Iterative Resource Allocation for Memory Intensive Parallel Search Algorithms on Clouds, Grids, and Shared Clusters

Alex Fukunaga
The University of Tokyo

Akihiro Kishimoto
Tokyo Institute of Technology

Adi Botea[*]
IBM Research, Dublin, Ireland

Abstract

The increasing availability of "utility computing" resources such as clouds, grids, and massively parallel shared clusters can provide practically unlimited processing and memory capacity on demand, at some cost per unit of resource usage. This requires a new perspective in the design and evaluation of parallel search algorithms. Previous work in parallel search implicitly assumed ownership of a cluster with a static amount of CPU cores and RAM, and emphasized wall-clock runtime. With utility computing resources, trade-offs between performance and monetary costs must be considered. This paper considers dynamically increasing the usage of utility computing resources until a problem is solved. Efficient resource allocation policies are analyzed in comparison with an optimal allocation strategy. We evaluate our iterative allocation strategy by applying it to the HDA* parallel search algorithm. The experimental results validate our theoretical predictions. They show that, in practice, the costs incurred by iterative allocation are reasonably close to an optimal (but a priori unknown) policy, and are significantly better than the worst-case analytical bounds.

1 Introduction

Cloud computing resources such as Amazon EC2, which offer computational resources on demand, have become widely available in recent years. In addition to cloud computing platforms, there is an increasing availability of massive-scale, distributed grid computing resources such as TeraGrid/XSEDE, as well as massively parallel, high-performance computing (HPC) clusters. These large-scale *utility computing resources* share two characteristics that have significant implications for parallel search algorithms. First, vast (practically unlimited) aggregate, memory and CPU resources are available on demand. Secondly, resource usage incurs a direct monetary cost.

Previous work on parallel search algorithms has focused on *makespan*: minimizing the runtime (wall-clock time) to find a solution, given fixed hardware resources; and *scalability*: as resource usage is increased, how are makespan and related metrics affected? However, the availability of virtually unlimited resources at some cost introduces a new context for parallel search algorithm research where an explicit consideration of cost-performance tradeoffs is necessary. For scalable algorithms, it is possible to reduce the makespan by allocating more resources (up to some point). In practice, this incurs a high cost with diminishing marginal returns. For parallel A* variants, under-allocating resources results in memory exhaustion. On the other hand, over-allocation is costly and undesirable.

It is often noted that memory-intensive search algorithms such as A* will run out of memory long before time is exhausted. With the vast amounts of aggregate memory available in utility computing, the *cost* (monetary funds) can be the new limiting factor, since one can exhaust funds long before allocating all of the memory resources available from a large cloud service provider.

We consider cost-efficient strategies for dynamically allocating utility computing resources. After an overview of utility computing and parallel search (Sections 2–3), we propose and analyze iterative allocation, a simple strategy that repeatedly runs a search algorithm with increasing resources until the problem is solved (Section 4). Bounds on the cost incurred by iterative allocation, compared to the optimal cost, are derived (Section 5). For a realistic class of utility computing environments and search problems, the cost suboptimality of our policy is bounded by a constant factor as small as 4. That is, we will never pay more than 4 times the a priori unknown optimal price.

We validate our analysis experimentally by applying iterative allocation to the HDA* (Kishimoto, Fukunaga, and Botea 2009) algorithm (Section 6). Results on classical planning and multiple sequence alignment problems, run on 3 distinct, massively parallel computing environments, indicate that the costs incurred by iterative allocation are reasonably close to optimal, and significantly better than the worst-case upper bounds.

2 Utility Computing Services

There are several types of utility computing services, including clouds, grids, and shared, massively parallel clusters. In all of these utility computing services, there is some notion of an atomic unit of resource usage. In general it is not possible to request arbitrary amounts of resources from a utility computing service (e.g., "3 CPU cores and 2345MB

[*]Part of this work was performed when this author was affiliated with NICTA and The Australian National University.

RAM"). Instead, hardware resources are allocated in discrete units, typically called "instances" or "nodes", which correspond to actual or virtual machine units.

Definition 1 (Hardware Allocation Unit)**.** *A hardware allocation unit (HAU), is the minimal, discrete resource unit that can be requested from a utility computing service.*[1] *It is characterized by a specific number of CPU cores and a given amount of RAM, e.g., 4 cores and 8GB.*

Various HAU types can be available, each with different performance characteristics and cost.[2]

In general, commercial clouds such as EC2 tend to have an immediate HAU allocation model with discrete charges, while grids and shared clusters tend to be batch-job based with a continuous cost model. We describe these cost models below.

2.1 Immediate/Dynamic Hardware Allocation vs. Batch Job Submission

Typical cloud services such as EC2 provision HAUs to the user immediately upon request – there is a delay, usually within 2 minutes, while the user's VM image is allocated, loaded and booted (Iosup et al. 2011). After the VM starts up, the user is free to use the system interactively as desired. Usage charges apply from the time that the allocated VM enters a "running" state to when it is stopped/terminated, regardless of the portion of time spent on actual computations. HAUs can be dynamically added to/removed from a running cluster of allocated HAUs.

In contrast, in HPC clusters and grids, users typically submit *jobs* to a centralized scheduler (resource manager), where a job is a request (script) specifying an executable program and the amount of resources to use. The scheduler decides when the submitted job actually executes (Feitelson et al. 1997). Usage charges apply for the time consumed by the user's job. In this type of model: (1) there are no guarantees about (nor fine-grained control over) when a job will actually get executed (it depends on congestion of the job queue and resource availability), and (2) jobs are independent binary executions - although some scheduling systems allow dependencies to be specified among submitted jobs, e.g., "job A must be executed after job B", communication of information between jobs must be through the file system.

