Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 774 – 783

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Simulation and performance study of large scale computer cluster configuration: combined multi-level approach

Grigory Rechistov[a,b], Alexander Ivanov[a,b], Pavel Shishpor[a,b], Vladimir Pentkovski[a,b]

[a]*Department of Radio Engineering and Cybernetics, Moscow Institute of Physics and Technology*
[b]*Intel Corporation*

Abstract

In this paper we describe an approach used to study the functional aspects and estimate the performance of large (read "ten times more cores than currently available") cluster computing system configurations running MPI applications. A combination of a functional simulation, performance models, traces analysis and real-world measurements is used to achieve these goals. We also present the first results of applying this methodology to actual applications of interest.

Keywords: simulation, Simics, cluster, performance, tracing

1. Introduction

A possibility to estimate the behavior and especially the performance for future, not yet constructed computer systems was always highly desirable by system architects and software developers as this leads to ability to look for their weak and strong points in advance and to have a chance to tune the software/hardware to fully employ their potential.

In this paper an approach to such modeling of cluster systems, that is a group of computers built from commodity hardware with no custom design hardware, is described.

There are many aspects of such huge systems that are contributing to the complexity of the performance estimation.

1. Hardware specifications are incomplete. This mostly applies to microarchitecture specifications of central processors which often comprise a trade secret, and only a limited amount of data is publicly available. Therefore we need to find a way to account the influence of a CPU not by direct simulation (which would turn out to be unacceptably slow anyway) but through other means.
2. A parallel execution is the general method of increasing the performance and it can actually be found on multiple levels of a complex system.
    - Vector or "single instruction multiple data" (SIMD) instructions. For the Intel IA-32 architecture they are presented with SSE, SSE2, SSE3, SSE4 and AVX ISA extensions [1] and can perform several floating

Email addresses: `grigory.rechistov@phystech.edu` (Grigory Rechistov), `alexander.a.ivanov@phystech.edu` (Alexander Ivanov), `pavel.shishpor@phystech.edu` (Pavel Shishpor), `vladimir.m.pentkovski@intel.com` (Vladimir Pentkovski)

1877-0509 © 2012 Published by Elsevier Ltd.
doi:10.1016/j.procs.2012.04.083

point operations in a single machine instruction. An accurate functional simulator that implements such instructions will be sufficient to account for their impact on the performance.
- In-core multiple execution devices and the out-of-order execution logic. Included in many contemporary CPUs, such techniques allow to effectively perform more than one instruction per cycle provided they do not depend on each other results.
- The simultaneous multithreading (SMT) allows to execute another thread when the first one is waiting for data from memory or is stalled for any other reason. While not offering a truly parallel execution, this technology is present in Intel processors (called Hyper-Threading) and, as shown below, introduces measurable changes to applications performance.
- Multicore CPUs have two or more cores capable to work on independent instruction streams.
- Multiple machines connected via Ethernet, Infiniband etc. networks. Applications that use message passing facilities like MPI are able to execute on all of them.

3. Multiple levels of communications with different latencies. They usually negatively affect the performance and therefore should be carefully taken into account.
   - Multiple caches – the communications between different layers to get a datum missing at the current level and message traffic between the cores to maintain a coherent memory view introduce disruptions to the computation.
   - Memory accesses in the modern systems are non-uniform, i.e. their latency depends on whether the request ends up in a local or a remote region.
   - Inter-host network communications. Its effectiveness depends on the basic link properties (speed, latency), the network topology, the messaging protocol implementation and the application behavior.
   - Non-volatile disk memory accesses should also be accounted for delays, but for this study we consider such influence as minor because of the nature of the applications studied – their data completely fits into RAM and no swapping is required.

The additional but major challenge is the speed of the simulation itself. The general rule is that the more precise model is the slower it is. We could not rely only on a simulation but had to employ other techniques to achieve a reasonable speed/accuracy compromise.

This paper describes the methodology created to catch all the described performance influencing aspects. Also the first results of measurements performed for real applications are given; the correspondence of simulation results to real world expectations is then discussed.

