Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence

External Memory Best-First Search for Multiple Sequence Alignment

Matthew Hatem and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
mhatem and *ruml* at *cs.unh.edu*

Abstract

Multiple sequence alignment (MSA) is a central problem in computational biology. It is well known that MSA can be formulated as a shortest path problem and solved using heuristic search, but the memory requirement of A* makes it impractical for all but the smallest problems. Partial Expansion A* (PEA*) reduces the memory requirement of A* by generating only the most promising successor nodes. However, even PEA* exhausts available memory on many problems. Another alternative is Iterative Deepening Dynamic Programming, which uses an uninformed search order but stores only the nodes along the search frontier. However, it too cannot scale to the largest problems. In this paper, we propose storing nodes on cheap and plentiful secondary storage. We present a new general-purpose algorithm, Parallel External PEA* (PE2A*), that combines PEA* with Delayed Duplicate Detection to take advantage of external memory and multiple processors to solve large MSA problems. In our experiments, PE2A* is the first algorithm capable of solving the entire Reference Set 1 of the standard BAliBASE benchmark using a biologically accurate cost function. This work suggests that external best-first search can effectively use heuristic information to surpass methods that rely on uninformed search orders.

Introduction

One real-world application of heuristic search with practical relevance (Korf 2012) is Multiple Sequence Alignment (MSA). As we explain in more detail below, MSA can be formulated as a shortest path problem where each sequence represents one dimension in a multi-dimensional lattice and a solution is a least-cost path through the lattice. To achieve biologically plausible alignments, great care must be taken in selecting the most relevant cost function. The scoring of *gaps* is of particular importance. Altschul (1989) recommends *affine* gap costs, described in more detail below, which increase the size of the state space by a factor of 2^k for k sequences.

Although dynamic programming is the classic technique for solving MSA (Needleman and Wunsch 1970), heuristic search algorithms can achieve better performance than dynamic programming by pruning much of the search space, computing alignments faster and using less memory (Ikeda and Imai 1999). A* (Hart, Nilsson, and Raphael 1968) uses an admissible heuristic function to avoid exploring much of the search space. The classic A* algorithm maintains an *open list*, containing nodes that have been generated but not yet expanded, and a *closed list*, containing all generated states, in order to prevent duplicated search effort. Unfortunately, for challenging MSA problems, the memory required to maintain the open and closed lists makes A* impractical. Due to the large branching factor of $2^k - 1$, the performance bottleneck for MSA is the memory required to store the frontier of the search.

Yoshizumi, Miura, and Ishida (2000) present a variant of A* called Partial Expansion A* (PEA*) that reduces the memory needed to store the open list by generating only the successor nodes that appear promising. This technique can significantly reduce the size of the open list. However, like A*, PEA* is limited by the memory required to store the open and closed list and for challenging alignment problems PEA* can still exhaust memory.

One previously proposed alternative to PEA* is Iterative Deepening Dynamic-Programming (IDDP) (Schroedl 2005), a form of bounded dynamic programming that relies on an uninformed search order to reduce the maximum number of nodes that need to be stored during search. The memory savings of IDDP comes at the cost of repeated search effort and divide-and-conquer solution reconstruction. IDDP forgoes a best-first search order and as a result it is possible for IDDP to visit many more nodes than a version of A* with optimal tie-breaking. Moreover, because of the wide range of edge costs found in the MSA domain, IDDP must rely on the bound setting technique of IDA*$_{CR}$ (Sarkar et al. 1991). With this technique, it is possible for IDDP to visit four times as many nodes as A* (Schroedl 2005). And even though IDDP reduces the size of the frontier, it is still limited by the amount of memory required to store the open nodes and for large MSA problems this can exhaust main memory.

When problems cannot fit in main memory, it is possible to take advantage of cheap and plentiful external storage, such as disks, to store portions of the search space. In the Delayed Duplicate Detection (DDD) (Korf 2003) technique, newly generated nodes are placed on disk and processed in a later step with external sorting and merging. Hash-based DDD (HBDDD, Korf 2008) is an efficient form of DDD that

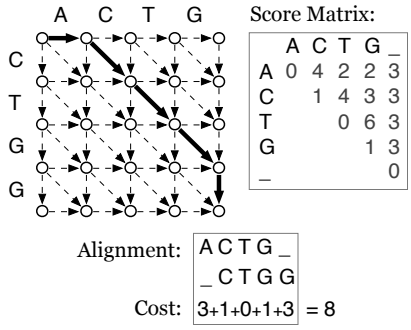

Figure 1: Optimal sequence alignment using a lattice and scoring matrix.

avoids the overhead of external sorting and is capable of processing duplicates in linear time. The same techniques used for external search may also be used to take advantage of multiple processors and overcome the latency of disk with parallel search.

