NO^2: Speeding up Parallel Processing of Massive Compute-Intensive Tasks

Yongwei Wu, *Member*, *IEEE*, Weichao Guo, Jinglei Ren, Xun Zhao, and Weimin Zheng, *Member*, *IEEE*

Abstract—Large-scale computing frameworks, either tenanted on the cloud or deployed in the high-end local cluster, have become an indispensable software infrastructure to support numerous enterprise and scientific applications. Tasks executed on these frameworks are generally classified into data-intensive and compute-intensive ones. However, most existing frameworks, led by MapReduce, are mainly suitable for data-intensive tasks. Their task schedulers assume that the proportion of data I/O reflects the task progress and state. Unfortunately, this assumption does not apply to most compute-intensive tasks. Due to biased estimation of task progress, traditional frameworks cannot timely cut off outliers and therefore largely prolong execution time when performing compute-intensive tasks. We propose a new framework designed for compute-intensive tasks. By using instrumentation and automatic instrument point selector, our framework estimates the compute-intensive task progress without resorting to data I/O. We employ a clustering method to identify outliers at runtime and perform speculative execution/aborting, speeding up task execution by up to 25%. Moreover, our improvement to bare instrumentation limits overhead within 0.1%, and the aborting-based execution only introduces 10% more average CPU usage. Low overhead and resource consumption make our framework practically usable in the production environment.

Index Terms—Distributed system, parallel processing, compute-intensive, resource management

1 Introduction

Cloud computing provides abundant computational resources to users. In a typical datacenter, over thousands of servers handle massive parallel tasks to support users' applications (e.g., information retrieval, analytical or scientific computation). Meanwhile, traditional clusters also reach much larger scale, commonly with thousands of CPU cores nowadays. On such a large-scale infrastructure, an efficient and robust processing framework is critical to users. This framework provides a viable way for users to organize and utilize the underlying tremendous computational resource pool. Specifically, it facilitates users' development/deployment of applications and provides guarantee on the quality of service (QoS). A set of easy-to-use but controlled interfaces are exported to users to code programs, and the framework takes charge of scheduling these programs to run on different server nodes in an efficient manner.

In this paper, we focus on the frameworks that achieve parallelism via SPMD (single program, multiple data) and support tasks with minimal mutual communication, rather than the message-passing tasks (e.g., MPI programs). A job is the unit that a user submits to the framework, and tasks refer to what a job is split into when executed on a large multi-node infrastructure. The tasks of different jobs can be generally classified into two types: the data-intensive and the compute-intensive. A data-intensive job/task issues a large amount of data I/O, so data I/O usually dominates and spreads along the whole task. Meanwhile, a compute-intensive job/task performs only limited data I/O but spends most of time on computation.

Our observation is that most existing computing frameworks are suitable for data-intensive tasks but lack consideration for compute-intensive ones. The main reason behind involves the important scheduling component in these frameworks. Many previous investigations [1], [2] have shown that *outliers* constitute a notorious performance killer in massive task processing. Outliers progress much more slowly than peer tasks and therefore dramatically delay the completion time of the whole job. Many common and unexpected factors may lead to outliers, like uneven task assignment, hardware/software problems, and network congestion. Thus outliers are pervasive in either a cloud or a cluster. In order to address this issue, computing frameworks heavily rely on a smart scheduler, which is capable of identifying outliers, timely aborting them, and re-executing replica tasks on other healthy nodes.

Although there have emerged various large-scale computing frameworks, among which MapReduce is one of the most successful and representative frameworks, they lack a mechanism to estimate the progress of compute-intensive tasks and are consequently incapable of action to outliers. For example, MapReduce assigns a specific amount of data to a task and naturally uses the processed data amount or proportion as the indicator of the task's progress. This approach embodies an assumption that the task progress is approximately linear with data I/O. Furthermore, main subsequent improvements on the scheduler of MapReduce also hold this assumption. However, it is not the case in compute-intensive tasks. When

• *The authors are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China and with Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China.*
E-mail: {wuyw, zwm-dcs}@tsinghua.edu.cn, {gwc11, renjl10, zhaoxun09}@mails.tsinghua.edu.cn.

Manuscript received 10 Feb. 2013; revised 19 May 2013; accepted 29 May 2013. Date of publication 13 June 2013; date of current version 12 Sep. 2014. Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.132

performing this kind of tasks, data I/O activities mostly take a small portion of time and are skewed within a small interval. Therefore the traditional mechanism of progress estimation and outlier identification becomes problematic if directly applied to compute-intensive tasks.

Considering the significance and pervasiveness of compute-intensive tasks (e.g., analytical and scientific computing), we propose NO^2, a framework designed for processing massive compute-intensive tasks in parallel. Our framework is based on ProActive and, more importantly, employs a scheduler that relies on instrumentation and sampling to precisely estimate task progress and accordingly cut off outliers.

There are several challenges to make our solution applicable to the production environment. (1) Instrumented programs may encounter unacceptable overhead. For example, we instrument function calls to reflect the position of the program that the task is executing, in the case that the program is well structured by functions. But if a frequently called function is chosen, the overhead shall increase. To meet such a challenge, we introduce a sampling phase to deliberately yet automatically select what functions/instructions to instrument. (2) Even though instrument points are wisely selected, there is no guarantee that these points are touched linearly along with the task progress. In the task duration, some instrument points appear temporally dense in the trace while others are sparse. So we have to design a scheduling policy to pick up potential outliers based on such skewed data. We borrow a data mining technique, the k-mean clustering, to distinguish different task states from their traces. (3) There exist many legacy code in scientific computing. We have to provide concise interfaces to execute them and integrate all the instrumentation, sampling and scheduling phases to form a user-friendly workflow. This target is also achieved in our design of framework.

To sum up, we make the following contributions in NO^2:

- We design and implement an easy-to-use, efficient and robust framework for parallel execution of massive compute-intensive tasks.
- We create an instrumentation-based technique to estimate the progress of compute-intensive tasks. Our approach introduces a sampling phase before final instrumentation to collect program characteristics, which enables a wise selection of instrumentation points and a better estimation of progress.
- At runtime, we design a scheduling policy based on the k-mean clustering method to identify outliers. This method is robust to uneven instrument points and diversity between data sets.

In the rest of this paper, we briefly introduce ProActive and outliers in Section 2, describe the system architecture in Section 3, address two main techniques: instrumentation and outlier clustering respectively in Section 4 and Section 5, and mentions implementation details in Section 6. Experimental results are reported in Section 7 followed by some discussion in Section 8, and related work is discussed in Section 9. Finally, we make a conclusion in Section 10.

2 Background

This section makes a brief introduction to ProActive [3], the basic computing framework on which our solution is built.

2.1 ProActive

ProActive Parallel Suite [4] developed by the French national laboratory INRIA, is an innovative open-source solution for the acceleration of applications. It is seamlessly integrated with the management of high-performance clouds, and simplifies the parallel program development in cluster, grid and cloud computing environment. ProActive is completely based on Java development for parallel distributed computing, and does not make any modification on Java Virtual Machine (JVM), so it can run on any operating system that supports a standard Java environment. With the ProActive platform, users tackle the acceleration and orchestration of all demanding applications easily.

To execute parallel tasks in distributed environments, such as clouds and clusters, requires a unified mechanism for scheduling tasks and managing resources. The original ProActive is equipped with a batch scheduler [5]. The scheduler provides an abstraction of resources for users. It allows users to submit jobs that contain one or several tasks, and to execute these tasks on available resources afterwards. It enables several users to share the same resource pool and to tackle the problem involved with distributed environment, such as failing tasks or resources. It also allows users to kill a specified task easily, and to restart the task on another node.

By default, the scheduler assigns tasks according to the default FIFO (First In First Out) priority policy. The users can increase the task's priority according to its emergency, or can change the basic policy. In ProActive, to create and add a new scheduling policy is very simple. Users only need to implement the policy interface, and execute the scheduler with a new policy.

The ProActive Scheduler is connected to the Resource Manager [6], a component for resource aggregation across the network. It sends compute units represented by ProActive nodes (Java Virtual Machines runs the ProActive Runtime) to the scheduler that manages the task workflow and distributes tasks to accessible resources. According to the deployment, it can retrieve computing nodes using different standards such as SSH, LSF, OAR, gLite, EC2.

With the Scheduler, the Resource Manager and other components, the ProActive platform can easily run on a cluster, a grid or a cloud or any mixture of them without modification.

