HPC on Competitive Cloud Resources

Paolo Bientinesi and Roman Iakymchuk and Jeff Napper

Abstract Computing as a utility has reached the mainstream. Scientists can now easily rent time on large commercial clusters that can be expanded and reduced on-demand in real-time. However, current commercial cloud computing performance falls short of systems specifically designed for scientific applications. Scientific computing needs are quite different from those of the web applications that have been the focus of cloud computing vendors.

In this chapter we demonstrate through empirical evaluation the computational efficiency of high-performance numerical applications in a commercial cloud environment when resources are shared under high contention. Using the Linpack benchmark as a case study, we show that cache utilization becomes highly unpredictable and similarly affects computation time. For some problems, not only is it more efficient to underutilize resources, but the solution can be reached sooner in realtime (wall-time). We also show that the smallest, cheapest (64-bit) instance on the studied environment is the best for price to performance ration.

In light of the high-contention we witness, we believe that alternative definitions of efficiency for commercial cloud environments should be introduced where strong performance guarantees do not exist. Concepts like *average, expected performance* and execution time, expected cost to completion, and variance measures—traditionally ignored in the high-performance computing context—now should complement or even substitute the standard definitions of efficiency.

Paolo Bientinesi
AICES, RWTH, Aachen, Germany, e-mail: pauldj@aices.rwth-aachen.de

Roman Iakymchuk
AICES, RWTH, Aachen, Germany, e-mail: iakymchuk@aices.rwth-aachen.de

Jeff Napper
Vrije Universiteit, Amsterdam, Netherlands, e-mail: jnapper@cs.vu.nl

1 Introduction

The *cloud computing* model emphasizes the ability to scale compute resources on demand. The advantages for users are numerous. Unlike conventional cluster systems, there is no significant upfront monetary or time investment in infrastructure or people and ongoing expenses are simplified. When resources are not in use, total cost can be close to zero. Instead of allocating resources according to average or peak load, the cloud user can pay costs directly proportional to current need. Individuals can quickly create and scale-up a custom compute cluster, paying only for sporadic usage. However, there are also some disadvantages. Costs can be divided into different categories that are billed separately: for example, network, storage, and CPU usage. This model can be complex when attempting to minimize costs [16]. Further, the setup time for computation resources is currently quite long (on the order of minutes for Amazon), and the granularity for billing CPU resources is coarse: by the hour. These two factors imply that resources should be very conservatively scaled in current clouds, reducing some of the benefits of scaling on demand. Finally, in many cloud environments, physical resources are shared among virtual nodes belonging to different users, which can negatively impact performance.

The ability to allocate on demand nodes in current commercial cloud environments implies a new problem in scientific computing: contention. Good high-performance computing clusters balance the CPU, memory, and network usage of applications to maintain efficiency while scaling up resource usage but assume exclusive access to these resources by an application. For example, the RoadRunner supercomputer at Los Alamos National Laboratory can solve a linear system using 100K cores while maintaining 90% efficiency of CPU usage. This feat requires carefully balancing the application's needs across CPU, memory, and the network. With exclusive access, developers of applications and compute libraries can achieve such balanced computing by reducing the noise in resource usage that prevents efficient synchronization. For example, one node out of a large system might run extra software for monitoring that causes it to reach the end of a computation slower than the other nodes. If all nodes must synchronize at the end of a computation, all other nodes in the system must wait for the slowest node to finish. Small amounts of noise can significantly affect the overall performance of a tightly coupled application [14].

Although cloud environments typically provide a small degree of resource availability guarantees, they do not provide a complete facsimile of an unshared virtual node or strong performance guarantees. Contention is visible along almost all resources: memory bandwidth, last-level cache space, network bandwidth and latency [20], and even CPU time. In addition to introducing the kind of noise discussed above, such contention can severely limit the use of resources shared between virtual environments. For example, memory contention can increase the compute time of a simple matrix multiplication by an order of magnitude.

To run scientific applications efficiently, cloud computing needs to provide resources to scientific applications comparable to current well-balanced, exclusive-access high-performance computing (HPC) systems. Isolation between different virtual machines in the cloud should provide predictable performance for devel-

opers of compute-intensive applications and libraries. However, our evaluation of the Amazon EC2 cloud—currently the largest commercial cloud environment—demonstrates that significant work is still needed to isolate virtual machines in the cloud or to adjust HPC libraries to adapt to the dynamic, contentious environment of the cloud. The Amazon EC2 cloud system was built for web service workloads (that is, without extremely fast interconnects) and cannot compare favorably with purpose-built, heavily-engineered HPC supercomputers for many tightly-coupled scientific computations such as dense linear algebra. Although we do not expect comparable performance, the dynamic scalability of the environments holds some promise for smaller workloads.

In order to determine the achievable efficiency of current cloud systems for HPC, we consider the execution of dense linear algebra, which provides a favorable ratio of computation to communication: $O(n^3)$ operations on $O(n^2)$ data. Dense linear algebra algorithms can thus overlap the slow communication of data with quick computations over much more data [2]. Cloud systems will generally be limited by slow communication more than specialized HPC systems given their relatively cheap, slow interconnects. We are concerned primarily with the efficiency of shared cloud systems as they scale up and show that the effects of contention can far outweigh (at least in current offerings) the imbalance caused by an underprovisioned interconnect. We thus focus on CPU costs, ignoring possible associated (and significantly less per computation) storage and networking costs.

This chapter empirically explores computational efficiency in a cloud environment when resources are shared under high contention. We consider single node performance of dense linear algebra operations while nodes are shared with unknown other applications on Amazon EC2 [15]. Further, we analyze how the performance changes on clusters of up to 64 compute cores. Our results show that the performance of single nodes available on EC2 can be as good as nodes found in current HPC systems [18], but the average performance is much worse and shows high variability. In fact, dense linear algebra algorithms do not appear to scale well on cloud systems with high levels of contention. Contention skews the balance of node's CPU and memory resource availability to network capacity to prevent efficient usage of the resources. Cache utilization becomes highly unpredictable and similarly affects computation time. We show that for some problems, not only is it more efficient to underutilize CPU resources, but the solution can even be reached sooner in realtime (wall-time).

In light of the high-contention we witness, we believe that alternative definitions of efficiency for cloud environments should be introduced where strong performance guarantees do not exist. Concepts like *average expected performance* and execution time, expected cost to completion, and variance measures—traditionally ignored in the HPC context—now should complement or even substitute the standard definitions of efficiency. In addition to standard metrics such as GFLOP/sec and efficiency used in HPC, we introduce $/GFLOP (dollars per billions of floating point operations) to analyze in depth the pros and cons of clouds. The $/GFLOP metric allows users to estimate straightforward costs—currently limited to CPU usage—for different applications with respect to computational efficiency.

