Auto-tuning Performance of MPI Parallel Programs Using Resource Management in Container-based Virtual Cloud

Hongyi Ma[1], Liqiang Wang[2], Byung Chul Tak[3], Long Wang[3], and Chunqiang Tang[4]

[1]University of Wyoming. {hma3}@uwyo.edu
[2]University of Central Florida. {lwang}@cs.ucf.edu
[3]IBM Thomas J. Watson Research Center. {btak, wanglo}@us.ibm.com
[4]Facebook, Inc. tang@fb.com

***Abstract*—Load imbalance problem is one of the major obstacles to achieving optimal performance of High Performance Computing applications. The approach of trying to distribute the problem pieces to each node with the hope of balancing execution time has limits since the performance depends not only on data size but also on many other dynamic factors. This paper describes an approach that uses adaptive resource management enabled by the container-based virtualization to solve the load imbalance problem of MPI programs running in the cloud. Our techniques dynamically adjust CPU resource allocation to MPI processes running as container instances according to the current program execution state and system resource status. The resource allocation among MPI processes is adjusted in two ways: the intra-host level, which dynamically adjusts resources within a host; and the inter-host level, which migrates containers together with MPI processes from one host to another host. We have implemented and evaluated our approach on Amazon EC2 platform using real-world scientific benchmarks and applications, which demonstrates that the performance can be improved up to 31% (with an average of 15%) when compared with the baseline.**

I. Introduction

With advances of High-Performance Computing (HPC) in scientific computing, the idea of the cloud as a platform for HPC applications draws large attentions since it has high potential to provide great cost benefits and convenience for researchers and engineers in implementing and running HPC program tasks. Users can simply rent computing resources in the form of a cluster of instances from the cloud providers such as Amazon EC2 [1] , IBM Bluemix [2] and Microsoft Azure [7] on demand with reasonable cost instead of owning and maintaining physical clusters [19] [14].

However, there are still several open research problems to be solved in order for the cloud to see wider adoption by the HPC community such as concerns over suboptimal performances, and concurrency errors [15] [16]. In the performance aspect, one of the major contributor of the performance degradation is known to be the load imbalance problem [21]. Load imbalance causes the nodes running in parallel to finish unevenly in time and the overall performance becomes the performance of the slowest node. In the cloud, traditional way of distributing equal-sized problem instances to each computing node may not always be an effective way of handling the load imbalance because of the dynamic and heterogeneous nature of the cloud environments such as hardware and virtual resource heterogeneity, sharing of storage I/O, network connection and varying computation capacities on different nodes. Load imbalance could be also caused by the interference of other unknown applications in the cloud.

Contrary to the traditional methods of focusing on balancing computing loads, such as [21], [12], we emphasize on on-the-fly resource management in response to the observed progress of each processing element using operating system level virtualization. There are two folds of benefits in using the approach of automatic online resource adjustment. Firstly, the approach allows us to transparently handle unpredictable performance issues as we take execution time into account as metrics in resource management. Secondly, dynamically adjusting compute resources in response to load imbalance avoids manual code modification or code translation.

Container-based virtualization is a lightweight virtualization technique for resource isolation and adjustment. Each container performs and executes exactly like a stand-alone server or an independent operating system, which can reboot independently, have root access to configure, and install softwares. A container may have independent IP address, memory, processes, files system, and system libraries. We

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.76

can dynamically adjust compute resources, such as CPU, memory, and I/O, allocated for each container within a host. Compare to hypervisor-based virtualization, which often has an overhead as high as 40% the overhead of container-based virtualization is as low as 2%, which is near native performance. In this work we have used OpenVZ container [20] because it supports live migration of processes and worked for public cloud platforms such as Amazon EC2. Other container tools, such as LXC [3] and Docker [8], did not support live migrations very well yet.