2.2 Continuous Cost vs. Discrete Cost Models

In a *continuous cost model*, the cost of resource usage is a linear function of the amount of resources used. Batch job based systems such as typical grid and shared cluster environments usually adopt a continuous cost model.

On the other hand, currently, three of the largest commercial cloud service providers (Amazon EC2, Windows Azure, Google App Engine) all apply a *discrete cost model* in which all charges are per "HAU hour". Usage of a HAU for any fraction of an hour is rounded up, e.g., a 1 hour, 1 second allocation of a HAU which costs \$0.68/hr will cost \$1.36. There are economic and technical reasons for the use of coarse grained, discrete units (e.g., HAU-hour) rather than fine-grained units. For example, in contrast to a HPC cluster where jobs are executed in a shared environment, cloud services such as EC2 provision virtual machine (VM) environments running user-specific operating systems and applications. Fine-grained cost policies would encourage users to frequently start/stop VMs, straining the cloud service provider's infrastructure.

3 Background: Distributed, Parallel Search

In this paper, we focus on parallel search algorithms for distributed memory systems. In particular, we focus on decentralized, parallel systematic search algorithms which distribute the search space among the processors, and are guaranteed to eventually find a solution, given enough resources. In this framework for distributed search, the root process initially generates some seed states (using sequential search) and assigns them to the available processors. Then, at each processor, a locally executed search algorithm begins to explore the descendants of its seed state. Various approaches to managing the distribution of work among the processors have been proposed with two main goals: (1) effective load balancing, and (2) avoiding duplicate work among processors.

Work stealing is one standard approach which moves work from busy processors to idle processors. Each processor maintains a local work queue. When processor P generates new work (i.e., new states to be expanded) w, it places w in Q_P, the local work queue for P. When Q_P is empty, P "steals" work from $Q_{P'}$, the work queue of some other processor P'. This basic strategy can be applied to depth-first search algorithms, including simple depth-first search, branch-and-bound, and IDA*, as well as breadth-first and best-first search algorithms such as A* (Rao and Kumar 1987; Powley, Ferguson, and Korf 1993; Mahanti and Daniels 1993).

While work-stealing is based on "pulling" work from other processors, an alternative is "pushing" work to other processors. Algorithms based on *hash-based work distribution* have a global hash function which maps each state in the search space to one "owner processor". Each processor has a local open/closed list. When a state is generated at any processor, its owner is computed using the hash function, and the state is sent to the open list of its owner processor. If the hash function is well-behaved, and the time to generate/evaluate a state is uniform, hash-based work distribution achieves uniform load balancing. Hash-based work distribution has been applied to A* and IDA* variants (Evett et al. 1995; Mahapatra and Dutt 1997; Kishimoto, Fukunaga, and Botea 2009; Romein et al. 2002).

[1]Although the industry standard term for a HAU is "instance" or "node", we use HAU to avoid confusion with problem instances and search nodes.

[2]Amazon EC2 has 13 instance types. They range from "Small" (a virtual machine with 1.7GB RAM and 1 "virtual core" potentially timesliced/shared with other users, costing \$0.09USD/hour), to "Cluster Compute Eight Extra Large", a dedicated (exclusive use, non-shared) machine with 8 cores and 23GB RAM, costing \$1.70USD/hour.

4 Iterative Allocation for Ravenous Algorithms

A scalable, *ravenous algorithm* is an algorithm (1) which can be executed on an arbitrary number of processors, and (2) whose memory consumption continually increases until either a solution is found, or the algorithm terminates (fails) due to memory exhaustion. For example, HDA* (Kishimoto, Fukunaga, and Botea 2009), a parallel variant of A* based on hash-based work distribution (see above), is a scalable, ravenous algorithm.

In contrast, Transposition-Driven Scheduling (TDS) (Romein et al. 2002), a parallel version of IDA* with a transposition table (Reinefeld and Marsland 1994), is an *any-space* algorithm. Additional memory tends to result in faster search, but beyond some minimal threshold, memory exhaustion does not result in algorithm termination.

Algorithm 1 Generic, Iterative Allocation (IA)

```
numHAUs ← 1
while true do
    result ← Run algorithm $a$ on problem $p$
    if result = solved then
        return success
    else
        numHAUs ← Increase(numHAUs)
```

The *iterative allocation* (IA) strategy outlined in Algorithm 1 repeatedly runs a ravenous algorithm a until the given problem is solved. This is very simple, but the key detail is the `Increase()` function, which decides the number of HAUs to allocate in the next iteration. We seek a policy for choosing the number of HAUs allocated on each iteration of IA which tries to minimize the total cost.

4.1 Analysis of Iterative Allocation: Preliminaries

We present formal properties of the generic allocation policy. For formal analysis, we make two assumptions.

Assumption 1 (Homogeneous hardware allocation units)**.** *All HAUs used by IA are identical hardware configurations.*

While it is possible to mix and match HAUs in some utility computing environments, the use of heterogeneous hardware configurations is beyond the scope of this paper.

