Sequential and Parallel Algorithms for Frontier A* with Delayed Duplicate Detection

Robert Niewiadomski[*] and **José Nelson Amaral** and **Robert C. Holte**
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
Email: {niewiado, amaral, holte }@cs.ualberta.ca

Abstract

We present sequential and parallel algorithms for Frontier A* (FA*) algorithm augmented with a form of Delayed Duplicate Detection (DDD). The sequential algorithm, FA*-DDD, overcomes the leak-back problem associated with the combination of FA* and DDD. The parallel algorithm, PFA*-DDD, is a parallel version of FA*-DDD that features a novel workload distribution strategy based on intervals. We outline an implementation of PFA*-DDD designed to run on a cluster of workstations. The implementation computes intervals at runtime that are tailored to fit the workload at hand. Because the implementation distributes the workload in a manner that is both automated and adaptive, it does not require the user to specify a workload mapping function, and, more importantly, it is applicable to arbitrary problems that may be irregular. We present the results of an experimental evaluation of the implementation where it is used to solve instances of the multiple sequence alignment problem on a cluster of workstations running on top of a commodity network. Results demonstrate that the implementation offers improved capability in addition to improved performance.

Introduction

Best-first search algorithms that record the states they have visited in order to avoid expanding the same states multiple times are necessary in problems, such as finding the optimal alignment of multiple biological sequences, where there are many paths to each state. These algorithms are inherently limited by their memory requirements, but two recent advances have extended their applicability considerably. Frontier Search (Korf 1999) reduces the memory required by significantly shortening the list of expanded states. Delayed Duplicate Detection (DDD) reduces the memory needed per node, and is expected to improve locality of reference (Korf 2004). DDD also can allow an algorithm to use disk instead of RAM, but we do not exploit this aspect of DDD in this paper. A technical obstacle, described in (Korf 2004), has prevented these two important ideas from being combined together with the heuristic best-first search algorithm A* (Hart, Nilsson, & Raphael 1968). The first contribution of this paper is to overcome this obstacle and provide a Frontier A* algorithm that uses a form of DDD, FA*-DDD.

This paper's main contribution is a parallel version, PFA*-DDD, of the new FA*-DDD algorithm, designed to run in the aggregate memory available on a cluster of workstations. The key difficulty that the parallel algorithm must overcome is distributing the workload evenly among the workstations without incurring a large overhead. The basic idea behind PFA*-DDD's workload distribution is to partition the current set of open states into n disjoint groups, where n is the number of workstations in the cluster. All work involving states in the i-th group is assigned to the i-th workstation. In particular, each state is represented by an integer, and the partition of the states is defined by dividing the full integer range ($-\infty$ to ∞) into n intervals. On each iteration the algorithm dynamically chooses between two different methods for defining these intervals. The simple method is to use the same intervals on the current iteration as were used on the previous iteration. Used repeatedly, this method can lead to imbalanced workload distribution, but it is a very low-cost method, so is ideal when the number of states to be processed is relatively small. When there are a large number of states to be processed on the current iteration, a more costly method is used. This method inspects a sample of the states to be processed and computes intervals that will balance the workload. By judiciously choosing between these two methods, PFA*-DDD maintains a balanced workload with a minimum of overhead, even when the number of states to be processed, and the distribution of the integers that represent them, varies enormously between iterations. PFA*-DDD copes well with the large network latencies encountered in clusters of workstations. Most of the inter-workstation data-movement is sequential in nature and can be done with a double-buffered communication technique that hides network latency.

PFA*-DDD differs significantly from the disk-based parallel Frontier Breadth-First Search with Delayed Duplicate Detection (FBFS-DDD) algorithm, described in (Korf & Schultze 2005). Their parallelization strategy depends on a user-specified hashing function. Thus the applicability and efficiency of their method depends on the quality and

[*]This research is supported by the Nationals Science and Engineering Research Council of Canada through the Collaborative Research and Development grant, by the Canadian Foundation for Innovation, and by IBM Corporation.

availability of hashing functions for each specific problem. By contrast, PFA*-DDD automatically defines a state partition function and adapts it dynamically as the search progresses. The user only supplies one parameter value, defining "sampling intensity". Moreover the parallelization strategy in (Korf & Schultze 2005) targets a shared-memory system and relies on centralized storage whereas PFA*-DDD targets a distributed-memory system, thereby making it applicable to a shared-memory system, and is oblivious to whether or not storage is centralized or distributed.