Key idea of our approach is to periodically adjust CPU resource allocations based on the measured information about the MPI process execution time in containers and the compute capacity of hosts. Main benefit of our auto-tuning approach is the ability to utilize the capabilities provided by the virtualization techniques to automatically adjust system resource allocation. Our evaluation shows that it achieves similar or better performance compared to these existing techniques. We conducted experiments on Amazon EC2 using NAS, UTS, stencil2D, and Jacob1D benchmarks and our technique could reduce execution time by an average of 15% and up to 31% compared to the original execution.

II. Background and Related Work

Developers designed specific load balancing algorithms according to the characteristics of data distribution or parallel computation to improve the performance of MPI applications [6]. These types of load balance algorithms usually achieve good performance, but are tightly coupled with specific applications or programming models. Various application-independent load balancing approaches have been proposed in response to these shortcomings. CrayPat [10] has a cost model that selects the best load balance algorithm for particular applications based on their load balance metrics. Charm++ [4] is a message-driven object-oriented parallel programming language that provides a high-level abstraction of parallel programs. Dynamic load balancing can be achieved by decomposing applications into Charm++ objects, which can be mapped to and migrated across available processors in Charm++ runtime. Adaptive MPI (AMPI) [13] is an implementation of MPI on top of Charm++ which provides virtual processors for applications and allows MPI processes to be used in Charm++ runtime. Based on Charm++ and AMPI, Gupta et al. proposes a dynamic load balancing approach for tightly coupled iterative HPC applications in the cloud through periodic refinement of task distribution [12]. The approach in [12] automatically applies a load balancing strategy for iterative computation. The workload is calculated based on the CPU time consumed by the tasks on VCPUs. Corbalan et al. proposes a processor balancing approach for hybrid OpenMP and MPI applications that can dynamically measure the percentage of computational load imbalance on different processes, then assign more CPU cores to the slow processes [9]. But this approach only focuses on resource adjustment for OpenMP threads within a single machine.

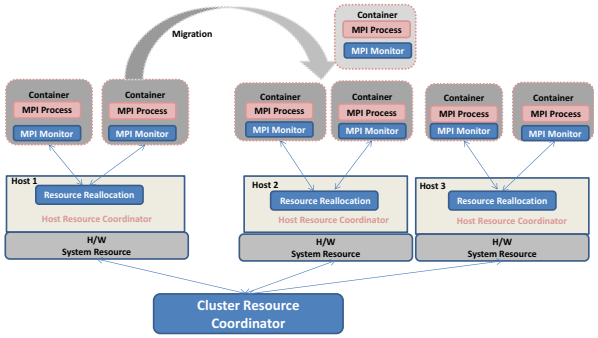

Figure 1. Architecture of our dynamic resource adjustment.

The main idea of our performance optimization for HPC applications in the cloud is to dynamically adjust CPU resource allocations based on the measured runtime information, *i.e.*, MPI process execution time in containers and the compute capacity of hosts. The adjustment is based on container virtualization on two levels: the host level and the cluster level. At the host level, our approach moves CPU resource allocation from under-utilized containers to the overloaded containers. At the cluster level, our approach further monitors resource utilization of physical hosts, then migrates containers from overloaded ones to less-utilized ones. This further reduces load imbalance across the entire cluster by effectively reassigning overloaded jobs.

III. Intra-host Auto-tuning Resource Reallocation

Our dynamic resource adjustment mechanism consists of three major components: *MPI monitor*, *resource coordinator*, and *cluster resource coordinator*, as illustrated in Figure 1.

MPI monitors are constructed as wrappers of the MPI library in order to collect execution information. At each host, a host resource coordinator runs an