2.2 Outliers

Many previous works have identified outliers as a critical performance killer in heterogeneous clouds. Even in high end clusters, outliers are an inconvenient issue. Take our experimental cluster for example. We share this large local cluster, which has 8800 CPU cores over 740 server nodes, with many other institutes and researchers. As the local cluster was mainly built for scientific computing, it only provides a Load Shared Facility (LSF) job scheduling system for users to submit jobs in parallel. This situation is similar to what typically happens in the cloud, where many virtual machines share some physical servers.

Additionally, we have another small cluster that is dedicated to our experiment. To better utilize all parts of computational resources, we deployed a uniform ProActive Scheduling System to administrate all available resources. In such a heterogeneous environment, outliers often emerge.

The traditional ProActive global scheduler does not support the collaboration and communication with the local job scheduler on a cluster, so that too much CPU contention on some nodes leads to outliers. The outliers prolong the completion time of a job and seriously harm the performance of ProActive.

3 System Overview

Inspired by the problem caused by outliers, we name our system NO Outliers (NO^2) which entails our aims as well as the NO^2's mission. It is learned that there is only one system that has a similar name with NO^2 in the world. It is named N2O, primarily concerning with introducing fuel and nitrous into an engine's cylinders for efficient combustion. N2O provides an instantaneously speedup for auto. Whereas, our NO^2 system can continuously speed up parallel processing for massive compute-intensive tasks in clusters and clouds.

NO^2 contains two main components (Fig. 1): the instrumenter ("Instrument Point Selector" + "Instrumenter") and tasks scheduling policy generater ("Outliers Clustering" + "Policy Generater"). The instrument point selector collects the statistics of function hits in a variety of executions and selects some function entries for an appropriate instrumentation granularity. Although the instrumenter can independently do the instrument task of tracing all function entries, but may incur unacceptable overheads in a production environment. With the help of the instrument point selector, such overhead can be reduced to a low level. Details will be discussed in Section 7. The tasks scheduling policy generater is supported by the outliers clustering. With the outliers information, an optimized tasks scheduling policy can be generated for the cloud or cluster. From the users' perspective, they first send the original binary code to be instrumented and then submit a job description. ProActive runs the instrumented binary code within so called "instrumented processes" and NO^2 collects traces to find outliers. Accordingly, NO^2 send controls to ProActive Scheduler to adjust scheduling.

We carefully designed NO^2 to assure its independence with the job scheduler that facilitates the users' adaptation to different job schedulers. As shown in Fig. 1, the only coupling between NO^2 and ProActive Scheduler is interfaces for obtaining task status information of a job and requesting to kill or start a task. These abilities can be easily satisfied by any job scheduler's control interface, such as control script provided by ProActive Scheduler or command line tools provided by Condor.

The instrumentation has two phases: function hits statistics and instrument with selected functions by the instrument point selector. For function hit statistics, another instrumentation is needed. As the statistics are offline, this instrumentation covers all functions in the binary code without considering instrumentation overhead. The modified binary code is executed with some input cases in production runs and the statistic results are saved in a file. With the statistics, the instrumentation for progress trace can be performed with the binary code in an appropriate granularity.

The speculation algorithm of tasks scheduling for a job is straightforward with improvement driven by several heuristics rules of thumb. These empirical rules have been verified in practice. The first rule is to assign a higher speculation priority to a worse outlier. As speculation is costly, this rule is obviously advisable. Making sure that the split task and merge task do not execute on irregular nodes is another important rule because we should not expose an outlier with a single task. On the other hand, the split and merge phases are always the critical stage of a job. The last one is to speculate as soon as possible without irregular nodes. The goal of speculation is to complete the job early. Earlier and faster speculation means better opportunity. The pseudo code of speculation algorithm is shown in Fig. 2.

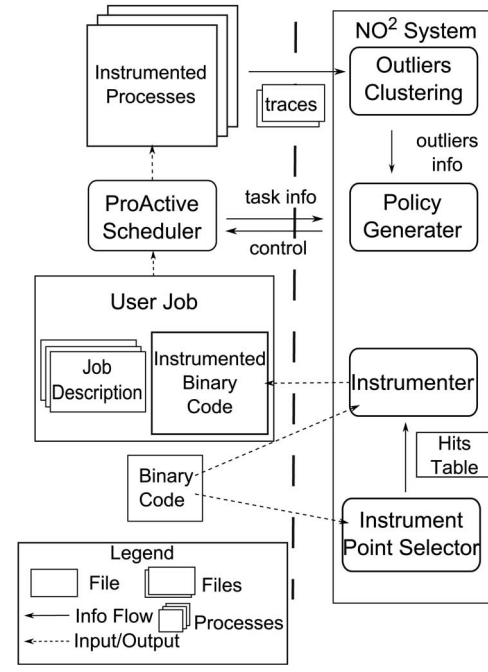


Fig. 1. NO^2 architecture.

In the next two sections, we will mainly illustrate the design of instrumentation based on function hits statistics and outliers clustering–the two key points in NO^2.

4 Instrumentation

There are two main challenges on tracing the progress of a process with binary instrumentation in production systems. One challenge is performance. Even a lightweight static binary instrumentation would increases execution time by tens of percentages. The other one is progress accuracy. The ideal progress tracepoints are uniformly distributed along the whole execution, which means that it can estimate accurately how much the task has been done and how long it lasts. Taking downloading a file for example if the downloading progress bar shows a 50% completion after 1 minute, it is clear that half of the task has been done and it may last another 1 minute with a stable network connection. But for a compute-intensive process, we only know that the process has met which tracepoint we instrumented. We can not know the actions of the process that has no instrumented tracepoints, and neither the subsequent actions of the process. In other words, it is hard to estimate the progress of a compute-intensive process that is not related to data I/O.

We primarily concern the performance of tracing in NO^2's instrumentation, and overcome the accuracy issue with our

Algorithm:TasksSpeculation

Input: instance of the job and sleep $interval$
Output: the job's state $job.state$

```
1: while job.state ≠ FinishedOrFaulty do
2:     tasks ← job.tasks
3:     clusters ← kmeans(tasks.traces, 2)
4:     if variance(clusters) > threshold then
5:         outliers ← min(clusters)
6:         sort(outliers, comparePriority)
7:         for outlier ∈ outliers do
8:             s ← submit(outlier.taskid)
9:             speculations ← [speculations, s]
10:        end for
11:    end if
12:    sleep(interval)
13:    job.update()
14:    for s ∈ speculations do
15:        if s.state = Finished then
16:            kill(s.outlier.taskid)
17:        else if s.outlier.state = Finished then
18:            kill(s.taskid)
19:        end if
20:    end for
21: end while
```

Fig. 2. Task speculation algorithm.

outlier clustering solution, which does not rely on accurate estimation of progress. This will be discussed in the Section 5.

With many instrumented execution cases, we have got two simple observations on binary code runtime behaviors. Functions in a binary can be placed on different levels of an abstract syntax tree (AST). Functions on the same level of AST are often called at similar frequency, and the minority of the functions on the low level of AST consumes the majority of execution time following a Pareto principle (80/20 principle). Table 1 shows the statistics of function hits of the instrumented ImageMagick dynamic link library. There are 1060 functions in the library and only 33 functions are called by more than 10,000 times in five executions.

As we know, the entry/exit points of functions and basic blocks are commonly used instrument points. For the purpose of progress tracing, an idealistic idea is to instrument each entry and exit point with all functions and basic blocks. Apparently, the overhead is proportional to the hit rates. So with a few more coarse-grained instrumentations, we can skip the minority of functions and reduce the majority of overhead. To achieve minimum overhead, we propose a hit-statistics-based function instrument approach.

The instrument point selector is designed to record each function call and maintain the function hit statistics, which are saved in a key-value pattern: function name and number of calls. After several runs, different AST levels of functions can be distinguished according to the function hits. The instrumenter first loads the statistics data and calculates the mean of hit numbers of all functions. Then it inserts a code snippet at entries of each function, but skips those with a larger number of hits than the mean. If it works well, the mean of all function hits will be set as the customized threshold. With the automatic selection, the instrumenter can instrument the binary codes in an appropriate granularity. The instrumented binary is written back as a seperate file and sent to the ProActive scheduler to execute.

TABLE 1
Statistics of Function Hits of ImageMagick

Hits	Functions	Example
above 10^7	6	CopyMagickMemory
10^6 - 10^7	16	GetCacheNexus
10^5 - 10^6	2	LocaleCompare
10^4 - 10^5	9	GetNexus
10^3 - 10^4	10	AddValueToSplayTree
10^2 - 10^3	17	FormatMagickStringList
10^1 - 10^2	78	NewSplayTree
0 - 10^1	922	GaussianBlurImage

Note: Only one function of each order of magnitude is shown in the table for example.