2 Related Work

Cloud computing has been proposed as a service to scale out or share existing scientific clusters. The Eucalyptus project provides an open source cloud management tool [12]. Similarly, the Nimbus project provides cloud management tools for clusters [7]. These tools do not typically provide for multi-tenancy—unlike the commercial cloud environments we study—and thus observe different performance characteristics closer to existing scientific clusters with exclusive access. We are concerned in this chapter with the effects of contention on HPC applications when multiple VMs share resources.

Tikotekar et al. run different HPC benchmarks in virtual machines to determine the effects of virtualization on high-performance computing applications [17]. The results show isolating the effects of multiple VMs on the same machine is difficult, but in general virtualization has little effect on the performance of compute intensive HPC applications. Similarly, Youseff et al. see little performance impact on the memory hierarchy behavior of linear algebra libraries under virtualization [21]. However, this study does not consider contention between multi-tenant VMs.

The impact of virtualization on the networking performance of EC2 is analyzed empirically by Wang and Ng [20]. They report heavy network instabilities and delays, especially on small 32-bit nodes. We observe poor performance on the HPL parallel benchmark using multiple instances but are more concerned with the scaling properties. Although the instances do not approach peak performance, the parallel job can scale (for tens of nodes) with poor network performance provided the instances have enough installed RAM due to the high ratio of computation to communication in dense linear algebra. Our experiments (see Figure 8) demonstrate this effect.

The effects of sharing the last-level cache in multi-core processors is discussed by Iyer et al. [6]. The paper discusses different approaches to improving the performance guarantees of each core with respect to cache behavior. Using these techniques could greatly increase the predictability of performance on cloud platforms.

Previously, Edward Walker has compared EC2 nodes to current HPC systems [19]. Our results here are similar to his for the small clusters of 4 nodes that he used. We previously reported preliminary results in [10].

3 Background

We perform our experiments on the Amazon Elastic Compute Cloud (EC2) service as a case study for commercial cloud environments [15]. Although there are competing cloud offerings that were publicly available at the time [9, 5], Amazon's service is the largest that provides highly configurable virtual machines. The nodes allocated by EC2 run a kernel or operating system configured by Amazon, but all software above this level is configured by the user. Many other cloud offerings by other

providers limit applications to certain APIs or languages. To use existing highly optimized dense linear algebra libraries, we use Amazon as a case study.

Nodes allocated through EC2 are called *instances.* Instances are allocated from Amazon's data centers according to unpublished scheduling algorithms. Allocations are initially limited to 20 total instances, but this restriction can be lifted upon request. Data centers are combined into entities known as an *availability zone*, Amazon's smallest logical geographic entity for allocation. These zones are further combined into regions, which consist of only the US and Europe at the moment.

After allocation, each instance automatically loads a user-specified *image* containing the proper operating system (in our case Linux) and user software (described below). Images are loaded automatically by Amazon services onto one or more virtualized processors using the Xen virtual machine (VM) [1]. Each processor is itself multi-core, resulting in a total of 2 to 8 virtual cores for the instances we reserved. The Terms of Service provided by Amazon do not provide strong performance guarantees. Most importantly for this study, they do not limit Amazon's ability to implement *multi-tenancy;* that is, to co-locate VMs from different customers. We discuss the performance characteristics of different instances below.

Tools written to Amazon's public APIs provide the abilities to allocate extra nodes on demand, release unused nodes, and create and destroy images to be loaded onto allocated instances. Using these tools and developing our own, we built images with the latest compilers provided by the hardware CPU vendors AMD and Intel. We use HPL 2.0 [13] from the University of Tennessee, compiled with GotoBLAS 1.26 [4] from the Texas Advanced Computing Center (TACC), and MPICH2 1.0.8 [8] from the Argonne National Laboratory. Using our tools we can allocate and configure variable size clusters in EC2 automatically, including support for MPI applications.

Although we developed tools to automatically manage and configure EC2 nodes for our applications, there are also other publicly available tools for running scientific applications on cloud platforms (including EC2) [11, 12]. Further, as the cloud computing platform matures, we expect much more development for specific applications such as high-performance computing to reduce or eliminate much of the initial learning curve for deploying scientific applications on cloud platforms. Already, for example, public images are available on EC2 supporting MPICH [3].

3.1 Overview of Amazon EC2 Setup

Our case study was carried out using various instance types on the Amazon Elastic Compute Cloud (EC2) service from November 2008 through January 2010. Table 1 describes the salient differences between the instance types: number of cores per instance, installed memory, theoretical peak performance, and the cost of the instance per hour. We only used instances with 64-bit processors so that we treat the m1.large as the smallest instance although Amazon provides a smaller 32-bit m1.small instance. The costs per node vary by a factor of 7 from $0.34 for the smallest to $2.40

Table 1 Information about various instances types: processor type, number of cores per instance, installed RAM (in Gigabytes), and theoretical peak performance (in GFLOP/sec). Prices are on Amazon EC2 as of January, 2010.

Instance	Processor	Cores	RAM (GB)	Peak (Gflops)	Price ($/hr)
m1.large	Intel Xeon E5430	2	7.5	21.28	$0.34
m1.large	AMD Opteron 270	2	7.5	8.00	$0.34
m1.xlarge	Intel Xeon E5430	4	15	42.56	$0.68
m1.xlarge	AMD Opteron 270	4	15	16.00	$0.68
c1.xlarge	Intel Xeon E5345	8	7	74.56	$0.68
m2.2xlarge	Intel Xeon X5550	4	34.2	42.72	$1.20
m2.4xlarge	Intel Xeon X5550	8	68.4	85.44	$2.40

for nodes with significant installed memory. We note that cost scales more closely with installed RAM than with peak CPU performance—the c1.xlarge instance being the outlier. Peak performance is calculated using processor-specific capabilities. For example, the c1.xlarge instance type consists of 2 Intel Xeon quad-core processors operating at a frequency of 2.3 GHz with a total memory of 7 GB. Each core is capable of executing 4 floating-point operations per clock cycle, leading to a theoretical peak performance of 74.56 GFLOP/sec per node. There are additional costs for bandwidth used into and out from Amazon's network and for long-term storage of data, but we ignore these costs in our calculations because they are negligible compared to the costs of the computation itself in our experiments.

In regards to multithreaded parallelism provided by the multi-core processors, extensive testing typically delivered the best performance when we set the Goto BLAS library to use as many threads as available cores per socket—4 and 2, for the Xeon and the Opteron, respectively. We provide the number of threads used to obtain specific results in the following section when presenting peak achieved efficiency. With these settings and using the platform-specific libraries and compilers, we reached 76% and 68% of theoretical peak performance (as measured in GFLOP/sec) for the Xeon E5345 and Opteron 270, respectively, for single node performance on xlarge instances. We thus believe the configuration and execution of LINPACK in HPL on the high-CPU and standard instances is efficient enough to use as an exemplar of compute-intensive applications for the purposes of our evaluation.