We examine the performance of our implementation on a cluster of workstations running on a commodity network by using it to solve two large instances of the multiple-sequence alignment problem. This is a problem well-suited for frontier-search algorithms. Traditional best-first search algorithms, such as A*, cannot solve large instances of this problem because of space constraints, and IDA* cannot handle them because of time constraints.

Experimental results demonstrate that our implementation: (1) attains excellent performance and scalability; (2) efficiently uses the aggregate memory capacity of a cluster of workstations to solve problems that could not be solved using a single workstation. Thus this implementation offers improved capability in addition to improved performance.

Algorithms

We present a sequential algorithm called Frontier A* with Delayed Duplicate Detection (FA*-DDD) and a parallel algorithm called Parallel Frontier A* with Delayed Duplicate Detection (PFA*-DDD). The algorithms combine the Frontier A* (FA*) algorithm (Korf 1999) with a form of Delayed Duplicate Detection (DDD) (Korf 2004).

Overview

Given a graph $G(V, E)$ that is directed and weighted, a pair of vertices $s, t \in V$, and a heuristic function h that is admissible and consistent for t, the algorithms compute the minimum-cost of a path from s to t. The algorithms maintain Open, the list of open vertices, ClosedIn, the list of edges from non-closed vertices to closed vertices, and ClosedOut, the list of edges from closed vertices to non-closed vertices. The algorithms begin with Open, ClosedIn, and ClosedOut being consistent with s being open and there being no closed vertices. While Open is non-empty the algorithms perform a *search step*.

The algorithms execute a search step as follows. Check if t is in Open and the f-value of t is $fmin$. If so, terminate and return the g-value of t as the minimum-cost of a path from s to t. Otherwise, expand every vertex in Open whose f-value is $fmin$ and collect the generated vertices and their g and f-values. Update Open, ClosedIn, and ClosedOut to make them consistent with the expanded vertices being made closed and each distinct non-open and non-closed generated vertex being made open. To update Open remove the expanded vertices and add each distinct generated vertex that neither is in the pre-update Open nor is an end-point of an edge in the pre-update ClosedIn. If a vertex in the post-update Open has a duplicate, make the g and f-values of the vertex equal the minimum of these values in all duplicates. To update ClosedIn remove edges that start at the expanded vertices and add edges that end at the expanded vertices but are not in the pre-update ClosedOut. To update ClosedOut remove edges that end at the expanded vertices and add edges that start at the expanded vertices but are not in the pre-update ClosedIn. If Open is empty at the end of a search step then terminate and return ∞ as the minimum-cost of a path from s to t.

In (Korf 2004) Korf suggests that there is no obvious way to combine FA* with DDD due to the potential of the resulting algorithm to suffer from a "leak-back problem" where a previously expanded vertex is re-expanded. In order for our algorithms to suffer from a leak-back problem a generated vertex that is closed or is to be closed would have to be added to Open during the update of Open. This does not happen because during the update of Open the use of the pre-update ClosedIn filters-out generated vertices that are closed while the use of the pre-update Open filters-out generated vertices that are to be closed.

When G is either undirected or bidirected the algorithms can be simplified by eliminating ClosedOut. The simplification is possible because when G is undirected ClosedIn is equivalent to ClosedOut and when G is bidirected the transpose of ClosedIn is equivalent to ClosedOut.

