Can Cloud Computing be Used for Planning? An Initial Study

Qiang Lu*, You Xu†, Ruoyun Huang†, Yixin Chen† and Guoliang Chen*

*University of Science and Technology of China, Heifei, Anhui, rczx@mail.ustc.edu.cn, glchen@ustc.edu.cn
†Washington University in St. Louis, St. Louis, MO, {yx2,ruoyun.huang,chen}@cse.wustl.edu

Abstract—Cloud computing is emerging as a prominent computing model. It provides a low-cost, highly accessible alternative to other traditional high-performance computing platforms. It also has many other benefits such as high availability, scalability, elasticity, and free of maintenance. Given these attractive features, it is very desirable if automated planning can exploit the large, affordable computational power of cloud computing. However, the latency in inter-process communication in cloud computing makes most existing parallel planning algorithms unsuitable for cloud computing.

In this paper, we propose a portfolio stochastic search framework that takes advantage of and is suitable for cloud computing. We first study the running time distribution of Monte-Carlo Random Walk (MRW) search, a stochastic planning algorithm, and show that the running time distribution usually has remarkable variability. Then, we propose a portfolio search algorithm that is suitable for cloud computing, which typically has abundant computing cores but high communication latency between cores. Further, we introduce an enhanced portfolio with multiple parameter settings to improve the efficiency of the algorithm. We implement the portfolio search algorithm in both a local cloud and the Windows Azure cloud. Experimental results show that our algorithm achieves good, in many cases superlinear, speedup in the cloud platforms. Moreover, our algorithm greatly reduces the running time variance of the stochastic search and improves the solution quality. We also show that our scheme is economically sensible and robust under processor failures.

Keywords—Cloud computing; automated planning; stochastic search;

I. INTRODUCTION

Graph search (such as depth first search and A*) is an important and pervasive technique for artificial intelligence (AI). Graph search has been employed by many AI techniques and applications, such as planning, scheduling, discrete optimization, game playing, and autonomous agents. A barrier that severely limits the applicability of AI technology is the high computational costs (in terms of time, memory, and storage) required by graph search algorithms. A natural way to improve the efficiency of search is to utilize advanced, more powerful computing platforms.

Cloud computing is emerging as a prominent computing model. It provides a low-cost, highly accessible alternative to other traditional high-performance computers. It allows small teams and even individual users to routinely have access to the same large-scale computing facilities as used by large companies and organizations, such as Amazon, Google, and Microsoft. Although cloud computing originated with a business root for the need of Internet based services, it is widely believed that cloud computing can also bring a revolution to scientific computing and applications. Given the high accessibility of cloud computing, if we can significantly improve the efficiency and quality of planning algorithm in the cloud, the cloud-based planning algorithms can be routinely used by all users and may fundamentally change the landscape of AI applications. This paper is an initial effort towards this direction.

In addition to accessibility, cloud computing has a number of advantages, including its elasticity, reliability, scalability, low set-up cost, and ease of maintenance. Such features make cloud computing a prominent and popular computing paradigm. We believe that cloud-based planning is important, beneficial, and feasible to be developed for the following reasons:

• Cloud computing offers an abundance of processors, memory, and storage which, when properly orchestrated, will likely significantly improve the speed and applicability of automated planning.

• Given the high accessibility and low cost of clouds, cloud computing is likely to become a prominent mode of computing for general users. Improving the speed of planning in the cloud is likely to have a more significant and broader impact than on other high-performance computing platforms, even if cloud computing leads to lower speedup for certain algorithms.

Although cloud computing has the potential to make planning much more efficient, there are also some key challenges that need to be addressed.

• The most important hurdle is the high latency in inter-process communication in cloud, due to the hardware implementation of cloud computing infrastructure. For example, recent measurement of communication in cloud systems shows much longer delays than MPI systems [1].

• Typically a job dispatched in a large-scale data center has a certain probability of failure, which might be caused by hardware failure or preemption in a shared environment.