Assumption 2 (Monotonicity)**.** *If a problem is solved on i HAUs, then it will be solved on $j > i$ HAUs.*

While not always explicitly stated, monotonicity is usually assumed in the previous work on parallel search. Otherwise, the only way to ensure completeness of a parallel search algorithm would be to increment the number of HAUs 1 by 1, until the problem is eventually solved. This can be very inefficient in cases when a problem requires a large number of HAUs to be solved. In practice, if a problem fails on a given number of HAUs, the sensible follow-up step is to increase the resources (especially RAM, in the case of ravenous algorithms). On the other hand, there may be pathological cases where, on some problems, the search overhead (states which are generated using $j > i$ processors but not by i processors) incurred when using additional CPUs could grow faster than the extra RAM made available by using extra HAUs.

The cost to solve a problem is defined in terms of HAU-hours. When the problem at hand can be solved on v HAUs, let T_v be the makespan time needed to solve the problem. In a *continuous cost model*, common in shared HPC clusters (Sec. 2.2), the cost of solving the problem on v HAUs is defined as $T_v \times v$.[3] In a *discrete cost model*, common among commercial cloud services (Sec. 2.2), the cost is $\lceil T_v \rceil \times v$. In the rest of the paper, unless the cost model (continuous vs discrete) is explicitly stated or clear from the context, our statements apply to both models.

Definition 2. *The* minimal width W^+ *is the minimum number of HAUs that can solve a problem with a given ravenous algorithm. The cost incurred by using W^+ HAUs is denoted C^+.*

Definition 3. *Given some cost model, the* minimal cost width W^* *is the number of HAUs that results in a minimal cost. When several width values achieve the minimal cost, W^* refers to the smallest such width. The minimal cost (i.e., the cost to solve the problem using W^* HAUs) is written as C^*.*

If W^* is known a priori, a cost-optimal strategy for solving the problem at hand is trivial: rent W^* HAUs in parallel until the problem is solved. Most of time, however, W^+ or W^* are not known a priori. Thus, the best we can hope for is to develop strategies that approximate the optimal values.

Definition 4 (Nonincreasing search efficiency)**.** *Given a ravenous algorithm, we say that the search efficiency is nonincreasing if the following two conditions hold: (1) If n_v is the number states generated using v HAUs, then $n_v \leq n_{v+1}, \forall v \geq W^+$, and (2) $T_v v \leq T_{v+1}(v+1), \forall v \geq W^+$.*

As the number of HAUs increases, search efficiency typically decreases because of factors such as an increased search overhead and communication overhead. While superlinear speed-ups can occur, they are less common in ravenous algorithms and often indicate an inefficiency in the underlying, serial version of the algorithm. In any-space search algorithms, such as IDA* with transposition tables and TDS, super-linear speed-ups are more common, due to the fact that extra memory used as a transposition table can directly speed up the search. In this paper, however, the focus is on ravenous, memory-intensive algorithms.

Proposition 1. *In a continuous cost model, if the search efficiency is nonincreasing, $C^* = C^+$ and $W^* = W^+$.*

Proof. $C^* \leq C^+$, by the definition of C^*. From Definition 4 and the fact that $W^* \geq W^+$, $T_{W^*} \times W^* \geq T_{W^+} \times W^+$, which can be re-written as $C^* \geq C^+$. It follows that $C^* = C^+$, and thus, $W^* = W^+$. □

In a discrete-cost model, the min-width cost C^+ is not necessarily the same as the minimal cost C^*. For example, suppose that running a ravenous algorithm using 1 HAU will exhaust memory, 2 HAUs can solve the problem in 1.3 hours

[3]Without loss of generality, we assume the cost per HAU-hour to be 1, slightly simplifying the formulas.

(which is rounded up to 2 hours), and 3 HAUs can solve the problem in 1 hour. In this case, the min-width is $W^+ = 2$, $C^+ = 4$, but the min-cost $C^* = 3$. However, the gap between C^* and C^+ is relatively small:

Proposition 2. *In a discrete cost model, if the search efficiency is nonincreasing, $C^+ < C^* + W^+$.*

Proof. $C^+ = \lceil T_{W^+} \rceil W^+$ and $C^* = \lceil T_{W^*} \rceil W^*$. According to Definition 4, we have that $T_{W^+} \times W^+ \leq T_{W^*} \times W^*$. This leads to $C^+ = \lceil T_{W^+} \rceil W^+ = (T_{W^+} + r)W^+ \leq T_{W^*} \times W^* + rW^+ < \lceil T_{W^*} \rceil W^* + W^+ = C^* + W^+$. $\square$

Later, we shall see that under realistic conditions, $W^* = W^+$ and $C^* = C^+$ in a discrete cost model, regardless of whether search efficiency is nonincreasing (Sec. 5.1).

Definition 5. *The* max iteration time *E is the maximum actual (not rounded up) time that an iteration can execute before at least 1 HAU exhausts memory and fails.*

The previous cost definitions are based on a fixed number of HAUs. On the other hand, IA varies the number of HAUs dynamically. In a continuous cost model, assuming that IA allocates v_i HAUs at iteration i, the cost incurred by IA is $I = \sum_{i=0}^{j} D_{v_i} v_i$. D_{v_i}, with $i < j$, is the time taken to fail for all but the last iteration. $D_{v_j} = s_{v_j}$ is the time to successfully complete the last iteration. In all cases, $D_{v_i} \leq E$. In a discrete cost model, times spent by individual HAUs are rounded up. HAUs will use any spare time left at the end of one iteration to start the next iteration. Thus, for each HAU used in solving a problem, the total time spent by that HAU across all iterations where it participated is rounded up. The next example clarifies this further.

