A Parallel External-Memory Frontier Breadth-First Traversal Algorithm for Clusters of Workstations

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

Abstract— **This paper presents a parallel external-memory algorithm for performing a breadth-first traversal of an implicit graph on a cluster of workstations. The algorithm is a parallel version of the sorting-based external-memory frontier breadth-first traversal with delayed duplicate detection algorithm. The algorithm distributes the workload according to intervals that are computed at runtime via a sampling-based process. We present an experimental evaluation of the algorithm where we compare its performance to that of its sequential counterpart on the implicit graphs of two classic planning problems. The speedups attained by the algorithm over its sequential counterpart are consistently near linear and frequently above linear. Analysis reveals that the algorithm is proficient at distributing the workload and that increasing the number of samples obtained by the sampling-based process improves workload distribution. Analysis also reveals that the algorithm benefits from the caching of external memory in internal memory that is done by the operating system.**

I. Introduction

A breadth-first traversal of an implicit graph can be used to solve problems such as determining whether or not certain vertices are reachable from a vertex and computing the minimum length of a path from one vertex to another. Applications for a breadth-first traversal of an implicit graph include model checking [6] and pattern database computation [4]. The standard method for performing a breadth-first traversal of an implicit graph involves using the Breadth-First Search (BFS) algorithm [3]. A fundamental limitation of BFS is its space requirement. The amount of space required by BFS is proportional to the number of vertices that are reachable from the starting vertex of the traversal. An alternative to BFS is the Frontier Breadth-First Traversal (FBFT) algorithm [7]. The amount of space required by FBFT is proportional to the maximum number of vertices found at any given distance from the starting vertex of the traversal. In practice, the amount of space required by FBFT is typically a fraction of the amount of space required by BFS.

For sufficiently large implicit graphs FBFT must use external memory (secondary storage), in addition to using internal memory (primary storage). Under such circumstances a FBFT variant called the Frontier Breadth-First Traversal with Delayed Duplicate Detection (FBFT-DDD) algorithm is recommended [8]. The sorting-based external-memory FBFT-DDD algorithm can be used to tackle a graph using external memory in a performance efficient manner [9].

The main contributions of this paper are:

- A parallel version of the sorting-based external-memory FBFT-DDD algorithm that is designed to run on a cluster of workstations. The algorithm distributes the workload among the workstations according to intervals that are computed at runtime via a sampling-based process.
- An experimental evaluation of the algorithm where we compare its performance to that of its sequential counterpart on the implicit graphs of two classic planning problems. The algorithm attains speedups that are consistently near linear and frequently above linear.
- An analysis of runtime measurements relating to the ability of the algorithm to distribute the workload. The analysis reveals that increasing the number of samples that are obtained by the sampling-based process improves workload distribution. The analysis also reveals that the algorithm benefits from the caching of external memory in internal memory that is done by the operating system.

The remainder of this paper is structured as follows. We begin by presenting core notations and definitions. We then proceed to outline the sequential algorithm and the parallel algorithm. We then present the results of an experimental evaluation of the parallel algorithm. We end by presenting related work and a conclusion of our findings.

II. Preliminaries

Let $G(V, E)$ be a directed graph where V is the set of vertices and E is the set of edges. If $(u, v) \in E$ then v is a successor of u and u is a predecessor of v. We assume that for any $u, v \in V$, u is a successor of v if and only if u is a predecessor of v. A path from $u \in V$ to $v \in V$ is a list of vertices where each vertex is a successor of the vertex preceding it in the list. Given $u, v \in V$, v is reachable from u if and only if there exists a path from u to v. The length of a path is the number of vertices in the path minus one. Given $u, v \in V$, the distance from u to v is the minimum length of any path from u to v.

Definition 2.1: Given $v \in V$, a breadth-first traversal *of G starting at v is a traversal of G where each vertex that is reachable from v is visited exactly once and where vertices are visited in a non-decreasing order of their distance from v.*

We assume that G is an implicit graph in the sense that each vertex in V is described in some language, that we obtain the description of every succcessor of a vertex by applying the successor function to the description of the vertex, and that we obtain the description of every predecessor of a vertex by applying the predeccessor function to the description of the vertex. We compare vertices in terms of the magnitudes of their descriptions. Let v_{min} and v_{max} be dummy vertices such that the magnitudes of the descriptions of v_{min} and v_{max} are equivalent to $-\infty$ and ∞, respectively.

Definition 2.2: A record *is a pair (v, S) where $v \in V$ and $S \subseteq \{(v, u) \in E\}$. Given a record r, let $r.v$ denote the v element of r and $r.S$ denote the S element of r.*

Given a record r, we refer to $r.v$ as the vertex of r and to $r.S$ as the successor edge-set of r.

Definition 2.3: Given a set of records $\mathcal{R}_0$, we reduce *$\mathcal{R}_0$ by computing a set of records $\mathcal{R}_1$ such that: (1) $\mathcal{R}_1$ consists of as many records as there are vertices in the set of the vertices of the records in $\mathcal{R}_0$, and (2) for each u in the set of the vertices of the records in $\mathcal{R}_0$, $\mathcal{R}_1$ contains a record whose vertex is u and whose successor edge-set is the set of the edges in the successor edge-sets of the records in $\mathcal{R}_0$ whose vertex is u.*

Given a set of records $\mathcal{R}$, we refer to the set of records obtained by reducing $\mathcal{R}$ as the reduced instance of $\mathcal{R}$. Given a set of records $\mathcal{R}$, we say that $\mathcal{R}$ is concise if and only if $\mathcal{R}$ is equal to the reduced instance of $\mathcal{R}$. Given two sets of records $\mathcal{R}_0$ and $\mathcal{R}_1$, we say that $\mathcal{R}_0$ is equivalent to $\mathcal{R}_1$ if and only if the reduced instance of $\mathcal{R}_0$ is equal to the reduced instance of $\mathcal{R}_1$.