Both challenges render it hard to use existing parallel search algorithms for planning. As we will see in a review of related work, most existing parallel search algorithms
are designed either for shared memory systems, which have limited scalability, or for distributed systems with low latency message passing mechanisms. In addition, one processor failure may cause the entire parallel algorithm to crash. Therefore, most existing parallel planning search strategies are not suitable for the cloud environment that are specifically designed to run on computer clusters with low-latency high throughput communication and assuming no node failure. These two challenges call for new parallel search strategies that are suitable for the cloud environment.

To address these challenges, we propose a portfolio search algorithm. In particular, we use Monte-Carlo Random Walk (MRW), a stochastic search algorithm for classical planning [2]. Our key observation is that some stochastic algorithms such as MRW exhibit high variability of their running time, which means running with different random seeds may have very different running time. Such a variability is attractive for cloud computing because even a simple scheme that launches parallel independent runs can have high (sometimes superlinear) speedup without requiring inter-processor communication.

In addition to the simple algorithm, we also develop an enhanced portfolio search algorithm with multiple parameter settings. In MRW, the parameters can greatly impact the search performance. By running a portfolio of searches with varying parameter settings, our scheme greatly enhances the possibility to get the parameter value that performs very well. Our experimental results show that this scheme further improves efficiency and solution quality significantly.

We further implement our algorithms in Windows Azure, a representative commercial cloud whose potential for scientific research remains largely unexplored. We study the performance characteristics of Windows Azure, and then develop a scalable Azure-based scheme for stochastic combinatorial search algorithms. We report experimental results to show the advantages of the proposed scheme, including high speedup, scalability, and reduced running time variance. We also show that our scheme is economically sensible.

The rest of this paper is organized as follows. In Section II, we briefly review related work on cloud computing, parallel search algorithms, and algorithm portfolio. In Section III, we give background on the MRW algorithm. In Sections IV and V, we study the MRW algorithm and its running time distribution, and propose a basic portfolio of MRW algorithm and an improved algorithm using multiple parameter settings. In Section VI, we study the performance characteristics of Windows Azure and present an implementation of the PMRW algorithm on it. We present our experimental results in Section VII and give conclusions in Section VIII.

II. RELATED WORK

A. Cloud computing

Cloud computing is becoming a popular computing paradigm in which dynamically scalable resources are provided as services over the Internet [3]. Cloud computing platforms have been used for scientific applications. Azure-Blast [4] studied the applicability of the Windows Azure cloud to the BLAST algorithm using a trivial parallelization with little communication due to the high communication latency. There is a work [5] that compared the utility of a supercomputer to that of public clouds, and analyzed the service times prior to actual execution. The conclusion is that while the supercomputer might be much faster, the turnaround time might actually be much better for the cloud because of the elapsed time from submission to the completion of execution, which gives another reason for our research.

B. Parallel search algorithms

Search is a key technique for planning. Early work on parallel search is surveyed in [6]. For shared memory systems, synchronized schemes such as layer synchronization and delayed duplicate detection, do not scale to very large amounts of processors; results with up to eight cores have been reported in [7]. For distributed memory architectures, inter-processor communication is generally needed to ensure efficiency and correctness [8]. Recently, the HDA* algorithm [9], based on asynchronous MPI communication, scales well to up to 128 processing cores. These parallel algorithms are not suitable for the cloud environment which has the high latency in inter-process communication and instance failures.

C. Portfolio search

A portfolio of algorithms is a collection of different algorithms and/or different copies of the same algorithm running in parallel on different processors or interleaved on one processor [10]. The portfolio idea has been applied to automated planning [11], SAT solver [12] and SMT solver [13]. Theoretical and experimental analyses show that portfolio search can significantly decrease variances of heavy-tailed distributions associated with SAT and constraint satisfaction solvers [14].

