University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Massively Parallel Breadth First Search Using a Tree-Structured Memory Model

Tom St. John[1], *Jack B. Dennis*[2],
Guang R. Gao[1]

CAPSL Technical Memo 114
December 23, 2011

[1] University of Delaware and
[2] Massachusetts Institute of Technology

Abstract

Analysis of massive graphs has emerged as an important area for massively parallel computation. In this paper, it is shown how the Fresh Breeze trees-of-chunks memory model may be used to perform breadth-first search of large undirected graphs. Overall, the computation can be expressed as a data flow process wherein a set of vertices to be searched is partitioned into a set of sub-domains and processed independently by many concurrent tasks.

The main contributions of the paper are listed below.

- We present the first case study demonstrating the power of the Fresh Breeze program execution model (PXM) in the exploitation of fine-grain parallelism found in irregular applications such as graph algorithms.
- We describe a unique sparse vector representation that represents the set of adjacencies for each vertex.
- We provide an experimental study and analysis of our implementation. An estimate is also made of the performance that might be achieved with a massively parallel system built according to Fresh Breeze principles.

1 Introduction

Analysis of massive graphs has emerged as an important area for massively parallel computation. In this paper, it is shown how the Fresh Breeze tree-based memory model may be used to perform breadth-first search of large undirected graphs.

The main contributions of the paper are:

- We present the first case study demonstrating the power of the Fresh Breeze program execution model in the exploitation of fine-grain parallelism found in irregular applications such as graph algorithms.
- This was achieved through the use of a novel parallel breadth-first search algorithm that is fully determinate. By parallelizing each level of the BFS search into a tree-based dataflow-style merging/sorting network, our algorithm avoids the need for expensive critical sections or atomic operations as seen in prevalent implementations [15].
- Furthermore, we chose a unique sparse vector representation for the set of adjacencies of each vertex. Since the structure of the sparse vectors mirrors the tree structure of the Fresh Breeze memory model, each of such vectors can be readily manipulated and stored using memory chunks in the Fresh Breeze system. The tree-structured memory hierarchy of the Fresh Breeze memory model can then be exploited to efficiently implement the tree-based merge operations.

The memory model and tasking scheme of the Fresh Breeze program execution model (PXM) are reviewed in Section 2. Section 3 introduces the graph search problem and breadth-first search. Our implementation strategy for graph search is presented in Section 4. We analyze

memory usage patterns in Section 5 and the performance on an envisioned massively parallel system architecture is estimated in Section 6. Experimental results from simulation runs are reported in Section 6. Conclusions and Acknowledgements complete the paper.

2 Fresh Breeze Overview

Precise models of the programmer's view of a computer system have a history dating back to the 1970s[17]. Moreover, there is at least one computer system, the Burroughs B6700 [16], that supports a well-conceived PXM encompassing multi-tasking, sharing of objects among concurrent users, and protection from inadvertent or malicious modification. Use of such models has disappeared since computer architecture has been driven by the need to continue running "legacy" software, and the HPC community has stressed performance above other qualities such as multi-user operation, sharing of objects, rational design for multi-threading, and universal data access. The present challenge to design usable massively multicore systems can benefit from a well-designed PXM to guide decisions about system architecture and software structure.

The Fresh Breeze PXM is a response to this challenge. It is defined as the combination of a tree-structured, global, virtual memory model and a tasking model suitable for a computer system supporting fine-grain task scheduling. It is a practical form for the heap-based data flow model of Dennis [7]. A computer system implementing this PXM is best equipped with hardware support for the memory and tasking model, resulting in elimination of all operating system execution cycles for memory management and task scheduling. The result is a system able to efficiently distribute large numbers of independent tasks over the processing cores of a massively parallel machine.