2. Overview of objects of the study

When talking about the performance one has to consider both the hardware used and the exact applications studied.

2.1. Hardware

The existing trend for demand for computational power is that a system once built will become obsolete very soon lest it should be constantly refreshed. Therefore we wanted to predict an evolution of the computer cluster system set up in Moscow Institute of Physics and Technology (“cluster” for short) as more hosts will be added to it over time. In the table 1 the characteristics of three generations (denoted as Gen1 – Gen3 or commonly “GenX”) of the cluster are given.

The Gen1 configuration has already been deployed. The configuration of Gen2 is finalized and it is currently being built. The specifications of Gen3 are not yet stabilized and therefore their estimations are shown. The strategy here is to increase the number of computing cores ten times when moving from GenX to Gen(X+1).

Of course, merely adding more computing cores does not mean a proportional increase of the performance observed on real applications, most obviously because of communication delays. That’s why we wanted to understand such impact well ahead.

The benefit of using a simulation is the possibility to “observe” systems not yet built: Gen2, Gen3 and the later ones.

Parameter	**Gen1**	**Gen2**	**Gen3**
The system status	Deployed	Under construction	Planned
Processor name	Intel Xeon X5680	Sandy Bridge based	Unknown
Processor frequency, GHz	2.8	2.66	Unknown yet
Number of computing hosts	16	112	1000 (Expected)
Number of processors in the host	2	2	Unknown yet
Number of cores in the CPU	6	8	8 (Expected)
Total number of cores in the system	192	1792	10 000 (Expected)
Interconnect network	Infiniband QDR	Infiniband QDR	Unknown
Theoretical peak performance, TFLOPS	2.1	19.0	200 (Expected)

Table 1: The generations of cluster systems. Most of the parameters for Gen3 system are not decided on yet.

2.2. Applications studied

- High Performance Linpack benchmark [2] (HPL for short) is a classic application for solving a system of linear equations and is widely used to measure supercomputer performance. It was our prime object of study for this paper.
- Netperfmeter [3] is a benchmark for measuring the speed of TCP/UDP communication. It was used to find the accuracy of performance measurements of network models.
- Gromacs [4] is a molecular dynamics simulation package. This is the one of “production” applications running on the GenX systems.

The following tools were used for building and running the applications.

- Compiler – Intel Composer XE 2011 SP1 6.233 (`icc` version 12.1.0)
- MPI library – MPICH 1.4.1p1.
- Applications: HPL 2.0, Netperfmeter 1.1.7, Gromacs 4.5.4 (double precision, MPI support).
- Operating system of real and simulated clusters: Debian GNU/Linux 6.0.3 x86_64.

2.3. Metrics studied

The first thing to perform within a full system simulation is to ensure a correct operation of the software on the simulated hardware configuration; that is, will it work and yield sane results. For this a functional model is enough as it ensures the correctness of the software algorithms. The next step is to find the performance.

We use FLOPS (floating point operations per second) as a measure of how fast an application is performing on the whole cluster. To characterize performance per a single core another value will be used, namely CPI – average processor clocks ticks required to retire one machine instruction. Having the core CPI one can easily get the cluster FLOPS with the formula

$$FLOPS = \frac{\alpha \times W \times F_{core}}{CPI} N_{cores} \tag{1}$$

Here α shows how many instructions are floating point operations ($0 \leq \alpha < 1$), W is the FPU unit vector width (e.g. for 128 bit SSE registers and double precision calculations $W = 2$, with 256 bit AVX registers $W = 4$), F_{core} is a frequency of computing cores[1], N_{cores} is the total number of cores in the cluster.

[1]We assume it to be constant throughout runs, while for modern CPUs this might no longer be true because of dynamic power saving schemes used; we assume that the average load is high enough to keep all cores at their highest speed.

3. Tools

Two main software tools were exploited throughout the work.

Wind River Simics [5] is a full system simulator and a framework for developing device models. Besides its high speed simulation, large range of existing device models, detailed documentation allowing to create new ones and extensive scripting facilities, it has a feature particularly suited to our needs, that is being able to run multithreaded on one host and distributed across the network. This allowed us to model Gen2 and Gen3 on existing Gen1 system and to fully employ its computing resources to speed up the simulation. We use version 4.6.

