Efficient and Cost-Effective HPC on the Cloud

Aditya Bhosale
adityapb@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, IL, USA

Laxmikant Kale
kale@illinois.edu
University of Illinois
Urbana-Champaign
Urbana, IL, USA

Sara Kokkila-Schumacher
saraks@ibm.com
IBM Thomas J. Watson Research
Center
Yorktown Heights, NY, USA

Abstract

HPC applications are increasingly utilizing cloud resources due to their cost-effectiveness. Among these resources, spot compute instances present an opportunity to run applications at deep discounts compared to on-demand instances. However, they present unique challenges for tightly-coupled HPC applications due to potential interruptions. Traditional parallel programming models like MPI are not inherently fault-tolerant, and existing methods to handle these interruptions are inefficient and require significant programmer effort. In this paper, we present Charm++ as an alternative solution that natively supports fault tolerance, dynamic load balancing, and resource rescaling. We present a tool to run Charm++ applications with a mix of on-demand and spot instances which can detect and efficiently handle spot interruptions without a shared filesystem. We show that using spot instances can result in up to 60% cost savings for our benchmark application.

Keywords

HPC, spot instances, fault-tolerance, cost-efficiency

ACM Reference Format:
Aditya Bhosale, Laxmikant Kale, and Sara Kokkila-Schumacher. 2025. Efficient and Cost-Effective HPC on the Cloud. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3731545.3744667

1 Introduction

HPC applications have traditionally been run on dedicated supercomputers with high-bandwidth, low-latency interconnects. However, due to its cost-effective nature, over the past decade, there has been an increasing adoption of cloud resources for running HPC applications. In the past few years, the rise in popularity of AI has led to an acceleration of cloud adoption in the HPC community for running ML jobs. This increasing adoption has also resulted in increased availability of compute nodes with high-performance interconnects that are designed for tightly-coupled HPC applications.

Cloud providers offer unused capacity at deep discounts in the form of spot compute instances. Unlike reserved and on-demand instances with a static pricing model, spot instances use a dynamic pricing model based on supply and demand for that instance type. These instances can be available at up to 90% cheaper than on-demand instances. However, these instances can be interrupted with short notice if there is an increase in demand for on-demand instances of that type.

While spot instances allow running workloads on the cloud at a fraction of the on-demand cost, using these instances for tightly-coupled HPC applications remains a challenge because of interruptions. Widely used parallel programming models such as MPI are not inherently fault-tolerant and cannot continue execution when some of the processes are interrupted. According to AWS®, less than 5% of all spot instances are interrupted [3]. This may seem like a small percentage; however, the probability of at least one interruption increases exponentially as the number of instances is scaled up.

Existing methods to use spot compute instances for HPC typically use checkpoint and restart to handle interruptions [4, 9]. The application state is checkpointed to disk every few iterations and restarted from the latest checkpoint in case of an interruption. Other methods try to avoid interruptions by setting a large bid price for a spot instance or using statistical models to predict instance lifetimes [8, 9, 13–15].

Setting a large bid price to avoid interruptions is not cost-efficient and can incur higher costs if the demand for an instance type spikes. Moreover, none of the major cloud providers like AWS and Google Cloud^{TM} use a bidding system for setting the price anymore, and interruptions are based solely on demand. Statistical models can be used to predict instance lifetimes to pick the right instance types for a given workload, but they are not effective for long- running jobs with a high chance of interruptions.

Checkpoint and restart is an effective method to handle interruptions. However, this involves significant programmer effort to write applications that checkpoint the state periodically and balance the load effectively on restart. This method also incurs the additional cost of lost progress when the application state is restored from a previous checkpoint. Moreover, checkpointing to disk is a high-latency operation, especially in a cloud environment. Checkpointing also requires a shared filesystem which has additional monetary costs associated with it.

Charm++ [1] is a parallel programming model that natively supports fault-tolerance, dynamic load balancing, and dynamic rescaling of resources without additional programmer effort [6]. In this paper, we study the effectiveness of Charm++ runtime-adaptivity on spot compute instances. We present a utility that can launch Charm++ jobs with a mix of on-demand and spot instances on AWS, and can detect and efficiently handle spot interruptions without requiring a shared filesystem. We demonstrate the robust fault-tolerant nature of Charm++ by artificially inducing spot interruptions and measure their effect on application runtime.