2.1 Memory Model

The memory model uses trees of fixed-size *chunks* of memory to represent all data objects. Chunks are 128 bytes in the present study; each chunk has a unique identifier, its *handle*, that serves to locate the chunk within the storage system, and is a globally valid means of reference to the chunk. Chunks may contain handles, permitting construction of trees of chunks to represent data objects. Chunks are created and filled with data, but are frozen before being shared with concurrent tasks. This policy eliminates data consistency issues and simplifies memory management. Low-cost reference-count garbage collection is used to recover for reuse memory chunks for which no references exist in the system. This supports modular programming in type-safe programming languages.

Such a memory model provides a global addressing environment, a virtual one-level store that may be shared by all user jobs and all processors of a many-core, multi-user computing system. It can extend to the entirety of online storage, replacing the separate access means for files and databases of conventional systems.

2.2 Tasking Model

In the proposed PXM, the basic unit of parallelism is the *task*, roughly the activity of performing a single instance of function activation. The organization of multiple tasks is expressed in a way similar to the spawn/join model for parallel programming of Cilk [12]. A *master task* may spawn one or more *worker tasks* executing independent instances of the same or different functions. Worker tasks may receive data objects (scalar values or handles of chunks) as arguments provided by the parent task, and each worker task contributes the results of its activity to a *continuation task* using a join mechanism [9]. The Fresh Breeze tasking model differs from Cilk in that the master task does not continue after spawning the workers and there is no interaction between the master and the worker or among the workers other than the contribution of each worker to the continuation task at the join. Through repeated use of this scheme, a program can generate an arbitrary hierarchy of concurrent tasks corresponding to available parallelism in the computation being performed. It is expected that the spawn/join mechanism would be implemented by special machine level instructions in a hardware realization of the proposed PXM.

2.3 Envisioned System Structure

A realization of the Fresh Breeze PXM is foreseen as consisting of a multitude of many-core processing chips and an off-chip memory hierarchy [8]. On-chip memory consists of L1 instruction and data caches at each processor, and a shared on-chip L2 cache. Off-chip storage is envisioned to be a multi-level storage system with associative directories at each level that map chunk handles to memory locations.

A significant departure from conventional wisdom is the omission of an interprocessor network for sending data between processors; in the Fresh Breeze system concept, data access by remote processors uses the highest level of memory that contains the data and is accessible by the processor. Given the fine-grain tasking model of the Fresh Breeze PXM, use of an I-structure-like [2] mechanism is expected to be competitive in performance.

There is a low bandwidth network among the processors that supports load distribution by means of a global work-stealing scheme.

Novel features of the many-core processor chip include: (1) Cache memories are organized around chunks instead of typical cache lines; (2) Processor registers are tagged to flag those holding handles of chunks; and (3) A hardware task scheduler implements fast switching among active tasks and a task stealing scheme [18, 13] for load distribution.

2.4 Simulation Studies

A simulator of the envisioned Fresh Breeze system has been built that can model systems with up to 40 processing cores and a two-level memory hierarchy. Programs written using a library interface to an implementation of the Fresh Breeze PXM have been developed for the linear

algebra kernels, dot product, matrix multiply, and the Fast Fourier Transform. Our simulation tool has shown that the use of the Fresh Breeze memory model in these kernel algorithms achieves full utilization of 40 processing cores even for modest problem sizes [10, 11]. Section 6 of this paper reports on simulation experiments using the breadth-first search algorithm discussed in Section 4. These experiments have demonstrated that the fine-grained tasking scheme, coupled with the use of a hardware task scheduler, permits effective automatic load distribution of tasks over 40 processors and suggests the scheme could be effective in massively parallel systems.

2.5 An Example of Fresh Breeze Programming

Programming for the Fresh Breeze PXM typically means identifying phases of the computation and setting up a hierarchy of tasks to perform the computation of each phase. We illustrate this with the program in Figure 1 for counting the number of defined elements in a vector. This simple example is included only to illustate the program structure and features of the present simulator API; it is not used in the BFS implementation. The code is simplified by assuming *size* is a power of 16.