Algorithm 1 Dynamic CPU resource adjustment within a host

```
1: % k is the current MPI process ID, and K is the total number of MPI processes.%
2: % mpiRankList is a list of MPI processes whose interval execution times have been received.%
3: % cpuUnits[k] is the CPU share units to be assigned to MPI process k.%
4: % exeT[k] is the accumulated interval time reported by MPI process k in the current sliding window. %
5: % slidingWin[k] is a queue where the Max number of synchronized MPI routine invocation are kept for MPI process k.%
6:
7: void DynamicCPUAdjustment(){
8: while (Upon receiving a message {interval, k} from an MPI routine call) do
9:     mpiRankList.put(k);
10:    slidingWin[k].enqueue(interval);
11:    if (slidingWin[k].size() >= Max) then
12:        slidingWin[k].dequeue();
13:    end if
14:    exeT[k] = sum_{i=0}^{Max-1} slidingWin[k][i]);
15:    cpuLimits[k] = 100%
16:    if (mpiRankList.size()! = K) then
17:        cpuUnits[k] = waitValue;
18:    else
19:        % reallocate resource for the next phase %
20:        for k = 0 to K − 1 do
21:            cpuUnit[k] = exeT[k]/ sum_{i=0}^{K-1} exeT[i];
22:        end for
23:        mpiRankList.clear();
24:    end if
25: end while
```

auto-tuning algorithm that dynamically adjusts CPU allocations among the hosted containers (*i.e.*, MPI processes) to reduce load imbalances. *Resource coordinator* takes CPU resources from under-utilized containers and gives them to more heavily loaded ones. In the context of this paper, each container runs one MPI process. The *cluster resource coordinator* aims to eliminate load imbalances across physical hosts by migrating containers on the busy hosts to these with surplus resource capacity.

A. Auto-Tuning at Host Level

The resource management algorithm within a host, *i.e.*, Algorithm 1, is executed whenever an interval report message is received by the host resource coordinator. OpenVZ supports CPU resource management for containers at the host level. It controls the allocation of CPU resources according to the ratio of CPU unit shares of all containing processes. Once time intervals of all processes from MPI monitors have been received, we adjust CPU unit shares among containers according to the ratio of execution time of all containers. If the execution time of some MPI processes have not been received, which indicates that the MPI processes are still running and have not reached the current rendezvous point, we use the elapsed wall clock time as the current execution time for adjusting CPU shares.

In Algorithm 1, for the containers that have already finished their work in the current phase, we release most of their CPU occupation but keep them in busy polling by assigning to them a small value of CPU units *waitValue*. At the beginning of the next phase, for each MPI process, its accumulated execution times within the recent a few MPI invocations (denoted by a sliding window, *i.e.*, $slidingWin$) is used to compute the CPU share units for the next phase. Specifically, due to the design of MPI collectives routines, even MPI processes are waiting for the others, they would still exchange the buffer information in order to synchronize. So, we consider these communication time caused by the collectives routines also belongs to the computation time.

Each MPI process is assigned with the ratio of its accumulated execution time against the total accumulated execution time of all the MPI processes, *i.e.*, *cpuUnits*. The larger the *cpuUnits* is, the more CPU time this container will get. One serial MPI process cannot utilize two or more physical CPU cores. To utilize the CPU cores more efficiently and adjust CPU allocations dynamically, we usually affiliate multiple containers to each physical core. In the configuration of the container, the parameter *cpuLimit* indicates the CPU upper-bound percentage that a container is allowed to use. The default CPU limit is 100 % in our system, which indicates that a serial MPI process can occupy at most 100% one physical CPU core in a physical host machine.

IV. Inter-host Auto-tuning Resource Reallocation

A. Resource Imbalance Measurement Metrics

To determine the host-level capacities and demands, we require a metric to evaluate the degree of runtime imbalance in running MPI processes. This metric allows us to both improve performance and balance resources, which is then used to make appropriate decisions on resource allocation and container migration. Similar to the traditional load imbalance percentage measurement [17], we define the load imbalance indicator as ψ.

$$\psi_p(i) = |interval_p(i) - ALM(i)| \qquad (1)$$

$$ALM(i) = \frac{1}{P}\sum_{p=0}^{P-1} interval_p(i) \quad (2)$$

$$LIP(i) = max(\frac{\psi_p(i)}{interval_p(i), p \in [0, P-1]}) \quad (3)$$

where $interval_p(i)$ denotes the execution time of interval i, p is the process *ID*, and P is the number of processes. *ALM(i)* is the average load metric, *i.e.*, average execution time of all MPI processes, under the time interval i, $\psi_p(i)$ is the difference between $ALM(i)$ and the execution time in process p, and $LIP(i)$ (Load Imbalance Percentage) indicates the load imbalance degree. The less LIP is, the more balance the current system obtains. When LIP is 0, then the current system is in a perfect balanced state. LIP provides an explicit way to evaluate and to show how load-imbalanced the application is.