Records and Record Operations

A *record* is a 5-tuple (v, g, f, In, Out) where $v \in V$ is the v-value, $g \in \mathbb{Z}$ is the g-value, $f \in \mathbb{Z}$ is the f-value, $In \subseteq \{(u, v) \in E\}$ is In-set , and $Out \subseteq \{(v, u) \in E\}$ is the Out-set. Given a set of records $\mathcal{R}$ we *reduce* $\mathcal{R}$ by computing a set of records $\mathcal{R}'$ such that $\mathcal{R}'$ contains as many records as there are distinct v-values of the records in $\mathcal{R}$ and such that for each such v-value u, $\mathcal{R}'$ contains the record r where: $r.v = u$; $r.f = \min_{r' \in \mathcal{R} | r'.v = u} r'.f$; $r.g = \min_{r' \in \mathcal{R} | r'.v = u} r'.g$; $r.In = \bigcup_{r' \in \mathcal{R} | r'.v = u} r'.In$; and $r.Out = \bigcup_{r' \in \mathcal{R} | r'.v = u} r'.Out$. If a set of records is equal to its reduced version then it is *concise*. Given a record r we *expand* r by computing:

- a record r' for each $u \in V$ where $(r.v, u) \in E$ and $(u, r.v) \in E$ and $(r.v, u) \notin r.Out$, such that: $r'.v = u$, $r'.g = r.g + c(r.v, u)$, $r'.f = r.g + c(r.v, u) + h(u)$, $r'.In = \{(r.v, u)\}$, and $r'.Out = \{(u, r.v)\}$;
- a record r' for each $u \in V$ where $(r.v, u) \in E$ and $(u, r.v) \notin E$ and $(r.v, u) \notin r.Out$, such that: $r'.v = u$, $r'.g = r.g + c(r.v, u)$, $r'.f = r.g + c(r.v, u) + h(u)$, $r'.In = \{(r.v, u)\}$, and $r'.Out = \emptyset$;
- a record r' for each $u \in V$ where $(r.v, u) \notin E$ and $(u, r.v) \in E$ and $(u, r.v) \notin r.In$, such that: $r'.v = u$, $r'.g = \infty$, $r'.f = \infty$, $r'.In = \emptyset$, and $r'.Out = \{(u, r.v)\}$.

Records that are computed in the expansion of a record are *generated* records. Given two sets of records $\mathcal{R}$ and $\mathcal{R}'$, we *reconcile* $\mathcal{R}$ with $\mathcal{R}'$ by computing a set of records $\mathcal{R}''$ such that $\mathcal{R}''$ is the set of records produced by reducing the union of the set of the records in $\mathcal{R}$ that have not been expanded and of the set of the records in $\mathcal{R}'$ whose v-value is not the v-value of any of the records in $\mathcal{R}$ that have been expanded.

The Sequential Algorithm

The algorithm computes a concise set of records $\mathcal{X}_d$ for increasing values of d. Each $\mathcal{X}_d$ encapsulates the information in Open, ClosedIn, and ClosedOut after d search steps. Let $fmin_d$ be the minimum f-value of any record in $\mathcal{X}_d$. Let $\mathcal{X}_d^{fmin_d}$ be the set of the records in $\mathcal{X}_d$ whose f-value is $fmin_d$. For $d = 0$, $\mathcal{X}_d = \{(s, 0, h(s), \emptyset, \emptyset)\}$. For $d \geq 0$, the algorithm computes $\mathcal{X}_{d+1}$ as follows. If $\mathcal{X}_d^{fmin_d}$ contains a record whose v-value is t then execution terminates and the g-value of that record is returned as the minimum-cost of a path from s to t. Otherwise, the algorithm expands the records in $\mathcal{X}_d^{fmin_d}$ to compute $\mathcal{Y}_d$ as the set of the records produced by the expansions. Next, $\mathcal{X}_d$ is reconciled with $\mathcal{Y}_d$ to compute $\mathcal{X}_{d+1}$ as the set of records produced by the reconciliation. If either $\mathcal{X}_{d+1}$ is empty or $fmin_d$ is ∞ then execution terminates and ∞ is returned as minimum-cost of a path from s to t.