5 Outlier Clustering

For speculative executions, the first task of NO^2 is to find out the outliers. With traces returned by the instrumented processes, NO^2 uses an approach based on K-means Clustering algorithm, which is commonly used in statistics analysis and data mining, to hunt outliers. With outliers exposed, NO^2 performs speculative execution or aborting, which needs to interact with the job scheduler.

There are several assumptions for the outliers clustering.

- Outliers is the minority of the massive parallel processes.
- A job is split into tasks with only slight imbalance.
- The server nodes of a cluster become abnormal in an indeterministic manner and may return to normal after a while.

We novelly select two properties of each process's traces to make outliers exposed, the number of triggered tracepoints N and the increment of tracepoints in an interval I. Fig. 3 shows an example of two processes of the same binary code with the same input at runtime. Ideally they should draw two very similar lines, but process B is actually slower than process A.

A straightforward idea for exposing outliers is to generate two classes using a one-degree K-means Clustering algorithm with all processes' number of tracepoints. If the normalized variation of the two classes' means is larger than a threshold T, the class of processes with a mean value N less than the other class will be judged as outliers.

But the number of tracepoints grows unevenly along the task execution. As illustrated in Fig. 3, four differences of tracepoint numbers of two processes have been marked as d_1, d_2, d_3, d_4 at four different time points. Although process B is always 0.8X slower than process A, the difference of process A and process B is not increased along the whole time line. An obvious case is $d_1 > d_2$, which may mislead the naive k-means clustering approach to missing process B when clustering outliers. Another type of mistake may be made in conditions just as $d_3 < d_4$. Process B may be just a little slower than process A, but it is judged as an outlier in the naive k-means clustering approach. So there is no fixed threshold T that is suitable for judging the outliers with the naive k-means clustering approach.

To eliminate the false positive and false negative mistakes, an improved approach has been proposed in NO^2. We

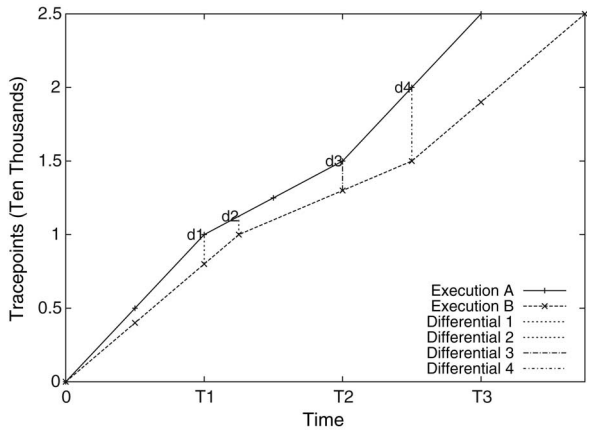


Fig. 3. Tracepoints of two example executions.

employ a new dynamic threshold T_d, which is determined by the threshold T, the mean of all processes' tracepoint increment I_m and the ith process's tracepoint increment I_i.

$$T_d = T \cdot (I_m / I_i).$$

As one of the assumptions is that outliers are the minority, the mean of the increment of tracepoints of all processes I_m is nearly the same to the majority. When the variance of the means of clusterings decreases exceptionally like in the case $d_1 > d_2$, the dynamic threshold T_d decreases correspondingly. When the variance of the means of clusterings increases exceptionally like in the case $d_3 < d_4$, the corresponding increase of T_d comes along. So dynamic threshold T_d can generally be adaptive to the uneven increase of the number of tracepoints. With this improved approach for outliers clustering, NO^2 can find out most outliers whereas introduce few false positives.

6 Implementation

For a clear understanding, we illustrate some important details of NO^2's implementation in this section. The technologies we used in development and some implementation tricks hidden behind are introduced in brief.

We developed the NO^2's instrument tool with a static instrumentation fashion based on the DynInst [7] library. The DynInst library is an Application Program Interface (API) [8] that permits the insertion of code into a static binary code or a running process. The goal of this API is to provide a machine independent interface to permit the creation of instrumentation tools and applications. The NO^2's implementation of function hits statistics is based on an example of DynInst tools named CodeCoverage [9].

Generally, there are three code snippets for the instrument point selector: the initial one is used for creating some data structures for statistics; the commonly used intermediate one is used for recording a function call; the final one is for writing back the function hits data to a file. There are also three code snippets for the instrumenter: the initial code snippet is used to open a file; the commonly used intermediate one is to increase the number of trace points and to write it to the file; the final one is to close the file. All the code snippets are carefully assembled with DynIsnt API and instrumented into the original binary.

To pursue excellent performance, the instrumenter completes the assembly of file operations through using the low level system call *open*, *write*, and *close* in Linux Opreating System. The file for saving traces is created in the directory that is mounted as a RAM file system. The RAM file system uses memory as a disk in the Virtual File System (VFS) level supported by the Linux Operation System. It means that only several memory operations overhead is introduced into the instrumentation of each function hit.

We provide a task speculation implementation with ProActive Scheduler [5], which can be interacted with a control script. The control script is interpreted in the javascript language based on a script engine, called 'Rhino', built-in integrated with the distribution of Java Standard Edition 6 (Java SE6). ProActive Scheduler provides the ability of interactive with the scheduler instance at runtime as script engine can access and invoke objects and methods in Java.

Our task scheduling policy generator calls the outliers clustering module to establish outliers clustering with updated trace data. Then it maintains a blacklist, each node of which has a penalty value. The penalty could be eliminated if no further outliers are found on a node and could be removed from the blacklist. In summary, the blacklist is used for keeping the speculative tasks away from the most possible nodes that may produce outlier tasks. For each outlier, the policy generater creates an urgency job description that has a higher priority in ProActive Scheduler. This urgency job has only one speculative task. Then all these urgency jobs are submitted to ProActive Scheduler by a speculation priority order. The constraint of no speculation taking place on irregular nodes can be achieved with a selection script and a claim of nodes selection in the job description. The default speculation interval is three seconds implying that the interactive with the job scheduler is moderate. Each update of traces information has little communication traffic and only several bytes indicating the number of trace points are transmitted. This lightweight implementation is acceptable to the job scheduler and the impact of performance can be ignored roughly.

We provide three extra shell scripts for splitting, wrapping native program and merging the results if needed. The shell scripts will be submitted and executed as normal tasks on the job scheduler. They can be used immediately in ProActive Scheduler. But it doesn't mean any dependence. It is easy to export to other schedulers with tiny modification. Taking a picture rendering job for example, there may be three phases for this job: 1. splitting the image into small pieces, 2. rendering them in parallel, and 3. merging the rendered parts. These three phases exactly match NO^2's three shell script templates : split script, operation script and merge script. Users just need to add an image cutting command into the split script, which may be optional for other jobs. The default procedure (i.e., copying input files to the destination), can also be modified if needed. In the operation script, the native program must be expressed as a command, a daemon process for traces transferring is also launched as default. When all parts of the results has been collected, a merge image command is executed in the merge script which is optional for other jobs too. As described above, users have an extreme flexibility to make their native

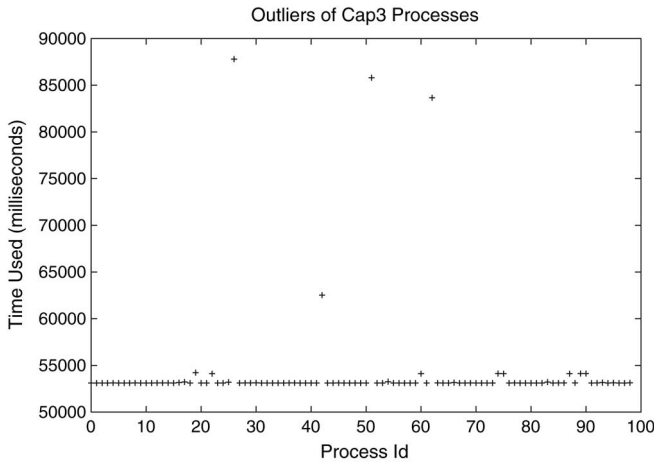


Fig. 4. Process durations of an Cap3 execution on 100 CPU cores in cluster at 8:00 am.