All instance types (with Intel or AMD CPUs) execute the RedHat Fedora Core 8 operating system using the 2.6.21 Linux kernel. The 2.6 line of Linux kernels supports autotuning of buffer sizes for high-performance networking, which is enabled by default. The specific interconnect used by Amazon is unspecified [15] and multiple instances might even share a single hardware network card [20]. Therefore, the entire throughput might not be available to any particular instance. In order to reduce the number of hops between nodes to the best of our ability, we run all experiments with cluster nodes allocated in the same availability zone.

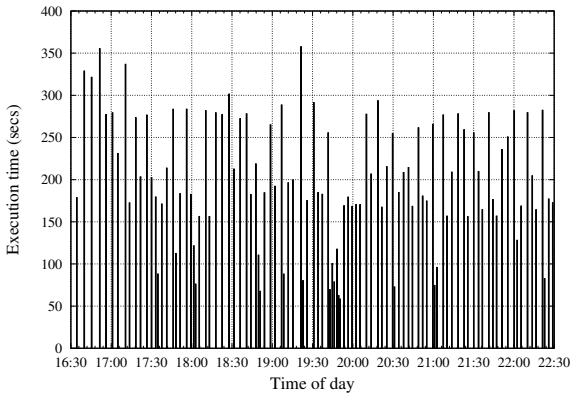

Fig. 1 Execution time of repeated DGEMM using 4 (of 8) cores over 6 hours on c1.xlarge instance. Average execution is 191.8 s with a standard deviation of 68.6 s. The input sizes for each DGEMM are identical. On a stand alone node all executions would be identical.

3.2 Overview of HPL

Our goal is to determine the suitability of commercial cloud environments for certain kinds of scientific applications. We focus on the HPL benchmark [13] as the exemplar of tightly coupled, highly parallel scientific applications. HPL computes the solution of a random dense system of linear equations via LU factorization with partial pivoting and triangular solves. This algorithm requires $O(n^3)$ floating point operations on $O(n^2)$ data; that is, HPL is compute-intensive, and represents a realistic upper bound for the performance of such scientific applications. The actual implementation is driven by more than a dozen parameters, all of which may have a significant impact on the resulting performance and therefore require fine tuning. We describe the HPL parameters that were tuned below:

1. Block size (*NB*). It is determined in relation to the problem size and the performance of the underlying BLAS kernels. We used four different block sizes, namely 192, 256, 512, and 768.
2. Process grid ($p \times q$). This is the number of process rows and columns of the compute grid. As with most clusters, we empirically observed that on EC2 it is better to use process grids where $p \leq q$. This is a product of the data flow of the algorithms used in HPL.
3. Broadcast algorithm (*BFACT*). It depends on the problem size and network performance. Testing suggested that the best broadcast parameters are 3 and 5. For large machines featuring fast nodes compared to the available network bandwidth, algorithm 5 is observed to be best.

With respect to the other HPL settings, we kept them fixed for all the experiments.

4 Intranode Scaling

We begin our empirical analysis of EC2 performance for linear algebra by evaluating the consistency of achievable performance on a single node. In order to evaluate the consistency in performance delivered by EC2 nodes, we executed DGEMM—the matrix-matrix multiplication kernel of BLAS—and HPL tests for 24 hours, repeating the experiment over different days. We first focus on the DGEMM results, then discuss results from HPL. DGEMM is at the core of the Basic Linear Algebra Subroutines (BLAS) library. Implementing the basic operation of multiplying two matrices, DGEMM is the building block of all the other Level-3 BLAS routines and of virtually every linear algebra algorithm. It is highly optimized for each target architecture and its performance, often in the 90+% range of efficiency, is ordinarily interpreted as the processor peak achievable performance.

4.1 DGEMM Single Node Evaluation

In our DGEMM experiment we initialize three square matrices A, B and C of a fixed size and then invoke the GotoBLAS implementation of DGEMM [4]. We only time the call to the BLAS library and not the time spent allocating and initializing the matrices. Due to the dense nature of the matrices involved in the experiments—most of the entries are non-zero—we expect little to no fluctuations in the execution time on a single node independent of problem size and the number of cores used.

Figure 1 presents the time to complete the same DGEMM computation repeatedly over six hours on EC2. For space reasons we focus on six hours; however, the rest of the time shows similar behavior. In this experiment, only four of the eight cores of a c1.xlarge instance were used. The results show very high variability in execution time with an average of 191.8 seconds and standard deviation of 68.6 seconds (36% of average). There are several possible sources for such variability: 1) the process is not being run for extended, variable periods of time, 2) the threads are being scheduled on different cores each time (reducing first-level cache performance), and 3) the last-level cache shared by all cores is less available to each thread (because it is being used by another thread on a different core).

Figure 2 shows a similar experiment to Figure 1 but using only one of the eight cores. The experiment shows none of the variability when using only one core that is present using more cores. With an average execution time of 227.9 s, the standard deviation is only 0.23 s. The results for using all eight cores shows even higher levels of variability than Figure 1. The reduced variability of a single core demonstrates cause (1) above is unlikely. The process is scheduled similarly, but performance is much more predictable. Cause (2), however, cannot be ruled out because Amazon EC2 provides no mechanism to *pin* threads to particular cores so that the thread always executes on the same physical core of the processor. Finally, we conjecture cause (3) plays at least as significant a role as (2), and we look at different experiments to explore these effects.

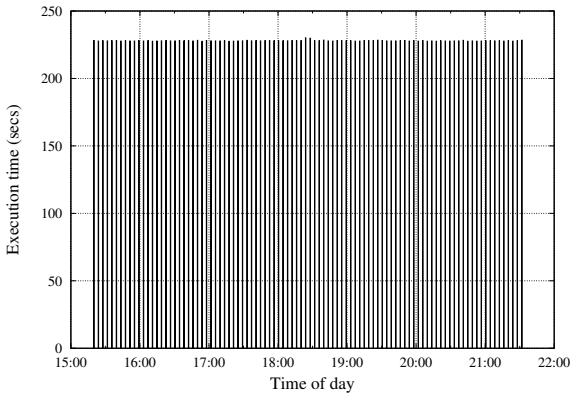

Fig. 2 Execution time of repeated DGEMM using 1 (of 8) core over 6 hours on c1.xlarge instance. Average execution is 227.9 s with a standard deviation of 0.23 s. The input sizes to each DGEMM are identical.