Table I
TABLE OF NOTATION IN MIGRATION PLAN

container[i]	container *i* in cluster system.
P	number of hosts in cluster system.
Host[i]	host *i* in cluster system.
$M_i[t]$	number of containers running inside of Host[i] at time *t*.
capacity[k]	load capacity of host *k*.
$vmload_{i,k}[t]$	container *i* load at host k at time *t*.
$load_k[t]$	load utilization at host *k* at time *t*.
$x_{i,j} = 1 \vee 0$	1 means container *j* is located in host *i*; 0 means not.
$u_i[t]$	resource utilization in host *i* at time *t*.
thresU[t]	average utilization for all hosts at time *t*.
NumECU[k]	number of ECPUs in host *k*.
$Mig[t] = 1 \vee 0$	1 for migration at time *t*, and 0 means not.
rest[k][t]	the rest load utilization in host *k* at time *t* .
Interfer[k][t]	load utilization of interference in host *k* at time *t* .

B. Container Migration Strategy

The dynamic container migration problem can be abstracted as a variant of on-the-fly multiple-knapsack problem (MKP) [22], which is an NP hard problem. There is no standard or classical solver for the problem. We designed an approximate heuristic solution as shown in Algorithm 2.

In order to formulate our migration plan decision algorithm, we define expressions in Table I. In general, the capacity of a host is constrained by runtime conditions and hardware.

$$(1) capacity[k] = NumECU[k]$$
$$(2) vmload_{k,i}[t] = \frac{exeT[k,i] \times NumECUs[k]}{max(exeT[])}$$
$$(3) u_i[t] = \sum_{k=1}^{M_i} x_{i,k}[t] \times \frac{vmload_{k,i}[t]}{capacity[i]}$$
$$(4) averageUsage[t] = \sum_{i=1}^{P} u_i[t]/P$$
$$(5) \sum_{k=1}^{M_k} x_{i,k}[t] \times vmload_{k,i}[t] \leq capacity[i]$$
$$(6) load_k[t] = \sum_{j=0}^{M_k-1} vmload_{j,k}[t]$$
$$(7) rest_k[t] = capacity[k] - load_k[t]$$
$$(8) Interfer[k][t] = load_k[t] \times \frac{InterferUsage[k]}{totalMPIusage[k]}$$

The above expressions are used to represent load and resource utilization status in containers and hosts at time *t*. Specifically, (1) denotes the capacity of host *k*, since different hosts may have different computing power. Thus, it is necessary to normalize the accumulated execution time according to the standard EC2 Computing Units (ECU). Expression (2) is to calculate the load utilization for each container. Equation (3) is to illustrate how to obtain the resource utilization at each host. Equation (4) is to obtain the average resource utilization in the whole cluster. Inequality (5) denotes that the sum of loads of containers on a host cannot exceed the capacity of the host. Equation (6) denotes the load of all containers in host *k* at time *t*. Expression (7) represents the rest load utilization in host k. In (8), let *Interfer[k]* denote the resource consumed by interference applications in host *k*, which can be obtained from host level monitoring using the ratio of *Interferusage[k]* (CPU usage of interference application in host k) to *totalMPIusage[k]* (total CPU usage of MPI applications in host k) then multiplied by the load in host *k*.

$$(1) \sum_{k=1}^{M_k} x_{i,k}[t+1] \times vmload_{i,k}[t+1] + Interfer[i][t] \leq capacity[i]$$
$$(2) \sum_{i=1}^{P} x_{i,j}[t+1] \leq 1$$
$$(3) capacity[i] = capacity[i] - norm(\lambda(x_{i,j}[t+1].dump)) \times Mig[t]$$
$$(4) |\sum_{i=1}^{P} u_i[t+1] - thresU[t+1]| \leq |\sum_{i=1}^{P} u_i[t] - thresU[t]|$$
$$(5) \sum_{i=1}^{P}(x_{i,j}[t+1] - x_{i,j}[t]) == 0$$
$$(6) \sum_{i=1}^{P} |x_{i,j}[t+1] - x_{i,j}[t]| == 2$$