Rather than suffer the overhead of an uninformed search order and divide-and-conquer solution reconstruction, we propose solving large problems by combining best-first search with external search. In this paper we present a new general-purpose algorithm, Parallel External Partial Expansion A* (PE2A*), that combines the best-first partial expansion technique of PEA* and the external memory technique of HBDDD. We compare PE2A* with in-memory A*, PEA* and IDDP for solving challenging instances of MSA. The results show that parallel external memory best-first search can outperform serial in-memory search and is capable of solving large problems that cannot fit in main memory. Contrary to the assumptions of previous work, we find that storing the open list is much more expensive than storing the closed list. We also demonstrate that PE2A* is capable of solving, for the first time, the entire Reference Set 1 of the BAliBASE benchmark for MSA (Thompson, Plewniak, and Poch 1999) using a biologically plausible cost function that incorporates affine gap costs. This work suggests that external best-first search can effectively use heuristic information to surpass methods that rely on uninformed search orders.

Previous Work

We first discuss the MSA problem in more detail, including computing heuristic estimates. Then we will review the most popular applicable heuristic search algorithms.

Multiple Sequence Alignment

In the case of aligning two sequences, an optimal *pair–wise* alignment can be represented by a shortest path between the two corners of a two dimensional lattice where columns and rows represent sequences. A move vertically or horizontally represents the insertion of a *gap*. Biologically speaking, a gap represents a mutation whereby one amino acid has either been inserted or removed; commonly referred to as an *indel*. A diagonal move represents either a conservation or mutation whereby one amino acid has either been conserved

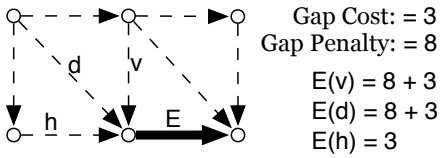

Figure 2: Computing edge costs with affine gaps.

or substituted for another. Figure 1 shows an alignment of two DNA sequences using a lattice.

In a shortest-path formulation, the indels and substitutions have associated costs; represented by weighted edges in the lattice. A solution is a least-cost path through the lattice. The cost of a partial solution is computed using the *sum-of-pairs* cost function; the summation of all indels and substitutions. The biological plausibility of an alignment depends heavily on the cost function used to compute it. A popular technique for assigning costs that accurately models the mutations observed by biologists is with a Dayhoff scoring matrix (Dayhoff, Schwartz, and Orcutt 1978) that contains a score for all possible amino acid substitutions. Each value in the matrix is calculated by observing the differences in closely related proteins. Edge weights are constructed accordingly. Figure 1 shows how the cost of an optimal alignment is computed using a score matrix. This technique can be extended to multiple alignment by taking the sum of all pair-wise alignments induced by the multiple alignment.

Of particular importance is the scoring of gaps. Altschul (1989) found that assigning a fixed score for gaps did not yield the most biologically plausible alignments. Biologically speaking, a mutation of n consecutive indels is more likely to occur than n separate mutations of a single indel. Altschul et al. construct a simple approximation called *affine* gap costs. In this model the cost of a gap is $a + b * x$ where x is the length of a gap and a and b are some constants. Figure 2 shows an example that incorporates affine gap costs. The cost of the edge E depends on the preceding edge; one of h, d, v. If the preceding edge is h then the cost of E is less because a gap is extended.

A pair-wise alignment is easily computed using dynamic programming (Needleman and Wunsch 1970). This method extends to alignments of k sequences in the form of a k dimensional lattice. For alignments of higher dimensions, the N^k time and space required render dynamic programming infeasible. This motivates the use of heuristic search algorithms that are capable of finding an optimal solution while pruning much of the search space with the help of an admissible heuristic. While affine gap costs have been shown to improve accuracy, they also increase the size of the state space. This is because each state is uniquely identified by the incoming edge; indicating whether a gap has been started. This increases the state space by a factor of 2^k for k sequences. Affine gap costs also make the MSA domain implementation more complex and require more memory for storing the heuristic.

Admissible Heuristics

The cost of a *k-fold* optimal alignment is computed by taking the sum of all pair-wise alignments using the same pair-wise cost function above. Carrillo and Lipman (1988) show that the cost of an optimal alignment for k sequences is greater than or equal to the sum of all possible *m-fold* optimal alignments of the same sequences for $m \leq k$. Therefore, we can construct a lower bound on the cost of an optimal alignment of k sequences by taking the sum of all *m-fold* alignments. Furthermore, we can construct a lower bound on the cost to complete a partial optimal alignment of k sequences by computing all *m-fold* alignments in reverse; starting in the lower right corner of the lattice and finding a shortest path to the upper left corner. The forward lattice coordinates of the partial alignment are then used in all reverse lattices to compute the cost-to-go estimate Figure 3 shows how to construct a cost-to-go estimate for aligning 3 sequences. The cost-to-go for the partial alignment is the pair-wise sum of alignments computed in reverse.