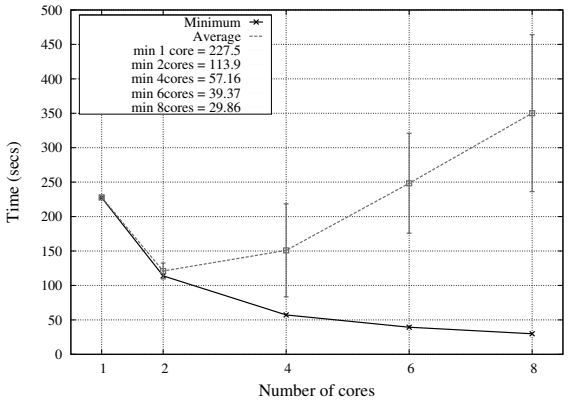

Fig. 3 Average execution time of repeated DGEMM with $n = 10k$ on c1.xlarge instance by number of cores used. The matrices cannot fit within the last-level cache. The inputs sizes for each DGEMM are identical. Error bars on average show one standard deviation.

The effects of using different numbers of cores is easily observed. The average and minimum execution time against the number of cores appear in Figure 3. In this graph, DGEMM was executed over several hours on a matrix of size $n = 10k$ so that the matrices (each 762 MB) cannot fit in the 8 MB last-level cache of the Intel XEON running the c1.xlarge instance. The results presented are average and minimum execution times for the DGEMM. Again, allocation and initialization of memory are not included in the timings. Error-bars show the standard deviation of the average. There are several notable characteristics of the graph:

1. The best performance in graphs (minimum line) shows that we can reach roughly 90% efficiency. Such performance is close to the optimal achievable. For example, such performance would be expected in a stand alone node. We note that virtualization alone clearly does not have a significant impact on peak performance.
2. The average performance shows that similar computations will not likely ever reach optimal. The average efficiency—as given by the inverse of the spread in graph between the min and average—drops dramatically. The best average

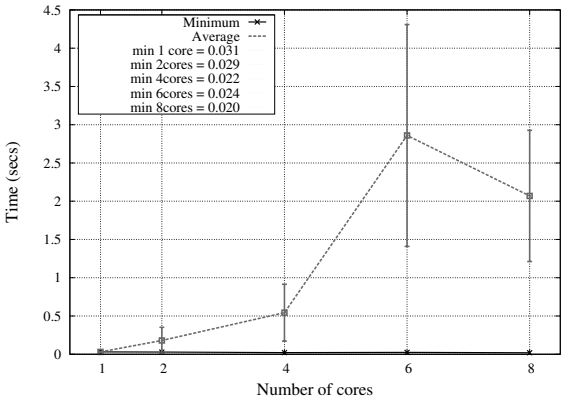

Fig. 4 Average execution time of repeated DGEMM with $n = 500$ on c1.xlarge instance by number of cores used. All the matrices fit within the last-level cache at once. The inputs sizes for each DGEMM are identical. Error bars on average show one standard deviation.

performance at two cores is still several times worse than the minimum execution time at eight cores.

3. As the number of cores increases, the average significantly increases along with the standard deviation. The standard deviation increases by four orders of magnitude. Expected performance diverges so significantly from best performance that when using eight cores one would rarely expect to achieve the best performance.
4. Underutilization is a good policy on EC2. Using two cores appears to be the sweet spot in this experiment. Efficiency is good and the average and expected performance are the best and in line with optimal. Using four cores already takes longer on average than using two, although using four is still slightly faster than a single core, and the trend only worsens as the number of cores increases. Note that the fastest expected time is obtained by only using only a quarter of the machine!

While we can attribute the performance degradation when using more than two cores per node in Figure 3 to ever worse cache behavior, we cannot easily distinguish which caches are being missed. In the multicore processors in these instances, cores have individual caches and a large shared last-level cache (LLC). Pinning a thread to a particular core would help the thread maintain cache consistency for the individual caches while reducing memory demand of the threads would reduce contention on the shared LLC. However, the kernels supplied by Amazon EC2 do not provide support to allow us to pin threads. To attempt to distinguish these effects without the ability to fix threads to particular cores, we performed similar DGEMM experiments with smaller matrices that all fit within the LLC of the processor.

Figure 4 shows the average and minimum execution times using different numbers of cores for a DGEMM on small matrices $n = 500$ yielding matrices of 1.9 MB that can easily fit within the 8 MB LLC. The error-bars on the average execution times give a standard deviation from the mean. Again, allocation and initialization of memory are not included in the timings. We note several characteristics of the graph:

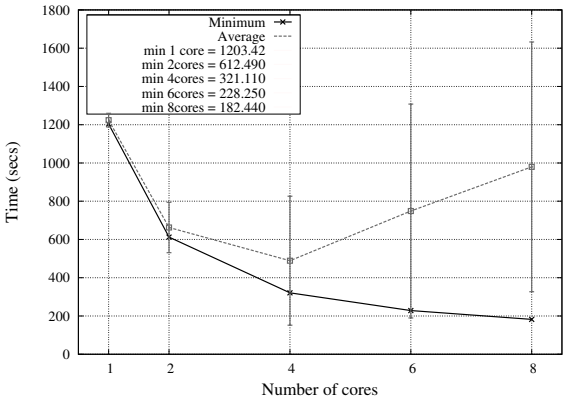

Fig. 5 Average execution time of repeated HPL with $n = 25\,\text{k}$ on c1.xlarge instance by number of cores used. The matrices cannot fit within the last-level cache. The input configuration to each execution of HPL is identical. Error bars on average show one standard deviation.

1. As with the out-of-cache DGEMM, the minimum execution times are consistent with the expected performance of a stand-alone node.
2. Contention is likely for the last-level cache. The two orders of magnitude performance degradation between a single core and eight cores is also the difference in time between accessing the last-level cache and main memory.
3. The ability to pin threads to cores would probably help significantly. Using multiple cores always performs worse than with a single core, implying significant overhead from coordination. Since the minimum scales well, we conjecture that the average suffers from poor cache behavior. However, the average time with multiple cores does not become orders of magnitude worse—as would be caused by going to main memory—until all of the cores are used. Lower level cache misses are thus more likely when underutilizing the instance with fewer than eight cores.

4.2 HPL Single Node Evaluation

In order to determine whether the effects we have seen using DGEMM scale to multiple nodes, we use the HPL benchmark [13] that solves a system of linear equations via LU factorization. HPL can scale to large compute grids. However, we first consider HPL given similar matrices to the previous DGEMM experiments, namely, solving a system of linear equations on a single node where the data does fit and does not fit into the LLC of the processor. In the following section, we extend HPL to examine internode scaling using parallel algorithms.