The migration solution includes 1) the source hosts; 2) which containers to be moved; and 3) the destination hosts, which are subject to the above conditions. (1) All containers' loads in host *i* cannot exceed to *capacity(i)* of Host *i*. Variable t indicates

the time before migration and $t+1$ indicates the time after migration. (2) One container can be deployed on only one host. (3) After each migration, the capacity of host is updated by subtracting the overhead. The migration overhead of a container, denoted by λ, is determined by all memory dumping size. (4) represents that the difference between u_i in all hosts should be smaller after migration. (5) and (6) are conditions to filter the plans of non-migration and only the migration plans are left. For example, if we have solutions to satisfy with these conditions on container *j*, we can migrate container *j* from host i_1 to host i_2 if $x_{i_1,j}[t] = 1 \wedge x_{i_2,j}[t+1] = 1 \wedge i_1 \neq i_2$.

C. Heuristic Migration Algorithm

Algorithm 2 utilizes a heuristic method to migrate a container in the migration list to a destination host. If host load utilization is less than $averageUsage$, then we put this host into the potential destination list that will receive migrated containers. If host load utilization is much more than $averageUsage$, which means this host has more load pressure, then we should move out some containers until the host load reaches the average level. These containers will be put into the potential migration list as candidates. The corresponding host will be also marked as *sourceHost*.

Algorithm 2 also determines the migration destinations of containers in the migration list. First, we pick a host in the destination list and a container from the migration list. If the rest capacity of the destination host is still greater than the average level after deducting the load of the migrated container and the migration overhead, we put it into the migration plan and continue to check other containers. We normalize the migration overhead to the CPU load using function $norm$, which is *(λ(y.dump) /max(exeT[])) * NumECUs[k]*. Thus, a migration that is initially considered too expensive to fulfill will become more feasible as the accumulated load imbalance could exceed the overhead of migration during the program's execution. Such a migration can help improve the performance if the load imbalance pattern remains similar in the future. In our experiment on EC2, the overhead is linear to the memory dumping speed, *i.e.*, around one second per 10MB. The memory size of a container is calculated by multiplying the OpenVZ "oomguarpages number" by 4KB per page. If the load imbalance indicator ψ is greater than the migration overhead of current container (*i.e.*, $\lambda(y.dump)$), there could be a potential performance improvement, hence container

Algorithm 2 Containers Migration Plan

```
1: %k ∈ [0, P) is the current host, P is the total number of hosts in cluster.%
2: void migrationCandidates(migrationList, destinationList) {
3: for i = 0 to P − 1 do
4:     %sort containers on host[i] according to their loads in an ascending order%
5:     tmp[] = sort(vmload[] ∈ host[i])
6:     if (u_i ≤ averageUsage) then
7:         %host i is relatively idle and could be a potential destination%
8:         destinationList.put(host i)
9:         continue
10:    end if
11:    if (u_i ≥ averageUsage) then
12:        %Continuously look for containers with the smallest load for migration in the current host until the rest capacity is reduced to the average level.%
13:        for j = 0 to M_i − 1 ∈ host[i] do
14:            %find a container to be migrated%
15:            k = getID(tmp[j])
16:            migrationList.put(container k)
17:            k.sourceHost = i
18:        end for
19:    end if
20: end for
21: % for each container, find which host to migrate. %
22: for each container y in (migrationList) do
23:    for each host k in (destinationList) do
24:        rest[k] = capacity[k] − load[k]
25:        tmpRest = rest[k] - vmload[y] - norm(λ(y.dump))
26:        avgRestCapacity = Σ_{i=0}^{P−1} rest[i] × 1/P
27:        % Recall ψ is load imbalance indicator; λ(y.dump) < ψ_k means that there is potential performance improvement after migration.%
28:        if ((tmpRest > avgRestCapacity) ∧ (λ(y.dump) < ψ_k)) then
29:            Migrate container y to host k
30:            rest[k] = tmpRest
31:            break
32:        end if
33:    end for
34: end for
35: }
```