The Parallel Algorithm

The parallel algorithm is a parallel version of the sequential algorithm that distributes the workload of the sequential algorithm among n workstations. Let $v_{min}, v_{max} \in V$ such that $v_{min} < u \leq v_{max}$ for any $u \in V$. An *n-interval list* is a list of $n + 1$ vertices where the first vertex is v_{min}, the last vertex is v_{max}, and each vertex is less-than or equal-to the vertex following it in the list. During the computation of $\mathcal{X}_d$, records are assigned to workstations according to an n-interval list Λ_d. All records assigned to workstation i have a v-value u such that $\Lambda_d[i] < u \leq \Lambda_d[i+1]$. $\mathcal{X}_{d,i}$ represents the records at workstation i at the beginning of iteration d.

For $d = 0$, $\mathcal{X}_{0,0} = \{(s, 0, h(s), \emptyset, \emptyset)\}$ and $\mathcal{X}_{0,i} = \emptyset$ for $i \neq 0$. Let $fmin_{d,i}$ be the minimum f-value of any record in $\mathcal{X}_{d,i}$, and let $fmin_d$ be the global minimum f-value for any record in $\mathcal{X}_{d,i}$ in any workstations. $\mathcal{X}_{d,i}^{fmin_d}$ is the set of records the records in $\mathcal{X}_{d,i}$ that have an f-value of $fmin_d$.

The algorithm proceeds by expanding the records in all $\mathcal{X}_{d,i}^{fmin_d}$'s. To ensure proper load balance, before this expansion takes place, the records have to be redistributed among the workstations. Therefore, in **phase 1** all workstations exchange the sizes of their local $\mathcal{X}_{d,i}^{fmin_d}$ to obtain the total number of records to be expanded. Workstation i can then determine the indexes of the first and last records that it has to expand. If not all records that are to be expanded by workstation i are stored in its local memory, remote records are transferred to workstation i.

In **phase 2** workstation i expands the records of each $\mathcal{X}_d^{fmin_d}$ assigned to it by the partition in phase 1 to compute $\mathcal{Y}_{d,i}$ as the set of the records produced by the expansions. $\mathcal{Y}_{d,i}$ may contain duplicate records, thus once the expansion is complete, workstation i sorts and reduces $\mathcal{Y}_{d,i}$ in place.

If during the expansion workstation i encounters a record r in $\mathcal{X}_d^{fmin_d}$ whose v-value is t, it raises a termination flag and records the g-value of r. At the end of phase 2, if any workstation has raised the termination flag, execution terminates and the g-value of r is returned as the minimum-cost of a path from s to t.

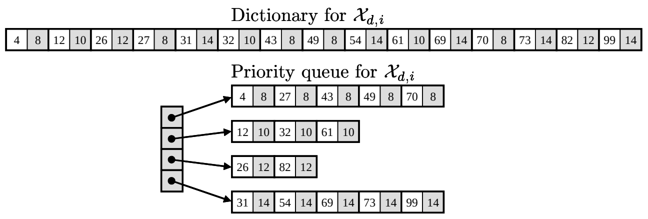

Figure 1: Dictionary and priority queue data structure for $\mathcal{X}_{d,i}$. Assume that a record consists of only the vertex (white) and the f-value (gray).

Next all records in all $\mathcal{X}_{d,i}$'s must be reconciled with all records in all $\mathcal{Y}_{d,i}$'s.

Phase 3 chooses between two reconciliation strategies. With strategy A, the same partition of $\mathcal{X}_d$ from the previous iteration is used and the records of each $\mathcal{Y}_{d,i}$ are sent to their home workstation as determined by that partition. Strategy B relies on a sampling technique to re-partition the records in $\mathcal{X}_d$ and $\mathcal{Y}_d$. To compute the *sampling stride* all workstations send the sizes of $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$ to workstation 0. Once the sampling stride is broadcast, each workstation sends a list of samples from its $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$ to workstation 0. Workstation 0 computes a new interval list Λ_d and broadcasts this list to all workstations. The decision between strategies A and B for reconciliation is based on an estimation of the maximum amount of work by each workstation for each strategy.