Lermen and Reinert (2000) use this lower bound in a heuristic function for solving MSA with the classic A* algorithm. Lower dimensional (*m-fold*) alignments for all $\binom{k}{m}$ sequences are computed in reverse using dynamic programming, generating a score for all partial solutions. The heuristic function combines the scores for all partial solutions of a given state. This heuristic is referred to as $h_{all,m}$.

Higher quality admissible heuristics can be obtained by computing optimal alignments with larger values for m. Unfortunately the time and space complexity of this technique make it challenging to compute and store optimal alignments of size $m > 2$. Kobayashi and Imai (1998) show that splitting the k sequences into two subsets and combining the scores for the optimal alignments of the subsets with all pair-wise alignments between subsets is admissible. For example, given a set of sequences S we can define two subsets $s_1 \subset S$, $s_2 \subset S$ such that $s_1 \cap s_2 = \emptyset$. A lower bound on the cost of an optimal alignment of all sequences in S can be computed by taking the sum of the optimal alignments for s_1, s_2 and all pair-wise alignments for sequences $x \in S, y \in S$ such that $\{x, y\} \not\subset s_1$ and $\{x, y\} \not\subset s_2$. This heuristic is referred to as the $h_{one,m}$ heuristic. The accuracy of the $h_{one,m}$ heuristic is similar to $h_{all,m}$ but it requires much less time and memory.

BAliBASE

Randomly generated sequences do not accurately reflect the sequences found in biology and provide no means of measuring the biological plausibility of the alignments that are produced. A popular benchmark for MSA algorithms is BAliBASE, a database of manually-refined multiple sequence alignments specifically designed for the evaluation and comparison of multiple sequence alignment programs (Thompson, Plewniak, and Poch 1999). Of particular interest to our study is a set of instances known as Reference Set 1. Each instance in this set contains 4 to 6 protein sequences that range in length from 58 to 993. The sequences in this set are challenging for optimal MSA programs because they are highly dissimilar; requiring that much of the state space

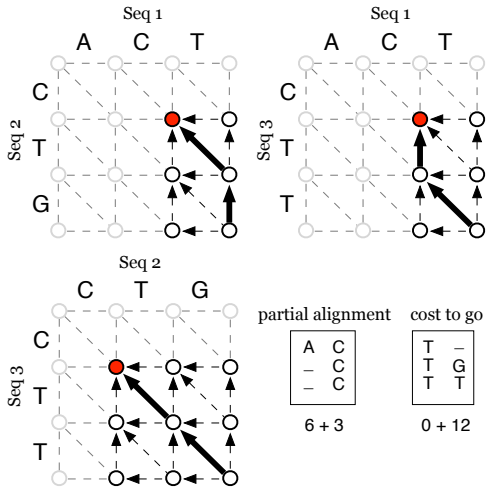

Figure 3: Computing cost-to-go by solving the lattice in reverse.

be explored to find an optimal alignment. To the best of our knowledge, no one has been able to compute optimal alignments for the entire Reference Set 1 using affine gap costs.

Iterative-Deepening Dynamic Programming

Iterative Deepening Dynamic-Programming (IDDP) (Schroedl 2005) is an iterative deepening search that combines dynamic programming with an admissible heuristic for pruning. IDDP uses a pre-defined search order much like dynamic programming. The state space is divided into levels that can be defined *row-wise*, *column-wise* or by *antidiagonals* (lower-left to upper-right). IDDP proceeds by expanding nodes one level at a time. To detect duplicates, only the adjacent levels are needed. All other previously expanded levels may be deleted. This pre-defined expansion order reduces the amount of memory required to store open nodes: if levels are defined by antidiagonals then only k levels need to be stored in the open list during search. In this case the maximum size of the open list is $O(kN^{k-1})$ for sequences of length N. This is one dimension smaller than the entire space $O(N^k)$.

IDDP uses a heuristic function, similar to A*, to prune unpromising nodes and in practice the size of the open list is much smaller than the worst case. At each iteration of the search an upper bound b on the solution cost is estimated and only the nodes with $f \leq b$ are expanded. IDDP uses the same bound setting technique of IDA*$_{\text{CR}}$ (Sarkar et al. 1991) to estimate an upper bound that is expected to double the number of nodes expanded at each iteration.

Schroedl (2005) was able to compute optimal alignments for 80 of the 82 instances in Reference Set 1 of the BAliBASE benchmark using IDDP with a cost function that incorporated affine gap costs. Edelkamp and Kissmann (2007) extend IDDP to external memory search using sorting-based DDD but were only able to solve one additional instance, gal4 which alone required over 182 hours of solving time.

Because IDDP deletes closed nodes, divide-and-conquer

solution reconstruction is required to recover the solution. As we will see later, deleting closed nodes provides a limited advantage since the size of the closed list is just a small fraction of the number of nodes generated during search.