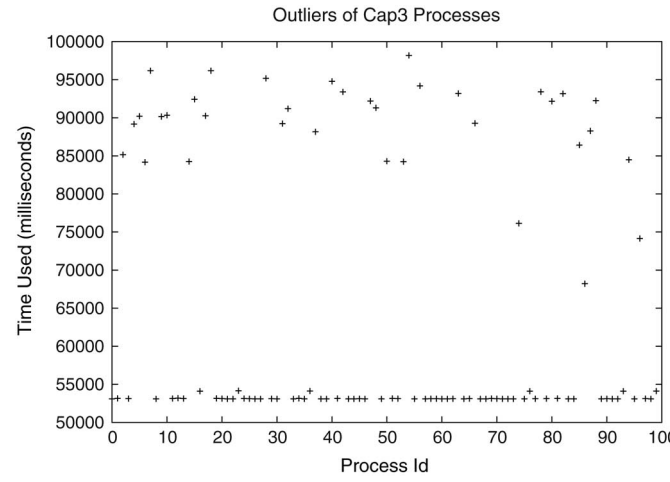


Fig. 5. Process durations of an Cap3 execution on 100 CPU cores in cluster at 8:00 pm.

programs massive parallel. It also means NO^2 does not provide any file transfer, data placement service, or anything like this, which have been provided by job schedulers or a distributed file system of cluster itself.

7 Evaluation

We deployed and evaluated NO^2 on both a local high end cluster and the cloud. Two representative applications, Cap3 and ImageMagick, were chosen for the evaluation purpose. We focus on reducing the completion time of jobs. Besides, the overhead and computing resource usage of NO^2 are also measured.

The first application, Cap3, is a well known sequence assembly program in bioinformatics. There are many Expressed Sequence Tag (EST) sequence files in FASTA format. So the job is split naturally according to the FASTA files. With the job description engine and the shell script templates we provided, it is simple to describe a job. The second application, ImageMagick, is a classic open source picture processing tool. Gaussian blur is a filter of ImageMagick, and it blurs an image by a Gaussian function to reduce image noise. This effect is widely used in graphics software, also known as Gaussian smoothing. In this case, the job renders an image of a huge size with Gaussian blur. The image is cut into small pieces with ImageMagick and then is filtered. All pieces are joined together in the end.

We use two metrics for the evaluation: job completion time and computing resource usage. As a job consists of lots of tasks that are processed in parallel, we define the job completion time as the duration between the submission of the job and the finish of all tasks. [1] We run a job repeatedly and use the average job completion time as the system metric. Considering speculative executions call for extra computing resource, we also measure the computing resource usage with CPU utilization, monitored by the ProActive Resource Manager. Given the baseline and NO^2 CPU utilization, the cost of speculative executions can be demonstrated.

1. There are several iterations in a workflow for most of the scientific computing jobs, and the successor jobs depend on the precursor ones. For instance, the precursor's output may be the the successor's input.

NO^2 acts as an extension to ProActive Scheduler. ProActive Scheduler is a high level job scheduling system based on ProActive library. With ProActive Resource Manager, jobs can be submitted and scheduled to use variable types of computing resources. A job is described with an XML description.

7.1 Verification

7.1.1 Outliers

In a high end cluster, each server node has powerful CPU, sufficient memory, and is equipped with the high performance network (e.g., InfiniBand) and usually a parallel file system. The heterogeneity of hardware is limited. Will outliers come out in a short several-minute execution?

We randomly selected a hundred of JVM nodes, and submitted a job with hundreds of parallel tasks. The test was repeated in different periods of day. We found that even in a several-minute execution, sometimes there are a few outliers, and they appeared on random server nodes and at different times.

We pick up two trials in this experiment, and make two scatter plots respectively as shown in Figs. 4 and 5. In Fig. 5, there are many more outliers than Fig. 4, and outliers need nearly twice the time to complete the same task. As one server node contains several CPU cores, the physical outliers are not as many as outliers of CPU cores. With this simple observation, we can conclude that even in high end clusters, there are a certain number of outliers in some server nodes and during some periods of day, due to common computing resource contention. For a further understanding about the outliers in the local cluster, we loop this experiment several times to last a whole day. Fig. 6 shows a rough statistics for the probability of outlier each hour in one day.

As this experiment may produce some computing resource contentions and the data is just in one day, the statistics is not accurate. For a non-quantitative study with the outliers, we simply make an assumption that a CPU core of the cluster happens to be an outlier is a random event, and the events are independent. As jobs often need lots of CPU cores to run

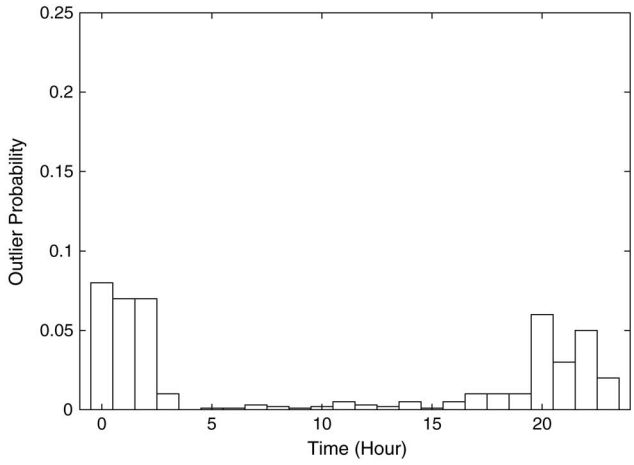


Fig. 6. Probability of outliers during each hour of the day.

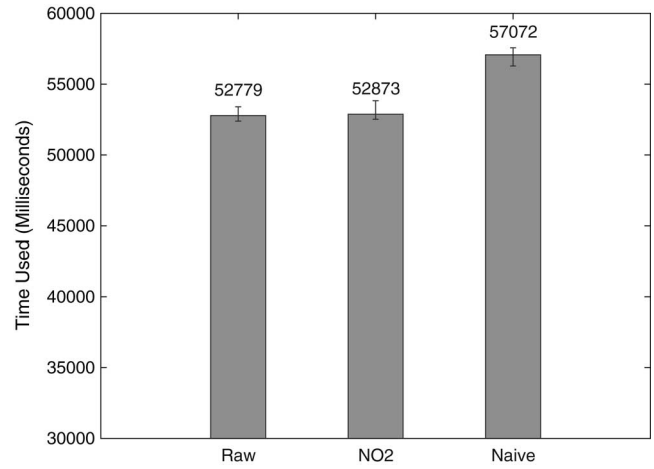


Fig. 7. Cap3 instrumentation overhead.

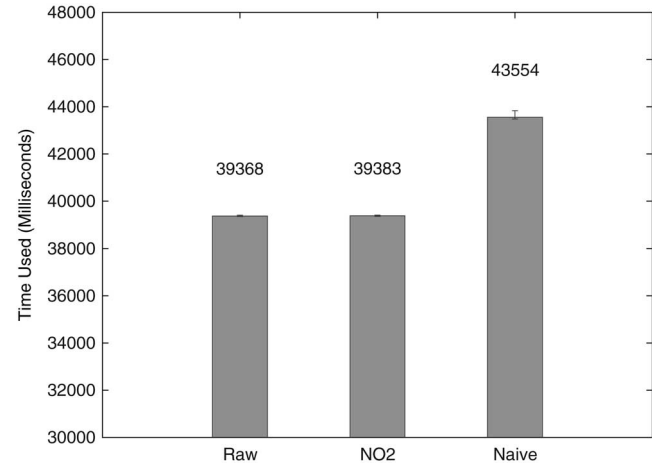


Fig. 8. Gaussianblur instrumentation overhead.

massive parallel tasks, the probability P of a job running N parallel tasks without outlier is:

$$P = \prod_{i=1}^{N} p_i(t).$$

While $p_i(t)$ is the probability of ith task of the job happens *not* to be a outlier at time t, $\overline{p}_i(t)$ is the probability of ith task of the job happens to be a outlier at time t. We choose the median of the Probability of outliers shown in Fig. 6: $\overline{p}_i(17) = 0.01$, for a job having 100 parallel tasks, the probability of running without outliers P is about 0.3660. When the job scales up to 1000 parallel tasks, even with a very little probability of outlier $\overline{p}_i(16) = 0.005$, the probability of running without outliers P is about 0.0067, which means that it is nearly impossible to keep away from outliers. So an outlier-aware tasks scheduling is extremely needed for a job with massive parallel tasks.

7.1.2 Overhead and Accuracy of Instrumentation

Binary instrumentation often costs a very high overhead, which may be tens of times slower than no instrumentation executions. Although the static binary instrumentation is more efficient than the dynamic, usually tens of percent points of overhead is needed if all function entries are traced, as shown in Figs. 7 and 8. Such high overhead prevents it from being used in production clusters.