A run is a list of records where the records appear in a non-decreasing order of their vertices and where no record appears more than once. An external-memory run is a run that resides in external memory. A subrun maps a sublist of a run. A subrun consists of the location of a run and the offsets of the first and last records of the subrun in the run. An external-memory subrun is a subrun that maps a sublist of an external-memory run.

Given a list of records L we sort r by sorting the records in L such that they appear in a non-decreasing order of their vertices. Given a run R, we reduce R by making the set of records in R concise. Given k runs, we merge the runs by performing a k-way merge of the runs during which records are merged in a non-decreasing order of their vertices. Similarily, given k sorted lists of vertices, we merge the lists by performing a k-way merge of the lists during which vertices are merged in a non-decreasing order of their vertices.

III. The Sequential Algorithm

Given $s \in V$, the sorting-based External-Memory Frontier Breadth-First Traversal with Delayed Duplicate Detection (FBFT-DDD$_{\text{EM}}$) algorithm performs a breadth-first traversal of G starting at s.

The algorithm performs the traversal by computing a set of records $\mathcal{X}_d$ for successive values of d from $d = 0$ to $d = d_{max}$ where d_{max} is the maximum distance from s of any vertex that is reachable from s. An $\mathcal{X}_d$ is a concise set of records where: the set of the vertices of the records in $\mathcal{X}_d$ is the set of the vertices in V whose distance from s is d, and the set of the edges in the successor edge-sets of the records in $\mathcal{X}_d$ is the set of the edges in E that start at vertices whose distance from s is d and end at vertices whose distance from s is $d-1$.

Definition 3.1: Given a record r, we expand *r by computing a record $(u, \{(u, r.v)\})$ for each $u \in V$ that is successor of $r.v$,* i.e. *$(r.v, u) \in E$, such that $(r.v, u) \notin r.S$.*

Given a record r, we refer to each record computed by expanding r as a record that is generated by expanding r. Let $\mathcal{Y}_d$ be the set of the records generated by expanding the records in $\mathcal{X}_d$.

Definition 3.2: Given two sets of records $\mathcal{R}_0$ and $\mathcal{R}_1$, we reconcile *$\mathcal{R}_0$ with $\mathcal{R}_1$ by computing a set of records $\mathcal{R}_2$ such that $\mathcal{R}_2$ is the reduced instance of the set of the records in $\mathcal{R}_0$ whose vertex is not the vertex of any record in $\mathcal{R}_1$.*

The algorithm stores each $\mathcal{X}_d$ as an external-memory run X_d and each $\mathcal{Y}_d$ as a list of external-memory runs $\boldsymbol{Y}_d$. The set of the records in X_d is concise and is equal to $\mathcal{X}_d$. The set of the records in each run in $\boldsymbol{Y}_d$ is concise and is disjoint from the set of the records in every other run in $\boldsymbol{Y}_d$ with the set of the records in the runs in $\boldsymbol{Y}_d$ being equivalent to $\mathcal{Y}_d$.

The algorithm uses two external-memory algorithms called Expand$_{\text{EM}}$ and Reconcile$_{\text{EM}}$:

Expand$_{\text{EM}}$: Given an external-memory run R_0, the algorithm expands the records in R_0 to produce a list of external-memory runs $\boldsymbol{R}_1$ such that the set of records in each run in $\boldsymbol{R}_1$ is concise and is disjoint from the set of the records in every other run in $\boldsymbol{R}_1$, and such that the set of the records in the runs in $\boldsymbol{R}_1$ is equivalent to the set of the records generated by expanding the records in R_0. The algorithm commences execution by setting $\boldsymbol{R}_1$ to empty and an internal-memory list of records T to empty. The algorithm then executes a scan of the records in R_0 during which it expands all the records in R_0 while appending generated records to T. Whenever T becomes full or every record in R_0 has been expanded and T is non-empty, the algorithm: sorts T in-place, reduces T in-place, creates a new external-memory run by writing T to external-memory, appends the newly created external-memory run to $\boldsymbol{R}_1$, and sets T to empty.

Reconcile$_{\text{EM}}$: Given two lists of external-memory runs $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$, the algorithm reconciles the set of the records in the runs in $\boldsymbol{R}_0$ with the set of the records in the runs in $\boldsymbol{R}_1$ to produce an external-memory run R_2 such that the set of the records in R_2 is concise and is equal to the set of records obtained by reconciling the set of the records in the runs in $\boldsymbol{R}_0$ with the set of the records in the runs in $\boldsymbol{R}_1$. The algorithm commences execution by setting R_2 to empty. The algorithm then executes a merge of the runs in $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$. Because records are merged in a non-decreasing order of their vertices the set of the records in the runs in $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$ that have the same vertex appears as a subsequence of the sequence of records produced by the merge. While executing the merge, the algorithm computes a record r and a boolean b for each such subsequence such that r is the single record in the reduced instance of the set of the records in the subsequence and such that b is false unless the subsequence contains a record from a run in $\boldsymbol{R}_1$. Whenever the algorithm obtains the last record in a subsequence, the algorithm stops executing the merge and completes the computation of r and b for the subsequence. The algorithm then appends r to R_2 if and only if b is false. The algorithm then deletes both r and b, and resumes executing the merge.

The algorithm commences execution by initializing d to 1. The algorithm then computes X_0 such that the set of the records in X_0 is $\{(s, \emptyset)\}$. The algorithm proceeds with execution by executing three phases of computation:

Phase 1: Expand$_{\text{EM}}$ is executed to expand the records in $\mathcal{X}_{d-1}$ to obtain $\boldsymbol{Y}_{d-1}$.

Phase 2: If $|\boldsymbol{Y}_{d-1}| = 0$ then execution terminates. Otherwise, Reconcile$_{\text{EM}}$ is executed to reconcile the set of the records in the runs in $\boldsymbol{Y}_{d-1}$ with the set of the records in the runs in $\langle X_{d-1} \rangle$ to obtain X_d.

Phase 3: If $|X_d| = 0$ then execution terminates. Otherwise, X_{d-1} and each run in $\boldsymbol{Y}_{d-1}$ are deleted, d is set to $d+1$, and execution proceeds to phase 1.