III. MONTE-CARLO RANDOM WALK

Stochastic search has been studied and applied to several areas of automated planning, such as sampling possible trajectories in automated planning [15][2], probabilistic planning [16] and robot motion planning [17]. The recent Monte-Carlo Random Walk (MRW) method is a stochastic search algorithm for classical STRIPS planning. MRW achieves comparable, and sometimes superior, performance to the best deterministic search algorithms in a number of testing domains [2]. It benefits from its exploration strategies.
Algorithm 1: MRW(II)

Input: a classical planning problem II
Output: a solution plan
1   s ← s_I;
2   h_min ← h(s_I);
3   counter ← 0;
4   while s does not satisfy s_G do
5       if counter > c^n or dead-end(s) then
6           s ← s_I;
7           h_min ← h(s_I);
8           counter ← 0;
9       s ← RandomWalk(s,II);
10      if h(s) < h_min then
11         h_min ← h(s);
12         counter ← 0;
13      else
14         counter ← counter + 1;
15   return plan;

The method is robust in presence of misleading heuristic estimates, since it obtains more information from the local neighborhood. Also, the random exploration can effectively escape from local minima.

Algorithm 1 shows the framework of MRW. Given a classical planning problem, MRW works as follows. At each step, it uses a RandomWalk() procedure to change the current state s to a neighboring state (Line 9). The RandomWalk() procedure tries a number (c^n) of paths, where each path is a random sequence of a number (c') of actions. RandomWalk() uses a heuristic function (such as FF [18]) to evaluate the ending state of each path and returns the best ending state out of the c^n paths. MRW search fails to find a solution when the minimum heuristic value is not improved in c^n iterations, or s ends up as a dead-end state (Line 5). In this case the MRW search simply restarts from the initial state s_I. The iteration stops when a state satisfies goal state s_G (Line 4), which means a solution is found.

IV. PORTFOLIO SEARCH WITH MRW

We choose MRW for planning in the cloud because it exhibits remarkable variability in the solving time of any particular problem instance. Such a variability is undesirable in practice. However, designing a portfolio of such algorithm gives a dramatic improvement in terms of overall performance [10]. In the portfolio of MRW, we combine independent, parallel executions of MRW with very limited communication. As we have discussed earlier, a key barrier of cloud computing is the high latency in inter-process communication. Hence, a scheme with little communication is highly desirable. We first study the running time distribution of MRW procedure in different planning domains. Then, based on the analysis of the distribution, we propose a parallel algorithm that can dramatically reduce the variance in the search behavior.

We extensively evaluate the running time distribution of MRW in planning domains that MRW can solve. For a given problem instance, we make 500 runs of MRW, each with a different random seed.

We show the running time of MRW (with its default parameter setting) and statistical results on six problem instances in Fig. 1 and Table I. We can see that two runs with different random seeds have significantly different running time. The results show that for all problems the running times of MRW have extremely large variances (often over one million). Specifically, it is common to observe that a MRW run with a different random seed solves the same instance much faster than another one. Such a large variability can benefit a portfolio scheme that makes multiple independent runs and terminates as soon as one run finds a solution.

Based on the above study of MRW running time distributions, we present a basic scheme for the cloud, shown as the parallel MRW (PMRW) procedure in Algorithm 2. It calls multiple (N) processors to run MRW independently with
different random seeds (Line 2). As soon as a processor finds a solution, all other processors will be halted (Line 4). Obviously, the solution time of PMRW is the minimum running time of the $N$ independent runs.

V. ENHANCED PMRW WITH MULTIPLE SETTINGS

In practice, it is usually impossible to get a parameter setting that is effective on all problem instances. While offline tuning may find the setting with the best average performance, it cannot guarantee that this setting will perform well on each individual problem. In this paper, we present an enhanced portfolio search with multiple parameter settings to deal with the parameter selection issue and further improve the performance without any increase of communication overhead.

The MRW algorithm has a few parameters affecting its performance, among which $c^0$ and $c^l$ are the most important since they directly control the process of escaping from extensive local minima and plateaus [2]. However, it is difficult to decide what is the best setting for a certain problem. Particularly, if $c^0$ and $c^l$ are too small, the local search method is greedy as it tries to immediately exploit their local knowledge instead of exploring the neighborhood of the current state. Following a misleading heuristic value may quickly lead to a much worse state than what could be achieved with a little more exploration. On the other hand, if they are too large, the search may take a long time on exploring the neighborhood of the current state. This exploitation-exploration trade-off has been extensively studied [19],[20]. One of the resulting search algorithms, UCT [20], has been successfully applied in many difficult domains.