y is put into the migration plan with the destination as k. We continue the `for` loop by checking containers one by one until the utilization usage of the current host is less than the average value. Then we start to check the next host in the destination list. If there are still containers in the temporary *migrationList* that have not found a destination host when this procedure has finished, then these containers will remain on their original hosts to avoid unnecessary migrations. We notice that migration may change the previous network topology that user defined, so we

only consider the total overhead instead of specified extra cost caused by network topology changes.

V. Experiments

A. System Setup

We set up a virtual cloud on Amazon EC2 using OpenVZ to manipulate CPU resource adjustment and container migration. This testbed has inherent heterogeneity as it consists of two types of physical hosts: VCPUs on C3 instances with 2.8 GHz Intel Xeon E5-2680v2 processors, and VCPUs on M3 instances with 2.5GHz Intel Xeon E5-2670 processors. We use 8 C3 instances and 8 M3 instances total. In our experiment setup, each container executes only one MPI process. The numbers, *e.g.*, 8, 16, 32, 64, 128, in the figures of this section represent the number of containers (as well as MPI processes) in our virtual clusters.

We refer to the first processor type as *fast* and the second one as *slow*. By default, we assume that each host/instance creates 8 containers to share its CPU cores. We conduct experiments on two different types of virtual clusters. The first type of virtual cluster is homogeneous and built on identical *fast* machines. The second type is heterogeneous and built on a mix of *fast* and *slow* machines. Also, we consider the effects caused by the application interference during runtime of MPI application. Our experiments are performanced on NAS Parallel Benchmark [18], Unbalanced Tree Search [11], Stencil2D [12], and Jacobi1D [4] open source benchmarks.

B. Case Study on Unbalanced Tree Search

Dinan et al. describes a parallel benchmark, called Unbalanced Tree Search (UTS) [11]. Our case study on UTS shows that it is possible to obtain good performance in solving an unbalanced tree search problem using our resource reallocation techniques. The imbalance issue is caused by the visiting depth of the tree. The UTS benchmark counts the number of nodes in a generated tree, as the depth and size of the subtrees can cause imbalances while searching. Highly unbalanced trees pose significant challenges for parallel traversal because the workload required for different subtrees may vary dramatically.

According to our experiments, our dynamic load balancing method is effective. Figure 2 presents the execution time in homogeneous and heterogeneous systems, respectively. The performance of the intra-host level load balance algorithm is shown in Figure 2. The average execution time reductions of the heterogeneous and homogeneous systems are 12% and 8%, respectively. Figure 3 shows the execution times for scenarios without auto-tuning algorithm ("NoLB"), using only the migration algorithm at cluster level ("Only_Migration"), using a combination of migration across hosts and dynamic adjustment within hosts ("Migration_Dynamic"). The "Migration" case reduces execution time in 16 containers, 32 containers, 64 containers, and 128 containers by an average of around 10%. The best performance improvement we obtained in the experiments is on the testing of UTS benchmark with one million nodes, which is 31% improvement over the baseline using 32 containers total in the cluster.

C. Case Study on Relatively Load-Balanced Benchmarks

We have also tested algorithms on real-world applications using NAS Parallel Benchmark, *i.e.*, LU, MG, SP, EP, BT, and IS with Class B size. In Figures 4, for the experiment on IS, we did not find any performance improvements using our load balancing approach. This is because the IS benchmark contains only a few MPI routines which are already load balanced. Similarly, the MG benchmark has little load imbalance, hence, few load balance adjustments are applied. As a result, there was not much improvement in these types of benchmarks. However, when we test other larger benchmarks with more load imbalance and longer execution times, the performance improvement is more obvious. Our algorithm is able to reduce the execution time by 22% in the best scenario. The average performance improvement is 13% on these relatively load-balanced benchmarks.