Intel VTune [6] is an utility mainly used to discover performance hotspots and bottlenecks. It uses performance counters embedded in a modern Intel CPU to measure numerous characteristics of its execution. In our study we use it to find miscellaneous performance and statistics data for single cores (in particular values of α, CPI). Intel VTune Amplifier XE 2011 (build 186533) is used in this work.

4. Estimating the performance of a parallel application

The used analysis of a single core is an elaboration of those proposed in [7]. A total CPI value is broken down to a sum of terms originating from different aspects of the architecture and the application behavior:

$$CPI = CPI_{core} + CPI_{caches} + CPI_{synch} + CPI_{network} \tag{2}$$

The first term, CPI_{core}, is for operation of a computing core subsystems and is described in section 4.1. CPI_{caches} represents a cumulative effect of the cache/local memory hierarchy and is an average delay of a memory reference per an instruction executed, see section 4.2 for details.

Two remaining terms are not directly related to the microarchitecture but rather to the algorithms application uses and network speed/topology etc. CPI_{synch} is an average length of POSIX thread operations: locking, waiting for a lock, barrier etc. For the applications studied (section 2.2) there were not thread communications implemented and therefore we assume this term to be equal to zero.

$CPI_{network}$ is delay attributed to LAN communications; for our study this is essentially MPI calls of different durations averaged as described in section 4.3.

4.1. Microarchitecture

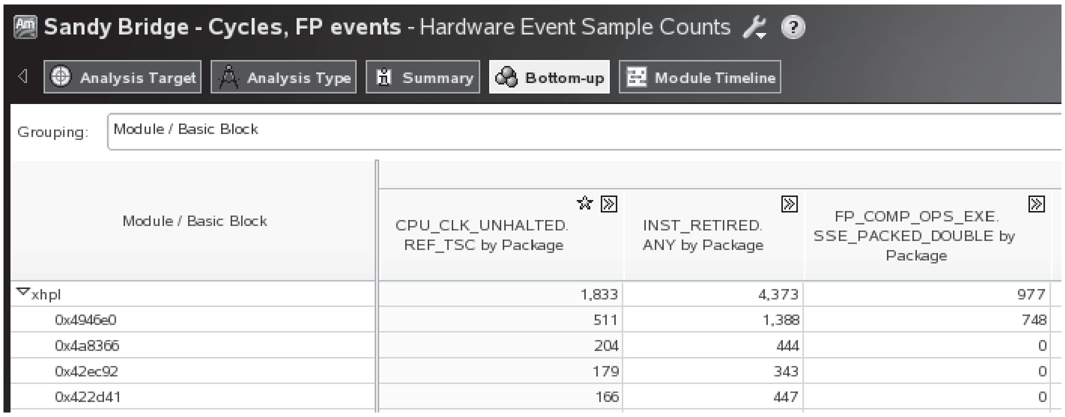
CPI_{core} from (2) can be further broken down as $CPI_{core} = CPI_{frontend} + CPI_{ideal} + CPI_{backend}$, being a sum of cycles per instruction spent in the frontend, the executing devices and the backend correspondingly. They can be found during a precise simulation. As it has already been noted simulating the internals of a single core is a complicated task that needs details of a microarchitecture; even if such a cycle-precise simulator is implemented it would be very slow because of taking so many details into account. With the amount of cores increasing the problem would become even worse.

Instead, we use Intel VTune to find out numbers of cycles and instructions for regions of an application code and to deduce CPI from them. There is another benefit of using VTune – it allows to find boundaries of hot spots[2] for applications (Fig. 1). It was found that the "hottest" block corresponds to an application's computing kernel, tends to remain the same across all the configurations observed and shows almost constant value for CPI_{core}. Finally, special counters exist for vector instructions (e.g. `FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE` column on Fig. 1) that allow to find α value for (1).