The `CreateJoin` command creates a place (a special *join chunk*) to collect the results of workers. It also identifes the function to be executed by the continuation task when all workers have contributed results. The `Spawn` command spawns a worker task with a specified index and the function code it is to execute. The `JoinFetch` command obtains the join chunk for use by the continuation task, which becomes ready for execution when the last worker finishes. The `JoinUpdate` command is the means used by a worker to put its result in the join chunk.

Let's consider time and space used by this code. The tree of tasks spawned to perform the computation consists of $size/16$ leaf-level tasks and fewer than $size/(16 \times 15)$ tasks processing non-leaf nodes. One chunk of memory is used for each non-leaf node to collect results from workers, and is released once its entries have been summed. Leaf-level tasks use only register memory in the processor. Thus the memory use is less than $size/15 \times 128$ bytes. Every task must include one `ChunkRead` operation to bring the chunk into L1 cache; this read is overlapped with execution of other tasks, so the cost is about ten cycles to save and restore task status.

In the body code, each leaf task performs 16 test and count instructions plus some load-multiples to bring the data in from the L1 cache: 40 cycles, assuming the loop is unrolled. Add a few more cycles for start, finish and the test at the top to see whether the node is leaf or non-leaf, and a total of 50 cycles would be conservative. The contribution of the non-leaf tasks is minor, as these involve just 16 spawn instructions, each interpreted by the hardware to create a task record for the scheduler, say two cycles each for a total of again around 50 cycles.

This illustrates the methodology used to estimate performance of search in Section 6.

```
long CountDefined (Handle vectHandl, long size) {

  if (size < 16) {

    // Leaf node: count defined elements
    // and report to parent continuation.
    long count = 0;
    for (int i = 0; i < 16; i++} {
      if (IsDefined(vector[i]) count++;
    }
    JoinUpdate (count);

  } else {
    // Not at leaf level of the tree.
    // Spawn a task for each subtree.

    handle event = CreateJoin (16, Continue());
    for (int i = 0; i < 16; i++} {
      Spawn (i, CountNonNull (vector[i], size / 16);
    }
    quit;
  }
}

void Continue () {
  Handle dataHandl = JoinFetch ();
  long sum = 0;
  for (int i = 0; i < 16; i++}
      sum += dataHandl[i];
  JoinUpdate (sum);
    // a JoinUpdate at the top level is
    // the same as a return.
}
```

Figure 1: Fresh Breeze code for the counting example.

3 Parallel Breadth First Search

Interest in graph analysis problems and algorithms has been heightened by recognition that data analysis problems of massive scale are amenable to solution through massively parallel computation. Breadth First Search is a method of traversing all vertices and edges of a graph, and is fundamental to many useful graph analysis algorithms.

Given an undirected graph **G** and a vertex of **G** chosen as the root, breadth-first search (BFS) starts from the root vertex and constructs a tree in the graph that includes all vertices reachable from the root (a *spanning tree*).

Let the vertices of **G** be indexed $0, ..., n-1$ where n is the number of vertices contained in **G**. The edges of **G** are specified by a relation, a subset of $\mathbf{V} \times \mathbf{V}$, containing pairs of vertices. A spanning tree of **G** may be represented by a function $P : V \to V$ that maps each vertex to

its parent vertex. By convention, the root vertex is taken to be its own parent.

A straightforward parallel BFS algorithm [4] proceeds in stages that correspond to levels of the spanning tree being constructed. The algorithm uses a search set $\mathbf{S}$, the vertices to be examined in a stage, and the partial function $\mathbf{P}_v : \mathbf{V} \rightarrow \mathbf{V}$ that assigns a parent vertex to each neighbor of vertex v.

For each stage, the algorithm consists of the following steps to produce a new search set $\mathbf{S}'$ and parent assignment $\mathbf{P}'$:

1. For each vertex v in $\mathbf{S}$, define the partial update function $\mathbf{U}_v$ that maps each neighbor of v to a candidate parent u.