To evaluate our progress trace approach, we run Cap3 and Gaussianblur from ImageMagick in one server node of the cluster. Each application is tested repeatedly with different inputs. Figs. 7 and 8 shows the results of this experiment. On average, instrumentation with all function entries costs 8-11% extra time to complete the execution, while instrumentation based on function hits statistics in NO^2 needs only 0.03-0.1% extra time costs.

Such low overhead makes NO^2 efficient enough to deploy on a production system. But the tracpoints are uneven, which means the instrumentation may sometimes miss parts of progress and the processes have caused the progress to report inaccurate. So we make an inspection with some trace cases of processes of Cap3, Gaussianblur and a combined image rendering all from ImageMagick. We noticed that the progress trace with instrumentations with all function entries is a little smoother than those with our instrumentation approach, such as ImageMagick cases shows in Fig. 9. But in some cases our approach is a little smoother than instrumentation with all functions, such as Cap3 case shows in Fig. 9.

Since each execution is completely arbitrary with different inputs and conditions, every function may last short or long period of time in different executions. Unfortunately, we can not make an inference of absolute execution progress with traces of the instrumentation with all function entries. Even for the instrumentation with all basic binary blocks, there is no absolute accuracy. A reasonable approach may be a customized instrumentation in the kernel level, requiring the modification of the Operating System kernel, with a large mount of time cost, which is nearly impossible to a production system.

But a relative smooth tracepoints hit rate can be guaranteed with the instrumentation with function entries in same granularity, as shown in Fig. 9, each line is composed of few smooth segments. The outliers clustering approach we proposed is inspired by this inspection. We use the relative progress to catch the outliers, so the approach instrumentation based on function hits statistics is enough in our context.

7.2 In Local Cluster

The local cluster we used is a production one in Tsinghua University, which has hundreds of server nodes. While each

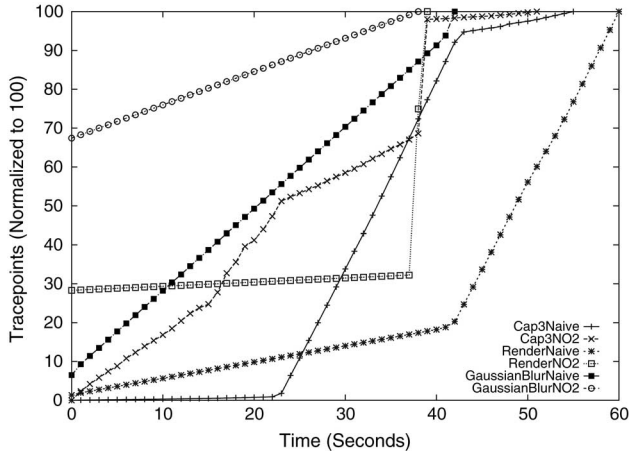


Fig. 9. Skewed tracepoint accumulation over time.

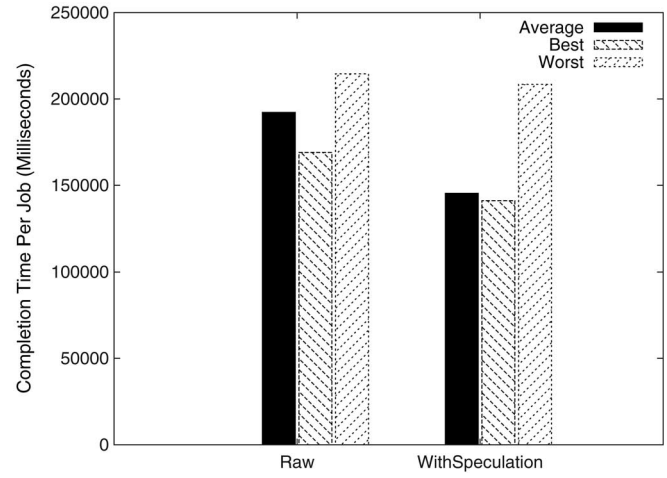


Fig. 11. Completion time of GaussianBlur jobs.

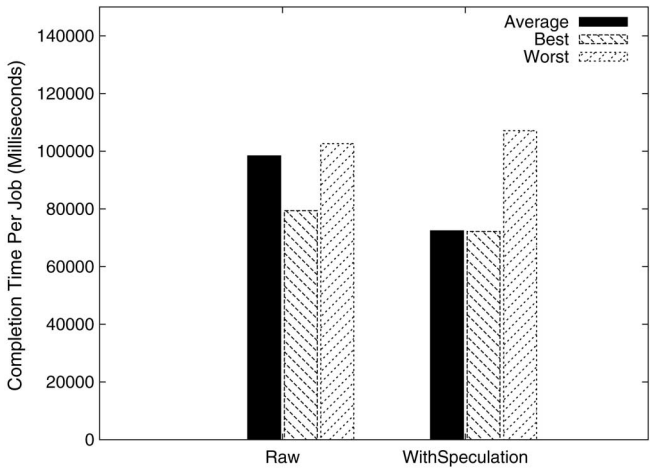


Fig. 10. Completion time of Cap3 jobs.

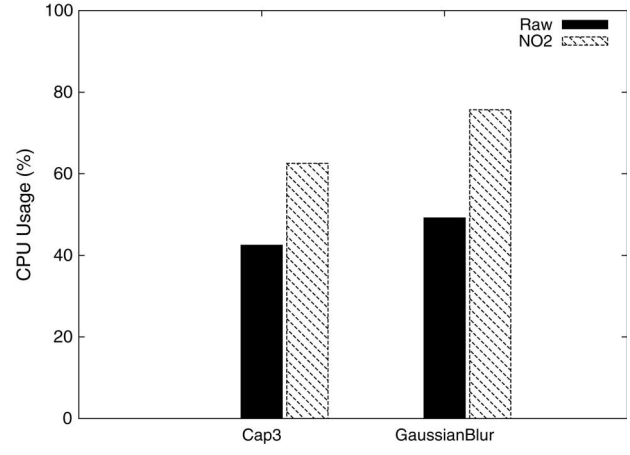


Fig. 12. Computing resource usage.

node has tens of GBs of RAM and two Intel Xeon X5670 2.93 GHz CPUs, each CPU has 6 cores. InfiniBand network and LUSTRE parallel file system are deployed in the cluster. With Load Sharing Facility (LSF) job scheduling system, hundreds of Java Virtual Machines (JVMs) are launched as a LSF job in the cluster. These JVMs are added as compute nodes in ProActive Resource Manager. We used the cluster just in this way: adding the cluster as a LSF node source to the ProActive Resource Manager.

Reduction in job completion time is a critical metric. We submitted Cap3 and Gaussian Blur jobs fifty times each to ProActive Scheduler with and without NO^2. Figs. 10 and 11 show that job completion time has been improved by roughly 25% on average. The histogram plots the best, the worst, and the average reduction of the Cap3 jobs and Gaussian Blur jobs. In the worst case of Cap3, there is a little increment of job completion with NO^2, caused by a worst choice of the speculative execution. On average, the reduction of job completion time with NO^2 is significant. Without NO^2, caused by the hinder of outliers, the completion time of jobs has a big deviation with the best case. However with NO^2, the completion time of jobs is reduced to the best case. These two experiments show that NO^2 mitigates the outliers efficiently.

NO^2 can benefits job schedulers with reduction of job completion time. At the same time, it needs some amount of computing resource in addition. In the aforementioned test, we collected the statistics data of CPU usage from ProActive Resource Manager and plotted a histogram as shown in Fig. 12.

As shown in Fig. 12, 20-30% more computing resource has been used by NO^2 for speculative executions. This makes sense for some local clusters, which have low utilization in most of the time of day. NO^2 helps them to scheduel jobs more efficient, and reduces the server idle time. But for a busy cluster or cloud, where computing resource is not free, we expect that speculative executions use resource as little as possible. We will analyze how to reduce waste of computing resource with little loss of reduction of the job completion time later, and for some error-prone environment NO^2 can even use less computing resource with less failure executions.