The main disadvantage of the described approach is that is does not take into account the changes in the microarchitecture between the CPUs generations. For example, the Gen1 configuration uses CPUs code named Westmere, Gen2 will have Sandy Bridge and Gen3 might have Ivy Bridge or Haswell cores. Until we acquire samples of the new hardware we have to cope with inaccuracies of CPI_{core} introduced by this or try to correct the values with data from sources other that the VTune experiments.

After the values for CPI_{core} are obtained from the experiments they can be used in the simulation model as Simics allows to specify a constant CPI value for a processor model so that every instruction execution (called "step") will take given amount of cycles; this value can be more or less than 1.0 (the default value).

[2]A hot spot is a continuous block of machine code where the execution spends most of its time.


Sandy Bridge - Cycles, FP events - Hardware Event Sample Counts
Analysis Target
Analysis Type
Summary
Bottom-up
Module Timeline

Grouping: Module / Basic Block

Module / Basic Block	CPU_CLK_UNHALTED.REF_TSC by Package	INST_RETIRED.ANY by Package	FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE by Package
▽xhpl	1,833	4,373	977
0x4946e0	511	1,388	748
0x4a8366	204	444	0
0x42ec92	179	343	0
0x422d41	166	447	0

Figure 1: The screenshot of Intel VTune hot spots view for the HPL. Linear blocks are sorted so that those consumed most of the processor's time are at the top. In this example the top basic block with start address 0x4946e0 is the main computational kernel.

4.2. Cache hierarchy

Cache/memory accesses CPI term in (2) is split as follows:

$$CPI_{caches} = P_{L1}L_{L1} + P_{L2}L_{L2} + P_{L3}L_{L3} + P_{local\ memory}L_{local\ memory} + P_{remote\ memory}L_{remote\ memory} \quad (3)$$

In (3) P_{Lx} is the probability of a cache level $x \in (1, 2, 3)$ access (following the previous level(s) miss event), L_{Lx} is the latency of such access; $P_{local\ memory}$ and $L_{local\ memory}$ are the probability and the latency of local memory access, $P_{remote\ memory}$ and $L_{remote\ memory}$ are for remote DRAM requests (this is for NUMA systems which are the becoming common).

The Simics simulation is used to get these probabilities. We extended the stock Simics **g-cache** model to represent the hierarchies of Gen1 and Gen2. The values for individual caches capacities and latencies are taken from the corresponding CPU specifications.

4.3. Network communication

There are two approaches to studying how network influences the performance using Simics:

1. To use Simics models of Ethernet devices that provide functional simulation of network and some basic performance characteristics, such as bandwidth (in our experiments we set it to be unlimited) and latency (see results section for more information on it). It was a straightforward approach but it turned out to give doubtful results (see section 5.3 for details) because of the high latency of Simics links, and setting it to low values would slow down the simulation significantly because of Simics synchronization mechanisms responsible for the determinism of models.
2. Collection and separate analysis of traces for MPI events.

The second approach required some additional development to create a distributed trace collector.

1. Running Simics with specially written tracer scripts that record all MPI functions invocations in all simulated threads on all simulated target machines.
   We used the MPI standard [8] profiling interface that states that every `MPI_*` function in the library to be a weak symbol aliased by `PMPI_*` which contains the actual code. It's possible to override such weak symbols at an apllication linking stage with a special library to instrument every call with special "magic" instruction 2. For the IA-32 architecture in Simics it is a `CPUID` variant with an unusual leaf value [1]. The library was generated from a corresponding MPI header; it can be recreated in order to study different MPI implementations such as Intel MPI or OpenMPI.
   When a "magic" instruction is encountered Simics stops the simulation and invokes a user-defined handler executed outside of the simulated environment so the application state is not modified. This handler inspects the target's execution context and collects the following data: function name, its parameters (ranks of sender/receiver,