In order to achieve memory savings, IDDP must expand nodes in an uninformed pre-defined expansion order. In contrast to a best-first expansion order, a node in one level may be expanded before a node in another level with a lower f. As a result, it is possible for IDDP to expand many more nodes in the final f layer than A* with good tie-breaking. In fact, the expansion order of IDDP approximates worst-case tie-breaking. The preferred tie-breaking policy for A* is to break ties by expanding nodes with higher g first. For MSA, the g of a goal node (g^*) is maximal among all nodes with equal f. Therefore, as soon as the goal node is generated it is placed at the front of the open list and search can terminate on the next expansion. In contrast, IDDP will expand all non-goal nodes with $f = f^*$ and $g < g^*$, assuming levels are defined by antidiagonals. This is because in the final iteration the bound is set to $b \geq f^*$ and all nodes with $f \leq f^* \leq b$ are expanded at each level. The level containing the goal is processed last and therefore all non-goal nodes with $f \leq f^*$ must be expanded first.

The situation is even worse when relying on the bound setting technique of IDA*$_{\text{CR}}$. This technique estimates each bound in an attempt to double the number of expanded nodes at each iteration. Since the search order of IDDP is not best-first, it must visit all nodes in the final iteration to ensure optimality, which can include many nodes with $f > f^*$. If doubling is achieved, then it is possible for IDDP to visit $4\times$ as many nodes as A* (Schroedl 2005).

Partial Expansion A*

When expanding a node, search algorithms typically generate all successor nodes, many of which have an f that is greater than the optimal solution cost. These nodes take up space on the open list, yet are never expanded. PEA* (Yoshizumi, Miura, and Ishida 2000) reduces the memory needed to store the open list by pruning the successor nodes that do not appear promising i.e., nodes that are not likely to be expanded. A node appears promising if its f does not exceed the f of its parent node plus some constant C. Partially expanded nodes are put back on the open list with a new f equal to the minimum f of the successor nodes that were pruned. We designate the updated f values as $F(n)$.

Yoshizumi, Miura, and Ishida (2000) show that for MSA, PEA* is able to reduce the memory requirement of the classic A* algorithm by a factor of 100. The reduced memory comes at the cost of having to repeatedly expand partially expanded nodes until all of their successors have been generated. However, these re-expansions can be controlled by adjusting the constant C. With $C = 0$, PEA* will only generate nodes with $f \leq f^*$ and will give the worst case overhead of re-expansions. With $C = \infty$ PEA* is equivalent to A* and does not re-expand any nodes. Yoshizumi et al. show that selecting a reasonable value for C can lead to dramatic reductions in the number of nodes in the open list while only marginally increasing the number of expansions.

PEA* is an effective best-first approach to handling problems with large branching factors such as MSA. However, it is still limited by the memory required to store open and closed nodes. This limitation motivates the use of external memory search.

Delayed Duplicate Detection

One simple way to make use of external storage for graph search is to place newly generated nodes in external memory and then process them at a later time. Korf (2008) presents an efficient form of this technique called Hash-Based Delayed Duplicate Detection (HBDDD). HBDDD uses a hash function to assign nodes to files. Because duplicate nodes will hash to the same value, they will always be assigned to the same file. When removing duplicate nodes, only those nodes in the same file need to be in main memory.

Korf (2008) described how HBDDD can be combined with A* search (A*-DDD). The search proceeds in two phases: an expansion phase and a merge phase. In the expansion phase, all nodes that have the current minimum solution cost estimate f_{min} are expanded and stored in their respective files. If a generated node has an $f \leq f_{\text{min}}$, then it is expanded immediately instead of being stored to disk. This is called a *recursive expansion*. Once all nodes within f_{min} are expanded, the merge phase begins: each file is read into a hash-table in main memory and duplicates are removed in linear time.

HBDDD may also be used as a framework to parallelize search (PA*-DDD, Korf 2008). Because duplicate states will be located in the same file, the merging of delayed duplicates can be done in parallel, with each file assigned to a different thread. Expansion may also be done in parallel. As nodes are generated, they are stored in the file specified by the hash function.

During the expand phase, HBDDD requires only enough memory to read and expand a single node from the open file; successors can be stored to disk immediately. During the merge phase, it is possible to process a single file at a time. One requirement for HBDDD is that all nodes in at least one file fit in main memory. This is easily achieved by using a hash function with an appropriate range. For MSA, one can use the lattice coordinates of states to derive hash values.

Parallel External PEA*

The main contribution of this paper is a new best-first external search that combines the partial expansion technique of PEA* with HBDDD to exploit external memory and parallelism. We call this new algorithm Parallel External Partial Expansion A* (PE2A*). Like A*-DDD, PE2A* proceeds in two phases: an expansion phase and a merge phase. PE2A* maps nodes to *buckets* using a hash function. Each bucket is backed by three files on disk: 1) a file of frontier nodes that have yet to be expanded, 2) a file of closed nodes that have already been expanded during an expansion phase and 3) a file of newly generated nodes that will be expanded in a subsequent expansion phase. PE2A* expands the set of frontier nodes that fall within the current f bound and keeps

track of the minimum f of all frontier nodes that exceed this bound. This value is used to select the f bound for the next expansion phase.