In this paper, we use parallel multiple settings [11], a simple formulation of dovetailing [21],[12], to deal with the parameter selection issue. PMRW with multiple parameter settings (PMRW\textsuperscript{ms}) is a strategy that takes in a candidate configuration set $C = \{c_0, c_1, \ldots, c_n\}$ where $c_i$ is a configuration for the search algorithm, and assigns a configuration $c_i \in C$ to each processor. Each $c_i$ contains settings for $c^0$ and $c^l$, denoted as $c^0_i$ and $c^l_i$, respectively. Each processor $p_i$ ($i$ in $1 \cdots n$) then performs search independently and simultaneously using the setting $c_i$.

A. Parameter settings

MRW by default sets $c^0 = 2000$ and $c^l = 10$, which are tuned offline and give good average performance [2]. However, there is no guarantee that this setting will perform well on each individual problem. As an example, we test different parameter values on a randomly selected problem Airport-17. For each parameter setting, we run MRW 10 times to get the average running time. The experimental results presented in Fig. 2 show that the performance with the default setting can be improved by arranging some other parameter settings. Based on a large number of experimental tests, we observe that a reasonable range for $c^0$ is $[200, 3000]$ and for $c^l$ is $[1, 15]$. These ranges are large enough to include all possible good settings based on our experiments. Specifically, we set each $c_i$ as follows. For each processor $p_i$, we set $c^0_i$ as $\left\lfloor 200 + \frac{15}{n-1} \right\rfloor$ and $c^l_i$ as $\left\lfloor 1 + \frac{15}{n-1} \right\rfloor$.

VI. IMPLEMENTATION IN WINDOWS AZURE

Windows Azure is a representative cloud computing system [22]. It provides a Windows-based cloud computing environment for running applications and storing data on servers in Microsoft data centers. Windows Azure offers web roles and worker roles that can be used for web hosting and computation in the queue respectively. It also provides communication mechanisms like Queue where web and worker roles can use to communicate with each other using asynchrized APIs. It is important to note that the latency of communication using Azure Queue is high. On average, adding or retrieving a message of 512 bytes in the queue requires as long as 20 ms, as measured in [1]. We also performed some test by ourselves and found that Windows Azure Queue can support at most 100 message insertion and retrievals using its RESTful API. Therefore, it is not practical to use parallel algorithms that are designed for high performance clusters with low latency communication.
A general framework for deploying computing applications in Windows Azure.

A general framework that we use to run PMRW algorithm in Windows Azure is presented in Fig. 3. It is a simplified version of the architecture used in AzureBlast [4]. We use a web role to provide a web portal for users to submit jobs. The web role dispatches jobs to worker roles using a job control queue. For the PMRW algorithm with \( N \) processors, a web role will insert \( N \) messages into the job control queue, each containing a job ID, a set of initial parameters and a pointer to the planning problem. Multiple worker roles will be launched and all will seek tasks from the job control queue to start independent searches. Messages will be popped by worker roles to initialize MRW search algorithms. Once the MRW search on a worker role finds a solution, it sends a done message, alone with the job ID, to the job control queue. Other worker roles that are working on the same planning problem, identified by the job ID, by checking the job control queue asynchronously and periodically, will then terminate promptly once they see the done message. The finished worker role will also send its solution and running time to the result queue for result collection. Such a design leads to minimum communication, while still giving prompt termination of the PMRW algorithm once a solution plan is found.

VII. EXPERIMENTAL RESULTS

In this section, we use two approaches to evaluate performances for three algorithms, i.e., MRW, PMRW and PMRW\textsuperscript{ms}. Same as the original MRW paper, we test all domains in IPC-4 [23]. For each domain, we present the two largest problems MRW can solve at least once in 10 independent runs, with a time limit of 3600 seconds for each run. These domains are Airport (air), Pipesworld Tankage (tank), Pipesworld NoTankage (notank), Philosophers (phi), Satellite (sat) and Power Supply Restoration-PSR Large (psr). For MRW, we use its default setting, which was tuned to give the best overall performance across different domains.