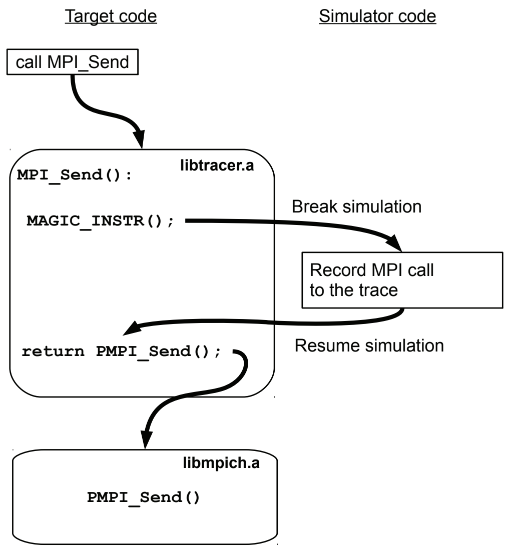


Figure 2: Sequence of instrumentation for an MPI call, example for `MPI_Send()`.

communicators, messages sizes etc), its return value and the timestamp. Then the simulation is resumed and the actual MPI function code is called. Such technique allows to capture the target program communication with minimal interference to its timings: one magic sequence consists of less than dozen of instructions.

2. Determining latencies of all the calls found in a trace for a particular application. We number all such calls from 1 to N and designate their latencies as L_{MPI_i} below.
   For this a set of real-world experiments should be carried out. While point to point MPI interactions are relatively simple, special attention should be paid to collective communications as their timespan depends on number of MPI processes interacting.
3. Replaying collected traces with correct latencies to discover application statistical characteristics.

After the data is collected we can find the last term in (2):

$$CPI_{network} = \sum_{i=1}^{N} P_{MPI_i} L_{MPI_i} \tag{4}$$

In (4) the sum is done for all MPI calls observed, P_{MPI_i} is the probability that an instruction is a call to the i-th MPI routine.

4.4. Accounting for simultaneous multithreading

The Intel Hyper-Threading technology [9] is used to partially hide the latency of outstanding cache misses by switching resources to another logical thread of the execution thus increasing the utilization ratio of the executing devices of a single core. To achieve this some of in-core and external resources are doubled (among them are architectural registers, local APIC devices, instruction TLBs), some are statically partitioned into the two smaller parts (e.g. instruction queues, reorder buffer) and the rest are time and space shared (decoder, executing devices, branch predictor, caches etc). Therefore hyper-threads are not real cores because not all of the resources are doubled. To account for this the following modifications to (1) are made: N_{cores} is doubled and CPI formula is adjusted:

$$CPI = (1 + q)CPI_{core} + CPI_{caches} + CPI_{synch} + CPI_{network} \tag{5}$$

Here in (5) q is used to reflect that the hyper-threads are "slower" than the regular cores because they possess less resources. To find q we use technique proposed in [9]. According to the `PAUSE` instruction documentation [1], its execution leads to immediate yielding of the control and switching to another thread. `PAUSE` does not occupy executing

```
#define PAUSE __asm__ ("pause\n")
int main () {
    while ( true ) {
        PAUSE; PAUSE;
        PAUSE; PAUSE;
        PAUSE; PAUSE;
        PAUSE; PAUSE;
        PAUSE; PAUSE;
        PAUSE; PAUSE;
        PAUSE; PAUSE;
        PAUSE; PAUSE;
    };
    return 0;
};
```

Figure 3: The source code of the `pauseloop` program. It consists of an endless loop of `PAUSE` instructions. The 16-times unrolling is used to minimize the effects of the ending jump instruction.

devices and does not experience cache misses but it does consume statically partitioned resources effectively taking them from the second thread. The experiment of finding q value is as follows: the application being studied runs in 12 threads (one per physical core on a single Gen1 host). Simultaneously with it 12 instances of the `pauseloop` application (Fig. 4.4) are started. Performance counters then are used to calculate CPI of the computational kernel, and q is derived from the comparison of two systems with and without Hyper-Threading.

5. Results

In this section athe vailable experimental results are presented. They are mostly collected for HPL application; network tests used Netperfmeter.

5.1. Functional simulation

We've installed the same operating system (64 bit version of Debian 6) and configured the software model to match the real hardware. There were no problems in running the applications in the Gen1 model. The most notable issue for the Gen2 model was discovered when a huge simulation of HPL involving all the simulated cores on all the hosts using command line `mpiexec -f hosts -n 1792 ./xhpl`. The guest OS reported that the opened file limit per user process was reached – the network sockets are created for MPI communication and are counted as files. After adjusting the limit no more problems were observed. From this point we were able to start the performance measurements.