Example 1 (Cost computation in discrete cost model)**.** *A doubling IA is executed: iteration 0 uses 1 HAU, iteration 1 uses 2 HAUs, iteration 2 uses 4 HAUs. In other words, one HAU is continuously used in all 3 iterations. One HAU is used in the last two iterations, and two HAUs participate only to the last iteration. The cost is \$1 per HAU-hour. Each failed iteration requires* .33 *hours. The last, successful iteration requires* .25 *hours. The total cost is:* 1 *HAU* $\times \lceil .33 + .33 + .25 \rceil$ *hours* $\times$ *\$1* $+ 1$ *HAU* $\times \lceil .33 + .25 \rceil$ *hrs* $\times$ *\$1* $+ 2$ *HAU* $\times \lceil .25 \rceil$ *hrs* $\times$ *\$1* $= 1 + 1 + 2 =$ *\$4.*

Definition 6 (Min-width cost ratio of a strategy)**.** *The min-width cost ratio R^+ is defined as $I(S)/C^+$, where $I(S)$ is the cost of IA (using a particular allocation strategy S) until the problem is solved, and C^+ is the min-width cost.*

Definition 7 (Min-cost ratio of a strategy)**.** *The min-cost ratio R^* is defined as $I(S)/C^*$, where C^* is the minimal cost.*

5 The Geometric (b^i) Strategy

Consider a simple strategy where the number of HAUs allocated on the i-th iteration is $\lceil b^i \rceil$, for some $b > 1$. For example, the 2^i (doubling) strategy doubles the number of HAUs allocated on each iteration: First, try 1 HAU; if it fails, try 2 HAUs, then 4 HAUs, and so on.

Suppose the b^i strategy solves a problem on the j-th iteration, i.e., $j = \lceil log_b W^+ \rceil$, where W^+ is the min width. The cost of a geometric allocation strategy with a continuous cost model is $I = \sum_{i=0}^{j} D_{b^i} b^i$. Recall that $D_{b^j} = s_{b^j} \leq E$ on the successful iteration, and $D_{b^i} \leq E$ on the failed iterations. By standard manipulations, $I \leq E\frac{b^j - 1}{b-1} + s_{b^j} b^j$. In the discrete case, $I \leq \lceil E \rceil \frac{b^j - 1}{b-1} + \lceil s_{b^j} \rceil b^j$. This latter upper bound is obtained by making the pessimistic relaxation that no spare time is used by a HAU from one iteration to the next. This explains in part why our experimentally measured cost ratios are better than the theoretical upper bounds introduced later in this section.

Next, we specialize the cost ratio analysis to a class of realistic cloud environments and ravenous search algorithms.

5.1 Discrete Cost Model with Fast Memory Exhaustion

Cloud platforms such as Amazon EC2 and Windows Azure typically have discrete cost models, where the discrete billing unit is 1 hour, and fractional hours are rounded up. This relatively long unit of time, combined with the fast rate at which search algorithms consume RAM, leads to the following, empirical observation, which has significant practical implications:

Observation 1. *Both the success time and the failure time are at most 1 billing time unit (1 hour), $D_k \leq E \leq 1, \forall k \geq 1$. A direct consequence is that $\lceil D_k \rceil = \lceil E \rceil = 1, \forall k \geq 1$.*

Currently, the amount of RAM per core on EC2 and Windows Azure ranges from 2 to 8.5 GB per core. Some of the RAM in each core must be used for purposes other than search state storage, e.g., the OS, MPI communication queue, and heuristic tables such as pattern databases. Assume (optimistically) that we have 8 GB RAM remaining after this "overhead" is accounted for. Suppose that a state requires 50 bytes of storage overall in the system. Generating at least 46,000 new states per second, which is a relatively slow state generation rate, will exhaust 8 GB RAM within an hour. Many search applications generate states much faster than this. For example, A* search on the Fast-Downward domain-independent planner using bisimulation merge-and-shrink heuristics generates O(100,000) states per second on a single core (Nissim, Hoffmann, and Helmert 2011); domain-specific search algorithms (e.g., sliding tiles puzzle solvers) can generate states even faster.

Thus, many (but not all) search applications will exhaust the RAM/core in a HAU within a single billing time unit in modern cloud environments. A single iteration of IA will either solve a given problem within 1 billing time unit, or fail (due to RAM exhaustion) within 1 billing time unit.

Observation 1 is not limited to current hardware, and will continue to apply in the future. Although the total RAM on a multi-core processor will continue to rise, the number of cores per processor will also continue to increase, because of fundamental architectural bottlenecks associated with increased RAM/core (Hennessy and Patterson 2007), so the amount of RAM *per core* will only increase moderately (if at all); furthermore, the speed per core is expected to increase, which will further offset increases in RAM/core so

	# Cores &	Max #								Continuous Model			Discrete Model			
	(RAM) per	HAUs	number of problems solved on iteration							Min-Width Cost Ratio (R^+)			Min-Cost Ratio (R^*)			# Solved w. Zero
	HAU	(cores)	2	3	4	5	6	7	Total	Mean	SD	Max	Mean	SD	Max	Cost Overhead
Planning: `HPC`	12(54GB)	64(768)	2	5	11	3	3	1	25	**2.18**	0.45	3.34	**1.29**	0.27	1.88	8
Planning: `Commodity`	8(16GB)	8(64)	5	1	2	-	-	-	8	**1.62**	0.29	2.29	**1.04**	0.12	1.33	7
Planning: `EC2`	4(15GB)	16(64)	6	1	1	5	-	-	13	**1.63**	0.25	2.27	**1.26**	0.25	1.64	6
Mult. Seq. Align: `HPC`	12(54GB)	64(768)	4	1	-	1	-	2	8	**2.02**	0.46	2.76	**1.54**	0.75	3.28	3

Table 1: Summary of IAHDA* on planning and multiple sequence alignment on `HPC`, `Commodity`, and `EC2` clusters.

that the rate of RAM consumption per core will either remain constant or increase in the future.