In Figure 5 we repeatedly execute HPL with $n = 25\,\text{k}$ on the Amazon EC2 c1.xlarge instance. We plot average and minimum execution times against the number of cores. As with the DGEMM experiments, error bars show one standard deviation. As with the first DGEMM experiment, these matrices do not fit within the LLC of the processor, but do fit within the main memory of a single instance. The

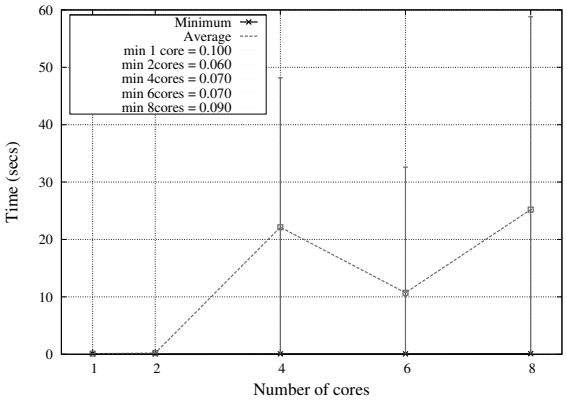

Fig. 6 Average execution time of repeated HPL with $n = 1\,\mathrm{k}$ on c1.xlarge instance by number of cores used. The matrices fit within the last-level cache. The input configuration to each execution of HPL is identical. Error bars on average show one standard deviation.

results are quite similar to DGEMM, but show slightly different trends. We do not directly compare the DGEMM and HPL experiments because of the use of very different algorithms. We only point out that the trends show similar behavior on a single node.

In the last single node figure, we show HPL with a matrix of size $n = 1\,\mathrm{k}$ so that all data for the computation fits within the LLC. Figure 6 plots the average (and one standard deviation) and minimum execution times of repeated runs of the HPL benchmark. The results show similar behavior to the corresponding DGEMM experiment. We could not find a source for the anomalous improved performance at six cores as compared to four or eight. In the following section, we extend HPL to multiple nodes to take a look at the effects of contention on parallel computations in a commercial cloud environment.

From the single node experiments we conclude that the very high variance in performance implies that the best achieved performance is not a good measure of expected performance. The average expected performance can be several orders of magnitude worse depending on the cache behavior of the algorithm in both efficiency and time to solution. In our experiments, the best average expected performance is obtained on Amazon using much less of the machine than is allocated: In an experiment accessing a large amount of main memory, using only a quarter of the machine provided the best expected performance!

5 Internode Scaling

The previous section demonstrates the significant effects of contention on single node performance in a commercial cloud environment. In this section, we extend our empirical analysis to parallel multi-node algorithms using the HPL benchmark. The HPL benchmark represents tightly coupled, highly parallel algorithms frequently used in scientific applications. We executed the HPL benchmark on different in-

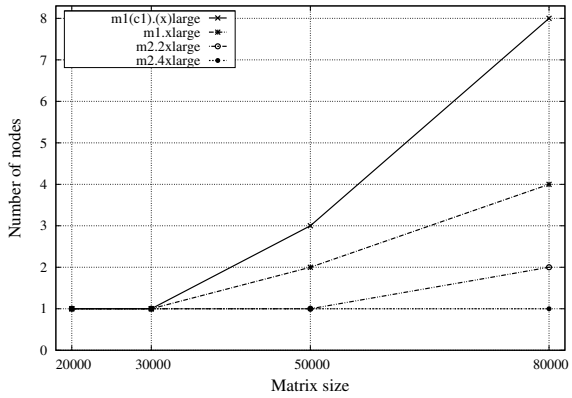

Fig. 7 Required number of nodes for different input matrix sizes n. The number of nodes is determined by maximizing usage of RAM on each node. We could not allocate enough m1.large nodes of AMD type to solve a problem of size $n = 80$k.

stance types on Amazon EC2, varying the parameters as described in Section 3.2. Using different instance types allowed us to see the effects on performance, efficiency, and cost of varying the amount of RAM available and the problem size.

Tables 2–5 show the parameters used to obtain the best results over different problem sizes, instance types, and number of cores used per node. To calculate the total number of cores used for a particular execution, use the result of $p \times q \times$ Threads. Due to the numerous parameters of HPL, it was necessary to try different input configurations to maximize performance. To maximize the performance, we minimized reliance on the interconnect by maximizing the memory allocation per node used to solve a particular problem size. Figure 7 shows the number of nodes used for different instance types to solve each problem size. Only the m2.4xlarge instance with 68.4 GB of RAM is large enough to solve all problem sizes on a single node. We used this as a reference point in Figure 8 to bound the effects of network usage on the performance.

Table 2 Results of HPL benchmarks for matrix size $n = 20$ k. Columns are, in order: Amazon EC2 instance type, CPU type (Xeon or Opteron), block size (NB), process grid ($p \times q$), number of threads used in the BLAS routines, broadcast algorithm (*BFACT*), best elapsed time, and corresponding efficiency using the theoretical peak performance from Table 1. A more detailed description of the parameters can be found in Section 3.2.

Instance	Proc.	Block size	p×q	Threads	Bcast	Time (min:sec)	Efficiency (% peak)
m1.large	Xeon	256	1x1	2	3	06:29.35	64.38
m1.large	Opt.	192	1x2	1	3	12:24.21	89.62
m1.xlarge	Xeon	256	1x1	4	3	05:07.30	40.78
m1.xlarge	Opt.	256	1x2	2	3	06:28.16	85.88
c1.xlarge	Xeon	512	1x1	8	3	01:51.85	63.96
m2.2xlarge	Xeon	256	1x2	2	3	03:19.74	62.50
m2.4xlarge	Xeon	256	1x1	8	3	01:44.55	59.71

Table 3 Results of HPL benchmarks for matrix size $n = 30\,\text{k}$.

Instance	Proc.	Block size	p×q	Threads	Bcast	Time (min:sec)	Efficiency (% peak)
m1.large	Xeon	256	1x1	2	3	20:20.82	69.31
m1.large	Opt.	256	1x2	1	3	41:21.01	90.75
m1.xlarge	Xeon	256	1x2	2	3	11:00.73	64.00
m1.xlarge	Opt.	192	1x2	2	3	21:20.46	87.88
c1.xlarge	Xeon	512	1x1	8	3	05:19.10	75.66
m2.2xlarge	Xeon	256	1x2	2	3	11:12.15	62.69
m2.4xlarge	Xeon	512	1x1	8	3	05:46.53	60.80

Table 4 Results of HPL benchmarks for matrix size $n = 50\,\text{k}$.