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3744667

2 Overview

2.1 Charm++

Charm++ [1] is an asynchronous, message-driven parallel programming model. It employs an object-oriented approach in which parallel programs are written using C++ objects called *chares*. These chares communicate via remote method invocations. The distribution of chares across processors is managed by the Charm++ runtime system. Chares are migratable and can be moved between processors by the runtime system. Charm++ is widely used for scientific applications, including molecular dynamics [11], computational astronomy [7], and discrete event simulation [10].

2.1.1 Shrink/Expand in Charm++. The migratability of chares enables the runtime system to move objects away from processors to shrink the number of processes at runtime, or to move objects into new processors to expand the number of processes [6]. The rescaling operation is initiated by signaling the Charm++ application from an external program. During the next load balancing step, the Charm++ runtime system performs the following actions to complete the rescaling operation:

To shrink:

(1) Load balancing moves objects away from the processors to be removed.
(2) The application state is checkpointed locally to Linux shared memory.
(3) The application is restarted with a reduced number of processors, and the checkpoint data is restored to continue execution.

To expand:

(1) The application state is checkpointed locally to Linux shared memory.
(2) The application is restarted with an increased number of processors, and the checkpoint data is restored.
(3) Load balancing distributes the workload among the newly added processors.

We also added logic to handle scenarios where processors are both removed and added simultaneously. The system first moves objects from processors being removed, then rebalances load across newly added processors after restart.

To enable shrink/expand in existing Charm++ applications, each chare, including the main chare, needs to define the serialization-deserialization routines using the *pup* framework [1] in Charm++.

2.2 Amazon EC2 Spot Instances

In this paper, we used AWS as our target cloud provider. AWS offers three types of instances:

- *On-demand* — Instances that can be started and terminated at any time without a contract. These instances remain uninterrupted until explicitly terminated by the user.
- *Reserved* — Instances available under long-term contracts (1 to 3 years) for a specific instance type in a region. These offer significant cost savings compared to on-demand instances.
- *Spot* — Unused EC2 capacity made available at a discount, which can be terminated with a two-minute warning.

```python
from charm_cloud_manager import CharmCloudManager

charm_manager = CharmCloudManager(<path_to_keypair>)
ami_id = 'ami-xxxxxxxxx'
workdir = '/home/ec2-user/charm/examples/jacobi2d-iter'
command = f'time {workdir}/charmrun +p%(num_pes)s'
          '{workdir}/jacobi2d 16384 256 20000 +balancer'
          'GreedyCentralLB ++nodelist /tmp/nodelist ++server'
          '++server-port 1234 ++verbose +LBTestPESpeed'
charm_manager.run(
    ami_id, ["c5.large", "c5.xlarge"], 'charm-example',
    command, total_target_capacity=64, on_demand_count=1,
    key_name='keypair-xxx', subnet_ids=['subnet-xxxxxxxxxxx'],
    security_group_ids=['sg-xxxxxxxxxxxx']
)
```

Figure 1: Example script to run a 2D Jacobi Charm++ application using CharmCloudManager

AWS originally used a bidding-based pricing model for spot instances, where instances were terminated immediately when spot prices exceeded user bids. This caused high price volatility and frequent interruptions, making these instances suitable only for stateless workloads. HPC applications could only recover from interruptions through checkpointing.

In late 2017, AWS switched to dynamic pricing with rates set internally based on demand [2]. They introduced price smoothing to reduce volatility and added a two-minute interruption notice. This brief warning enables us to use Charm++'s shrink/expand capability to rescale applications and continue execution without disk-based checkpointing.

Even though our experiments are limited to AWS, these concepts can be used to target any other cloud provider that offers spot compute instances. For example, Google Cloud offers spot VMs that are preemptible with a 30 second notice [5].

3 Methodology

We wrote CharmCloudManager, a Python utility, to run Charm++ applications on Amazon EC2$^{®}$ spot instances. We use boto3 to interact with EC2. Figure 1 shows an example script for running a Charm++ application using CharmCloudManager.