Our experiments (Sec. 6) validate Observation 1 for all of our planning and sequence alignment benchmarks. In addition, HDA* has been observed to exhaust memory within 20 minutes on every planning and 24-puzzle problem studied in (Kishimoto, Fukunaga, and Botea 2012).

Observation 2. *In a discrete cost model with $E \le 1$, the cost to solve a problem on v HAUs is proportional to v. As a direct consequence, $W^+ = W^*$ and thus $R^+ = R^*$.*

Remarkably, Observation 2 holds independently of whether the search efficiency is nonincreasing (Definition 4). Although we used Def. 4 to obtain general case results (Propositions 1 and 2), under the stronger condition that $E \le 1$, all of the results below hold regardless of whether the search efficiency is nonincreasing.

The cost overhead of IA consists of two components: (1) unnecessary HAUs allocated on the final, successful iteration, and (2) repeated allocation of HAUs due to failed iterations. When the min-width happens to be a power of b, then the former overhead is 0. In a discrete pricing model, the latter overhead can be reduced significantly when iterations terminate faster than a single billing unit, and thus u iterations fit in $v < u$ billing units. Furthermore, with a sufficiently small E, *all iterations can be executed within a single billing time unit*, entirely eliminating the repeated allocation cost overhead. Indeed, in our experiments below, for all our planning benchmark problems, all iterations fit in a single billing time unit.

Let us consider the best, worst and average cases for the min-cost ratio $R^+ = R^*$. Clearly, in the *best case*, $R^* = R^+ = 1$. Recall Example 1. Since $E < 1$ hour, if the minimum width W^+ is 4, then $I = C^+ = C^* = 4$ and therefore $R^* = R^+ = 1$.

The *worst case* is when the $(j-1)$-th iteration is barely insufficient, and on the final j-th iteration, only $W^+ = b^{j-1} + 1$ HAUs (out of the b^j allocated) are necessary:

$$\begin{aligned} R^*_{wo} &= R^+_{wo} = \frac{I}{\lceil s_{W^+}\rceil (b^{j-1}+1)} \le \frac{\lceil E\rceil \frac{b^j-1}{(b-1)} + \lceil s_{bj}\rceil b^j}{\lceil s_{W^+}\rceil (b^{j-1}+1)} \\ &= \frac{(b^j-1)}{(b-1)(b^{j-1}+1)} + \frac{b^j}{b^{j-1}+1} \le \frac{b}{b-1} + b = \frac{b^2}{b-1} \end{aligned}$$

According to Observation 2, the *average case* is when the number of required HAUs is halfway between the worst and best cases, i.e., $W^+ = ((b^{j-1}+1)+b^j)/2$ of the HAUs allocated on the final, j-th iteration are necessary. With a similar computation as for R^*_{wo}, we obtain the following average-case upper bound:

$$R^*_{avg} = R^+_{avg} = \frac{I}{\lceil s_{W^+}\rceil (b^j + b^{j-1} + 1)/2} \le \frac{2b^2}{b^2-1}$$

Observation 3. *When $E \le 1$, the worst case bound $R^*_{wo} \le b^2/(b-1)$ is minimized by the doubling strategy ($b = 2$).*

As b increases above 2, making the iterative allocation more aggressive, the upper bound for R^*_{avg} improves, but the worst case gets worse. Therefore: *the simple, doubling strategy is the natural allocation policy to use in practice.*

Note that for the doubling strategy (2^i), the average case ratio is bounded by $8/3 \approx 2.67$, and the worst case cost ratio does not exceed 4. In other words, with the simple doubling strategy in a discrete cost model when $E \le 1$, we will *never pay more than 4 times the optimal cost* that we would have paid if we knew the optimal width in advance.

6 Experimental Results

We experimentally evaluate iterative allocation applied to HDA* (**IAHDA***). We focus on the doubling strategy, for the reasons outlined in the previous section. Domain-independent planning and multiple sequence alignment problems are solved on 3 parallel clusters: (1) `HPC` - a large-scale, high-performance computing cluster, where each HAU has 12 cores (Intel Xeon 2.93GHz), 4.5GB RAM/core, and a 40GB Infiniband network. (2) `Commodity` - a cluster of commodity machines, where each HAU has 8 cores (Xeon 2.33GHz) and 2GB RAM/core, and a 1Gbps (x3, bonded) Ethernet network. (3) `EC2` - Amazon EC2 cloud cluster using the `m1.xlarge` ("Extra Large" instance) HAU, which have 4 virtual cores, (3.75GB RAM per core, and an unspecified network interconnect).