The pseudo code for PE2A* is given in Figure 4. PE2A* begins by placing the initial node in its respective bucket based on the supplied hash function (lines 1–2). The minimum $bound$ is set to the f of the initial state. All buckets that contain a state with f less than or equal to the minimum bound are divided among a pool of threads to be expanded.

An expansion thread proceeds by expanding all frontier nodes in the buckets that fall within the f bound (this f layered search order is reminiscent of Fringe search (Björnsson et al. 2005)). Expansion generates two sets of successor nodes for each expanded node n; nodes with $f \leq F(n) + C$ and nodes with $f > F(n) + C$ (lines 15–16). Successor nodes that do not exceed the f bound are recursively expanded. Nodes that exceed the f bound but do not exceed $F(n) + C$ are appended to files that collectively represent the frontier of the search and require duplicate detection in the following merge phase. Partially expanded nodes that have no pruned successor nodes are appended to files that collectively represent the closed list (lines 22–23). Partially expanded nodes with pruned successor nodes are updated with a new F and appended to the frontier (lines 25–26).

Because PE2A* is strictly best-first, it can terminate as soon as it expands a goal node (line 14). We approximate optimal tie-breaking by sorting buckets. If a solution has not been found, then all buckets that require merging are divided among a pool of threads to be merged in the next phase (line 7).

During the merge phase, each merge thread begins by reading the closed list for the bucket into a hash-table. Like A*-DDD, PE2A* requires enough main memory to store at least the largest closed-list of all buckets that need to be merged. Next, all frontier nodes generated in the previous iteration are streamed in and checked for duplicates against the closed list (lines 28–29). The nodes that are not duplicates are written back out to files that collectively represent the open list for the next iteration. If a duplicate node is found in open, the node with the smaller g value is moved to the open list (lines 30–32).

```
SEARCH(initial)
 1. bound ← f(initial); bucket ← hash(initial)
 2. write(OpenFile(bucket), initial)
 3. while ∃bucket ∈ Buckets : min_f(bucket) ≤ bound
 4.   for each bucket ∈ Buckets : min_f(bucket) ≤ bound
 5.     ThreadExpand(bucket)
 6.   if incumbent break
 7.   for each bucket ∈ Buckets : NeedsMerge(bucket)
 8.     ThreadMerge(bucket)
 9.   bound ← min_f(Buckets)

THREADEXPAND(bucket)
10. for each state ∈ Read(OpenFile(bucket))
11.   if F(state) ≤ bound
12.     RecurExpand(state)
13.   else append(NextFile(bucket), state)

RECUREXPAND(n)
14. if IsGoal(n) incumbent ← n; return
15. SUCC_p ← {n_p | n_p ∈ succ(n), f(n_p) ≤ F(n) + C}
16. SUCC_q ← {n_q | n_q ∈ succ(n), f(n_q) > F(n) + C}
17. for each succ ∈ SUCC_p
18.   if f(succ) ≤ bound
19.     RecurExpand(succ)
20.   else
21.     append(NextFile(hash(succ)), succ)
22. if SUCC_q = ∅
23.   append(ClosedFile(hash(n)), n)
24. else
25.   F(n) ← min f(n_q), n_q ∈ SUCC_q
26.   append(NextFile(hash(n)), n)

THREADMERGE(bucket)
27. Closed ← read(ClosedFile(bucket)); Open ← ∅
28. for each n ∈ NextFile(bucket)
29.   if n ∉ Closed ∪ Open or g(n) < g(Closed ∪ Open[n])
30.     Open ← (Open − Open[n]) ∪ {n}
31. write(OpenFile(bucket), Open)
32. write(ClosedFile(bucket), Closed)
```

Figure 4: Pseudocode for the PE2A* algorithm.

Experiments

To determine the effectiveness of this approach, we implemented A*, PEA*, IDDP, PA*-DDD and PE2A* in Java. To verify that we had efficient implementations of these algorithms, we compared them to highly optimized versions of A* and IDA* written in C++ (Burns et al. 2012). The Java implementations use many of the same optimizations. In addition we use the High Performance Primitive Collection (HPPC) in place of the Java Collections Framework (JCF) for many of our data structures. This improves both the time and memory performance of our implementations (Hatem, Burns, and Ruml in press).

The 15-Puzzle

The first set of rows in Table 1 summarizes the performance of A* on Korf's 100 15 puzzles (Korf 1985). These experiments were run on *Machine-A*, a dual quad-core machine with Intel Xeon X5550 2.66 GHz processors and 48 GB RAM. From these results, we see that the Java implementation of A* is just a factor of 1.7 slower than the most optimized C++ implementation known. These results provide confidence that our comparisons reflect the true ability of the algorithms rather than misleading aspects of implementation details.

