PGI in all iterations.

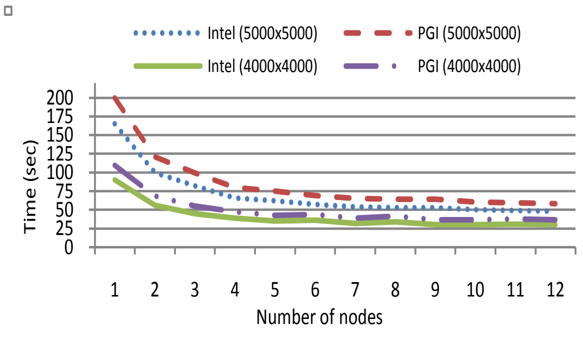

Figure 4: Intel vs. PGI using MPI in matrix multiplication

Figure 4 shows performance of the Intel and PGI compiled code using MPI routines. In this test, the C code (too long to be included in the paper, but can be found in [15]) was compiled using MVAPICH with Intel and PGI parallel `mpicc` compilers. Similarly, all advance options were ignored. It is noticeable that the Intel compiled MPI-run on a single node/core took around 165 seconds, whereas it took only 150 seconds when running OpenMP on a single core. This is due to the fact that the MPI-based matrix multiplication C code has more routines and functions to call, making the code more complex, and thus more time to run. Another observation is the slight increase in the run time when multiplying the 4000x4000 size matrices on 11 and 12 cores. This increase is related to the additional communication overhead with respect to the computation time. This communication is lessened in the 5000x5000 multiplication as the computation time gets larger with respect to the communication overhead. Similar to the OpenMP test, the Intel compiler outperformed PGI in both 4000 and 5000 iterations.

To magnify the effect of MPI communication overhead with respect to computation time, we extended the MPI matrix multiplication benchmark runs to 32 nodes. Figure 5 shows the effect of this communication overhead as the number of nodes increases.

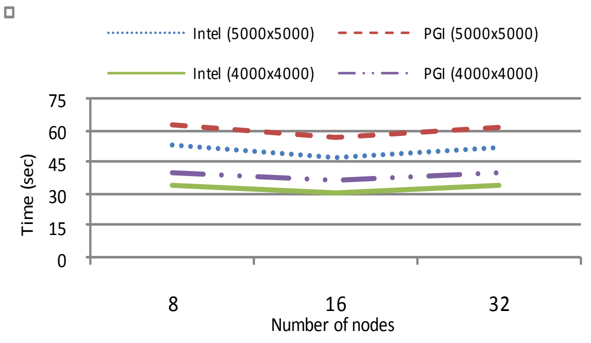

Figure 5: MPI scalability in 5000x5000 and 4000x4000 cells matrix multiplication using up to 32 nodes

IV. CONCLUSION

Intel and Portland Group have been designing their parallel compilers to leverage the rich set of performance enabling features in modern CPUs and parallel systems. This is achieved by tightly integrating OpenMP directives and advanced MPI optimizations to generate efficient multithreaded code for exploiting parallelism at various levels. In this paper, we evaluated the performance of two commonly used parallel compilers, Intel and Portland's PGI, using a state-of-the-art Intel Westmere-based HPC cluster. The performance evaluation was based on two sets of experiments, once evaluating the compilers' performance using an MPI-based code, and another using OpenMP. Our results show that, for scientific applications that are matrices-dependant, the MPI and OpenMP features of the Intel compiler supersede PGI when using the defined HPC cluster.

Our future work includes testing both compilers using hybrid OpenMP and MPI codes and evaluating the scalability and efficiency of each on the high performance computing cluster.

REFERENCES

[1] R. AlShaikh, M. Ghuson, M. Baddourah, “Performance Evaluation of Myrinet and Cisco Infiniband Using Intel MPI Middleware", the 9th LCI International Conference on High Performance Computing, NCSA, Univerity of Illinois, USA, May 2008.

[2] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda, “Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand”, Int’l Parallel and Distributed Processing Symposium (IPDPS 04), April, 2004.

[3] C. Bell, D. Bonachea, Y. Cote and et al. “An Evaluation of Current High-Performance Networks”, Int’l Parallel and Distributed Processing Symposium (IPDPS’03), April 2003.

[4] J. Liu, B. Chandrasekaran, J. Wu and et al. “Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics”, Supercomputing, ACM/IEEE, pages 58- 58, Nov. 2003.

[5] Myrinet, Myricom. Available at: http://www.myri.com

[6] R. Fatoohi, K. Kardys, S. Koshy and el at. “Performance evaluation of high-speed interconnects using dense communication patterns”, Parallel Computing Volume 32, Issue 11-12, pages 794-807, 2006.

[7] Intel Inc. Available at: http://www.intel.com

[8] Portland PGI. Available at: http://www.pgroup.com/

[9] The top500 supercomputers. Available at: http://www.top500.org

[10] MVAPICH: MPI over InfiniBand and iWARP. Available at: http://mvapich.cse.ohio-state.edu

[11] T. Typou, V. Stefanidis, P.D. Michailidis and K.G, “ Margaritis, Implementing Matrix Multiplication on an MPI Cluster of Workstations”, in Proceedings of the 1st In’t Conference "From Scientific Computing to Computational Engineering" (IC-SCCE'2004), Athens, Greece, vol. II, pp. 631-639, 2004

[12] B. Madani, R. Al-Shaikh, “Performance Modeling and MPI Evaluation Using Westmere-based Infiniband HPC Cluster”, 4th European Modelling Symposium on Mathematical Modelling and Computer Simulation, Pisa, Italy, 2010

[13] Lawrence Livermore National Laboratory – OpenMP tutorial. Available at: https://computing.llnl.gov/tutorials/openMP/

[14] Simple matrix multiplication on MPI. Available at: http://sushpa.wordpress.com/2008/05/20/simple-matrix-multiplication-on-mpi/