The EC2 configuration is a less favorable environment than the `HPC` and `Commodity` clusters because of physical processor sharing with other users, and highly variable network performance (Wang and Ng 2010; Iosup et al. 2011). Although Amazon offers higher-performance (more expensive) HAUs intended for high-performance scientific computing (i.e., "Cluster Compute" instances), we intentionally selected this particular configuration in order to evaluate the behavior of IAHDA* under adverse conditions. Boot times for newly allocated HAUs before each iteration, required in a fully automated script for a cloud environment such as EC2, are not included in the times we report. All the newly allocated HAUs boot in parallel, and there is no charge while waiting for allocation to succeed. Booting our unoptimized

VM image on EC2 currently takes around 30 seconds. Optimizing the Linux VM image with standard OS configuration techniques can reduce this further.

We evaluated IAHDA* for domain-independent planning on a Fast-Downward based planner using the merge-and-shrink (M&S) heuristic (Helmert, Haslum, and Hoffmann 2007). We use 7 standard benchmark domains: Depot, Driverlog, Freecell, Logistics, Mprime, Pipesworld-Notankage, Pipesworld-Tankage (142 problems total). These are domains where the M&S "HHH" heuristic we used is competitive with the state of the art (Nissim, Hoffmann, and Helmert 2011).

We also evaluated IAHDA* on multiple sequence alignment (MSA) using the variant of HDA* in (Kobayashi, Kishimoto, and Watanabe 2011), without the weighted-A* preprocessing/upper-bounding step. The test set consisted of 28 standard alignment problems for 5-9 sequences (`HPC` only).

For each problem, on each cluster, the min-width W^+ was found by incrementing the number of HAUs until the problem was solved. We evaluate the data under both the continuous and discrete cost models: For both the continuous and discrete models, we computed the min-width cost ratio R^+. In the discrete model, we assume the industry standard 1 hour granularity. In all our test problems, max iteration time E (Def 5) turns out to be less than 1 hour. Thus, by Observation 2, discrete $R^* = R^+$ (the number of HAUs which minimizes cost is equal to min-width).

Table 1 summarizes the results. For all 3 clusters, we only consider problems which required ≥ 2 iterations on that cluster. For each system, we show how many problems were solved using exactly i iterations for each $2 \leq i \leq \log_2(\text{Max\#HAU's})$, as well as the total number of problems solved using ≥ 2 iterations. For both the continuous and discrete cost models, we show the mean, standard deviation, min, and max values of the min-width cost ratio R^+ or min-cost ratio R^* (see above). The column "# solved with zero overhead" shows the number of problems where the discrete $R^* = 1$. Details for individual runs (for solved problems) on `HPC` and `EC2` are shown in Tables 2-3. Details for individual runs on `Commodity` are not shown due to space, but are qualitatively similar to the HPC results, as indicated by the aggregate results in Table 1.

From the experimental results (Tables 1-3), we observe:
(a) The mean discrete min-cost ratios R^* for all problems, on all 3 clusters (Table 1) is significantly less than the theoretical worst case bound (4.0) and average case bound (2.67) for the doubling strategy (Sec. 5.1); The continuous min-width cost ratio R^+ was never higher than 3.34.
(b) For all our benchmarks, $E < 1$, satisfying the conditions of Sec. 5.1. On all planning problems, all iterations were performed within a single billing time unit (hour). Furthermore, on some problems, W^* is a power of 2, and the discrete $R^* = 1.0$, i.e., no additional cost was incurred by IA, compared to the optimal (minimal) cost.
(c) IAHDA* displays nonincreasing search efficiency behavior (Def 4) on `HPC` and `Commodity` (verified for all runs on these systems). This strongly suggests that on these clusters, Prop. 1 applies, and continuous model $R^+ = R^*$. On the other hand, some iterations on EC2 sometimes took longer than later, (failed) iterations. This is most likely due to the extreme variability in network performance (Wang and Ng 2010).

7 Related Work

There is little prior work on cost-based analysis of search algorithms in a utility computing environment. An experimental investigation of a cloud-based portfolio for stochastic planning algorithms is in (Lu et al. 2011). Work on resource allocation in utility computing services has focused on the service provider's perspective, i.e., resource management/scheduling (Feitelson et al. 1997; Garg, Buyya, and Siegel 2010). In contrast, we investigate efficient resource usage from an end-user's perspective.

The main idea of IA, i.e., repeatedly running a search algorithm with increasing machine resources, is related to other approaches which iteratively apply a search algorithm with an increasing bounds/threshold on some parameter, such as iterative deepening (Korf 1985). Prior approaches to parallel search have focused on mechanisms for effectively using a limited amount of memory. For example, TDS is a parallel IDA* which uses hash-based work distribution and a distributed transposition table (Romein et al. 2002), and PRA* is a parallel A* which uses node retraction mechanism when memory is exhausted (Evett et al. 1995).

Iterative Allocation differs fundamentally from these previous approaches by dynamically allocating hardware resources on demand. Techniques such as transposition tables and node retraction, which seek to reuse limited memory given static resources, incur significant overhead in terms of search efficiency and wall-clock time. The performance of TDS and PRA* degrades as the gap between the amount of physical RAM vs. the size of the search space grows.

On the other hand, IA incurs wall-clock time and monetary overheads because of work that is discarded after each iteration as well as the difference between the width of the final iteration and the minimal width. Furthermore, the effectiveness of IA requires that the underlying search algorithm (such as HDA*) continues to scale well as the amount of resources allocated is increased. A quantitative comparison of the various tradeoffs made by IA and static-resource approaches such as TDS is an area for future work.