The algorithm exhibits good spatial data-reference locality in external-memory because Expand$_{\text{EM}}$ and Reconcile$_{\text{EM}}$ access external-memory in manner where consecutive accesses map to consecutive external-memory addresses. As a result, consecutive external-memory accesses made by Expand$_{\text{EM}}$ and Reconcile$_{\text{EM}}$ can be performed in a single external-memory I/O operation. Furthermore, the manner in which Expand$_{\text{EM}}$ and Reconcile$_{\text{EM}}$ access external-memory permits the use of double-buffered and non-blocking external-memory I/O to hide external-memory I/O latency via the overlapping of external-memory I/O with computation.

In Expand$_{\text{EM}}$ the capacity of T determines the number of runs in $\boldsymbol{R}_1$. The capacity of T should be large enough as to ensure that the number of runs in $\boldsymbol{R}_1$ is managable. In particular, if the number of runs produced by Expand$_{\text{EM}}$ is too large then the efficiency of Reconcile$_{\text{EM}}$ suffers because Reconcile$_{\text{EM}}$ has to merge the runs produced by Expand$_{\text{EM}}$. As the number of runs to be merged by Reconcile$_{\text{EM}}$ increases so does its internal-memory requirement and the overhead it incurs in conducting external-memory I/O.

IV. The Parallel Algorithm

Given $s \in V$, the sorting-based Parallel External-Memory Frontier Breadth-First Traversal with Delayed Duplicate Detection (P-FBFT-DDD$_{\text{EM}}$) algorithm performs a breadth-first traversal of G starting at s.

The algorithm is a parallel version of FBFT-DDD$_{\text{EM}}$ that is designed to run on a cluster of workstations where each workstation has its own internal memory and its own external memory with neither being directly accessible by the other workstations. Let n be the number of workstations to be used by the algorithm. We label the n workstations from 0 to $n-1$.

Definition 4.1: Given a positive integer β and a list of vertices I, I is an β-interval list *if and only if: $|I| = \beta+1$, $I[0] = v_{min}$, $I[\beta] = v_{max}$, and for each $0 \leq j \leq \beta - 1$, $I[j] \leq I[j+1]$.*

Definition 4.2: Given a set of records $\mathcal{R}$, an β-interval list I, and a non-negative integer j where $0 \leq j \leq \beta - 1$, the j-th subset *of $\mathcal{R}$ defined by I is the set of the records in $\mathcal{R}$ whose vertex is greater-than $I[j]$ but less-than or equal-to $I[j+1]$.*

Definition 4.3: Given a run R, an β-interval list I, and a non-negative integer j where $0 \leq j \leq \beta - 1$, the j-th subrun *of R defined by I is the subrun of R that consists of the records in the j-th subset of the set of the records in R defined by I.*

The algorithm represents each $\mathcal{X}_d$ with n sets of records $\mathcal{X}_d^0, \mathcal{X}_d^1, \ldots, \mathcal{X}_d^{n-1}$ and each $\mathcal{Y}_d$ with n sets of records $\mathcal{Y}_d^0, \mathcal{Y}_d^1, \ldots, \mathcal{Y}_d^{n-1}$. Let Γ_d be an n-interval list. For each $0 \leq i \leq n-1$, $\mathcal{X}_d^i$ is the i-th subset of $\mathcal{X}_d$ defined by Γ_d and $\mathcal{Y}_d^i$ is the set of the records generated by the expanding the records in $\mathcal{X}_d^i$. Consequently, each $\mathcal{X}_d^i$ is disjoint from the other $\mathcal{X}_d^i$'s with the set of the records in the $\mathcal{X}_d^i$'s being equal to $\mathcal{X}_d$, and each $\mathcal{Y}_d^i$ is disjoint from the other $\mathcal{Y}_d^i$'s with the set of the records in the $\mathcal{Y}_d^i$'s being equal to $\mathcal{Y}_d$.

The algorithm stores each $\mathcal{X}_d^i$ as an external-memory run X_d^i and each $\mathcal{Y}_d^i$ as a list of external-memory runs $\boldsymbol{Y}_d^i$. The set of the records in each X_d^i is concise and is equal to $\mathcal{X}_d^i$. The set of the records in each run in each $\boldsymbol{Y}_d^i$ is concise and is disjoint from the set of the records in every other run in $\boldsymbol{Y}_d^i$. The set of the records in the runs in $\boldsymbol{Y}_d^i$ is equivalent to $\mathcal{Y}_d^i$. For each $0 \leq i \leq n-1$, X_d^i and each run in $\boldsymbol{Y}_d^i$ resides in the external-memory of workstation i.

Definition 4.4: Given a positive integer α and a run R, the stride-α sample list *of R is the sorted list of j vertices $\langle R[\alpha - 1].v, R[2\alpha - 1].v, \ldots, R[j\alpha - 1].v \rangle$ were $j = \lfloor |R|/\alpha \rfloor$.*

Definition 4.5: Given a positive integer θ and a sorted list of vetices L, S is an θ-splitter list *of L if S is a sorted list of $\theta - 1$ vertices computed as:*

- *if $|L| < \theta - 1$ then S is formed by L followed by $\theta - 1 - |L|$ copies of v_{max};*
- *if $|L| = \theta - 1$ then $S = L$;*
- *if $|L| > \theta - 1$ then $S = \langle L[j-1], L[2j-1], \ldots, L[(k-1)j - 1] \rangle$ where $j = \lfloor |L|/\theta \rfloor$ and $k = \lfloor |L|/j \rfloor$.*

Figure 1 illustrates the location of runs, subruns, sample lists, and splitter lists within the memory hierarchies of the workstations during the execution of the parallel algorithm. The elipse corresponds to the processor, the rectangle corresponds to the internal-memory, and the cylinder to the external-memory.

The algorithm executes two proccesses on each workstation, a worker process and a server process. We refer to the worker process and the server processor of node i as worker i and server i, respectively. All workers and servers execute six phases of computation such that no worker or server proceeds onto the next phase until all workers and servers are finished executing the current phase.