Figure 2 shows the design of CharmCloudManager. We create an EC2 fleet based on the user-specified on-demand capacity, with the rest of the instances being spot instances to meet the target capacity. We use a cluster placement group in a single region and availabilty zone, and a single subnet to maximize network performance. We use a *maintain*-type fleet by default. A *maintain* fleet replaces interrupted spot instances to maintain the target capacity. The other option is to use an *instant* fleet, which doesn't replace interrupted spot instances.

The on-demand capacity needs to be at least 1, since we need the main node for Charm++ to be non-preemptible. CharmCloudManager then launches two asynchronous tasks: the execution task for running the execution command, and the monitoring task to monitor the instances for any interruptions.

Execution task. The execution task first creates the nodelist file and writes it to the volume attached to the main node, which is one of the on-demand instances. It then runs the execution command on the main node. When the execution completes, the task terminates the monitoring task and the EC2 fleet.

Monitoring task. The monitoring task runs a polling loop every 10 seconds. This loop first checks the instance metadata for all spot instances to find any interruption notices. It then looks up the active instances in the fleet to find any new instances that may have been launched to replace previously interrupted instances. If there are any interruption notices or new instances, the nodelist file on the main node is updated by removing entries corresponding to interrupted instances and adding entries for new instances. A signal is then sent to the Charm++ application on the main node to rescale accordingly.

4 Experiments

For these experiments, we used CharmCloudManager with the c5.xlarge instance type, which provides 4 vCPUs per instance. We used the *maintain* fleet type. A 2D stencil application was used as the benchmark for our experiments. This application solves the Laplace equation over a 2D grid using Jacobi iterations.

We employed a dummy load balancer for our runs, as the application does not exhibit any load imbalance. The dummy load balancer invokes load balancing only if the application is to be rescaled during that step; otherwise, it continues execution without performing load balancing.

4.1 Scaling performance

Figure 3 shows strong scaling performance of the 2D stencil application with two different grid sizes. We see near-perfect strong scaling up to 128 cores (or 32 instances).

4.2 Performance with interruptions

There are two sources of overhead when a spot instance is interrupted during a run. First, the rescaling overhead of Charm++. Second, the temporary drop in capacity that occurs after the Charm++ application is rescaled but before the interrupted spot instance is replaced in the fleet.

To evaluate application performance under spot interruptions, we injected interruptions into our spot allocation using the AWS Fault Injection Simulator. We conducted two experiments in this setting:

- To measure rescaling overhead, we injected a varying number of instance interruptions over a short duration, ensuring that all interruptions were handled within a single rescaling step in Charm++.
- To measure the impact of the temporary capacity drop, we recorded the end-to-end runtime while varying the number of instance interruptions.

For these runs, we used a total target capacity of 64 vCPUs and executed 20,000 iterations of the 2D stencil benchmark on a $16k \times 16k$ grid.

Rescaling Overhead. We measure the four components of rescaling overhead separately as a varying number of instances are interrupted [6].

- *Checkpoint* – Time to checkpoint the application state to Linux shared memory
- *Load balance* – Time to move objects away from interrupted instances and balance the load after the interrupted instance is replaced
- *Restart* – Time to restart the application and reconnect to all instances
- *Restore* – Time to restore the application state from Linux shared memory

Figure 4 shows the rescaling overhead for shrinking (when an interruption notice is detected) and for expanding (when the instance is replaced). We observe that in both cases, the largest contributor to the rescaling overhead is load balancing. This is due to the high cost of migrating objects over the network.

The restart time is low during shrinking because no new processes are started. During expansion, new processes are launched on the replacement instances which results in a higher restart time. The checkpoint and restore times are the smallest contributors to the total overhead, as these are fast in-memory operations. We found that in-memory checkpointing for data of size of 2GB takes only 0.046 seconds.

Figure 4c shows the total rescaling overhead of a shrink followed by an expand when a single instance is interrupted, for varying data sizes. For smaller problem sizes, we find the total overhead to be dominated by the restart time. As the problem size increases, the load balancing step dominates the total cost.