D. Comparing with State-of-the-Art Techniques of MPI Performance Optimization

Now we compare the performance of our solution with state-of-the-art techniques of MPI performance optimization. Figure 5 shows the load balance improvements on two benchmarks, Stencil 2D and Jacobi1D. For Stencil2D, problem size is 5000×5000. For Jacobi1D, the number of steps is set to 10000 iterations while running.

Figure 5 compares the performance improvements by our approach (auto-tuning performance algorithm, or AP) and Adaptive MPI (AMPI), Charm++, respectively. The default load balancing mode in Charm++ uses HybridLB. For AMPI, it can automatically support Charm++ features plus migration of processes. We use the benchmarks in AMPI and Charm++ package for our experiments in Figure

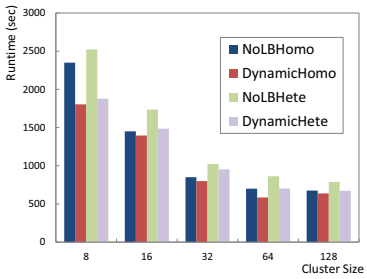

Figure 2. Experiment of unbalanced tree search on homogeneous system and heterogeneous

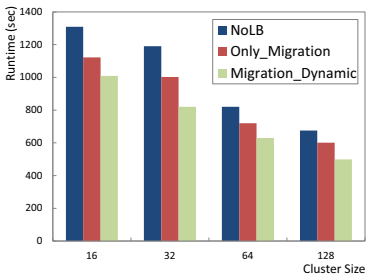

Figure 3. Migration using different load balance adjustments in different number of containers.

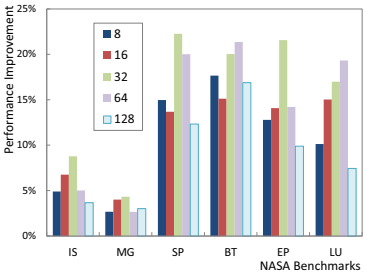

Figure 4. Experiment on NAS parallel computing benchmarks.

5. In the experiments, our approach, AMPI, and Charm++ achieve an average of 15%, 14% and 13% performance improvement, respectively.

E. Performance Analysis

This section discusses several performance issues, including scalability, load imbalance ratio, and scalability. We find that the overhead increases with the total number of containers on hosts, because more containers incur more wrapper communication and resource adjustment. More containers than the number of physical cores in a host would cause resource overcommitment, which increases resource utilization but potentially decreases the performance of applications. We are targeting MPI program performance rather than overall resource utilization.

Figure 6 shows the load imbalance percentage for all benchmarks. One test is called *NoLB*, which is the load imbalance percentage of original benchmark. The other is named as *LB*, which is the load imbalance percentage after applying our resource adjustment algorithm. In the smaller benchmarks, such as IS and MG with short execution time, the load imbalance percentage is low because all processes finish the jobs in a short time.

F. Interference

Interference from other applications can have an unpredictable impact on HPC programs in a cloud environments, where physical resources are shared. Typically, an MPI process will exclusively occupy a whole container. Interference generally occurs when there are applications running on co-located containers on the same physical host. In order to emulate a real-world computing environment, we run an interference program called `ParMETIS` [5] , which implements algorithms on graphs and meshes, while executing the UTS benchmark [11] in our own containers.

Figure 7 shows the performance improvement under interference for homogeneous and heterogeneous systems, respectively. Our algorithm can improve performance in an average of 12% in ("Heter") heterogenous system *without* interference, and an average of 8% in homogeneous system *without* interference ("Homo"). The underlying reason for this phenomenon is that there is more load imbalance in heterogeneous system. The interference application will influence the performance improvement of our approach. The load imbalance is potentially increased by interference because it takes away extra resources from MPI applications. Our algorithm can improve performance by an average of 16% for heterogenous system ("Hete_ITFN") and 10% for ("Homo_ITFN") homogeneous system *with* the presence of the interference application.