2. Combine the maps $\mathbf{U}_v$ for all vertices in $\mathbf{S}$ to obtain $\mathbf{U}$ defined on $\mathbf{S}$, the update function for all neighbors of vertices in $\mathbf{S}$. Conflict occurs if two or more vertices in $\mathbf{S}$ have a common neighbor u. If so, choose one arbitrarily as the candidate parent of u.

3. Update: For the next stage of search, the new search set is the set of neighbors found in this stage (the domain of $\mathbf{U}$), excluding vertices that have already been assigned as parents (the range of function $\mathbf{P}$). The new parent function $\mathbf{P}'$ is the union of $\mathbf{P}$ and $\mathbf{U}$.

The BFS computation has been chosen as representative of large-scale non-numeric problems of interest for future large-scale computing. It has been posed as a challenge to workers with supercomputers, and up-to-date results are posted at [1]. Implementations of parallel BFS have been constructed for both distributed memory and shared memory computer systems.

In a typical distributed memory implementation [19], one divides the set of vertices to be examined into many *domains* and assigns one processor to each domain. This yields many sets of vertices to search; producing sets of pairs $< u, v >$, each identifying a candidate parent v for neighbor u. These candidates must be passed to the processors responsible for the domains containing each neighbor. In conventional multiprocesors, this is frequently accomplished, in part, by interprocessor messaging, often using MPI. Because many messages arrive at each processor asynchronously, there are possibilities for non-repeatable behavior without careful design[15].

The problem of delivering candidate parents to the appropriate procesing domain is a sorting problem – one for which an interconnection network is effectively a Batcher sorting network [3]. The sorting problem is special in that the set of keys is exactly the same size as the set of items to be sorted. As we shall see, the Fresh Breeze implementation uses the merging of sparse vectors as a means of achieving the same effect – essentially a "bucket sort" [4] with a separate bucket for each key.

When using shared memory systems, individual vertices belonging to the set $\mathbf{S}$ are processed in parallel by the system's processing elements. Since it is possible that two vertices found in set $\mathbf{S}$ share a common neighbor, updates to the set $\mathbf{P}$ must be performed atomically. This introduces a source of contention between processing elements and can create bottlenecks during execution[14, 15]. Since there is no possibility of concurrent writes to a common location

in the Fresh Breeze PXM, such mechanisms are not needed. This results in a determinate implementation free from interference among processing elements.

4 Fresh Breeze Implementation

The Fresh Breeze memory model makes use of sparse vectors especially attractive. In the 16-ary tree representation of vectors, subtrees can be omitted if the subtree contains no defined elements of the vector.

To make this concrete, consider a vector of size $n = 16^d$ in which there are k defined elements. if k is much less than $n/16$, then the number of chunks needed is no greater than $d-1$ for each defined element, and no greater than $k \times (d-1)$ for the entire vector. The memory need grows linearly with k, and is only slightly dependent on n.

The principal contribution of this paper is the demonstration that the Fresh Breeze memory model, when implemented together with fine-grain scheduling, can provide competitive performance in massively parallel non-numeric computations such as BFS.

In the Fresh Breeze implementation, vectors are represented by trees of chunks of degree 16. The parent map and search set are each represented by sparse vectors of length n. By "sparse" we mean that many elements of the vector are undefined, and subtrees of the tree-of-chunks representation that hold only undefined elements are omitted. Elements of the search vector **S** consist of a value located at the index of each vertex whose neighbors are to be searched at the current level of the graph traversal. The parent function **P** is represented by a sparse vector with a value at the index of each vertex in the domain of **P**; the value is the index of the parent vertex.

As illustrated in Figure 2, the search uses the graph and a specified root vertex to produce the parent vector **P** representing a spanning tree from the specified root. Its essential work is performed by three functions: BreadthFirstSearch, MergeSparseVectors and UpdateParent, which are discussed in Sections 4.2, 4.3 and 4.4. In this implementation, it is possible to avoid a separate update computation at the end of each phase by identifying vertices in the search set during execution of the BreadthFirstSearch function. Further, because the updated parent function is not needed for the next search phase, this processing may proceed in parallel with search computation.