8 Discussion and Conclusions

This paper explored some implications of having access to vast (but costly) resources for parallel search algorithms. We analyzed a general, iterative resource allocation strategy for scalable, memory-intensive search algorithms, including parallel A* variants such as HDA*. We presented bounds on the relative cost of a simple, geometric allocation policy, compared to an (a priori unknown) optimal allocation. Under realistic assumptions and a discrete pricing model used in commercial clouds such as Amazon EC2 and Windows Azure, we showed that the average and worst case cost ratios for a doubling strategy were at most 2.67 and 4, respectively. While our analysis primarily focused on cost ratio

	Min-Width		Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Iter 6	Iter 7	Total	R^+	R^*
Problem	Cores	Time	12 cores	24 cores	48 cores	96 cores	192 cores	384 cores	768 cores	Time	Continuous	Discrete
Domain-Independent Planning												
Depot-pfile16	96	706	445	544	616	706				2311	1.71	1.00
Driverlog-pfile13	60	167	200	228	325	104				856	3.34	1.60
Freecell-pfile9	192	521	422	481	510	575	521			2509	1.96	1.00
Freecell-pfile6	24	389	429	389						818	1.55	1.00
Freecell-pfile11	96	459	435	467	474	459				1835	1.89	1.00
Freecell-pfile12	276	582	422	493	486	550	576	412		2939	2.25	1.39
Freecell-5-1	108	897	402	530	626	764	496			2818	2.23	1.78
Freecell-5-2	72	753	441	600	673	570				2285	1.97	1.33
Freecell-5-3	72	653	522	593	817	477				2409	2.24	1.33
Freecell-5-4	72	691	484	573	727	534				2319	2.12	1.33
Freecell-5-5	84	953	470	553	682	807				2512	1.61	1.14
Logistics00-8-1	36	362	448	463	268					1179	2.25	1.33
Logistics00-9-0	48	272	406	527	272					1205	2.34	1.00
Mprime-prob15	48	323	236	307	323					865	1.66	1.00
Pipesworld-Notankage-p18	36	218	248	258	164					669	2.17	1.33
Pipesworld-Notankage-p20	60	247	255	289	333	163				1039	2.80	1.60
Pipesworld-Notankage-p25	384	406	316	341	368	394	471	406		2296	2.01	1.00
Pipesworld-Notankage-p27	636	313	316	337	357	380	452	469	252	2563	2.65	1.21
Pipesworld-Notankage-p32	60	286	282	301	327	185				1094	2.57	1.60
Pipesworld-Notankage-p33	156	362	294	325	339	365	286			1609	2.08	1.23
Pipesworld-Notankage-p35	204	305	249	283	292	311	332	172		1638	2.94	1.88
Pipesworld-Tankage-p09	36	704	634	777	526					1937	2.03	1.33
Pipesworld-Tankage-p10	24	710	652	710						1362	1.46	1.00
Pipesworld-Tankage-p14	60	297	252	239	360	196				1047	2.52	1.60
Pipesworld-Tankage-p22	72	394	299	347	364	289				1299	2.02	1.33
Multiple Sequence Alignment												
MSA 08_BB12003	24	2982	3193	2982						6175	1.54	1.50
MSA 09_BB12032	36	4061	3765	4592	3050					11408	2.06	1.50
MSA 07_BBS11026	432	3115	1978	2233	2458	2650	2847	3000	1739	16905	2.59	3.28
MSA 05_BB11035	636	892	656	699	705	733	781	792	758	5122	2.05	1.28
MSA 05_BB12009	108	694	652	702	733	760	388			3234	2.77	1.78
MSA 05_BB12019	24	741	741	741						1482	1.50	1.00
MSA 05_BB12023	24	550	728	550						1279	1.66	1.00
MSA 05_BBS11037	24	372	716	372						1088	1.96	1.00

Table 2: Detailed results for domain-independent planning and multiple sequence alignment (MSA) on the `HPC` cluster. Solved problems that required ≥ 2 iterations are shown. R^+ = "min-width cost ratio", R^* = "min-cost ratio".

bounds for the more interesting, discrete cost model, similar derivations can be applied to the continuous cost model.

Experiments with planning and sequence alignment validated the theoretical predictions, and showed that the cost ratios can be quite low, showing that IA with a doubling policy is a reasonable strategy in practice. For our domain-independent planning benchmarks, under a discrete pricing model typical for commercial clouds, there was *no* cost overhead due to repeated allocations because all iterations were completed within a single billing unit (hour), and the only cost overhead was due to the difference between the number of instances "guessed" by IA on the final iteration and the actual, optimal allocation. Domain-specific solvers for standard search domains such as the sliding tiles puzzle have even higher state generation rates and exhaust memory faster than our benchmarks, providing a greater opportunity to run multiple iterations within one billing time unit.

While our experiments applied IA to HDA*, our theoretical analysis is quite general, and applies to any scalable, ravenous algorithm that satisfies the assumptions, including, for example, scalable work-stealing A* or breadth-first search implementations.

IA is simple to implement as a script on top of any scalable ravenous algorithm. The downside of this simplicity is that IA discards all work done in previous iterations. While we showed that total overhead of IA is quite low (and can even be zero), there are cases where more efficiency is required. It is possible to start a new iteration from the existing open and closed lists, after re-distributing these among a larger number of CPUs available in the new iteration. This requires significant modifications of the underlying parallel search algorithm, and is an area for future work.