To estimate the accuracy of the performance predictions obtained from the simulation we compared reported GFLOPS of HPL runs collected on Gen1 with different matrix sizes N. As it can be seen from the comparison of Fig. 4 and Fig. 5 the peak performance of the simulated system is about 3.5 times lower than of the real one. This is explained by the fact that the simulation experiment used an in-order core with CPI fixed to 1.0. As it was expected we cannot use these data for direct performance evaluation.

5.2. Intel VTune measurements

On Fig. 6 the dependency of two values: CPFLO (short for "cycles per floating point operation") and the matrix size N is given. The upper curve represents the reported value, while the lower almost-horizontal line is for the hot spot of the application. It can be seen that it does not depend from N.

More experiments with miscellaneous HPL parameters varied confirmed that CPI_{core} does not depend of total amount of cores and the input data configuration.

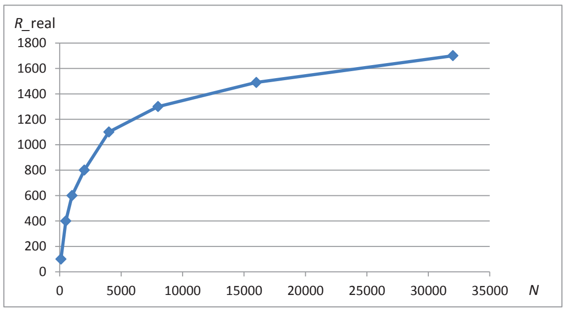


Figure 4: HPL performance on real Gen1.

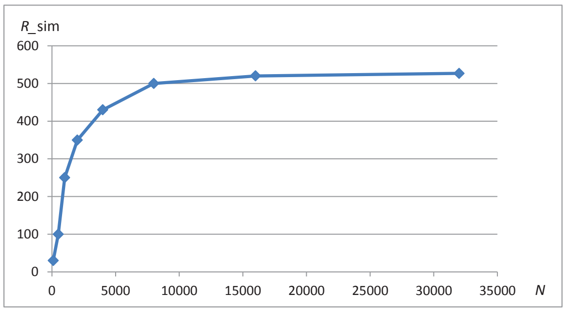


Figure 5: HPL performance on simulated Gen1.

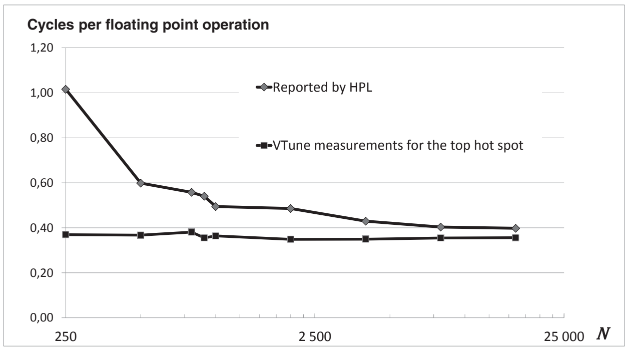


Figure 6: The comparison of performance for the HPL computing kernel measured by Intel VTune and total self-reported HPL performance for different matrix sizes N on a single machine from Gen1.

5.3. Network performance

To find network speed limits on the Simics network model a set of experiments was carried out with Netperfmeter between two simulated nodes. The benchmark creates a TCP connection, sends packets of different sizes (exponentially distributed with average 2000 bytes) in both directions for the 60 simulated seconds.

System	**Link latency, μsec**	**Incoming rate, MBytes/s**	**Outgoing rate, MBytes/s**	**Loss rate, kBytes/s**
Real Gen1, 1Gb Ethernet	$\approx$ 25	110	137	267
Simulated Gen1	400	8	18	19
Simulated Gen1	240	14	30	40
Simulated Gen1	160	23	42	60
Simulated Gen1	80	41	90	110

Table 2: The TCP connection performance of the real and simulated environments.