4.1 Graph Representation

In our implementation, a graph of size n is represented as a Fresh Breeze vector **G** containing n elements, where $\mathbf{G}_v$ is the handle of a sparse vector of size n that represents the adjacency list of vertex v. An element of the adjacency vector for vertex v is defined if an edge exists between v and the vertex corresponding to the element's position in the vector. The value of each defined element is set to v, so that each adjacency vector of the graph contains a unique value for each defined element. In this way, the adjacency vector represents both vertices of

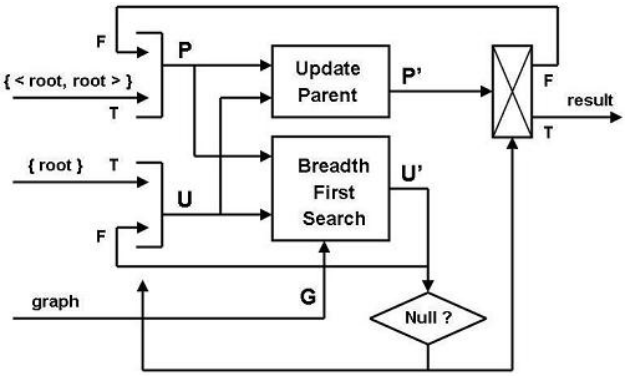

Figure 2: Dataflow diagram of parallel breadth first search.

each edge connecting vertex v with a neighbor vertex: the index u of a defined element is the neighbor vertex, and its value v is the origin of the edge.

The amount of memory needed for a graph represented this way is discussed in Section 5. Note that this graph representation is independent of the chosen root vertex and may be used for any number of BFS computations for different roots.

4.2 Searching the Graph

The search process at each level of graph traversal is guided by a function called `BreadthFirstSearch` (Figure 3. The inputs for this function are `graph`, `update` and `parent`, which correspond to segments of the graph, update and parent vectors. In cases where the inputs correspond to non-leaf chunks, the function recursively calls `BreadthFirstSearch` on individual subtrees which contain defined elements of `update` (since the elements contained in `search` are a subset of the elements contained in `update`) and creates a continuation task called `CollectVectors, Figure 4` which structures the hierarchical merging process. In cases where the inputs are leaf chunks, the function calls `MergeSparseVectors` on the set of adjacency lists rooted in the `graph` chunk which also have a defined element in the corresponding index of the `update` chunk and an undefined element in the corresponding index of the `parent` chunk, since the combination of these two conditions implies that a vertex belongs to the `search` vector.

The `CollectVectors` function receives the vectors produced after the lower-level merge operations have completed and then merges them further. At each successive level, the number of vectors is reduced by a factor of 16. This process continues until the adjacency lists for all vertices examined in the current level of the graph traversal are merged into a single vector.

4.3 Merging Sparse Vectors

The merging of sparse vectors is perhaps the most interesting part of our BFS implementation.

Input to this function is `vecHandl`, the handle of a chunk that contains 16 handles of chunks

```
Chunk BreadthFirstSearch (Handle graph, Handle update, Handle parent, long size ) {

  if (size < 16 ) {
    // The given handles represent leaf chunks
    // of the respective vectors.

    Chunk vectors = new Chunk;
      // A new chunk for the roots of the
      // individual adjacency lists
    int count = 0;
      // A count of elements the
      // sub-domain to be searched
    Handle defVector;
      // The handle of a non-null
      // adjacency list

    for (int i = 0; i < 16; i++) {

      if ((IsDefined(update[i])) && !(IsDefined(parent[i]))) {
        subtree = graph[i];
        vectors[count] = subtree;
        count++;
      }
    }
    if (count > 1) {
      CreateJoin(1, ForwardHandle() );
      Spawn(0, MergeSparseVectors(vectors) );
      quit();
    }
    if (count == 1)
      JoinUpdate (defVector);
    if (count == 0)
      JoinUpdate (UnDef);

  } else {
    // The given handles represent non leaf
    // chunks of the respective vectors.

    createJoin(16, CollectVectors() );

    for (int i = 0; i < 16; i++) {
      if(IsDefined (update[i]))
        Spawn(i, BreadthFirstSearch(graph[i], update[i], parent[i]));
      else
        Spawn(i, UpdateNull());
    }
    quit();
  }
}
```