In the first evaluation, we report the average running time, standard deviation, speedup, and solution length based on a large number of runs in a local cloud. Second, we report results in Windows Azure.

A. Evaluation in a local cloud

We report results in a local cloud. We use the local cloud since it is free and we can make extensive testing on it to accurately measure the mean and variance of the running time of our algorithms. For comparison, we also test Fast Downward [24] and LAMA [25], two state-of-the-art planners. The local cloud contains 112 CPUs and 400Gb memory in total. Due to the limitation on the number of processors in our local cloud, we use multiple independent runs of MRW algorithms to simulate the running of PMRW. Namely, we pick the shortest time of \( N \) independent runs as the running time of PMRW on running \( N \) processors. This estimation reflects the actually running time of PMRW algorithms correctly for the two reasons listed below. First, in both PMRW and PMRW\textsuperscript{ms}, searches are conducted independently and in parallel. In other words, there is no communication that changes the running time of either algorithm. Second, our experiments show that for both clusters and in the cloud, the overhead of controlling \( N \) parallel, independent MRW or MRW\textsuperscript{ms} algorithms is negligible.

Since both PMRW and PMRW\textsuperscript{ms} terminate promptly once a solution is found, for \( N \) parallel searches, the shortest running time among these \( N \) searches is almost identical to the actual running time of these algorithm portfolios.

For each of the 12 problems, we conduct a large number of independent runs of MRW to estimate the expected running time for both PMRW and PMRW\textsuperscript{ms} on 4, 8, 16, 32, 64 or 128 processors. The average running times of the PMRW
algorithms with $N$ processors are computed as follows: 1) Make $N$ independent runs with different random seeds; let $t$ be the minimal running time of these $N$ samples. If a sample run times out after 3600 seconds, we repeat until a successful run is made and use the total time for that sample. 2) Repeat the sampling until the mean and the variance of the successful run is made and use the total time for that sample.

The mean of running time converges when

$$|\mu(T^m) - \mu(T^{m-1})| < \tau,$$

whereas the variance converges when

$$|\sigma^2(T^m) - \sigma^2(T^{m-1})| < \tau'.$$

Here we set $\tau = 0.001$ and $\tau' = 0.1$ seconds.

We give the results of our algorithms, Fast Downward (FD) and LAMA in Fig. 4, Tables II, and III. From Figures 4(a), 4(b), and 4(c), we can see that both PMRW and PMRW$^ms$ largely reduce the running times and the standard deviation, and provide large speedups.

Tables II and III give more detailed results. Columns 2-3 of Table II are the average running time and standard deviation of sequential MRW search ($N=1$). In problems such as air-46, air-47, phi-12, phi-13 and psr-24, they achieve super linear speedups (e.g. 487.5 on phi-13 of PMRW$^{ms}$). Comparing to PMRW, PMRW$^{ms}$ improves the efficiency by getting even smaller running time and hence a larger speedup. More importantly, PMRW and PMRW$^{ms}$ largely reduce the standard derivation of running time, which makes them a much more predictable and hence favorable choice over the original MRW algorithm. MRW tends to have very large running time variance, while PMRW and PMRW$^{ms}$ are much more stable. We also see that PMRW and PMRW$^{ms}$ scale well even with 128 processors.

We compare MRW, PMRW$^{ms}$, FD, and LAMA in Table III. In addition to running time and variance, we also report the length of the solution plans from these algorithms (Fig. 4(d) and Table III). For PMRW$^{ms}$, we report the average solution length. We make the following observations. 1) Comparing to MRW, PMRW$^{ms}$ not only greatly reduces running time and variance, but also largely reduces the solution length. For example, PMRW$^{ms}$ with 128 cores finds shorter plan than MRW does on all problems, and nearly halves the length in many cases. This may be explained since MRW is a random walk. The quickest run found by PMRW$^{ms}$ tends to have explored a shorter path. 2) Comparing to FD and LAMA, PMRW$^{ms}$ is also very competitive in terms of running time and solution length. For example, for air-46, PMRW$^{ms}$ can find shorter plan and use less time than FD and LAMA do. For phi-12 and phi-13, PMRW$^{ms}$ achieves similar quality while taking much less time than FD and LAMA.