The results are shown in Tab. 2. We did not test the simulated latencies lower than 80 μsec as they slow the simulation too much to be usable. The latency value of the real 1Gb Ethernet is our estimation and should not be considered accurate as it can vary significantly on different hardware and with OS settings.

These results show that performance from the network simulation are questionable for high latencies. Therefore we chose to develop the MPI tracer described in the section 4.3 to overcome this limitation.

Another set of experiments was targeted to discover the dependency between reported HPL performance and the network latency. The results are shown on Fig. 7. We tested different algorithm data distribution schemes controlled by PxQ values in `HPL.dat` configuration file as this is known to affect the speed significantly. It turned out that it does not depend of link latency for wide range of its values. As expected, the bandwidth required depends significantly on the algorithm parameters. The network bandwidth usage of HPL is shown on Fig. 8.

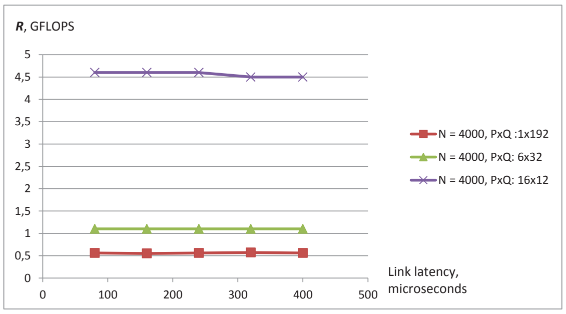


Figure 7: The dependency of the simulated HPL performance of network latency for different algorithm parameters PxQ.

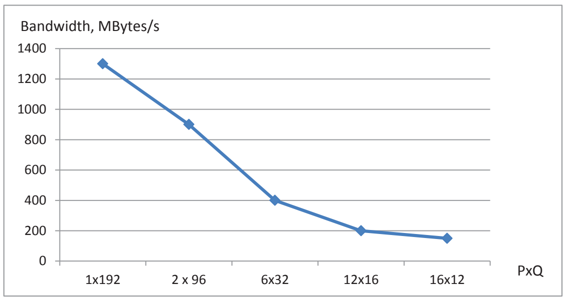


Figure 8: The dependency of the simulated network bandwidth consumed for different HPL values of PxQ. The simulated latency is 400 μsec, $N = 4000$.

We also performed an initial MPI trace collections for HPL applications with following observations for the MPICH library.

- The head node (where `mpiexec` program is started) does not call any of MPI functions at all – all computation and communinication occur at the slave nodes.
- HPL uses lots of `MPI_Irecv()` (asynchronous receive) calls. We even had to suppress its logging to keep the log readable.
- Most of communications are point-to-point, and only a few of collective communication routines were observed.

5.4. Hyper-Threading

The experiments for Hyper-Threading evaluation were carried out for the Gromacs workloads. The results for this runs for two organic molecule models are shown on Fig. 9. It turned out that the computational kernel performance has little dependency from the particular molecule being modeled. The value we obtained for the Gromacs computational kernel is $q = 0.2 \pm 0.02$.

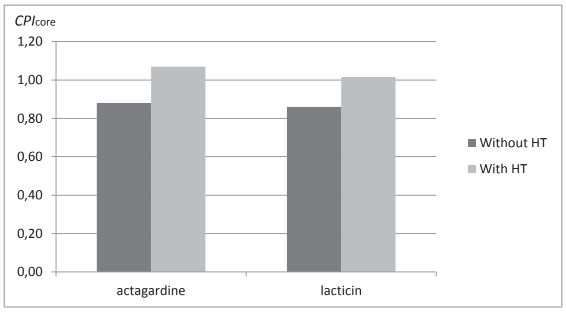


Figure 9: The comparison of CPI for hosts with/without Hyper-Threading. Gromacs models for molecules of lacticin and actagardine are used.