Figure 3: The BreadthFirstSearch function.

that are root chunks of trees representing sparse vectors to be merged. As in the illustrative code above, the function body has two parts, shown in Figures 5 and 6: The first part processes leaf chunks; the second part processes non-leaf chunks that represent subtrees of the set of 16 vectors.

For leaf chunks, the program chooses, for each each index from 0 to 15, a single value from

```
void CollectVectors() {
  Handle vecHandl = JoinFetch();

  Chunk new_vectors = new Chunk;
    // A chunk for the roots of
    // individual vectors
  int count = 0;
    // A count of non-null vectors
  Handle defVector;
    // The handle of a non-null vector

  for (int i = 0; i < 16; i++) {
    // Iterate over root chunks of the
    // given vectors
    if (IsDefined(vecHandl[i])) {
      defVector = vecHandl[i];
      new_vectors[count]=defVector;
      count++;
    }
  }

  if (count > 1) {
    CreateJoin(1, ForwardHandle() );
    Spawn(0, MergeSparseVectors(new_vectors) );
    quit();
  }
  if (count == 1)
    JoinUpdate(defVector);
  if (count == 0)
    JoinUpdate(UnDef);
}

void ForwardHandle() {
  Handle dataHandl = JoinFetch();
  JoinUpdate (dataHandl[0]);
}

void UpdateNull () {
  JoinUpdate (Undef);
}
```

Figure 4: The CollectVectors function.

the last among the 16 chunks containing a defined element at that index (I). Defined values found at lesser indices are overwritten.

For non-leaf chunks, the merge process is continued in parallel for each of the 16 leaf or non-leaf chunks at the next lower level. Of course, if all elements at some index are undefined, a null reference is placed in the join chunk instead of the handle of a subtree or leaf chunk. Also, if only one of the sixteen given chunks has a non-null element at some index, then the handle of that subtree or leaf chunk is placed in the join chunk and no recursive call of MergeSparseVectors is made.

```
Chunk MergeSparseVectors (Handle vecHandl, long size) {

  Chunk vecChunk = ChunkRead (vecHandl);
    // Read the chunk containing handle
    // of trees to be merged.

  if ( size < 16 ) {

    // The given chunks are leaf
    // chunks of the 16 vectors.

    Chunk leaf = new Chunk;
      // A new leaf chunk for the
      // merged vector

    for (int i = 0; i < 16; i++) {
      // Iterate over indices of elements
      // in the 16 given root chunks

      long element;
      for (int j = 0; j < 16; j++) {
        // Iterate over the 16 leaf chunks
        // at the current element index

        element = vectors[i][j];
        if (IsDefined(element))
          leaf[i] = element;
      }
    }

    // Pass the chunk of merged values to the
    // parent continuation
    JoinUpdate (HandleOf(leaf));
    quit();

  } else {
```

Figure 5: MergeSparseVectors, case of leaf chunks.

4.4 Update Parent

At each level, a new parent vector is generated by using the `MergeSparseVectors` function to combine the candidates vector with the current parent vector. The existing parent vector is given priority to ensure that the assignment obtained in earlier stages remains unchanged.