The goal of the reconciliation in **phase 4** is to obtain $\mathcal{X}_{d+1,i}$ in each workstation such that the union of all $\mathcal{X}_{d+1,i}$ form $\mathcal{X}_{d+1}$. $\mathcal{X}_{d+1}$ contains one record for each vertex in $\mathcal{X}_d$ and $\mathcal{Y}_d$ that was not in $\mathcal{X}_d^{fmin*}$. If strategy A is used in phase 4, all records of $\mathcal{X}_{d,i}$ are reconciled locally. The records in each $\mathcal{Y}_{d,i}$ that represent a vertex v such that $\Lambda_d[i] < v \leq \Lambda_d[i+1]$ are sent to workstation i for reconciliation. If strategy B is used in phase 4, the new interval list Λ_d, computed in phase 3, is used to re-partition both the $\mathcal{Y}_{d,i}$'s and $\mathcal{X}_{d,i}$'s before reconciliation. At the end of phase 4, the workstations exchange the information to determine the value of $fmin^*_{d+1}$ If $fmin^*_{d+1} = \infty$ the algorithm terminates and ∞ is returned as $\delta(s,t)$.

In **phase 5** workstation i deletes $\mathcal{X}_{d,i}$, increments d and proceeds to phase 1.

Efficiency Issues

Dictionaries and Priority Queues

The reconciliation strategies in the parallel algorithm require techniques to efficiently find a record representing a given vertex v in $\mathcal{X}_{d,i}$; to find a record with a given f-value in $\mathcal{X}_{d,i}$; and to eliminate from $\mathcal{X}_{d,i}$ all the records that have an f-value of $fmin$. Our solution is to maintain two representations of $\mathcal{X}_{d,i}$: a dictionary indexed by vertex, and a bucket-based priority queue ordered by f-value, as shown in Figure 1. The combination of these two data structures has the following advantages: (1) during expansion, a constant-time operation finds the records in $\mathcal{X}_{d,i}$ that have a minimum f-value in the priority queue; (2) because records are sorted in the dictionary, the computation of sampling points is trivial, and the number of data transfers between workstations after repartitioning is reduced; (3) when strategy A is used, a search for existing records that represent the same vertex in the dictionary requires $\log(k)$ time, where k is the number of records in the dictionary; (4) the elimination of all the records with f-value equal $fmin$ from the priority queue can be done in constant time; (5) the traversal of $\mathcal{X}_{d,i}^{fmin_d}$ during reconciliation is sequential, thus improving data locality.

Interval-Based Sampling

The parallel algorithm relies on an interval-based sampling technique. The goal of sampling is to produce a list of vertices that can then be used to partition several ordered lists of vertices. These vertices define the intervals of records assigned to each workstation after the partition. Optimal intervals would distribute the total number of records in the $\mathcal{X}_{d,i}$'s and $\mathcal{Y}_{d,i}$'s evenly. Our implementation approximates these optimal intervals using the following strategy:

- all workstations send the sizes of $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$ to workstation 0.
- using a user-specified sample intensity π, workstation 0 determines the sampling stride to be broadcast to all workstation. The value of the sampling stride is adjusted to ensure that the total number of samples computed by all workstations will be as close as possible to π.
- Each workstation i computes samples from $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$ and sends the vertices to workstation 0. These values are merged into a single sorted list. This list is then divided into n equal-sized intervals to generate the list of vertices used to partition all the $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$. This list of vertices is then broadcast.

Data Streaming

Once workstation i receives the list of vertices that defines the intervals to processed by each workstation it starts sending its records to the appropriate workstations. The communication is efficiently set-up as a series of data streams between the workstations. An advantage of the interval-based partitioning combined with data streaming is that there is no need for the consumer to request records.

At the receiving end, records arrive from each workstation in increasing order of their $r.v$ value. Therefore, fetching a record from each stream and comparing with the records in $\mathcal{X}_{d,i}^{fmin_d}$ and with the records in the dictionary representation of $\mathcal{X}_{d,i}$ is an efficient way to implement duplicate detection and reconciliation with the expanded vertices in a parallel environment. A detailed description of the separation of $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$ into runs and subruns to enable efficient data streaming will be published in a future work.