6. Related work

Several simulators exist for a research in the field in many-core supercomputer systems. BigSim [10] was used for IBM BlueGene development. It uses a specialized programming environment Charm++ and PowerPC CPUs while we focused on the conventional MPI library and Intel IA-32 architecture. The MIT Graphite [11] is an actively developed parallel functional simulator capable to model a thousand of IA-32 cores. It is an application level simulator and is designed for systems with shared memory, while our study implies systems with distributed memory. There also exist specialized simulators for MPI applications such as MPI SIM [12]; it is the interconnection performance that is addressed in it, not CPU cores; while Simics allows to account for both aspects. There are not many publicly available cycle-precise perfomance simulators of the IA-32 architecture but they still exist e.g. PTLSim [13].

Analytical descriptions of performance of modern super-scalar processors [7] and multiprocessor systems [14] performance were used as the base for this research when we faced the limitations of a pure simulation.

7. Future work

The research outlined in this paper is only at its beginning. Much of experimental data should still be obtained to get final values for the applications performance.

As it was confirmed during this work, a functional simulation alone (as it is in Simics) can not give satisfactory performance estimations. What it can give are accurate event traces that can later be analyzed off line, and this is the approach we use for cache and MPI calls studies.

Detailed performance data for Gromacs [4] is to be collected. We also target to study Amber [15] – another popular molecular dynamic simulation application to be used the cluster. This would allow us to verify the applicability of the approach to a wide range of applications of practical value, not just synthetic tests.

Gen3 study on Gen1 would assume that we simulate a hundred of cores on a single real one which might turn out to be challenging. The situation will be more relaxed when Gen2 is ready – it would yield ten-to-one simulation ratio instead, just like it's today for the Gen2-on-Gen1 study. Obtaining the performance prediction results for Gen3 would allow us to give certain recommendations to our hardware vendor in order to redistribute the funds properly to build the optimal computing system.

8. Acknowledgments

This work was supported under the grant of the Russian Government order number 220 of 09 April 2010.

References

[1] Intel Corporation, Intel 64 and IA-32 Architectures Software Developers Manual. Volume 3B.

[2] High performance Linpack benchmark.
URL `http://www.netlib.org/linpack`

[3] T. Dreibholz, Netperfmeter: A TCP/UDP/SCTP/DCCP Network Performance Meter Tool.
URL `http://www.iem.uni-due.de/~dreibh/netperfmeter`

[4] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, H. J. C. Berendsen, GROMACS: Fast, flexible, and free, Journal of Computational Chemistry 26 (16) (2005) 17011718.

[5] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: A Full System Simulation Platform, Computer 35 (2002) 5058. doi:10.1109/2.982916.

[6] D. Levinthal, Performance Analysis Guide for Intel® Core^{TM}i7 Processor and Intel® Xeon^{TM}5500 Processors (2009).
URL `http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`

[7] L. J. Simonson, L. He, Micro-architecture Performance Estimation by Formula, in: SAMOS'05, 2005, p. 192201.

[8] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. Version 2.2 (September 2009).
URL `http://www.mpi-forum.org/docs/docs.html`

[9] N. Tuck, D. M. Tullsen, Initial Observations of the Simultaneous Multithreading Pentium 4 Processor, in: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, PACT '03, IEEE Computer Society, Washington, DC, USA, 2003, p. 26.

[10] G. Zheng, G. Kakulapati, L. V. Kale, BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines, Parallel and Distributed Processing Symposium, International 1 (2004) 78b.
URL `http://charm.cs.uiuc.edu/research/bigsim`

[11] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, A. Agarwal, Graphite: A Distributed Parallel Simulator for Multicores, The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA).

[12] R. Riesen, A Hybrid MPI Simulator, in: CLUSTER, 2006, pp. 1–9.

[13] M. T. Yourst, PTLsim User's Guide and Reference (2007).
URL `http://www.ptlsim.org/Documentation/PTLsimManual.pdf`

[14] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, D. A. Wood, Analytic evaluation of shared-memory systems with ilp processors, in: 25th annual International Symposium on Computer Architecture, IEEE Computer Society, 1998, p. 380391.

[15] D. A. Case, T. A. Darden, I. T E Cheatham, C. L. Simmerling, J. Wang, R. E. Duke, R. Luo, R. C. Walker, W. Zhang, K. M. Merz, et al., Amber 11 Users' Manual, University of California, 2010.