5 Analysis

In this section, we formulate upper and lower bounds on the number of memory chunks used in the representation of a sparse vector and use this information to determine bounds on the number of tasks executed while performing BFS. We also discuss the amount of memory space occupied by our chosen graph representation and the necessary memory bandwidth. Throughout our analysis, we use n for the number of vertices in the graph and assume that n is a power

```
    // The given chunk holds handles of
    // non-leaf chunks of the 16 trees.

    join = JoinCreate (16, DoneMerge () );

    for (int i = 0; i < 16; i++) {

      // Iterate over element indices of chunks
      Chunk new_vectors = new Chunk;
        // A new chunk for the roots of the
        // individual subtrees
      int count = 0;
        // A count of vectors having a non-null
        // element at index i.
      Handle subtree;
        // The handle of a non-null subtree

      for (int j = 0; j < 16; j++) {

        // Iterate over root chunks of the
        // given vectors.
        if (vectors[i][j] != UnDef) {
          subtree = vectors[i][j];
          new_vectors[count] = subtree;
          count++;
        }
      }
      if (count > 1)
        Spawn (i, MergeSparseVectors(new_vectors) );
      if (count == 1)
        Spawn (i, MergeOne(subtree) );
      if (count == 0)
        Spawn (i, MergeOne(UnDef) );
    }
    quit();
  }
}

void MergeOne (Handle tree) {
  JoinUpdate (tree);
}

void DoneMerge () {
  Handle dataHandl = JoinFetch ();
  JoinUpdate (dataHandl)
    // Pass the chunk containing subtrees
    // to the parent task and quit.
}
```

Figure 6: MergeSparseVectors, case of non-leaf chunks.

of 16, $n = 16^d$. We use m for the number of edges in the graph.

5.1 Chunks to Represent a Sparse Vector

We consider a sparse vector $\mathbf{V}$ of size n with x defined elements. The 16-ary tree of chunks representing $\mathbf{V}$ will have a depth of d, where there is a single chunk at level 0 and a maximum of 16^{d-1} chunks at the leaf level $d-1$. To determine an upper bound, we consider the worst case, where a separate leaf chunk is used for each defined element. Let $S[h]$ be the number of chunks in the representation of $\mathbf{V}$ at level h. $S[h]$ cannot be any greater than x, but also it cannot be any greater than the number of chunks 16^h needed to represent a fully defined vector of size n. Thus an upper bound on the number of chunks at level h is

$$S[h] \leq \min(x, 16^h)$$

and the total for the tree is

$$S_T \leq \sum_{h=0}^{d-1} \{\min(x, 16^h)\ \}$$

Using $b = \lceil log_{16}(x) \rceil$ and assuming $x \geq 16$, we can split this sum into two parts, the first where the number of defined elements bounds, and the second where the maximum size of representation bounds:

$$S_T \leq \sum_{b}^{d-1} x \ + \ \sum_{h=0}^{b-1} 16^h$$

For simplicity, in the remaining discussion we will use

$$S_T \leq \sum_{0}^{d-1} x \ = \ d \cdot x$$

as the upper bound, ignoring the constraint imposed by the maximum possible size at each depth.

For a lower bound, we assume a best case in which the defined elements of the sparse vector are densely packed into the smallest number of chunks sufficient to hold them. The leaf level of the tree will have $\lceil x/16 \rceil$ chunks containing defined elements, and each higher level will have fewer chunks by a factor of 16. Levels 0 through b will have just a single chunk, where $b = d - \lceil log_{16}(x) \rceil$. Using the sum of geometric series, we find

$$S_T \geq \frac{16^b - 1}{16 - 1} \ + \ b$$

or, dropping the second term

$$S_T \geq (x - 1)/15$$

5.2 Merge Sparse Vectors

In our implementation of BFS, each level of search is performed by merging the set of adjacency vectors of vertices in the search set. These vectors are gathered by a task hierarchy that visits successively smaller domains of vertices. Let us call the tasks in this hierarchy Master Tasks. Each Master Task is the root task of a hierarchy of Merge Tasks that performs the merge of up to 16 sparse vectors.