The second set of rows in Table 1 shows a summary of the performance results for parallel A* with delayed duplicate detection (PA*-DDD). These experiments were run on *Machine-B*, a dual hexa-core machine with Xeon X5660 2.80 GHz processors, 12 GB of RAM and 12 320 GB disks. We used 24 threads and the states generated by PA*-DDD were distributed across all 12 disks. In-memory A* is not able to solve all 100 instances on this machine due to memory constraints. We compare PA*-DDD to Burns et al.'s highly optimized IDA* solver implemented in C++ (Burns et al. 2012) and a similarly optimized IDA* solver in Java. The results show that the base Java implementation of PA*-DDD is just 1.7× slower than the C++ implementation of

	Time	Expanded	Nodes/Sec
A* (Java)	925	**1,557,459,344**	1,683,739
A* (C++)	516	**1,557,459,344**	3,018,332
PA*-DDD (Java)	1,063	3,077,435,393	2,895,047
PA*-DDD$_{tt}$ (Java)	**447**	1,970,960,927	6,572,005
IDA* (Java)	1,104	18,433,671,328	16,697,166
IDA* (C++)	634	18,433,671,328	**29,075,191**

Table 1: Performance summary on the 15-puzzle. Times reported in seconds for solving all instances.

	Expanded	Generated	Time
IDDP	163,918,426	474,874,541	1,699
A*	**56,335,259**	1,219,120,691	1,054
PEA*	96,007,627	**242,243,922**	1,032
A*-DDD	90,846,063	1,848,084,799	8,936
PE2A*(1 thread)	177,631,618	351,992,993	3,470
PA*-DDD	90,840,029	1,847,931,753	1,187
PE2A*	177,616,881	351,961,167	**577**

Table 2: Results for the 75 easiest instances. Times reported in seconds for solving all 75 instances.

IDA* but faster than the Java implementation. We can improve the performance of HBDDD with the simple technique of using transposition tables to avoid expanding duplicate states during recursive expansions (PA*-DDD$_{tt}$). With this improvement the Java implementation is 1.4× faster than the highly optimized C++ IDA* solver. This gives further evidence that efficient parallel external search can surpass serial in-memory search (Hatem, Burns, and Ruml 2011).

Multiple Sequence Alignment

Next, we evaluated the performance of PE2A* on MSA using the PAM 250 Dayhoff substitution matrix with affine gap costs and the $h_{all,2}$ heuristic. We compared PE2A* with in-memory A*, PEA* and IDDP. All experiments were run on *Machine-B*. The files generated by external search algorithms were distributed uniformly among all 12 disks to enable parallel I/O. We found that using a number of threads that is equal to the number of disks gave best performance. We tried a range of values for C from 0 to 500 and found that 100 performed best.

One advantage of an uninformed search order is that the open list does not need to be sorted. IDDP was implemented using an open list that consisted of an array of linked lists. We found IDDP to be sensitive to the number of bins used when estimating the next bound; 500 bins performed well. We stored all closed nodes in a hash table and to improve time performance did not implement the *Sparsify-Closed* procedure as described in Schroedl (2005). Since we did not remove any closed nodes we did not have to implement divide-and-conquer solution reconstruction.

We used the pair-wise ($h_{all,2}$) heuristic to solve 80 of the 82 instances in the BAliBASE Reference Set 1 benchmark. This was primarily because the memory required to store the $h_{all,3}$ heuristic exceeded the amount of memory available. For example, instance 1taq has a maximum sequence length of 929. To store just one *3-fold* alignment using affine gap costs, we would need to store $929^3 \times 7 \times 4$ bytes or approximately 21 GB, assuming 32-bit integers. The maximum sequence length for the hardest 2 instances is 573, allowing us to fit the $h_{one,3}$ heuristic. For the 75 easiest instances, we used a hash function that used the lattice coordinate of the longest sequence. For all other instances we used a hash function that used the lattice coordinates of the two longest sequences.

Table 2 shows results for solving the 75 easiest instances of BAliBASE Reference Set 1. In the first set of rows we see that IDDP expands 1.7× more nodes than PEA* and nearly 3× more nodes than A*. The total solving time for IDDP takes approximately 1.6× longer than A* and PEA*. These results are consistent with results reported by Schroedl (2005) for alignments consisting of 6 or fewer sequences. We also see that the partial expansion technique is effective in reducing the number of nodes generated by a factor of 5. IDDP generates 1.96× more nodes than PEA*.

In all instances, A* generates many more nodes than it expands. For example, for instance 1sbp A* generates approximately 64,230,344 unique nodes while expanding only 3,815,670 nodes. The closed list is a mere 5.9% of all nodes generated during search. This means that simply eliminating the closed list would not significantly reduce the amount of memory required by search. PEA* generates just 5,429,322 nodes and expands just 5,185,887 nodes, reducing the number of nodes generated by a factor of 11.8 while increasing the number of nodes expanded by a factor of just 1.3.