This paper focused on the analysis and evaluation of the monetary cost of IA. Since iterations terminate quickly due to fast RAM consumption (Observation 1), the wall-clock runtime of ravenous algorithms using IA tends to be short, and budget and resource availability limitations will tend to be a more significant concern than excessive runtimes. The hardest planning problem in our experiments, `Pipesworld-Notankage-36` (Table 2), required 7 itera-

	Min-Width		Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Total	R^+	R^*
Problem	Cores	Time	4 cores	8 cores	16 cores	32 cores	64 cores	Time	Continuous	Discrete
Freecell-pfile6	40	750	366	603	677	571	511	2727	2.27	1.60
Freecell-pfile7	8	585	422	585				1007	1.36	1.00
Logistics00-7-1	8	495	432	495				927	1.44	1.00
Logistics00-8-1	44	1297	337	516	595	577	941	2966	1.64	1.45
Logistics00-9-0	48	1116	328	469	530	489	838	2654	1.55	1.33
Logistics00-9-1	12	584	353	436	498			1287	1.84	1.33
Mprime-prob30	8	423	420	423				843	1.50	1.00
Pipesworld-Notankage-p16	8	255	275	255				529	1.54	1.00
Pipesworld-Notankage-p18	44	525	215	319	361	326	389	1610	1.93	1.45
Pipesworld-Notankage-p19	8	223	227	223				450	1.51	1.00
Pipesworld-Notankage-p24	8	305	305	305				611	1.50	1.00
Pipesworld-Tankage-p09	44	1750	419	811	879	797	1265	4170	1.67	1.64
Pipesworld-Tankage-p10	32	1558	441	852	921	1558		3772	1.47	1.12

Table 3: Detailed results for domain-independent planning on EC2 `m1.xlarge` HAUs. Solved problems that required ≥ 2 iterations are shown. R^+ = "min-width cost ratio", R^* = "min-cost ratio".

tions of IA, and the 7th iteration used 768 cores and 3.456 terabytes of RAM on the `HPC` cluster (the min-width was 53 HAUs = 636 cores, 2.9TB RAM). Extrapolating from this, a hypothetical 2-hour long run of IAHDA* on a harder planning problem from the same class would execute 13 iterations, where the final iteration of this would use 49,152 cores and 221 terabytes of RAM, which is unaffordable today. While resources will become cheaper in the future, the exponential growth of resource usage by successive iterations of IA means that the maximum wall-clock time for IA will be limited in practice. Comparison of the tradeoff between wall-clock time and resource usage for IA vs. strategies that use static resources (e.g., parallel IDA* variants) is an area for future work.

Acknowledgements

This research is supported by the JST PRESTO program and a grant from JSPS.

References

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1995. PRA*: Massively Parallel Heuristic Search. *Journal of Parallel and Distributed Computing* 25(2):133–143.

Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U.; Sevcik, K. C.; and Wong, P. 1997. Theory and practice in parallel job scheduling. In Feitelson, D. G., and Rudolph, L., eds., *JSSPP*, volume 1291 of *Lecture Notes in Computer Science*, 1–34. Springer.

Garg, S. K.; Buyya, R.; and Siegel, H. J. 2010. Time and cost trade-off management for scheduling parallel applications on utility grids. *Future Generation Computer Systems* 26(8):1344 – 1355.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling ICAPS-07*, 176–183.

Hennessy, J., and Patterson, D. 2007. *Computer architecture: a quantitative approach, 4th ed.* Morgan Kaufmann.

Iosup, A.; Ostermann, S.; Yigitbasi, N.; Prodan, R.; Fahringer, T.; and Epema, D. H. J. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst.* 22(6):931–945.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *Proc. ICAPS*, 201–208.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2012. Evaluation of a simple, scalable, parallel best-first search strategy. *arXiv:1201.3204v1*. http://arxiv.org/abs/1201.3204.

Kobayashi, Y.; Kishimoto, A.; and Watanabe, O. 2011. Evaluations of Hash Distributed A* in Optimal Sequence Alignment. In *Proc. IJCAI*, 584–590.

Korf, R. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 97:97–109.

Lu, Q.; Xu, Y.; Huang, R.; Chen, Y.; and Chen, G. 2011. Can cloud computing be used for planning? an initial study. In *Proc. IEEE CloudCom*.

Mahanti, A., and Daniels, C. 1993. A SIMD Approach to Parallel Heuristic Search. *Artificial Intelligence* 60:243–282.

Mahapatra, N., and Dutt, S. 1997. Scalable Global and Local Hashing Strategies for Duplicate Pruning in Parallel A* Graph Search. *IEEE Trans. on Parallel and Distributed Systems* 8(7):738–756.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *Proc. IJCAI*, 1983–1990.

Powley, C.; Ferguson, C.; and Korf, R. 1993. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence* 60:199–242.

Rao, V. N., and Kumar, V. 1987. Parallel Depth-First Search on Multiprocessors Part I: Implementation. *International Journal of Parallel Programming* 16(6):479–499.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.* 16(7):701–710.

Romein, J. W.; Bal, H. E.; Schaeffer, J.; and Plaat, A. 2002. A Performance Analysis of Transposition-Table-Driven Work Scheduling in Distributed Search. *IEEE Transactions on Parallel and Distributed Systems* 13(5):447–459.

Wang, G., and Ng, T. S. E. 2010. The impact of virtualization on network performance of Amazon EC2 data center. In *Proc. IEEE INFOCOM*.