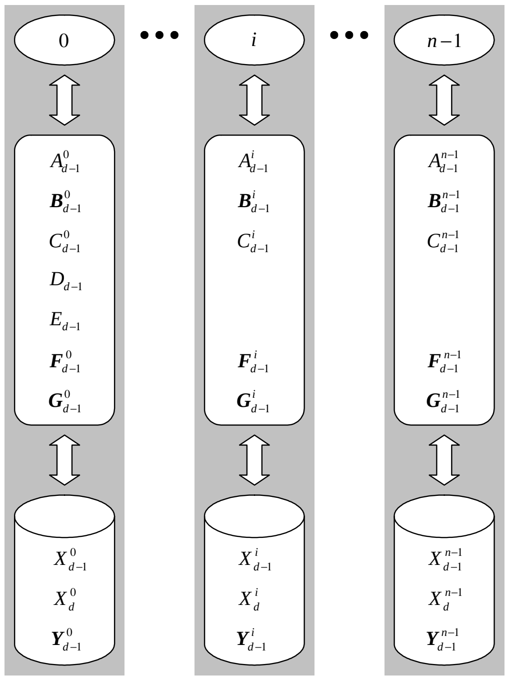


Fig. 1. Location of runs, subruns, sample lists, and splitter lists within the memory hierarchies of the workstations during the execution of the parallel algorithm.

During initialization all workers and servers initialize d to 1, all workers initialize Γ_{d-1} to the list consisting of v_{min} followed by n copies of v_{max}, and all workers initialize a positive integer α, called the sampling stride, to a predetermined value. Worker i then computes X^i_{d-1} such that the set of the records in X^i_{d-1} is the i-th subset of $\{(s, \emptyset)\}$ defined by Γ_{d-1}.

The algorithm proceeds with execution as follows. In phase 1, worker i expands the records in $\mathcal{X}^i_{d-1}$ to obtain $\mathcal{Y}^i_{d-1}$. In phase 2, all workers and servers compute the size of $\mathcal{Y}_{d-1}$. If $\mathcal{Y}_{d-1}$ is empty then all workers and servers terminate execution. Otherwise, worker i computes a sample, in a manner that is dicated by α, of the vertices of the records in $\mathcal{X}^i_{d-1}$ and $\mathcal{Y}^i_{d-1}$. In phase 3, worker 0 computes Γ_d based on the samples obtained by all workers in phase 2. In phase 4, worker i partitions $\mathcal{X}^i_{d-1}$ into the n subsets defined by Γ_d and $\mathcal{Y}^i_{d-1}$ into the n subsets defined by Γ_d. In phase 5, worker i reconciles the set of the records in the i-th subsets of $\mathcal{Y}^0_{d-1}, \mathcal{Y}^1_{d-1}, \ldots, \mathcal{Y}^{n-1}_{d-1}$ defined by Γ_d with the set of the records in the i-th subsets of $\mathcal{X}^0_{d-1}, \mathcal{X}^1_{d-1}, \ldots, \mathcal{X}^{n-1}_{d-1}$ defined by Γ_d to obtain $\mathcal{X}^i_d$. During this phase worker i obtains records that reside in the external memory of remote workstations via the servers of those workstations. In phase 6, all workers and servers compute the size of $\mathcal{X}_d$. If $\mathcal{X}_d$ is empty then all workers and servers terminate execution. Otherwise, worker i deletes $\mathcal{X}_{d-1}$ and $\mathcal{Y}_{d-1}$, all workers and servers set d to $d+1$, and all workers and servers proceed to phase 1. A detailed description of each phase follows.

Phase 1: Worker i executes Expand$_{\text{EM}}$ to expand the records in X^i_{d-1} to obtain $\boldsymbol{Y}^i_{d-1}$. Worker i then sends $|\boldsymbol{Y}^i_{d-1}|$ to all workers and servers.

Phase 2: All workers and servers compute $\sum_{i=0}^{n-1} |\boldsymbol{Y}^i_{d-1}|$. If $\sum_{d-1}^{n-1} |\boldsymbol{Y}^i_{d-1}|$ is zero then all workers and servers terminate execution. Otherwise, worker i computes A^i_{d-1}, the stride-α sample list of X^i_{d-1}, and $\boldsymbol{B}^i_{d-1}$, a list of $|\boldsymbol{Y}^i_{d-1}|$ lists of vertices where for each $0 \le j \le |\boldsymbol{Y}^i_{d-1}| - 1$, $\boldsymbol{B}^i_{d-1}[j]$ is the stride-α sample list of $\boldsymbol{Y}^i_{d-1}[j]$. Worker i then executes a merge of A^i_{d-1} and $\boldsymbol{B}^i_{d-1}[0], \boldsymbol{B}^i_{d-1}[1], \ldots, \boldsymbol{B}^i_{d-1}[|\boldsymbol{B}^i_{d-1}| - 1]$ to obtain C^i_{d-1}. Worker i then sends C^i_{d-1} to worker 0.

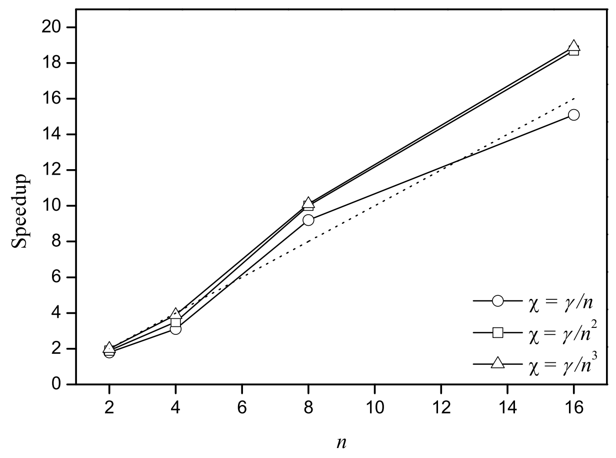
Phase 3: Worker 0 executes a merge of $C^0_{d-1}, C^1_{d-1}, \ldots, C^{n-1}_{d-1}$ to obtain D_{d-1}. Worker 0 then computes E_{d-1} as the n-splitter list of D_{d-1}. Worker 0 then computes Γ_d as $\langle v_{min}, E_{d-1}[0], E_{d-1}[1], \ldots, E_{d-1}[n - 2], v_{max} \rangle$. Worker 0 then sends Γ_d to all workers.