Instance	Proc.	Block size	p×q	Threads	Bcast	Time (hr:min:sec)	Efficiency (% peak)
m1.large	Xeon	512	1x3	2	5	35:55.66	60.56
m1.large	Opt.	256	1x3	2	3	1:09:44.47	83.00
m1.xlarge	Xeon	512	1x2	4	5	28:07.85	58.00
m1.xlarge	Opt.	512	1x4	2	5	1:00:46.59	71.41
c1.xlarge	Xeon	512	3x1	8	5	18:14.36	34.04
m2.2xlarge	Xeon	256	1x2	2	3	51:09.45	63.11
m2.4xlarge	Xeon	256	1x2	8	3	26:16.52	61.87

Table 5 Results of HPL benchmarks for matrix size $n = 80\,\text{k}$.[a]

Instance	Proc.	Block size	p×q	Threads	Bcast	Time (hr:min:sec)	Efficiency (% peak)
m1.large	Xeon	768	2x4	2	5	1:01:06.31	54.69
m1.xlarge	Xeon	512	1x4	4	3	1:14:57.67	44.58
m1.xlarge	Opt.	512	2x2	4	5	3:27:46.61	42.78
c1.xlarge	Xeon	768	1x8	8	5	2:17:50.49	0.07
m2.2xlarge	Xeon	512	1x2	4	3	1:50:56.69	60.02
m2.4xlarge	Xeon	512	1x1	8	3	1:46:06.05	62.76

[a] We could not allocate enough m1.large nodes of Opteron type to solve a problem of size $n = 80k$.

In Figures 8–13 we present different aspects of the results of our HPL benchmark experiments given in Tables 2–5. We first discuss the best results obtained to give a reasonable lower bound on performance in a contentious commercial cloud environment. In these parallel experiments, the minimum times do not differ as greatly from the average as the single node experiments for several reasons: 1) the elapsed time required (more than an hour for large problem sizes) implies an averaging effect and 2) the overhead for parallel jobs is higher than single node experiments due to network usage. Finally, we note that in these figures the line for m1.large instances using the Opteron processor does not extend to $n = 80\,\text{k}$ because we could not allocate enough m1.large instances with Opteron CPUs in a single availability zone to solve a problem of size $n = 80k$.

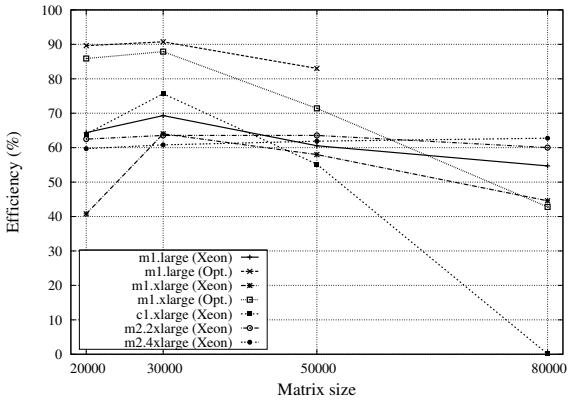

Fig. 8 Efficiency scaling by problem size. Efficiency is determined with respect to the theoretical peak given in Table 1 multiplied by the number of nodes used.

5.1 HPL Minimum Evaluation

Figure 8 demonstrates that efficiency is generally better than 60% for problem sizes below 80 k. Here, efficiency is considered to be percentage of theoretical peak given in Table 1 multiplied by the number of nodes used. We consider 60% to be reasonable performance given the relatively slow interconnect provided by Amazon EC2 compared to purpose-built HPC systems. The balance of resources available to the computation is clearly important. For example, the m2.2xlarge and m2.4xlarge instances have roughly equal performance at 80 k although HPL needs two of the m2.2xlarge instances and only one m2.4xlarge. In this case the interconnect does not have a significant effect because the nodes are provisioned with enough RAM to keep the CPU busy. The c1.xlarge instances suffer severe performance degradation, however. At $n = 80\,\text{k}$, the c1.xlarge instances perform two orders of magnitude worse than a single node. We conjecture that the 7 GB of RAM of c1.xlarge nodes is insufficient to keep the CPU busy given the same interconnect between instances. In general, nodes with more RAM clearly scale up in problem size much better as expected due to the high ratio between the required number of operations and the data size to solve a dense linear system.

While efficiency is important to gauge the scalability of the implementation, for any particular experiment the most important concern is generally time to solution. Figure 9 shows the total time to solution for different instance types by problem size. This graph shows that although the efficiency of larger nodes is somewhat better than that of the smaller ones, in most cases the smaller nodes reach the solution faster. The c1.xlarge instances, recommended for high-compute applications, and the m2.2xlarge and m2.4xlarge instances, recommended for high-memory applications, are generally the slowest.

One of the important concerns when using commercial cloud environments are the differing costs of different instances. Figure 10 provides the cost to solution for different instance types by problem size. In this graph, the cost is prorated to the second to illustrate the marginal costs incurred when allocating a cluster to solve

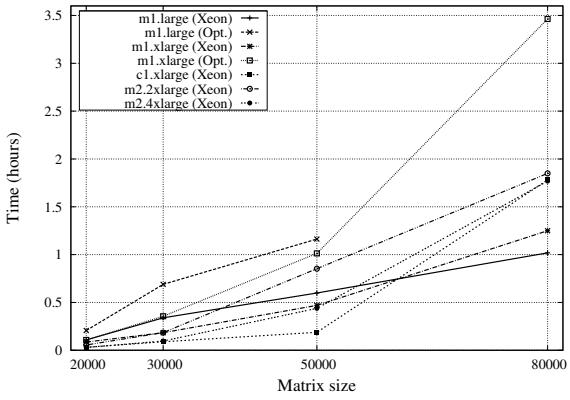

Fig. 9 Total time to solution for different instance types by problem size.

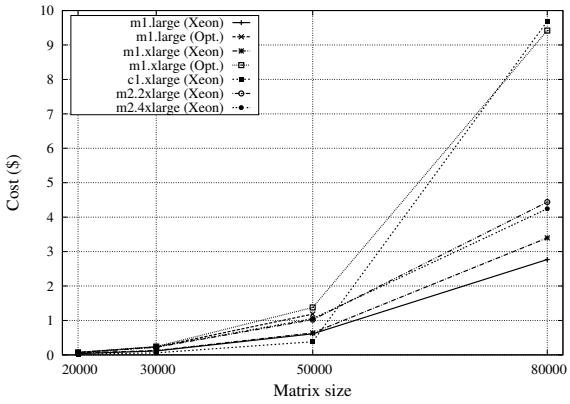

Fig. 10 Cost to solution prorated to actual time spent for different instance types by problem size.

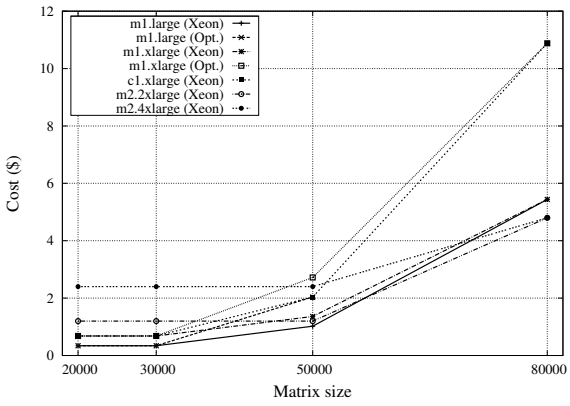

Fig. 11 Actual cost to solution for different instance types by problem size. Execution time is rounded to the nearest hour for cost calculation.

several problems. Figure 11 provides the comparable actual costs of using the different instance types to solve a single execution each problem size; that is, this graph includes the costs of the remaining hour after the execution completes. The salient difference between the figures is the relation of the largest nodes to the smaller nodes. Using absolute cost, the largest nodes are cheapest for large problems, but using prorated costs they are more expensive. Since the prorated trends are more informative for different problem sizes and for multiple jobs, we conclude the smaller instances m1.large and m1.xlarge are the most cost effective for parallel jobs.