7.3 In Cloud

With attractive low price and demand payment, there are more and more computing tasks transferred to clouds in both academia and industry. The virtual clusters has been more and more popular, StarCluster [10] is a toolkit for Amazons Elastic Compute Cloud (EC2), designed to automate and simplify the process of building, configuring, and managing clusters of virtual machines. We deployed such a virtual cluster on EC2 with StarCluster. This cluster has hundreds

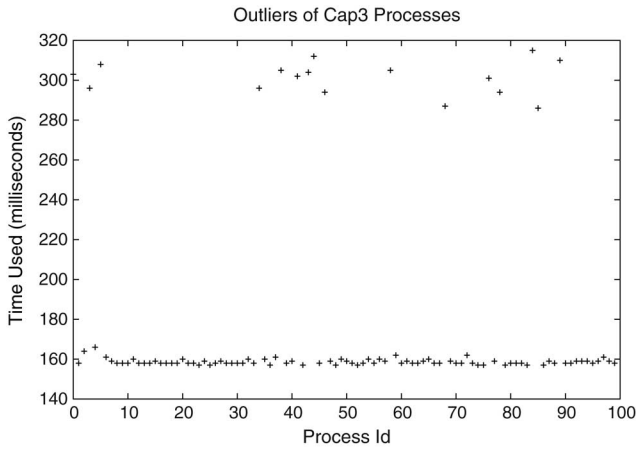


Fig. 13. Process durations of an Cap3 execution on 100 instances in the cloud.

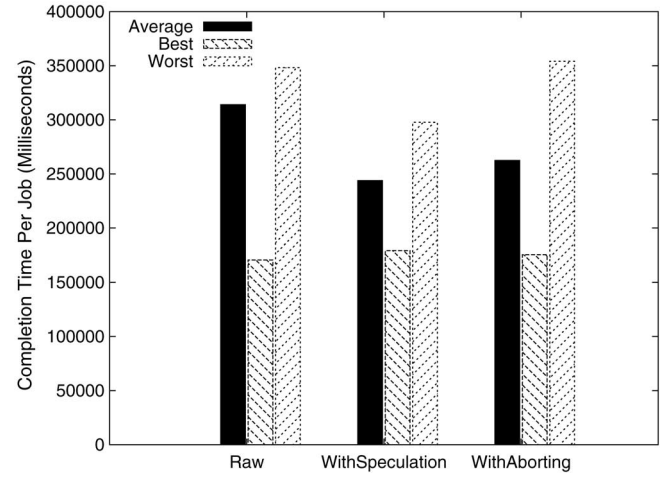


Fig. 14. Completion time of Cap3 jobs in the cloud.

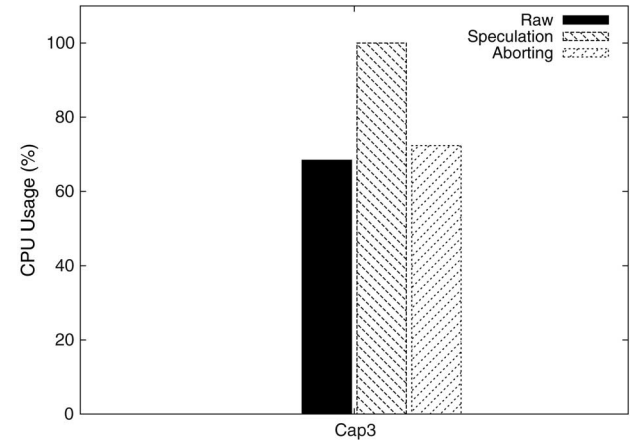


Fig. 15. Computing resource usage in the cloud.

of virtual machine nodes. Each node is a M1 small instance, which has 1.7 GB RAM and one virtual CPU core with 1 EC2 Compute Unit. An Elastic Block Storage (EBS) device is mounted as share storage and NFS file system is deployed in the cluster. We deployed ProActive Scheduler on this virtual cluster like in local cluster, by adding these virtual machine nodes as an SSHInfrastructure type node source. Hundreds of Java Virtual Machines (JVMs) are launched and added as computing nodes in ProActive Resource Manager.

We began our evaluation in the cloud by measuring outliers, same as the verification we have done in the local cluster. Unlike the high-end server nodes of local cluster, the EC2 instances are mediocre and less powerful. The outliers in the cloud are more obvious than the local cluster and prolong a 2X or more completion time for jobs, which has been shown in Fig. 13.

We submitted Cap3 jobs in the same scale as local one, and plot the histogram in Fig. 14. As computing resources is not sufficient in the cloud, we also try an aborting strategy, also known as 'kill then restart' which costs less computing resources than speculation. We will discuss this strategy in the next section.

There is roughly 15-22% job completion time reduction on average shown in Fig. 14. At the same time, the resource usage with the speculation strategy is 100%, implying that there may be more speculations not performed for any sufficient computing resources and other jobs may be blocked with the same reason. The aborting strategy is more economic, as it only costs a 4-5% additional computing resource.

8 Discussion

We will make a discussion on the threshold tuning in NO^2 system in this section and take one step further by evaluating a different strategy to mitigate outliers.

8.1 Threshold Tuning

How slow will outliers mostly be? Or to what extent is the process lagged behind the majority when we could judge it as an outlier? The threshold of outlier clustering is critical for NO^2. In our outlier clustering approach, we rely on a naive one-degree kmeans clustering algorithm where $K = 2$, and we use the normalized variation of the centers of clusterings to judge if one of the clusterings contains outliers. When the normalized variation is larger than a dynamic threshold T_d, NO^2 realizes outliers exist. But the dynamic threshold T_d depends on the original threshold T was determined a little bit arbitrarily. So a fine-grained tuning with the threshold T is an indispensable procedure for NO^2.

We use three fix JVM nodes running a shell script and costing lots of CPU to act as outliers. In order to make sure other nodes are running normally, if there are some unexpected outliers, we just drop the execution result. In this way, the other variation is limited. Tuning the threshold T from low to high, we run a Cap3 job that was split into 100 parallel tasks to study the sensitivity of the threshold T. In Fig. 16, as threshold rises, the number of waste speculations declines and keeps low, but successful speculations have no obvious deviation. When the threshold still increases, NO^2 finds out few outliers and becomes lag to response.

A low threshold may be false positive, and cause a large amount of speculative executions, as shown in Fig. 16, implying a low success rate of speculations and waste of a lot of computing resource. With an optimized threshold, our outliers clustering approach can mitigate the false positive risk and keep sensitive with outliers, which is critical for NO^2.

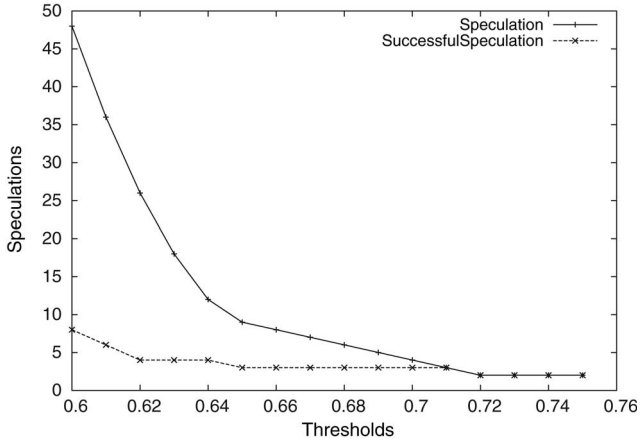


Fig. 16. Number of speculative executions over threshold values.

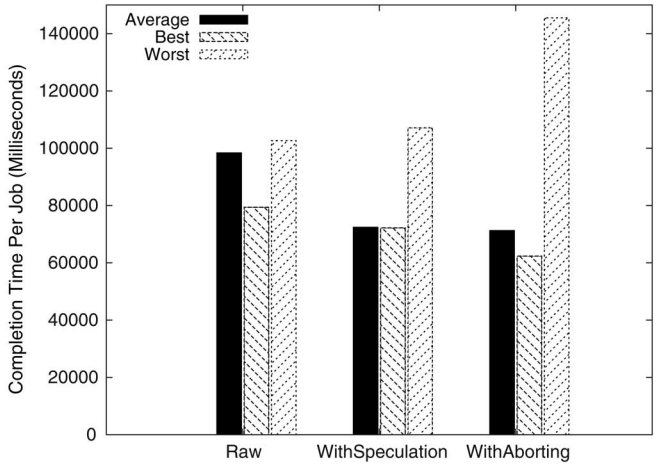


Fig. 17. Cap3 job completion time with aborting strategy.

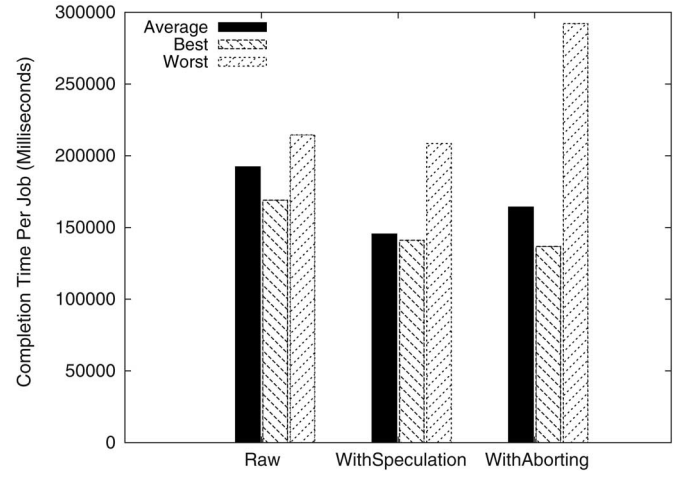


Fig. 18. GaussianBlur job completion time with aborting strategy.