In the second set of rows of Table 2 we show results for serial and parallel versions of A*-DDD and PE2A*. PE2A* expands nearly 2× more nodes than A*-DDD but generates about 5× fewer nodes; as a result PE2A* incurs less I/O overall and is over 2.6× faster than A*-DDD. The parallel versions of A*-DDD and PE2A* show good speedup. PA*-DDD is faster than IDDP and just 1.1× slower than serial in-memory A*. PE2A* outperforms all other algorithms and is nearly 1.8× faster than serial in-memory PEA* despite using external memory and being more scalable.

The external memory algorithms expand and generate more nodes than their in-memory counterparts for two reasons: 1) the search expands all nodes in the current f layer a bucket at a time and therefore is not able to perform perfect best-first tie-breaking and 2) recursive expansions blindly expand nodes without first checking whether they are duplicates and as a result duplicate nodes are expanded and their successors are included in the generated counts. However, we approximate perfect tie-breaking by sorting buckets and recursive expansions do not incur I/O, so their effect on performance appears minimal.

Finally, Table 3 shows results for solving the 7 most difficult instances of BAliBASE Reference Set 1 using the scalable external memory algorithms, PA*-DDD and PE2A*. We used *Machine-A*, with 48 GB of RAM, to solve the hardest two instances (gal4 and 1pamA). The additional RAM was necessary to store the $h_{one,3}$ heuristic. For all instances PE2A* outperforms PA*-DDD by a significant margin and

	PA*-DDD			PE2A*		
	Expanded	Generated	Time	Expanded	Generated	Time
2myr	**50,752,511**	761,287,665	38:31	122,427,176	**269,511,980**	**8:49**
arp	**64,911,504**	2,012,256,624	55:53	180,889,669	**417,982,883**	**17:05**
2ack	**209,141,122**	6,483,374,782	6:47:50	962,397,295	**2,884,534,067**	**1:45:32**
1taq	**1,328,889,371**	41,195,570,501	2:10:04:05	4,280,877,760	**8,856,703,240**	**8:19:04**
1lcf	**2,361,798,608**	148,793,311,664	3:02:13:07	11,306,418,334	**31,385,983,782**	**1:00:10:08**
ga14	**1,160,112,879**	35,963,499,249	2:01:31:55	3,632,786,687	**8,410,512,588**	**9:30:00**
1pamA	**14,686,835,898**	455,291,912,838	32:14:01:48	67,730,312,850	**173,509,054,532**	**9:08:42:39**

Table 3: Results for the 7 hardest instances of the BAliBASE Reference Set 1. Times reported in days:hours:minutes:seconds.

only PE2A* is able to solve the hardest instance (1pamA) in less than 10 days. To the best of our knowledge, we are the first to present results for solving this instance optimally using affine gap costs or on a single machine. The most relevant comparison we can make to related work is to External IDDP, which took over 182 hours to solve gal4 (Edelkamp and Kissmann 2007) with a slightly stronger heuristic.

Other Related Work

Frontier Search (Korf et al. 2005) saves memory by only storing the open list. To ensure expanded nodes are never regenerated, Frontier Search must keep track of which operators were used to generate each of the states on the open list.

Niewiadomski, Amaral, and Holte (2006) combine parallel Frontier A* search with DDD (PFA*-DDD). A sampling based technique is used to adaptively partition the workload at runtime. PFA*-DDD was able to solve the two most difficult problems (gal4 and 1pamA) of the BAliBASE benchmark using a cluster of 32 dual-core machines. However, affine gap costs were not used, simplifying the problem and allowing for higher-dimensional heuristics to be computed and stored with less memory. The hardest problem required 16 machines with a total of 56 GB of RAM. In their experiments, only the costs of the alignments were computed. Because Frontier Search deletes closed nodes, recovering the actual alignments requires extending PFA*-DDD with divide-and-conquer solution reconstruction. Niewiadomski et al. report that the parallelization of the divide-and-conquer strategy with PFA*-DDD is non-trivial.

Structured Duplicate Detection (SDD, Zhou and Hansen 2004) is an alternative to DDD that exploits structure in the problem to localize memory references. Rather than delay duplicate checking, duplicates are processed immediately, avoiding I/O overhead as duplicate nodes never need to be stored to disk. However, to reduce the number of times a state has to be read from and written to disk, an uninformed search order, such as breadth-first search is often used. It is interesting future work to combine PE2A* with SDD.