Deciding Between Strategies A and B

The motivation for the two separate partition strategies in phase 3 is to only use adaptive load balancing when it is cost-effective. Strategy A is the best choice when the number of records to be reconciled in a given iteration is small. Strategy A only updates, as opposed to rebuilding, the dictionary and the priority queue. Thus strategy A implements binary searches of both the dictionary and the record lists in the priority queue.

When the number of records to be reconciled is large, then it is cost-effective to apply strategy B. Strategy B uses the sampling technique to compute new n-interval list, thus adaptively re-balancing the workload through data movement. In this case both the priority queue and the dictionary in each workstation are rebuilt from scratch. Strategy B increases the algorithm's memory requirement because each workstation must store the original $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$while it is computing the new versions of these based on the new partition. To enable problems that generate many records to be solved, we delete portions of $\mathcal{X}_{d,i}$ and $\mathcal{Y}_{d,i}$ as soon as they are sent to their destinations.

Experimental Evaluation

Setup

We implemented PFA*-DDD in ANSI C and used the MPICH implementation of the MPI-2 communication library for communication and synchronization. Each workstation executes two processes, a worker and a server. The worker executes the PFA*-DDD algorithm. The server facilitates access to records residing on its workstation to remote workers. We implemented lists of records as a blocked-and-indexed linked lists. This data structure is akin to a B$^+$-tree with all but the two bottommost levels absent. We used a trivial n-interval list as Λ_0. We used a π of $3n^2$. Though not discussed in the text, we established that, in general, when startegy B is used no workstation reconciles more than $1 + \frac{3n^2}{\pi}$ times the average number of records reconciled by a workstation. A double-buffered and non-blocking scheme is used for the streaming of records in phases 2 and 4 as a means of hiding communication latency. We also implemented the sequential algorithm FA*-DDD in ANSI C in the same manner as PFA*-DDD but without its parallel aspects, such as phases 1 and 3, communication and synchronization, and the server process.

We ran our experiments on two clusters of workstations: C-AMD and C-IBM. C-AMD has 32 dual-processor workstations with 2.4 GHz AMD Opteron 250 processors run and 8 GB of RAM of which 6.8 GB is available to user processes. C-IBM consists of 12 dual-processor workstations with 1.6 GHz IBM PPC970 processors and 4 GB of RAM of which 3.4 GB is available to user processes. Both C-AMD and C-IBM run on top of a dedicated Gigabit Ethernet network and a Linux-based operating system.

We performed experiments using the exact multiple sequence alignment problem. We adopt the formulation of the problem from a work on the use of A* on the problem (McNaughton *et al.* 2002). We focused our experiments on two problem instances drawn from the BAliBASE problem set (Thompson, Plewniak, & Poch 1999), gal4 and 1pama. gal4 has five sequences with lengths varying from 335 to 395. 1pama has fives sequences with lengths varying from

n	Time (hours)	Speedup (vs. $n = 1$)	Speedup Efficiency
1	13.63		
2	6.07	2.2	112%
3	4.12	3.3	110%
4	3.17	4.3	108%
5	2.55	5.3	107%
6	2.14	6.4	106%
7	1.86	7.3	105%
8	1.62	8.4	105%
9	1.45	9.4	104%
10	1.34	10.2	102%
11	1.23	11.1	101%
12	1.15	11.9	99%
13	1.07	12.8	98%
14	1.00	13.7	98%
15	0.95	14.3	95%
16	0.89	15.3	96%
17	0.84	16.2	95%
18	0.80	17.0	94%
19	0.74	18.4	97%
20	0.69	19.8	99%
21	0.66	20.7	99%
22	0.65	21.0	95%
23	0.65	21.0	91%
24	0.65	20.9	87%
25	0.66	20.7	83%
26	0.67	20.4	78%
27	0.68	20.1	74%
28	0.67	20.2	72%
29	0.69	19.9	69%
30	0.70	19.4	65%
31	0.70	19.4	63%
32	0.70	19.5	61%

Table 1: gal4 on C-AMD.

n	Time (hours)	Speedup (vs. $n = 2$)	Speedup Efficiency
2	12.19		
3	7.18	1.7	113%
4	5.42	2.2	112%
5	4.45	2.7	109%
6	3.80	3.2	107%
7	3.26	3.7	107%
8	2.85	4.3	107%
9	2.56	4.8	106%
10	2.32	5.3	105%
11	2.12	5.8	105%
12	1.94	6.3	105%

Table 2: gal4 on C-IBM.

n	Time (hours)	Speedup (vs. $n = 16$)	Speedup Efficiency
16	24.43		
17	21.91	1.1	105%
18	20.77	1.2	105%
20	19.27	1.3	101%
24	15.86	1.5	103%
32	11.92	2.0	102%

Table 3: 1pama on C-AMD.