Effect of Capacity Drop. To measure the overhead caused by the intermittent drop in capacity due to spot interruptions, we measured the end-to-end application runtime with a varying number of instance interruptions. The total overhead is defined as the difference between the total runtime with interruptions and the runtime without interruptions. The overhead specifically due to the capacity drop is then calculated as the difference between the total overhead and the rescaling overhead.

Table 1 presents the total runtime, total overhead, and the overhead due to capacity drop for the 2D stencil application under varying numbers of instance interruptions. Figure 5 shows a comparison of the two sources of overhead.

We observe that rescaling overhead dominates the total cost when only a single instance is interrupted. However, as more instances are interrupted within a short duration, the overhead due to the drop in capacity increases sharply.

Instances Interrupted	Total time (s)	Total overhead (s)	Overhead due to capacity drop (s)
0	568.5	0	0
1	628.65	60.15	20.48
2	651.19	82.69	41.86
4	673.31	104.81	61.08
8	696.71	128.21	74.78

Table 1: End-to-end runtime and overhead due to drop in capacity with a varying number of interrupted instances for a 2D stencil application

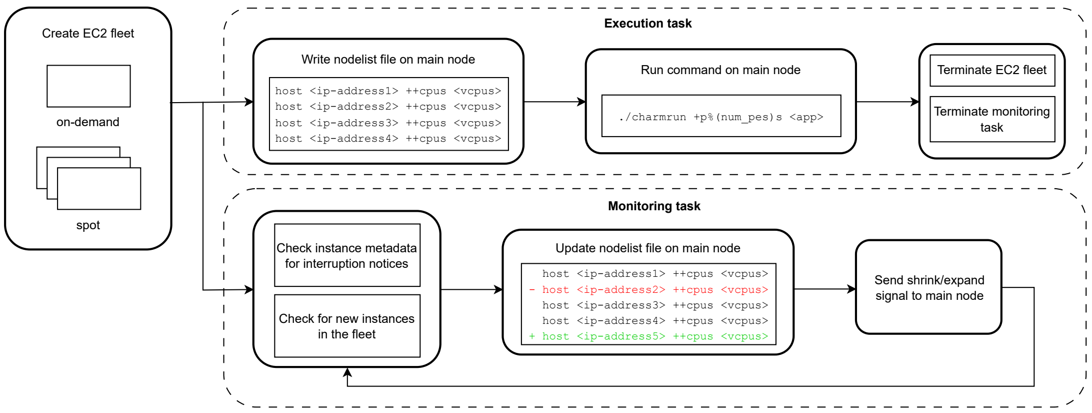

Create EC2 fleet
on-demand
spot
Execution task
Write nodelist file on main node
host <ip-address1> ++cpus <vcpus>
host <ip-address2> ++cpus <vcpus>
host <ip-address3> ++cpus <vcpus>
host <ip-address4> ++cpus <vcpus>
Run command on main node
./charmrun +p%(num_pes)s <app>
Terminate EC2 fleet
Terminate monitoring task
Monitoring task
Check instance metadata for interruption notices
Check for new instances in the fleet
Update nodelist file on main node
host <ip-address1> ++cpus <vcpus>
- host <ip-address2> ++cpus <vcpus>
host <ip-address3> ++cpus <vcpus>
host <ip-address4> ++cpus <vcpus>
+ host <ip-address5> ++cpus <vcpus>
Send shrink/expand signal to main node

Figure 2: The CharmCloudManager utility for running Charm++ jobs on EC2 spot instances

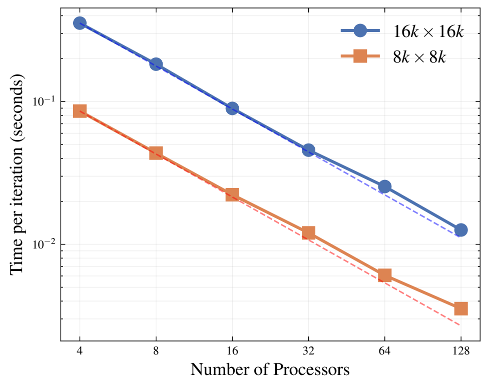


Figure 3: Strong scaling performance for a 2D stencil application for different grid sizes on c5.xlarge instances

4.3 Cost comparison

Table 2 shows the monetary cost comparison of running the 2D stencil application from the last section using 1 on-demand and 15 spot instances with the cost using all 16 on-demand instances. We see that with no interruptions, using spot instances can result in 59.78% cost savings over on-demand. We also see that even with a large number of instances being interrupted, the run using spot instances costs 50.71% lower than the all on-demand run.