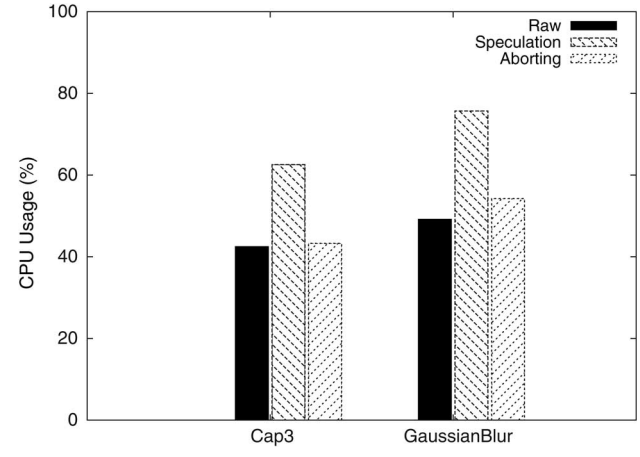


Fig. 19. Computing resource usage of aborting strategy.

8.2 Aborting Strategy

Considering that the idle computing resources may be insufficient to meet the speculations' need in the cloud. We try an aborting strategy, which is a bit different from speculation. This strategy is more aggressive and adventurous. Rather than speculates, it immediately aborts potential outliers and restarts these tasks on other nodes that are not in the blacklist. This extreme scheduling decision cuts lots of computing resource usage for speculation, with the drawback of slowing the completion time of a job. Because it is not sure if a speculative process will complete earlier than a outlier.

We verified this strategy by repeating the same experiment with modified NO^2 equipmented with the aborting scheduling strategy in the aforementioned local cluster. Figs. 17 and 18 show that the aborting strategy has a larger variation than the speculation one in terms of job completion time. In the worst case of Cap3 and Gaussian Blur, the aborting strategy is obviously slower than the case without NO^2 and with the NO^2's speculation strategy. On average for Cap3 case, similar to the speculation strategy, the aborting strategy reduces 25% of job completion time. In contract, the aborting strategy has a slowdown in the Gaussian Blur case. There is nearly 10% more time cost than the speculation strategy. These two experiments show that the aborting strategy can reduce the completion time of jobs, but is a little bit slower than the speculation strategy.

With a minor performance degration, the aborting strategy cuts down lots of CPU resource usage. As the same as the last experiment, we collected the statistics data of CPU usage from ProActive Resource Manager and plotted a histogram as shown in Fig. 19.

As shown in Fig. 19, about 20% CPU usage reduction was obtained. With the aborting strategy, the waste of CPU resource is bounded about 10%, which verifies that the aborting strategy is valuable. This is also comfirmed by the experiment in the cloud shown in Fig. 15.

9 Related Work

A variety of studies have been done on the classic problem of scheduling policies for task assignment in distributed systems, and they were designed for different circumstances and various requirements. TAGS [11] and SITA-V [12] deal with scheduling independent tasks among lots of server nodes in a cluster, such as web servers that process HTTP requests. With the goal of achieving low mean response time and low mean

slow down, they reach consensus to overcome the challenge of heavy-tailed task size distribution due to load unbalancing. Task duplication (TD) [13] for massive parallel processing has been analyzed for its effectiveness. Using task duplication based on dependency graphs of tasks, TDS [14] and some other studies [15], [16] trade off between maximizing concurrency and minimizing inter-processor communication with the primary objective of minimizing schedule length and scheduling time. M. Silberstein et al. [17] propose a scheduling policy for heterogeneous dynamic workloads composed of massively parallel short tasks. Considering the communication cost and heterogeneous computing environment, some dynamic task scheduling approaches [18], [19] have been proposed. An extended version of Condor MasterWork [20] dynamically measures the execution times of tasks and uses this information to dynamically adjust the number of workers to achieve a desirable efficiency, minimizing the impact in loss of speedup. Some of these previous works are old fashioned, as proposed more than ten years ago. Our work has a different background and different requirements with these prior ones. We aim at improving response time for a job consisting of massive parallel computing-intensive tasks of a similar size.

Some job schedulers were designed for global administration over clusters or datacenters. Most of them focus on overall throughput and fairness scheduling [21]. HTCondor [22] is a scheduler for coarse-grained distributed parallelization of computationally intensive tasks in hybrid computing environment, which is well known for its high throughput computing framework. ProActive is a platform and middleware for parallel, distributed and multicore computing with a java library provided and lots of features supported. ProActive Scheduler [5] is a job scheduler base on ProActive [23], which is similar with HTCondor. Cooperating with ProActive Resource Manager [6], it can be used with a variety of computing resources. HTCondor and ProActive are widely deployed and used in grids and clusters. Our NO^2 system is implemented to interact with ProActive Scheduler's interfaces. Our work mainly addresses the outliers problem of massive parallel executions and helps Proactive Scheduler to reduce the job completion time.

Speculative execution is borrowed from distributed file systems [24] and some other work [25]. Unlike those that treat parallel processes as a black box, our work launches speculations in a more reasonable situation. With instrumentation, we change the view to a white box.

It is attractive to use instrumentation for performance analysis, system modeling and debugging. Although there are still many open research problems in developing the instrument tools, it is among the most reasonable ways to understand complex system behaviors. Magpie [26] is a tool chain for automatically modeling workload of cluster and predicting performance with fine-grained instrumentation in kernel, middleware, and application components. DTrace [27] is an instrumentation framework on Solaris Operating System, providing an option for lightweight dynamic instrumentation in kernel. Similar approaches on other operating systems include the Linux Trace Toolkit (LTT) [28] and Event Tracing for Windows (ETW) [29]. The PMaC Prediction Framework [30] implements an automated prediction model that takes into account attributes of applications, input data, and target machine hardware (and other factors). The model computes the expected performance as output. It is a generalized framework for similar requirements with our system. In our work, we use instrumentation for a simple clear goal. In a specialized environment, our instrumentation is easier to implement than PMaC. As far as we know, our work is the first case introducing an instrumentation approach to the scheduling of Massive Parallel Processing.

Driven by the success of MapReduce [31] and its open-source implementation Hadoop, lots of improvements [1], [2] and adaptations [32] to varieties of applications have been done. Dryad [33] is a general-purpose distributed execution engine for coarse-grain data-parallel applications sharing the same goal with MapReduce but extends the programming model of MapReduce. J. Ekanayake et al. [34] make a study on comparison and evaluation of these technologies in scientific computing. There are various strategies to choose which tasks to duplicate. Hadoop uses slots that free up to duplicate any task that has read less data than the others after all tasks have been started. Dryad duplicates those that have been running for longer than 75 percentage of task durations. LATE [1] designs a new scheduling algorithm to robust Hadoop with heterogeneity. Being cause-aware and resource-aware, Mantri [2] detects and acts on outliers early in their lifetime. All these prior scheduling strategies motivate our work with the speculation idea. As MapReduce is designed for data-intensive computing, these works naturally predict the progress of executions using data processing progress. But with computing-intensive executions, this basic approche is not always valid.

10 Conclusion

As most existing computing frameworks are oriented to data-intensive tasks, we propose an efficient and robust framework, NO^2, for massive compute-intensive tasks. Our framework is based on ProActive, and we focus on improvement of its scheduler, one of the most performance-influencing components. In a large-scale computing framework, a key responsibility of the scheduler is to avoid outliers that may severely prolong the job completion. To sum up, we introduced three new techniques to the scheduler and the computing framework: firstly, we use instrumentation to indicate the progress of compute-intensive tasks, instead of using I/O proportion for data-intensive tasks; secondly, we sample the program before instrumentation to automatically select instrument points so that overhead are limited; thirdly, the k-means clustering method is employed in outlier identification without resorting to biased progress calculation. Evaluation has shown that NO^2 largely reduces the job completion time with small instrumentation overhead and acceptable extra resource utilization.