VI. Conclusion and Future Work

In this paper, we have presented both intra-host and inter-host resource management techniques for improving performance of MPI applications in container based cluster, which are shown to be effective in improving MPI program performance. We have demonstrated that our technique works well with various real-world benchmarks on homogeneous and heterogeneous virtual cloud environments with or without interference. Experimental results show that our techniques can reduce execution time by an average of 15% compared to the original execution without resource management. The average performance improvement is higher in heterogeneous environments than in homogeneous environments.

In the future, it is possible to leverage our approach for the other more general applications using Docker containers. In addition, we can proactively conduct resource adjustment of MPI applications in advance based on the estimation of computation workload.

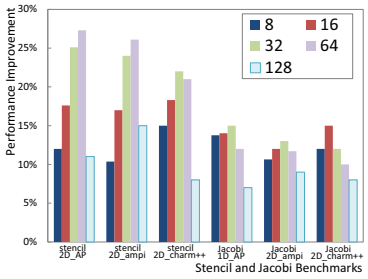

Figure 5. Summary of performance improvement using Stencil2D and Jacobi1D.

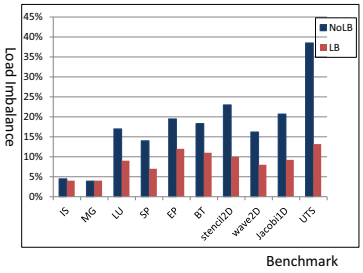

Figure 6. Load imbalance percentage (LIP).

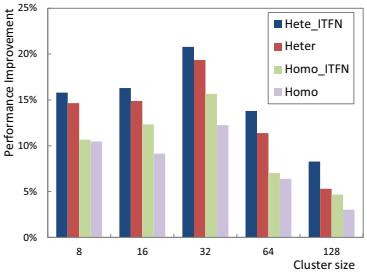

Figure 7. Execution time reduction percentage in heterogeneous and homogeneous systems with interference application.

VII. Acknowledgment

This work was supported in part by NSF-CAREER-1622292.

References

[1] Amazon EC2. http://http://aws.amazon.com/ec2/.

[2] Ibm Bluemix. http://www.ibm.com/cloud-computing/bluemix/.

[3] Linux containner. http://www.lxc.org/.

[4] Parallel language/paradigms: Charm++ parallel objects. http://ppl.cs.illinois.edu/research/charm.

[5] Parmetis. http://en.wikipedia.org/wiki/Xen.

[6] Vampir performance optimization. http://www.vampir.eu/.

[7] Windows Azure. https://azure.microsoft.com/en-us/.

[8] C. Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 2015.

[9] J. Corbalan, A. Duran, and J. Labarta. Dynamic load balancing of mpi+openmp applications. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, Washington, DC, USA. IEEE Computer Society.

[10] L. DeRose, B. Homer, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, pages 150–159, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA. ACM.

[12] A. Gupta, O. Sarood, L. V. Kal, and D. S. Milojicic. Improving hpc application performance in cloud through dynamic load balancing. In *CCGRID, IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 402–409. IEEE Computer Society, 2013.

[13] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance evaluation of adaptive mpi. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, New York, NY, USA, 2006. ACM.

[14] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, Washington, DC, USA, 2013.

[15] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. Symbolic analysis of concurrency errors in openmp programs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 510–516, Washington, DC, USA, 2013. IEEE Computer Society.

[16] H. Ma, L. Wang, and K. Krishnamoorthy. Detecting thread-safety violations in hybrid openmp/mpi programs. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*.

[17] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, New York, NY, USA. ACM.

[18] Nasa advanced supercomputing division. http://www.nas.nasa.gov/publications/npb.html/.

[19] V. Subramanian, H. Ma, L. Wang, E.-J. Lee, and P. Chen. Rapid 3d seismic source inversion using windows azure and amazon ec2. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, Washington, DC, USA, 2011. IEEE Computer Society.

[20] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Charlottesville, VA, USA, 2000.

[21] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Pract. Exper.*, 3(5), Oct. 1991.

[22] G. J. Woeginger and Z. Yu. On the equal-subset-sum problem. *Inf. Process. Lett.*, 42(6):299–302, July 1992.