Phase 4: Worker i executes $n-1$ binary-searches on X^i_{d-1} to obtain $\boldsymbol{F}^i_{d-1}$, a list of n external-memory subruns where for each $0 \le j \le n-1$, $\boldsymbol{F}^i_{d-1}[j]$ is the j-th subrun of $\boldsymbol{X}^i_{d-1}$ defined by Γ_d, and $n-1$ binary-searches on each run in $\boldsymbol{Y}^i_{d-1}$ to obtain $\boldsymbol{G}^i_{d-1}$, a list of $|\boldsymbol{Y}^i_{d-1}|$ lists of n external-memory subruns where for each $0 \le j \le |\boldsymbol{Y}^i_{d-1}| - 1$ and $0 \le k \le n-1$, $\boldsymbol{G}^i_{d-1}[j][k]$ is the k-th subrun of $\boldsymbol{Y}^i_{d-1}[j]$ defined by Γ_d. For each $0 \le j \le n-1$, worker i sends the run locations and the first and last record offsets of $\boldsymbol{F}^i_{d-1}[j]$ and $\boldsymbol{G}^i_{d-1}[0][j], \boldsymbol{G}^i_{d-1}[1][j], \ldots, \boldsymbol{G}^i_{d-1}[|\boldsymbol{G}^i_{d-1}| - 1][j]$ to worker j and server i.

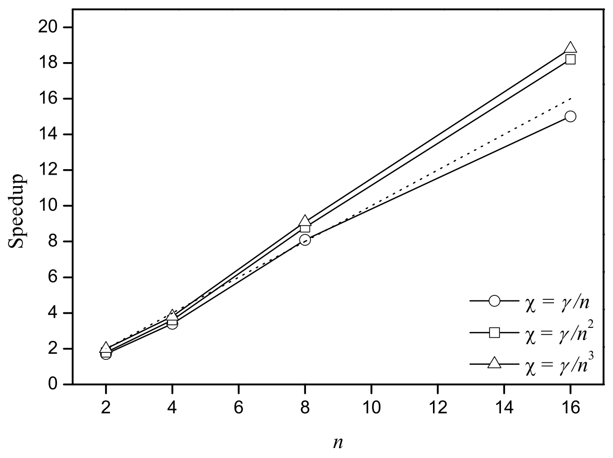
Phase 5: Worker i executes Reconcile$_{\text{EM}}$ to reconcile the set of the records in the subruns in $||_{j=0}^{n-1} \langle \boldsymbol{G}^j_{d-1}[0][i], \boldsymbol{G}^j_{d-1}[1][i], \ldots, \boldsymbol{G}^j_{d-1}[|\boldsymbol{G}^j_{d-1}| - 1][i] \rangle$, where $||$ is the list concatenation operation, with the set of the records in the subruns in $\langle \boldsymbol{F}^0_{d-1}[i], \boldsymbol{F}^1_{d-1}[i], \ldots, \boldsymbol{F}^{n-1}_{d-1}[i] \rangle$ to obtain X^i_d. In executing Reconcile$_{\text{EM}}$, worker i treats the subruns as runs and obtains records in a subrun that resides in the external-memory of node j via server j. Worker i then sends $|X^i_d|$ to all workers and servers.

Phase 6: All workers and servers compute $\sum_{i=0}^{n-1} |X^i_d|$. If $\sum_{i=0}^{n-1} |X^i_d|$ is zero then all workers and servers terminate execution. Otherwise, worker i deletes X^i_{d-1} and each run in $\boldsymbol{Y}^i_{d-1}$. All workers and servers then set d to $d+1$ and proceed to phase 1.

In phases 2 and 3, the algorithm strives to compute Γ_d such that the workers reconcile equal amounts of records in phase 5. The algorithm does so by computing Γ_d based on a sample of the vertices of the records to be reconciled. The size of the sample is inversely proportional to α. The larger the sample the more information that the algorithm has at its disposal to compute Γ_d. It can be shown that for a given value of α, the difference between the average number of records reconciled



(a) STP.



(b) TOH.

Fig. 2. Speedup attained by P-FBFT-DDD$_{\text{EM}}$ over FBFT-DDD$_{\text{EM}}$. The dotted line corresponds to linear speedup.

by a worker in phase 5 and the maximum number of records reconciled by a worker in phase 5 is bounded from above by $(n + x + y)\alpha$, where x is the number of non-empty X^i_{d-1}'s and y is the number of runs in the $\boldsymbol{Y}^i_{d-1}$'s. Thus, the smaller the α the smaller the largest possible imbalance between the amounts of records reconciled by the workers in phase 5. A caveat of using a small α is that the cost of phase 2 and 3 is inversely proportional to α due to the number of samples that are aquired and processed in phase 2 and 3 being inversely proportional to α.

V. Experimental Evaluation

This section presents the results of an experimental evaluation of the parallel algorithm. The main findings of the evaluation are:

- The parallel algorithm achieves speedups over the sequential algorithm that are consistently nearly linear and frequently above linear. In general, as the number of workstations used by the parallel algorithm increases so does the speedup efficiency attained by the parallel algorithm.
- The use of a smaller sampling stride, *i.e.* α, improves the degree of load balancing attained by the parallel algorithm. When using a sampling stride that yields approximately n^2 samples per external-memory run, the parallel algorithm attains a degree of load balancing that is nearly perfect.
- Because the aggregate internal-memory capacity of a cluster of workstations is larger than the internal-memory capacity of a single workstation, the parallel algorithm achieves above linear speedups over the sequential algorithm due to the caching of external-memory in internal-memory that is done by the operating system.