Figure 5. Results of PMRW algorithm in Windows Azure, including the summation of the running time and the financial costs, and the average speedup.

B. Evaluation in Windows Azure

We also evaluate both PMRW and PMRW$^{ms}$ using Windows Azure, one of the major commercial cloud computing platforms [22]. For the following evaluation, we request up to 120 processors in Windows Azure. Given the time and resource limit, for each instance and setting, we make 10 runs and calculate the average running time and cost.

Financial cost is a major concern for cloud users. Cloud computing adopts a pay-as-you-go model for computational resources. Therefore, although theoretically we can employ a large amount of processors for our algorithm portfolios, it is necessary to consider the tradeoff between speedup and cost.

We report the solution time, monetary cost, and speedup in Fig. 5 and Table IV. The cost is calculated based on a unit cost of $0.12 per hour per CPU core, which is the standard rate for small instances in Windows Azure.

From Figures 5(a) and 5(c), we see that the results of our algorithms in Windows Azure are similar to the results from our local cloud, which largely reduce the running times and provide large speedups. For each instance, we report the detailed average time and cost in Table IV. Comparing to PMRW, PMRW$^{ms}$ largely improves the efficiency and also reduces the total monetary cost since it finishes sooner. For example, for phi-13 with 120 processors, not only can PMRW$^{ms}$ achieve a great speedup, the financial charge (2.7 US cents) is also the lowest among all reported $N$s. Hence, for this problem, it is economical to use 120 processors instead of fewer processors.

For other problems, our scheme also achieves good tradeoff. For example, for air-47 and psr-25, to increase the number of processors from 16 to 120 only increases the total charge by less than 50%, but improves the speedup.
Table II

<table>
<thead>
<tr>
<th>P</th>
<th>MRW 4</th>
<th>MRW 8</th>
<th>MRW 16</th>
<th>MRW 32</th>
<th>MRW 64</th>
<th>MRW 128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>S</td>
<td>T</td>
<td>S</td>
<td>T</td>
<td>S</td>
</tr>
<tr>
<td>air-46</td>
<td>6039.9</td>
<td>5972.2</td>
<td>1598.3</td>
<td>1461.5</td>
<td>8.6</td>
<td>8.6</td>
</tr>
<tr>
<td>air-47</td>
<td>9310.3</td>
<td>9144.6</td>
<td>2503.9</td>
<td>2259.7</td>
<td>3.7</td>
<td>3.7</td>
</tr>
<tr>
<td>tank-40</td>
<td>1134.7</td>
<td>1036.4</td>
<td>429.4</td>
<td>205.4</td>
<td>2.6</td>
<td>2.6</td>
</tr>
<tr>
<td>tank-41</td>
<td>43.0</td>
<td>28.8</td>
<td>24.5</td>
<td>6.7</td>
<td>20.7</td>
<td>2.2</td>
</tr>
<tr>
<td>tank-42</td>
<td>84.5</td>
<td>67.9</td>
<td>44.7</td>
<td>11.7</td>
<td>9.3</td>
<td>2.2</td>
</tr>
<tr>
<td>tank-43</td>
<td>286.3</td>
<td>383.4</td>
<td>91.9</td>
<td>38.9</td>
<td>3.1</td>
<td>76.0</td>
</tr>
<tr>
<td>sate-29</td>
<td>1119.4</td>
<td>1119.4</td>
<td>694.9</td>
<td>179.3</td>
<td>1.7</td>
<td>55.1</td>
</tr>
<tr>
<td>sate-30</td>
<td>2030.5</td>
<td>1209.1</td>
<td>1221.3</td>
<td>300.8</td>
<td>1.7</td>
<td>1051.5</td>
</tr>
<tr>
<td>phi-12</td>
<td>1236.6</td>
<td>1210.3</td>
<td>325.1</td>
<td>319.5</td>
<td>3.8</td>
<td>164.5</td>
</tr>
<tr>
<td>phi-13</td>
<td>5011.2</td>
<td>5039.1</td>
<td>1256.8</td>
<td>1219.3</td>
<td>4.0</td>
<td>565.3</td>
</tr>
<tr>
<td>phi-24</td>
<td>1463.0</td>
<td>1459.4</td>
<td>396.7</td>
<td>347.6</td>
<td>3.7</td>
<td>220.6</td>
</tr>
<tr>
<td>phi-25</td>
<td>65.3</td>
<td>57.8</td>
<td>18.5</td>
<td>12.3</td>
<td>3.3</td>
<td>12.6</td>
</tr>
</tbody>
</table>