435 to 572. The implictly-defined graphs of both problems are weighted and directed with edge costs ranging from -17 to 8 and an average in-degree and out-degree of approximately 31. The graph of gal4 has 6.20×10^{12} vertices. The graph of 1pama has 2.82×10^{13} vertices. For both problems the heuristic is computed using two exact 3-way alignments and four exact 2-way alignments, with the longest sequence being involved in the both 3-way alignments. These alignments require approximately 200 MB for gal4 and approximately 480 MB for 1pama.

Results and Analysis

For gal4, 3.11×10^8 records are expanded, 1.03×10^{10} records are generated, and the peak memory requirement was 4.6 GB. For 1pama, 7.95×10^9 records are expanded, 2.57×10^{11} records are generated, and the peak memory requirement was 55.8 GB. The peak memory requirements vary depending on the value of n but are very close to these values. Because of the peak memory requirement we were unable to obtain execution times for the sequential algorithm for gal4 on C-IBM, for the sequential algorithm for 1pama on either cluster, and for the parallel algorithm for 1pama on C-AMD for values of n less than 16 and on C-IBM for any value of n.

Table 1 reports execution time, speedup, and speedup efficiency for gal4 on C-AMD. Table 2 reports execution time, speedup, and speedup efficiency for gal4 on C-IBM. Speedup efficiency compares the speedup obtained by increasing from n_0 to n_1 processors to the ratio $\frac{n_1}{n_0}$. A speedup efficiency of 100% means the speedup is exactly $\frac{n_1}{n_0}$, a greater efficiency indicates super-linear speedup. In the case of C-IBM results, speedup and speedup efficiency are measured using the results for PFA*-DDD for $n = 2$ as a baseline ($n_0 = 2$). On C-AMD speedup efficiency is above linear until n reaches 11, and then becomes sublinear. On C-IBM, speedup efficiency is above linear for all the values of n we used ($n \leq 12$). The decrease in speedup efficiency on C-AMD is the result of the workload being spread too thin, as n increases, to mitigate the increasing overhead of synchronization and communication initiation. Table 3 reports execution time, speedup, and speedup efficiency for 1pama on C-AMD. Speedup and speedup efficiency are measured using the results for PFA*-DDD for $n = 16$ as a baseline. Speedup efficiency starts at above linear and stays above linear regardless of the number of workstations that are used.

Phases 2 and 4 accounted for the majority of the execution time in both problems. The amount of execution time spent in phases 2 and 4, which gradually dropping as n increases, are approximately as follows: in gal4 the amount 98% for $n = 2$ and 65% for $n = 32$; in 1pama 84% for $n = 16$ and 67% for $n = 32$. We assessed workload distribution using the Relative Deviation From the Average (RDFA) metric. RDFA is the ratio of the maximum amount of work performed by any workstation over the average amount of work performed by a workstation. Thus, the lower the RDFA the better, with 1 being the minimum value, in which case all workstations performed an equal amount of work, and n being the maximum value, in which case a single workstation performed all the work. For both problems, phase 2 RDFA was nearly always 1. For both problems, phase 4 RDFA ranged from 1.9 to n when strategy A was used and from 1.0 and 1.2 when strategy B was used. Even though strategy A was used almost 3 times as frequently as strategy B, the amount of execution-time spent in phase 4 when strategy A was used is negligible compared to the amount of execution-time spent in phase 4 when strategy B was used.