Acknowledgments

This work is supported by National Basic Research (973) Program of China (2011CB302505), Natural Science

Foundation of China (60963005, 61170210), National High-Tech R&D (863) Program of China (2012AA012600, 2011AA01A203), and Chinese Special Project of Science and Technology (2012ZX01039001).

References

[1] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, pp. 29-42, 2008.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters Using Mantri," *Proc. 9th USENIX Conf. Operating Systems Design and Implementation*, pp. 1-16, 2010.

[3] F. Huet, D. Caromel, and H.E. Bal, "A High Performance Java Middleware with a Real Application," *Proc. Supercomputing Conf.*, Nov. 2004, pp. 2-17.

[4] The OASIS Research Team and ActiveEon Company, *Proactive Parallel Suite*, http://proactive.activeeon.com/index.php, accessed 2012.

[5] The OASIS Research Team and ActiveEon Company, *ProActive Scheduling Version 3.3.2*. The OASIS Research Team and ActiveEon Company, http://hudson.activeeon.com/view/Scheduling/job/Scheduling_3.3.x/lastSuccessfulBuild/artifact/doc/built/Scheduling/multiple_html/index.html, accessed 2012.

[6] The OASIS Research Team and ActiveEon Company, *ProActive Resource Manager Version 3.3.2*. The OASIS Research Team and ActiveEon Company, http://hudson.activeeon.com/view/Scheduling/job/Scheduling_3.3.x/lastSuccessfulBuild/artifact/doc/built/Resourcing/multiple_html/index.html, accessed 2012.

[7] G. Ravipati, A.R. Bernat, B.P. Miller, and J.K. Hollingsworth, "Towards the Deconstruction of Dyninst," Univ. of Wisconsin, technical report, 2007.

[8] Computer Science Department University of Wisconsin-Madison and Computer Science Department University of Maryland, *Paradyn Parallel Performance Tools Dyninst Programmer Guide Release 8.0*. Computer Science Dept. Univ. of Wisconsin-Madison and Computer Science Dept. Univ. of Maryland, http://www.paradyn.org/release8.0/doc/DyninstAPI.pdf, accessed 2012.

[9] Computer Science Department University of Wisconsin-Madison and Computer Science Department University of Maryland, *codeCoverage*, http://www.paradyn.org/html/tools/codecoverage.html, accessed 2012.

[10] OEIT of MIT, *Starcluster Documentation (v. 0.93.3)*, http://star.mit.edu/cluster/docs/latest/index.html, accessed 2012.

[11] M. Harchol-Balter, "Task Assignment with Unknown Duration," *Proc. Int'l Conf. Distributed Computing Systems*, pp. 214-224, 2000.

[12] M.E. Crovella, M. Harchol-Balter, and C.D. Murta, "Task Assignment in a Distributed System (Extended Abstract): Improving Performance by Unbalancing Load," *Proc. ACM SIGMETRICS Joint Int'l Conf. Measurement and Modeling of Computer Systems*, pp. 268-269, 1998.

[13] S. Manoharan, "Effect of Task Duplication on the Assignment of Dependency Graphs," *Parallel Computing*, vol. 27, no. 3, pp. 257-268, Feb. 2001.

[14] S. Ranaweera and D.P. Agrawal, "A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems," *Proc. Symp. Parallel and Distributed Processing*, 2000, pp. 445-450.

[15] I. Ahmad and Y. Kwong Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872-892, Sept. 1998.

[16] A. Dogan and F. Özgüner, "LDBS: A Duplication Based Scheduling Algorithm for Heterogeneous Computing Systems," *Proc. Int'l Conf. Parallel Processing*, pp. 352-359, 2002.

[17] M. Silberstein, D. Geiger, A. Schuster, and M. Livny, "Scheduling Mixed Workloads in Multi-Grids: The Grid Execution Hierarchy," *Proc. IEEE Int'l Symp. High Performance Distributed Computing*, pp. 291-302, 2006.

[18] I. Ahmad and A. Ghafoor, "Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems," *IEEE Trans. Software Eng.*, vol. 17, no. 10, pp. 987-1004, Oct. 1991.

[19] B. Uar, C. Aykanat, K. Kaya, and M. Ikinci, "Task Assignment in Heterogeneous Computing Systems," *J. Parallel and Distributed Computing*, vol. 66, pp. 32-46, 2006.

[20] E. Heymann, M.A. Senar, E. Luque, and M. Livny, "Adaptive Scheduling for Master-Worker Applications on the Computational Grid," *Proc. 1st IEEE/ACM Int'l Workshop on Grid Computing*, pp. 214-227, 2000.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," *Proc. ACM Symp. Operating Systems Principles*, pp. 261-276, 2009.

[22] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor—A Distributed Job Scheduler," *Beowulf Cluster Computing with Linux*, T. Sterling, ed., MIT Press, Oct. 2001.

[23] The OASIS Research Team and ActiveEon Company, *ProActive Programming Reference Manual Version 5.3.2*. ActiveEon Company, in collaboration with INRIA, http://hudson.activeeon.com/job/ProActive_5.3.x/lastSuccessfulBuild/artifact/doc/built/Programming/ReferenceManual/multiple_html/index.html, accessed 2012.

[24] E.B. Nightingale, P.M. Chen, and J. Flinn, "Speculative Execution in a Distributed File System," *ACM Trans. Computer Systems*, vol. 24, no. 4, pp. 361-392, Nov. 2006.

[25] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: Improving Configuration Management with Operating System Causality Analysis," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles*, pp. 237-250, 2007.

[26] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modelling," *Proc. 6th Conf. Symp. Operating Systems Design & Implementation*, pp. 18-18, 2004.

[27] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal, "Dynamic Instrumentation of Production Systems," *Proc. Ann. Conf. USENIX Ann. Technical Conf.*, pp. 2-2, 2004.

[28] K. Yaghmour and M.R. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging," *Proc. Ann. Conf. USENIX Ann. Technical Conf.*, pp. 2-2, 2000.

[29] I. Park, "Event Tracing for Windows: Best Practices," *Proc. Computer Measurement Group Conf.*, pp. 565-574, 2004.

[30] L. Carrington, A. Snavely, and N. Wolter, "A Performance Prediction Framework for Scientific Applications," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 336-346, Feb. 2006.

[31] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Operating Systems Design and Implementation*, pp. 137-150, 2004.

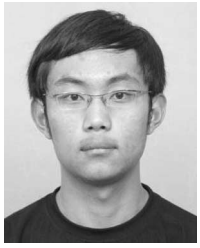
[32] S.N. Srirama, P. Jakovits, and E. Vainikko, "Adapting Scientific Computing Problems to Clouds Using MapReduce," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 184-192, Jan. 2012.

[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *SIGOPS Operating Systems Rev.*, vol. 41, no. 3, pp. 59-72, Mar. 2007.

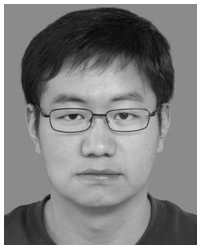
[34] J. Ekanayake, T. Gunarathne, and J. Qiu, "Cloud Technologies for Bioinformatics Applications," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 6, pp. 998-1011, June 2011.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences, Beijing, China in 2002. He is currently a professor of computer science and technology at Tsinghua University of China, Beijing, China. His research interests include parallel and distributed processing, and cloud storage. He has published over 80 research publications and has received two Best Paper Awards. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He can be reached at: wuyw@tsinghua.edu.cn.

Weichao Guo received the BE degree from Harbin Institute of Technology, China in 2011. He is a PhD student with Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include distributed systems, software reliable, and operating systems.



Jinglei Ren received the BE degree from Northeast Normal University, Changchun, China, in 2010. He is a PhD candidate with Department of Computer Science and Technology, Tsinghua University, Beijing, China. He has developed a storage system for virtual machines with enhanced manageability, and is currently working on the flash-aware and energy-efficient smartphone filesystem. His research interests include distributed systems, storage techniques, and operating systems.



Xun Zhao received the BE degree from Harbin Institute of Technology, China, in 2009. He is a PhD candidate with Department of Computer Science and Technology, Tsinghua University, China. He has developed an Urgent Scheduler for Xen virtual machine. His current work primarily involves VM scheduling on multi-core machines. His research interests include cloud computing, virtual machine scheduling, and distributed file systems.

Weimin Zheng received the BS and MS degrees from Tsinghua University, China, in 1970 and 1982, respectively, where he is currently a professor of Computer Science and Technology. He is the research director of the Institute of High Performance Computing at Tsinghua University, and the managing director of the Chinese Computer Society. His research interests include computer architecture, operating system, storage networks, and distributed computing.