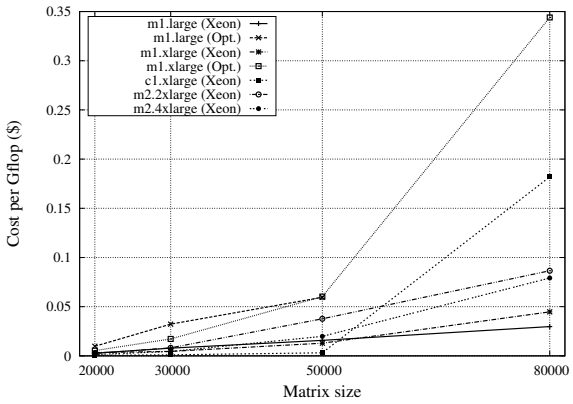

Fig. 12 Cost per GFLOP ($/GFLOP) prorated to actual time spent for different instance types by problem size.

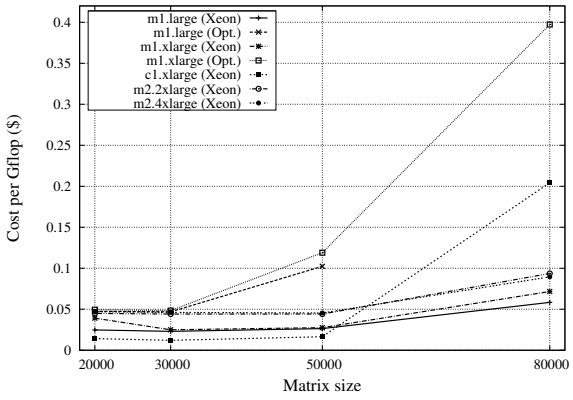

Fig. 13 Actual cost to solution per GFLOP ($/GFLOP) for different instance types by problem size. Execution time is rounded to the nearest hour for cost calculation.

In addition to cost to solution, we also consider a more general cost measure, $/GFLOP, calculated using the ratio between the total GFLOPS returned by the HPL benchmark and the (prorated) cost for the specific computation. This measure allows a rough conversion of expected cost for other problem sizes including other scientific applications characterized by similar computational needs. Figures 12 and 13 show the results of the HPL benchmarks giving the cost per GFLOP for differ-

ent instances by problem size. The cost per GFLOP measure magnifies the differences between the instance types, but of course the best instance type in Figures 10 and 11—m1.large—is still the best for the price to performance ratio.

The m1.large instance is the fastest by Figure 9 and the best for price to performance by Figure 10, leading us to recommend the smallest (64-bit) instance for most parallel linear algebra compute jobs on the Amazon EC2 cloud environment according to the empirical upper bound on performance. The high-compute and high-memory instances are not worth the extra costs in our experiments. Given the high variability in performance of single nodes, we examine in the following section the expected performance from instances instead of the upper bound to determine if our recommendation holds also for expected average performance.

5.2 HPL Average Evaluation

The minimum execution times from the previous section provide a rough upper bound for performance. In this section, we examine the average execution times for HPL to provide a better estimate of the expected performance of applications with dense linear algebra. Figures 14–17 provide the minimum and average (and standard deviation) execution times for different instances by problem size. We do not show the m2.2xlarge or m2.4xlarge instances due to insufficient data. These large instances are also quite expensive to allocate.

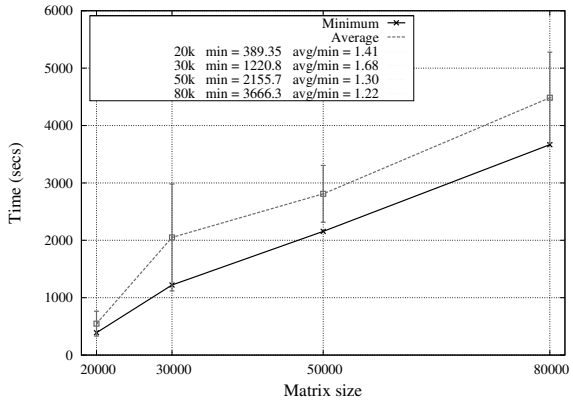

Fig. 14 Minimum and average execution time of HPL on m1.large (Xeon) instances by problem size. Error bars on average show one standard deviation.

As with the single node experiments, Figures 14 through 17 show that the expected performance is worse than the best performance, but by a much smaller margin. Indeed, for the m1.large instances in Figure 14, the expected performance at $n = 80\,\mathrm{k}$ is only 22% worse than the best execution time. Generally, the average for the HPL experiments is between 30% and $2\times$ worse than the minimum execution times. As we mentioned at the beginning of this section, we believe the smaller

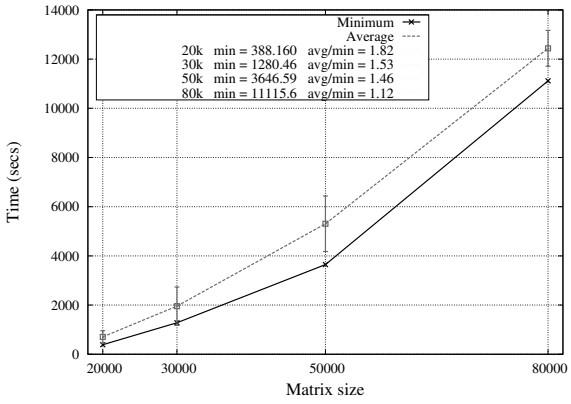

Fig. 15 Minimum and average execution time of HPL on m1.xlarge (Opt.) instances by problem size. Error bars on average show one standard deviation.

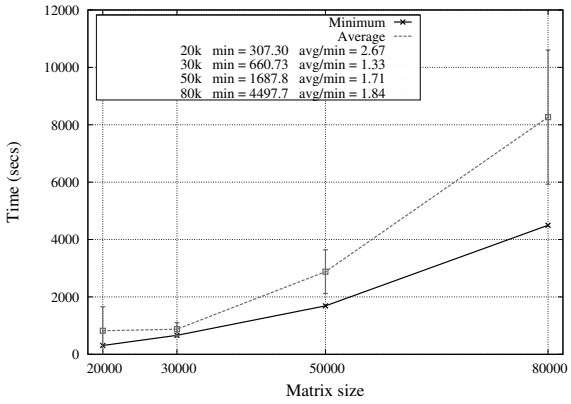

Fig. 16 Minimum and average execution time of HPL on m1.xlarge (Xeon) instances by problem size. Error bars on average show one standard deviation.