Comparison of the running time, standard deviation and the speedup of MRW algorithm. ‘T’ is the average running time in seconds, ‘σ’ is the standard deviation and ‘S’ is the speedup.

Table III

<table>
<thead>
<tr>
<th>P</th>
<th>MRW 4</th>
<th>MRW 8</th>
<th>MRW 16</th>
<th>MRW 32</th>
<th>MRW 64</th>
<th>MRW 128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>S</td>
<td>T</td>
<td>S</td>
<td>T</td>
<td>S</td>
</tr>
<tr>
<td>air-46</td>
<td>176.2</td>
<td>177.0</td>
<td>82.5</td>
<td>82.5</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>air-47</td>
<td>164.9</td>
<td>450.0</td>
<td>90.3</td>
<td>334.7</td>
<td>72.5</td>
<td>50.1</td>
</tr>
<tr>
<td>tank-40</td>
<td>177.0</td>
<td>114.8</td>
<td>50.3</td>
<td>54.6</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>tank-41</td>
<td>49.5</td>
<td>13.7</td>
<td>5.1</td>
<td>47.2</td>
<td>10.4</td>
<td>4.1</td>
</tr>
<tr>
<td>tank-42</td>
<td>177.0</td>
<td>68.3</td>
<td>114.3</td>
<td>124.2</td>
<td>6.4</td>
<td>6.4</td>
</tr>
<tr>
<td>tank-43</td>
<td>211.1</td>
<td>35.2</td>
<td>22.8</td>
<td>181.2</td>
<td>25.6</td>
<td>13.3</td>
</tr>
<tr>
<td>sate-29</td>
<td>667.7</td>
<td>1806.7</td>
<td>632.7</td>
<td>346.4</td>
<td>57.9</td>
<td>31.5</td>
</tr>
<tr>
<td>sate-30</td>
<td>679.3</td>
<td>1924.4</td>
<td>715.4</td>
<td>617.5</td>
<td>72.6</td>
<td>56.9</td>
</tr>
<tr>
<td>phi-12</td>
<td>1487.4</td>
<td>1541.0</td>
<td>887.2</td>
<td>330.9</td>
<td>90.9</td>
<td>32.2</td>
</tr>
<tr>
<td>phi-13</td>
<td>159.9</td>
<td>749.0</td>
<td>59.0</td>
<td>429.9</td>
<td>50.9</td>
<td>39.2</td>
</tr>
<tr>
<td>phi-24</td>
<td>89.9</td>
<td>3169.4</td>
<td>8.5</td>
<td>436.8</td>
<td>8.4</td>
<td>43.8</td>
</tr>
<tr>
<td>phi-25</td>
<td>78.9</td>
<td>16.8</td>
<td>11.6</td>
<td>5.4</td>
<td>8.7</td>
<td>7.3</td>
</tr>
</tbody>
</table>

Comparison of the results of MRW, Fast Downward (FD), and LAMA. ‘T’ is the average running time in seconds, ‘σ’ is the standard deviation, ‘S’ is the speedup, and ‘L’ is the solution length. ‘-’ means it does not find a solution in 3600 seconds and ‘NA’ means LAMA cannot parse the problem. The time and variance of MRW are the same as Table II.