Instances Interrupted	On-Demand Cost (USD)	Spot Cost (USD)	Cost savings
0	0.43	0.17	59.78%
1	0.43	0.19	55.52%
2	0.43	0.20	53.93%
4	0.43	0.20	52.36%
8	0.43	0.21	50.71%

Table 2: Cost comparison of running with all 16 on-demand instances and with 1 on-demand and 15 spot instances with varying number of instances interrupted

5 Related Work

Price Prediction. Several studies [8, 9, 13–15] have explored statistical modeling to predict future spot prices based on historical pricing data, with the goal of selecting optimal bidding strategies and minimizing the risk of interruptions. However, since most cloud providers have moved away from the bidding-based pricing model, these strategies are no longer applicable.

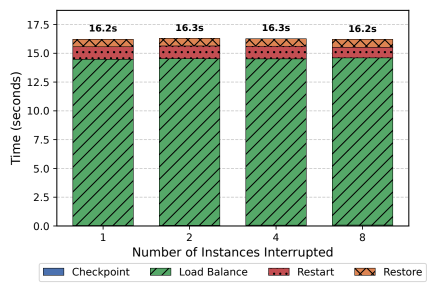
Migrating VMs to Minimize Cost. Hotspot [12] is a resource container that can self-migrate between VMs by copying all memory pages from the source to the destination VM. It employs a cost-based migration policy to move containers to a cheaper VM when the price of a spot VM increases. FarSpot [16] is an optimization framework for HPC applications in the cloud that uses ensemble-based learning methods to predict the long-term price of spot instances under the new pricing model. These price predictions are used to migrate tasks between spot instances to minimize execution costs.

Checkpointing and Replication. Marathe et al.[9] explored policies for determining when to checkpoint HPC applications to efficiently utilize spot instances and introduced redundancy to improve fault tolerance. Gong et al.[4] presented a cost optimization model to minimize the execution cost of MPI applications using checkpointing and replicated execution.

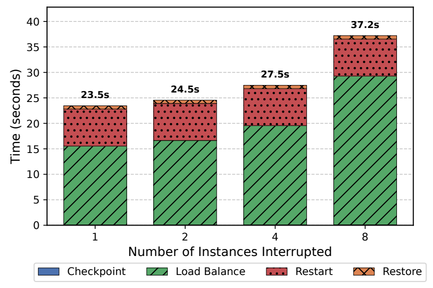
6 Conclusion and Future work

We presented CharmCloudManager, a Python tool for running Charm++ applications in the cloud using a mix of on-demand and spot instances. The tool can detect and gracefully handle spot instance interruptions without requiring a checkpoint to disk. We measured and classified the overhead caused by spot interruptions into two categories: the rescaling overhead of Charm++, and the overhead due to a drop in EC2 fleet capacity.

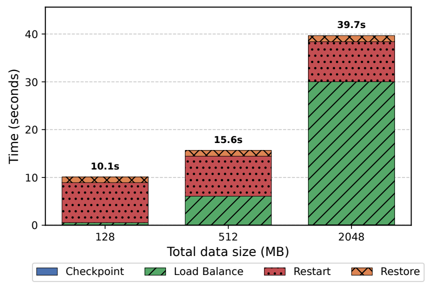
One avenue for future exploration is the use of the *capacity rebalancing* feature in EC2 fleets, which can proactively launch replacement instances before a spot instance is interrupted. This approach can help minimize the temporary drop in capacity caused by spot interruptions. However, a potential downside is that the frequency of rebalance recommendations may exceed the actual



(a) Overhead of shrink when instances are interrupted



(b) Overhead of expand when interrupted instances are replaced



(c) Total rescaling overhead with 1 instance interruption with varying data sizes