A. Algorithm Implementation

Both FBFT-DDD$_{\text{EM}}$ and P-FBFT-DDD$_{\text{EM}}$ are implemented in ANSI C. Each run is stored in its own file. The MPICH 1.2.6 implementation of MPI is used for communication and synchronization.

Expand$_{\text{EM}}$ uses Quicksort to sort T in-place and the following reduction algorithm to reduce T in-place. The reduction algorithm executes two scans of T, a head scan and a tail scan. The head scan proceeds first. As it moves across T, the head scan computes consecutive records in the reduced instance of T. Whenever the head scan computes a record that record is written to T by the tail scan at the location of the tail scan. Having written a record the tail scan moves one position to the right. When the head scan is finished, the portion of T at the location of the tail scan is deleted.

Reconcile$_{\text{EM}}$ uses a binary-heap to conduct the merge of the runs in $\boldsymbol{R}_0$ and $\boldsymbol{R}_1$. Each element in the heap consists of a record and a tag that indicates whether the record came from a run in $\boldsymbol{R}_0$ or a run in $\boldsymbol{R}_1$.

Expand$_{\text{EM}}$ and Reconcile$_{\text{EM}}$ use double-buffered and non-blocking disk I/O to read from files and to write to files. Reconcile$_{\text{EM}}$ uses non-blocking network I/O to read from remote files.

B. Benchmarks

In our experiments we execute a breadth-first traversal of the implicit graphs of two classic planning problems, Sliding-Tile Puzzle (STP) and four-peg Towers Of Hanoi (TOH). For an excellent overview of these problems we refer the reader to [4]. A record for STP is stored as a 9-byte data structure consisting of a 64-bit integer that encodes the vertex and an 8-bit integer that encodes the successor edge-set. A record for TOH is stored as a 10-byte data-structure consisting of a 64-bit integer that encodes the vertex and a 16-bit integer that encodes the successor edge-set. The starting vertex in STP experiments is the vertex that corresponds to the tiles being positioned in the increasing order of their labels from left to right and from top to bottom with the blank being in the top left corner. The starting vertex in TOH experiments is the vertex that corresponds to all the disks being arranged on

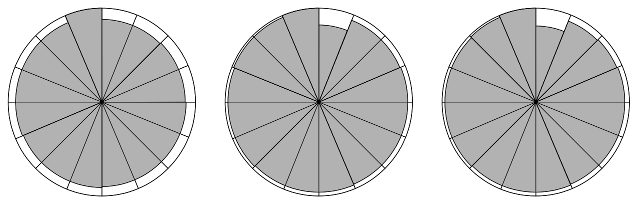

(a) $\alpha = \gamma/n$. (b) $\alpha = \gamma/n^2$. (c) $\alpha = \gamma/n^3$.

Fig. 3. Phase 1 workload distribution for STP and $n = 16$.

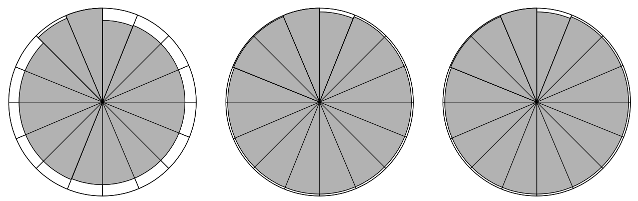

(a) $\alpha = \gamma/n$. (b) $\alpha = \gamma/n^2$. (c) $\alpha = \gamma/n^3$.

Fig. 4. Phase 5 workload distribution for STP and $n = 16$.

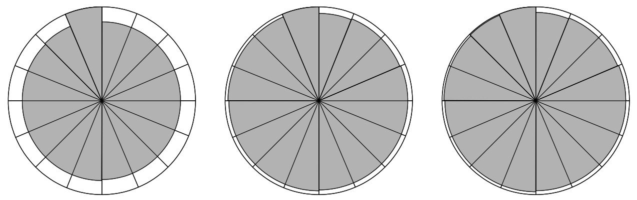

(a) $\alpha = \gamma/n$. (b) $\alpha = \gamma/n^2$. (c) $\alpha = \gamma/n^3$.

Fig. 5. Phase 1 workload distribution for TOH and $n = 16$.

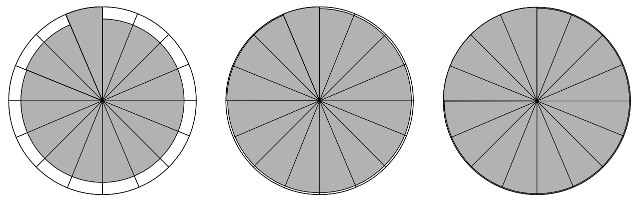

(a) $\alpha = \gamma/n$. (b) $\alpha = \gamma/n^2$. (c) $\alpha = \gamma/n^3$.

Fig. 6. Phase 5 workload distribution for TOH and $n = 16$.

the first peg in the increasing order of their size from top to bottom.

C. Hardware Setup

We run our experiments on a cluster of workstations consisting of 16 dual AMD Opteron 248 workstations. Each node comes equipped with approximately 5 GB of internal memory and approximately 120 GB of external memory. The workstations are connected to each other via a dedicated 1000Base-T Ethernet network running on top of a switch that has a switching capacity of 48 Gb/s. On a per workstation basis, the effective disk I/O bandwidth is approximately one-half of the effective network I/O bandwidth.

The cluster runs the RedHat Enterprise 3.3 Linux operating system. All compilation is done using GCC 3.2 with the -O3 optimization flag.

D. Results

We ran FBFT-DDD$_{\text{EM}}$ and P-FBFT-DDD$_{\text{EM}}$ on the implicit graph of the 2×7 instance of STP and on the implicit graph of the 18-disk instance of TOH. These were the largest instances of either problem that we were able to tackle using FBFT-DDD$_{\text{EM}}$. For simplicity, we refer to the execution of the breadth-first traversal of the implicit graph of the 2×7 instance of STP simply as the *STP problem* and to the execution of the breadth-first traversal of the implicit graph of the 18-disk instance of TOH simply as the *TOH problem*. In the case of STP, 4.36×10^{10} records are expanded and 5.92×10^{10} records are generated. In the case of TOH, 6.87×10^{10} records are expanded and 2.91×10^{11} records are generated.