Table IV

<table>
<thead>
<tr>
<th>P</th>
<th>MRW 4</th>
<th>MRW 8</th>
<th>MRW 16</th>
<th>MRW 32</th>
<th>MRW 64</th>
<th>MRW 128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>S</td>
<td>T</td>
<td>S</td>
<td>T</td>
<td>S</td>
</tr>
<tr>
<td>air-46</td>
<td>6999.7</td>
<td>6624.9</td>
<td>1240.4</td>
<td>1155.9</td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td>air-47</td>
<td>10812.5</td>
<td>9231.5</td>
<td>1066.1</td>
<td>1028.4</td>
<td>3.8</td>
<td>3.8</td>
</tr>
<tr>
<td>tank-40</td>
<td>1355.4</td>
<td>631.2</td>
<td>512.7</td>
<td>28.8</td>
<td>6.4</td>
<td>6.4</td>
</tr>
<tr>
<td>tank-41</td>
<td>53.0</td>
<td>18.8</td>
<td>35.6</td>
<td>1.4</td>
<td>5.5</td>
<td>5.5</td>
</tr>
<tr>
<td>tank-42</td>
<td>96.7</td>
<td>59.3</td>
<td>16.6</td>
<td>0.8</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>tank-43</td>
<td>367.6</td>
<td>123.0</td>
<td>119.3</td>
<td>3.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>sate-29</td>
<td>1234.5</td>
<td>412.0</td>
<td>676.0</td>
<td>18.0</td>
<td>9.0</td>
<td>9.0</td>
</tr>
<tr>
<td>sate-30</td>
<td>2181.5</td>
<td>727.2</td>
<td>909.8</td>
<td>24.1</td>
<td>12.1</td>
<td>12.1</td>
</tr>
<tr>
<td>phi-12</td>
<td>1385.0</td>
<td>460.6</td>
<td>27.6</td>
<td>224.3</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>phi-13</td>
<td>5156.7</td>
<td>1217.2</td>
<td>919.5</td>
<td>56.0</td>
<td>12.3</td>
<td>12.3</td>
</tr>
<tr>
<td>phi-24</td>
<td>17796.9</td>
<td>5491.5</td>
<td>365.6</td>
<td>67.7</td>
<td>9.2</td>
<td>9.2</td>
</tr>
<tr>
<td>phi-25</td>
<td>73.7</td>
<td>234.3</td>
<td>28.2</td>
<td>0.6</td>
<td>4.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Comparison of the running time and cost of MRW algorithms using different number of nodes in Windows Azure. ‘T’ is the running time in seconds, ‘S’ is the speedup and ‘C’ is the average total cost in US cents.

by almost 4 times. Therefore, using more processors is still desirable when users are willing to trade a slightly higher cost for a significant speedup. After all, the total charge in a cloud is not very expensive using our algorithm. For the different numbers of processors we tested, it takes less than a US dollar to solve most instances.

Finally, we point out that, our algorithm is robust under processor failures, which are commonly seen in cloud environments. For MRW and MRW*, a failed run does not affect other parallel runs since they are independent and do not require communication. Previous parallel planning algorithms requiring intricate coordination are much more
vulnerable to processor failures.

VIII. CONCLUSIONS

In this paper, we proposed a portfolio search algorithm that takes advantage of and is suitable for cloud computing. We first studied the running time distribution of the MRW search algorithm for planning and showed that its running time distribution usually has remarkable variability. Then, we proposed a portfolio of MRW algorithm that is suitable for cloud computing and an enhanced algorithm with multiple settings to further improve the efficiency of the algorithm. We implemented our portfolio of MRW algorithm in a local cloud and the Windows Azure platform.

Our experimental results showed that our algorithm achieves good, in many cases superlinear, speedup in the cloud platforms and greatly reduces the running time variability of the stochastic search which makes the performance of our algorithm much more stable and predictable. Our algorithm scales well to large number of processors. It also compares competitively against some state-of-the-art heuristic search planners, in both running time and solution quality. The proposed algorithm is economically sensible in clouds and robust under processor failures. These results suggest that our algorithm can potentially be a practical option for fast planning on the cloud.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.61033009 and No.60873210), China Scholarship Council, NSF grants IIS-0535257, DBI-0743797, IIS-0713109, and Microsoft Research New Faculty Fellowship.

REFERENCES