Figure 4: Overhead of rescaling with spot interruptions

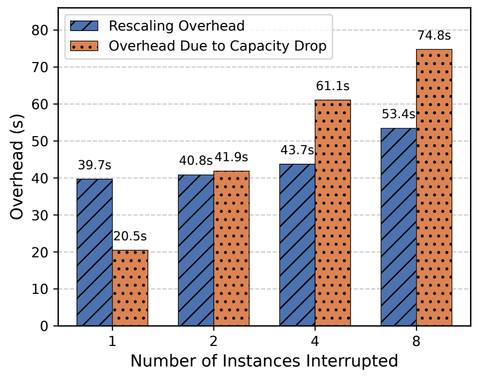


Figure 5: Comparison of sources of overhead during interruptions

interruption frequency, leading to increased overhead from more frequent rescaling of the Charm++ application.

We also demonstrated that running Charm++ applications with a single on-demand instance for the main node and using spot instances for compute nodes results in monetary savings of over 50%, even in the presence of interruptions, compared to using only on-demand instances.

In the future, we plan to extend the shrink/expand capability of Charm++ to GPUs. The additional cost of data movement between the device and the host can significantly increase the rescaling overhead on interruptions. Using GPUDirect RDMA with Elastic Fabric Adapter (EFA) will be critical in managing the rescaling overhead with GPUs.

7 Acknowledgments

This work is supported by the IBM-Illinois Discovery Accelerator Institute (IIDAI).

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice *(SC)*.

[2] Amazon Web Services. 2018. New Amazon EC2 Spot Pricing Model. https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/.

[3] Amazon Web Services. 2025. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/instance-advisor/.

[4] Yifan Gong, Bingsheng He, and Amelie Chi Zhou. 2015. Monetary cost optimizations for MPI-based HPC applications on Amazon clouds: checkpoints and replicated execution. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2807591.2807612

[5] Google Cloud. 2025. Google Cloud Spot VMs. https://cloud.google.com/solutions/spot-vms.

[6] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kale. 2014. Towards Realizing the Potential of Malleable Parallel Jobs. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC '14)*. Goa, India.

[7] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. 2008. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*.

[8] Mikhail Khodak, Liang Zheng, Andrew S. Lan, Carlee Joe-Wong, and Mung Chiang. 2018. Learning Cloud Dynamics to Optimize Spot Instance Bidding Strategies. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2762–2770. doi:10.1109/INFOCOM.2018.8486291

[9] Aniruddha Marathe, Rachel Harris, David K. Lowenthal, Bronis R. de Supinski, Barry Rountree, and Martin Schulz. 2016. Exploiting Redundancy and Application Scalability for Cost-Effective, Time-Constrained Execution of HPC Applications on Amazon EC2. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2016), 2574–2588. doi:10.1109/TPDS.2015.2508457

[10] Eric Mikida, Nikhil Jain, Elsa Gonsiorowski, Peter D. Barnes, Jr., David Jefferson, Christopher D. Carothers, and Laxmikant V. Kale. 2016. Towards PDES in a Message-Driven Paradigm: A Preliminary Case Study Using Charm++. In *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS) (SIGSIM PADS '16 (to appear))*. ACM.

[11] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. 2005. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 26, 16 (2005), 1781–1802. doi:10.1002/jcc.20289 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.20289

[12] Supreeth Shastri and David Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 493–505. doi:10.1145/3127479.3132017

[13] Jiayi Song and Roch Guerin. 2017. Pricing and bidding strategies for cloud computing spot instances. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 647–653. doi:10.1109/INFCOMW.2017.8116453

[14] Cheng Wang, Qianlin Liang, and Bhuvan Urgaonkar. 2017. An Empirical Analysis of Amazon EC2 Spot Instance Features Affecting Cost-effective Resource Procurement. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/3030207.3030210

[15] Qi Zhang, Eren Gürses, Raouf Boutaba, and Jin Xiao. 2011. Dynamic resource allocation for spot markets in clouds. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (Boston, MA) *(Hot-ICE'11)*. USENIX Association, USA, 1.

[16] Amelie Chi Zhou, Jianming Lao, Zhoubin Ke, Yi Wang, and Rui Mao. 2022. FarSpot: Optimizing Monetary Cost for HPC Applications in the Cloud Spot Market. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 2955–2967. doi:10.1109/TPDS.2021.3134644