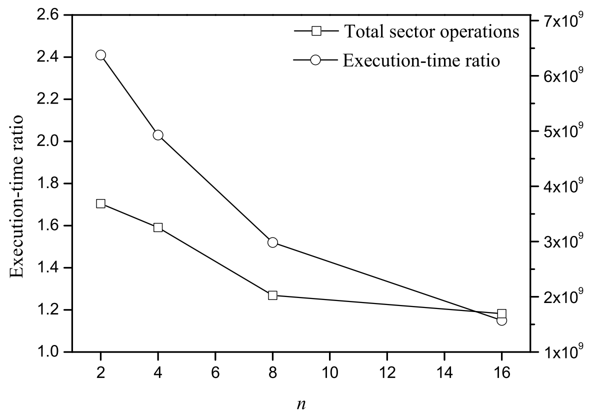
FBFT-DDD$_{\text{EM}}$ took 20.0 hours to solve STP and 47.9 hours to solve TOH. Figure 2 reports the speedup attained by P-FBFT-DDD$_{\text{EM}}$ over FBFT-DDD$_{\text{EM}}$ on the two problems for all combinations of four values of n and three values of α where n is either 2, 4, 8, or 16, and where α is either γ/n, γ/n^2, or γ/n^3. The value of γ is 2^{26} and corresponds to the capacity of T in Expand$_{\text{EM}}$. We found this value of γ to yield the best overall performance on the two problems in a series of tests involving FBFT-DDD$_{\text{EM}}$. The use of $\alpha = \gamma/n^c$ results in approximately n^c samples per run.

In instances where n is either 2 or 4 speedup ranges from near linear to linear. In instances where n is either 8 or 16 speedup ranges from near linear to above linear. In all instances for a given value of n, the use of a smaller α leads to a higher speedup.

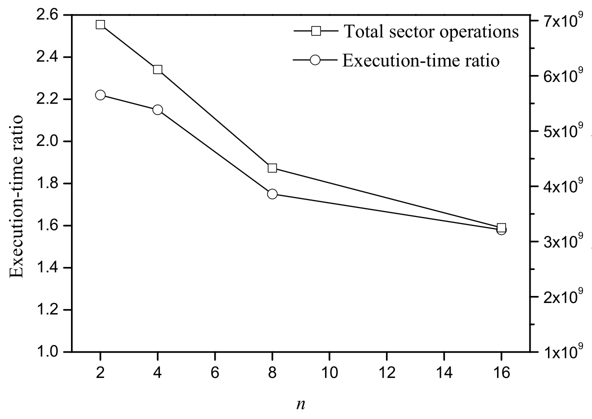
E. Analysis of Load Balancing

Because P-FBFT-DDD$_{\text{EM}}$ is a parallel algorithm its proficiency at distributing the workload is important. We assess the proficiency of the algorithm at distributing the workload by assessing its proficiency at distributing the disk I/O workload in phases 1 and 5. This approach is prudent because phases 1 and 5 account for nearly all of the execution time in all experiments — even in the experiments where n is 16 and α is γ/n^3 — and because all experiments are disk I/O bound. Furthermore, the degree to which the disk I/O workload is distributed in phases 1 and 5 is indicative of the degree to which the computational workload is distributed in phases 1 and 5.

We focus on the experiments where n is 16. For each experiment and for each of phase 1 and phase 5, we examine the *disk I/O distribution ratio* for each workstation. The disk I/O distribution ratio of a workstation is the number of disk I/O operations performed by the workstation over the maximum number of disk I/O operations performed by any workstation. The closer that the disk I/O distribution ratios are to one the better the workload distribution. Figures 3 through 6 report the disk I/O distribution ratios. In these figures a circle is divided into 16 equally sized wedges. Each wedge corresponds to one of the 16 workstation. The ratio of the area of the wedge that is gray over the area of the wedge is the disk I/O distribution ratio for the workstation. Thus, the larger the portion of the circle that is gray the better the workload distribution. From these figures two things are evident: (1) the algorithm is proficient



(a) STP.



(b) TOH.

Fig. 7. The ratio of the wall-time execution time over the process-time execution time (execution-time ratio) and the total number of disk-sector read operations and write operations (total sector operations) in the P-FBFT-DDD$_{\text{EM}}$ experiments where α is γ/n^3.

at distributing the workload, and (2) the use of a smaller α results in better workload distribution.

The degree to which the speedup improves as a result of using a smaller α correlates to the degree to which workload distribution improves as a result of using a smaller α. Using a α of γ/n^2 instead of a α of γ/n results in a large improvement in speedup while using a α of γ/n^3 instead of a α of γ/n^2 results in a marginal improvement in speedup. Concordantly, using a α of γ/n^2 instead of a α of γ/n results in a large improvement in workload distribution while using a α of γ/n^3 instead of a α of γ/n^2 results in a marginal improvement in workload distribution.

F. Analysis of Above Linear Speedups

An investigation of disk I/O sub-system activity reveals that P-FBFT-DDD$_{\text{EM}}$ is able to achieve above linear speedups over FBFT-DDD$_{\text{EM}}$ because of the caching of external memory in internal memory that is done by the operating system. Recall that X^i_{d-1} and each run in $\boldsymbol{Y}^i_{d-1}$ resides in the external memory of workstation i. As n increases the size of these runs becomes smaller while the portion of these runs that can be cached in internal memory becomes larger. As a result, as n increases the number of external-memory accesses that hit the external-memory cache increases and the algorithm becomes less disk I/O bound. Figure 7 reports total sector operations and the execution-time ratio for the runs where α is γ/n^3. Total sector operations is the sum of the total number of disk sector read operations and write operations performed by all workstations. To obtain the execution-time ratio for an execution with n workstations, we compute the ratio of the wall-clock execution time over the process execution time for each workstation and then take the maximum over the n workstations. The figure illustrates the effect of the algorithm becoming less disk I/O bound as n increases as a result of the number of disk I/O operations that hit the external-memory cache increasing as n increases. In particular, as n increases both the total sector operations and the execution-time ratio decrease.