Related Work

The Frontier A* (FA*) algorithm is due to Korf (Korf 1999) and is an extension of the A* algorithm (Hart, Nilsson, & Raphael 1968). FA* has been shown to be as time efficient as A* while being more space efficient than A* (Korf *et al.* 2005). The Delayed Duplicate Detection technique is also due to Korf (Korf 2003).

The space-efficiency advantage of FA* over A* stems from it not storing closed vertices. Because of this practice, however, FA* computes only the minimum-cost of a path from s to t. Fortunately, FA* can be extended to also compute a minimum-cost path from s to t without sacrificing its space efficiency advantage over A* while rendering its time efficiency to be only marginally worse than that of A*. The extension computes one or more vertices on a minimum-cost path from s to t and constructs a minimum-cost path from s to t using a divide-and-conquer strategy (Korf & Zhang 2000). Both FA*-DDD and PFA*-DDD can be modified to compute one or more vertices on a minimum-cost path. However, the parallelization of the divide-and-conquer strategy that uses the modified instance of PFA*-DDD is non-trivial. After the initial call to the modified instance of PFA*-DDD, up to two calls can be executed in parallel, with the number of additional calls that can be executed in parallel potentially growing by two with each subsequent call. The issue at hand is whether to allocate all processors to each call and execute them sequentially or to allocate the processors among the calls and execute them in parallel. In addition, if the former approach is to be taken, then the issue of how to allocate the processors among the calls also arises.

Existing parallel formulations of A* often assume that the graph is a tree (Dutt & Mahapatra 1994; Kumar, Ramesh, & Rao 1988; Kumar *et al.* 1994). When the graph is a tree, there is no need to keep a closed list or to check if a generated vertex is in the open or closed list. Moreover, in a tree there is no need to perform reconciliation, which results in negligible communication. In instances where the graph is not a tree the use of a static hashing function is typically recommended as a means of distributing the reconciliation workload, an approach that is limited.

Our sampling-based workload distribution method is akin to that of the parallel sorting algorithm Parallel Soring by Regular Sampling (PSRS) (Shi & Schaeffer 1992). We extend the use of sampling in PSRS by handling arbitrarily sized lists of keys as well as by computing the sampling interval at runtime based on a sampling intensity.

Conclusion

In this paper we presented the Frontier A* with Delayed Duplicate Detection (FA*-DDD) and Parallel Frontier A* with Delayed Duplicate Detection (PFA*-DDD) algorithms. FA*-DDD is a version of the Frontier A* (FA*) algorithm augmented with a form of Delayed Duplicate Detection (DDD). FA*-DDD overcomes the leak-back problem associated with combining FA* with DDD. The novel workload distribution in PFA*-DDD is based on intervals. Experimental results on the multiple sequence alignment problem demonstrate that the implementation offers improved capability as well as improved performance.

References

Dutt, S., and Mahapatra, N. R. 1994. Scalable load balancing strategies for parallel a* algorithms. *Journal of Parallel and Distributed Computing* 22(3):488–505.

Hart, P.; Nilsson, N.; 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.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-05)*, 1380–1385.

Korf, R. E., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2000)*, 910–916.

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

Korf, R. E. 1999. A Divide and Conquer Bidirectional Search: First Results. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1184–1189.

Korf, R. E. 2003. Delayed duplicate detection: Extended abstract. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 1539–1541.

Korf, R. E. 2004. Best-first frontier search with delayed duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-04)*, 650–657.

Kumar, V.; Grama, A.; Gupta, A.; and Karypis, G. 1994. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc.

Kumar, V.; Ramesh, K.; and Rao, V. N. 1988. Parallel best-first search of state-space graphs: A summary of results. In *National Conference on Artificial Intelligence*, 122–127.

McNaughton, M.; Lu, P.; Schaeffer, J.; and Szafron, D. 2002. Memory-efficient A* heuristics for multiple sequence alignment. In *National Conference on Artificial Intelligence (AAAI-02)*, 737–743.

Shi, H., and Schaeffer, J. 1992. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distibuted Computing* 14(4):361–372.

Thompson, J. D.; Plewniak, F.; and Poch, O. 1999. BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics* 15(1):87–88.