Sweep A* (Zhou and Hansen 2003) is a space-efficient algorithm for domains that exhibit a partial order graph structure. IDDP can be viewed as a special case of Sweep A*. Sweep A* differs slightly from IDDP in that a best-first expansion order is followed within each level. This helps find the goal sooner in the final level. However, if levels are defined by antidiagonals in MSA, then the final level contains very few nodes and they are all goals. Therefore, the effect of sorting at each level is minimal. Zhou and Hansen (2004) combine Sweep A* with SDD on a subset of the BAliBASE benchmark without affine gap costs. In our experiments we compared with IDDP as the algorithms are nearly identical and results provided by Schroedl (2005) and Edelkamp and Kissmann (2007) for IDDP used affine gap costs and are thus more relevant to our study.

EPEA* (Felner et al. 2012) is an enhanced version of PEA* that improves performance by predicting the cost of nodes, generating only the promising successors. Unfortunately it is not clear how to predict successor costs in MSA since the cost of an operator cannot be known *a priori*.

Conclusion

We have presented a parallel external-memory best-first search algorithm, PE2A*, that combines the partial expansion technique of PEA* with hash-based delayed duplicate detection. We showed empirically that PE2A* performs very well in practice, solving 80 of the 82 instances of the BAliBASE Reference Set 1 benchmark with the weaker $h_{all,2}$ heuristic and the hardest two instances using the $h_{one,3}$ heuristic. In our experiments, PE2A* outperformed serial PEA*, IDDP and PA*-DDD and was the only algorithm capable of solving the most difficult instance in less than 10 days using affine gap costs, a more biologically plausible cost function. These results add to a growing body of evidence that a best-first search order can be competitive in an external memory setting.

Acknowledgments

We gratefully acknowledge support from the NSF (grants 0812141 and 1150068) and DARPA (grant N10AP20029). We also thank Rong Zhou for helpful discussion and sharing his code.

References

Altschul, S. F. 1989. Gap costs for multiple sequence alignment. *Journal of Theoretical Biology* 138(3):297–309.

Björnsson, Y.; Enzenberger, M.; Holte, R. C.; and Schaeffer, J. 2005. Fringe search: beating A* at pathfinding on game

maps. In *In Proceedings of IEEE Symposium on Computational Intelligence and Games*, 125–132.

Burns, E.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *Proceedings of the Symposium on Combinatorial Search (SoCS-12)*.

Carrillo, H., and Lipman, D. 1988. The multiple sequence alignment problem in biology. *SIAM J. on Applied Mathematics* 48(5):1073–1082.

Dayhoff, M. O.; Schwartz, R. M.; and Orcutt, B. C. 1978. A model of evolutionary change in proteins. *Atlas of protein sequence and structure* 5(suppl 3):345–351.

Edelkamp, S., and Kissmann, P. 2007. Externalizing the multiple sequence alignment problem with affine gap costs. In *KI 2007: Advances in Artificial Intelligence*. Springer Berlin / Heidelberg. 444–447.

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; Sturtevant, N. R.; Schaeffer, J.; and Holte, R. 2012. Partial-Expansion A* with selective node generation. In *Proceedings of AAAI-2012*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics* SSC-4(2):100–107.

Hatem, M.; Burns, E.; and Ruml, W. 2011. Heuristic search for large problems with real costs. In *Proceedings of AAAI-2011*.

Hatem, M.; Burns, E.; and Ruml, W. in press. Implementing fast heuristic search code in Java. *IBM developerWorks*.

Ikeda, T., and Imai, H. 1999. Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoretical Computer Science* 210(2):341–374.

Kobayashi, H., and Imai, H. 1998. Improvement of the A* algorithm for multiple sequence alignment. 120–130.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, 1034–1036.

Korf, R. E. 2003. Delayed duplicate detection: extended abstract. In *Proceedings of the Eighteenth International Joint Conference on Articial Intelligence (IJCAI-03)*, 1539–1541.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 55(6).

Korf, R. 2012. Research challenges in combinatorial search. In *Proceedings of AAAI-2012*.

Lermen, M., and Reinert, K. 2000. The practical use of the A* algorithm for exact multiple sequence alignment. *Journal of Computational Biology* 7(5):655–671.

Needleman, S. B., and Wunsch, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology* 48:443–453.

Niewiadomski, R.; Amaral, J. N.; and Holte, R. C. 2006. Sequential and parallel algorithms for Frontier A* with delayed duplicate detection. In *Proceedings of AAAI-2006*, 1039–1044. AAAI Press.

Sarkar, U.; Chakrabarti, P.; Ghose, S.; and Sarkar, S. D. 1991. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence* 50:207–221.

Schroedl, S. 2005. An improved search algorithm for optimal multiple-sequence alignment. *Journal of Artificial Intelligence Research* 23:587–623.

Thompson; Plewniak; and Poch. 1999. BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with partial expansion for large branching factor problems. In *Proceedings of AAAI-2000*, 923–929.

Zhou, R., and Hansen, E. A. 2003. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-03)*.

Zhou, R., and Hansen, E. A. 2004. Structured duplicate detection in external-memory graph search. In *Proceedings of AAAI-2004*.