VI. Related Work

External-memory algorithms and data structures play an important role in applications that process large amounts of data [13]. Classic examples of such applications include file systems and relational databases. Because of the importance of these applications, external-memory algorithms and data structures are a well researched area of computing science. In a survey paper, Vitter uses the widely adopted parallel disk model to perform asymptotic analysis of the efficiency of external-memory versions of fundamental algorithms and data structures [15].

The pioneering work on the frontier graph-search and graph-traversal algorithms can be attributed to Korf [7]. In a work addressing the efficient utilization of external memory in frontier graph-search and graph-traversal algorithms Korf outlines the FBFT-DDD and FBFT-DDD$_{\text{EM}}$ algorithms [9]. Korf and Zhang outline a method for dealing with non-bidirected directed graphs in a frontier graph-traversal algorithm and a method for obtaining a minimum-length path from one vertex to another using a frontier graph-traversal algorithm [11]. Both of these methods are readily applicable to P-FBFT-DDD$_{\text{EM}}$. Zhou and Hansen [17] outline a method for using a breadth-first traversal algorithm to mimick the A* algorithm [5] in graphs featuring uniform cost edges. Their method is also readily applicable to P-FBFT-DDD$_{\text{EM}}$.

The techniques of runtime data consumption patterns [16] and informed internal-memory management [12] produce performance gains that are orthogonal to the improvements generated by P-FBFT-DDD$_{\text{EM}}$ and thus these two techniques can be incorporated in a future implementation of P-FBFT-DDD$_{\text{EM}}$. Phases 2 through 5 of P-FBFT-DDD$_{\text{EM}}$ can be classified as an augmented instance of parallel two-pass disk-to-disk sorting based on sample sorting [14]. Previous work on parallel two-pass disk-to-disk sorting has been based on bucket sorting [1]. The use of the sample sorting approach instead of the bucket sorting approach is motivated by the proficiency of sample sorting to deal with large keys and arbitrary key distributions. Whether the sample sorting or the bucket sorting approach is used, performance is limited by the bandwidth available for disk-to-disk data streaming [2].

Korf and Schultze present a P-FBFT-DDD$_{\mathrm{EM}}$ algorithm that is designed to run on a shared memory-system [10]. Their algorithm employs a workload distribution strategy that relies on a perfect-hashing function that is specified by the user. In contrast, our algorithm employs a workload distribution strategy that relies on intervals that are automatically computed by the algorithm. As a result, our algorithm is a general-purpose solution whereas theirs is not: perfect-hashing functions are not always feasible or practical. In addition, because our algorithm is based on sorting, it is readily extensible to support variable length records. Finally, by virtue of our algorithm being designed for a distributed-memory system, we believe that our algorithm has better scalability.

VII. Conclusion

In this paper we present a parallel external-memory algorithm for performing a breadth-first traversal of an implicit graph on a cluster of workstations. The algorithm distributes the workload according to intervals that are computed at runtime via a sampling-based process. We present an experimental evaluation of the performance of the algorithm where we compare its performance to that of its sequential counterpart on the implicit graphs of two classic planning problems. The speedups attained by the algorithm are consistently near linear and frequently above linear. Analysis reveals that the algorithm is proficient at distributing the workload and that increasing the number of samples obtained by the sampling-based process improves workload distribution. Analysis also reveals that the algorithm benefits from the caching of external memory in internal memory that is done by the operating system.

The main strength of the algorithm is that its workload distribution strategy is both adaptive and automated. Consequently, the algorithm is readily applicable to arbitrary implicit-graphs. In addition, the algorithm does not require the user to specify the the manner in which the workload is to be distributied, *e.g.* by specifying a workload-mapping function. Another strength of the algorithm is its ability to harness the aggregate external-memory capacity of a cluster of workstations. The algorithm can be used to tackle implicit graphs that are otherwise too large to be tackled on a single workstation.

The performance attained by the algorithm on a cluster of workstations consisting of sixteen nodes running on top of a commodity network bodes well for the scalability of the algorithm on larger clusters of workstations especially on those running on top of more capable networks.

Achnowledgements

This research is supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canadian Foundation for Innovation (CFI), and by IBM.

References

[1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 243–254, 1997.

[2] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. The architecural costs of streaming I/O: a comparison of workstations, clusters, and SMPs. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 90–101, 1998.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[4] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artifi cial Intelligence Research*, 22:279–318, 2004.

[5] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

[6] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[7] R. E. Korf. A divide and conquer bidirectional search: first results. In *Proceedings of the Sixteenth International Joint Conference on Artifi cial Intelligence*, pages 1184–1189, 1999.

[8] R. E. Korf. Delayed duplicate detection: extended abstract. In *Proceedings of the Eighteenth International Joint Conference on Artifi cial Intelligence*, pages 1539–1541, 2003.

[9] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the Twentieth National Conference on Artifi cial Intelligence*, pages 650–657, 2004.

[10] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *Proceedings of the Twenty-First National Conference on Artifi cial Intelligence*, pages 1380–1385, 2005.

[11] R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the Seventeenth National Conference on Artifi cial Intelligence*, pages 910–916, 2000.

[12] P. Larson and G. Graefe. Memory management during run generation in external sorting. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 472–483, 1998.

[13] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

[14] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Paralle and Distibuted Computing*, 14(4):361–372, 1992.

[15] J. S. Vitter and E. A.M. Shriver. Algorithms for parallel memory I: two-level memories. Technical report, Brown University, 1992.

[16] L. Zheng and P. Larson. Speeding up external mergesort. *IEEE Transactions on Data and Knowledge Engineering*, 8(2):322–332, 1996.

[17] R. Zhou and E. A. Hansen. Breadth-first heuristic search. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 92–100, 2004.