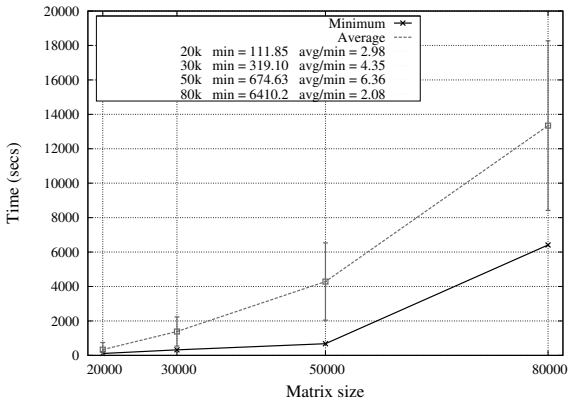

Fig. 17 Minimum and average execution time of HPL on c1.xlarge instances by problem size. Error bars on average show one standard deviation.

differences between the expected execution time and the best times are due to an averaging effect from the length of the experiments and the greater overhead of the HPL benchmark from network traffic (as compared to the DGEMM experiments in Section 4.1).

The m1.large instance—the smallest that we tested—remains the best instance on EC2 for tightly coupled, compute intensive, parallel jobs. Although the average expected time is longer than the minimum by 20% for the largest problem size, the expected time is still faster than the next fastest minimum time (m1.xlarge Xeon). Given that the m1.large also costs half as much as the m1.xlarge, the smallest instance is the clear winner for the tightly-coupled, dense linear algebra computation that we evaluated, and the high marginal costs for high-compute or high-memory nodes is not cost effective.

6 Conclusions

In this chapter we demonstrated empirically the computational efficiency of high-performance numerical applications in a commercial cloud environment when resources are shared under high contention. Through a case study using the Linpack benchmark, we show that cache utilization becomes highly unpredictable and similarly affects computation time. For some problems, not only is it more efficient to underutilize resources, but the solution can be reached sooner in realtime (wall-time). We also show that the smallest, cheapest (64-bit) instance on the Amazon EC2 commercial cloud environment is not only the fastest, but also the best for price to performance ratio.

We presented the *average expected* performance and execution time, expected cost to completion, and variance measures—traditionally ignored in the high performance computing context—to determine the efficiency and performance of the Amazon EC2 commercial cloud environment. Under omnipresent contention for resources, the expected performance in the cloud environment diverges by an order of magnitude from the best achieved performance. We conclude that there is significant space for improvement in providing predictable performance in such environments.

Acknowledgements The authors wish to acknowledge the Aachen Institute for Advanced Study in Computational Engineering Science (AICES) as sponsor of the experimental component of this research. Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged. Also, support from the XtreemOS project, which is partially funded by the European Commission under contract #FP6-033576 is gratefully acknowledged.

References

1. Barham, P.T., Dragovic, B., Fraser, K., Hand, S., Harris, T.L., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Symposium on Operating Systems Principles, pp. 164–177. Bolton Landing, New York (2003)
2. Dongarra, J., van de Geijn, R., Walker, D.: Scalability issues affecting the design of a dense linear algebra library. Journal of Parallel and Distributed Computing **22**(3), 523–537 (1994)
3. Gemignani, C., Skomoroch, P.: Elasticwulf: Beowulf cluster run on Amazon EC2. Available via the WWW. Cited 1 Jan 2010. http://code.google.com/p/elasticwulf/
4. Goto, K.: GotoBLAS. Available via the WWW. Cited 1 Jan 2010. http://www.tacc.utexas.edu/resources/software/#blas
5. Hosting, S.D.: GoGrid cloud hosting. Available via the WWW. Cited 1 Jan 2010. http://gogrid.com
6. Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., Reinhardt, S.: Qos policies and architecture for cache/memory in cmp platforms. SIGMETRICS Perform. Eval. Rev. **35**(1), 25–36 (2007). DOI http://doi.acm.org/10.1145/1269899.1254886
7. Keahey, K., Freeman, T., Lauret, J., Olson, D.: Virtual workspaces for scientific applications. In: SciDAC 2007 Conference (2007)
8. Laboratory, A.N.: MPICH2: High-performance and widely portable MPI. Available via the WWW. Cited 1 Jan 2010. http://www.mcs.anl.gov/research/projects/mpich2/
9. xcalibre communications ltd: FlexiScale cloud computing. Available via the WWW. Cited 1 Jan 2010. http://www.flexiscale.com
10. Napper, J., Bientinesi, P.: Can cloud computing reach the Top500? In: Unconventional High-Performance Computing (UCHPC) (2009)
11. Nimbus science clouds. Available via the WWW. Cited 1 Jan 2010. http://workspace.globus.org/
12. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus open-source cloud-computing system. In: Proceedings of Cloud Computing and Its Applications [online] (2008)
13. Petitet, A., Whaley, R.C., Dongarra, J., Cleary, A.: HPL - a portable implementation of the high-performance LINPACK benchmark for distributed-memory computers. Available via the WWW. Cited 1 Jan 2010. http://www.netlib.org/benchmark/hpl/
14. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In: SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, p. 55. IEEE Computer Society, Washington, DC, USA (2003)
15. Services, A.W.: Amazon elastic compute cloud (EC2). Available via the WWW. Cited 1 Jan 2010. http://aws.amazon.com/ec2
16. Strebel, J., Stage, A.: An economic decision model for business software application deployment on hybrid cloud environments. In: M. Schumann, L.M. Kolbe, M.H. Breitner (eds.) Tagungsband Multikonferenz Wirtschaftsinformatik (2010). Forthcoming
17. Tikotekar, A., Vallée, G., Naughton, T., Ong, H., Engelmann, C., Scott, S.L.: An analysis of HPC benchmarks in virtual machine environments. In: Euro-Par 2008 Workshops - Parallel Processing: VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers, pp. 63–71. Springer-Verlag, Berlin, Heidelberg (2009). DOI http://dx.doi.org/10.1007/978-3-642-00955-6_8
18. TOP500.Org: Top 500 supercomputer sites. Available via the WWW. Cited 1 Jan 2010. http://www.top500.org/
19. Walker, E.: Benchmarking Amazon EC2. ;LOGIN: pp. 18–23 (2008)
20. Wang, G., Ng, E.: The impact of virtualization on network performance of Amazon EC2 data center. In: INFOCOM '10: Proceedings of the 2010 IEEE Conference on Computer Communications. IEEE Communication Society (2010)

21. Youseff, L., Seymour, K., You, H., Dongarra, J., Wolski, R.: The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In: HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing, pp. 141–152. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1383422.1383440

Index