Assume that the set of adjacency vectors to be merged in level s of search contains a total of x defined elements. To determine an upper bound on the number of Master Tasks, note that at most x leaf-level domains can spawn a Merge Task hierarchy. This constraint also holds at all higher levels of Master Tasks, so the number of Master Tasks executed for search level s is subject to the bound:

$$U(s) \leq \ (d-1) \ \cdot \ x$$

In a Merge Task Hierarchy, a merge task is performed only if there are two or more subtrees or leaf chunks to be merged. For x defined elements, there can only be x leaf level merge tasks among all Merge Task hierarchies. As before, the same bound applies at all higher levels. Therefore the number of Merge Tasks is subject to the same bound:

$$L(s) \leq \ (d-1) \ \cdot \ x$$

The total number of tasks performed while merging adjacency vectors in a complete search is

$$T_S \leq \ \sum_s (\ U(s) + L(s)\ ) \ = \ 2 \ \cdot \ (d-1) \ \cdot \ \sum_s x(s)$$

But $\sum_s x(s)$ cannot be greater than twice the number of edges in the graph, so our upper bound is

$$T_S \leq \ 4 \ \cdot \ (d-1) \ \cdot \ m$$

A lower bound is determined by assuming the $e(s)$ defined elements of the final merged sparse vector of a search level are densely packed. The sum of $e(s)$ over all search levels must be at least $n-1$, one less than the number of vertices in the graph (or the number of vertices in the connected component being traversed, if the graph is not fully connected). The number of leaf-level tasks in the collection of Merge Task hierarchies must be at least $e(s)$ / 16, and each higher level less by a factor of 16. As in determining the least number of chunks to represent a sparse vector, the total for all levels of the Merge Task tree is no less than

$$e(s) \ / \ 15$$

Similarly, the number of leaf tasks in the Master Task tree must be at least $e(s)$, and because the sum of $e(s)$ over all search levels must be at least $n-1$, our lower bound is:

$$T_S \geq 2 \cdot (n - 1) / 15$$

The conclusion of this analysis is that the number of tasks executed in BFS grows at worst linearly with the number of edges in the graph, increased by a factor proportional to $log_{16}(n)$. In the case of scale-free graphs generated for the Graph500 benchmark, the number of edges in the graph is at most $16 * n$, so we can say that the number of tasks grows at worst linearly with the number of vertices in the graph.

5.3 Memory for the Graph

The graph representation used in the BFS algorithm is an n-vector having sparse vectors as its elements. The memory space required to store this form of a graph is no more than the memory to store

$$d \cdot 2 \cdot m$$

chunks with 128 bytes for each chunk plus some metadata.

The additional memory required to perform BFS consists of the single chunks created to accumulate results passed up to parents by subtasks. Most of these chunks have a short lifetime, so the amount of memory in use depends on the dynamics of the computation.

5.4 Memory Bandwidth

While executing MergeSparseVectors, each leaf-level merge task may read up to 16 chunks from the graph representation and create a single new chunk for its results. The number of such tasks is, as before, bounded by x, the number of defined elements of the search set for one level of search. Many of the input chunks for these tasks will be read from lower levels of the memory system. The non-leaf level tasks will read recently created chunks which will therefore occupy upper levels of the memory hierarchy. Evidently, the read bandwidth required will be substantially greater than the write bandwidth, and writes of chunks to lower memory levels will occur only if memory capacity limits at higher levels are exceeded.

6 Experiments

For simulation with a model of a Fresh Breeze system, the FAST simulation tool [5] available at the University of Delaware was used. This simulation tool was developed by a collaboration of IBM and E.T. International, for testing and evaluating the IBM Cyclops 64 many-core chip [6]. The Cyclops 64 chip contains 80 processing assemblies, each consisting of two independent thread units (TUs) sharing a 64-bit floating point unit. Each TU has an associated 30 KB

block of SRAM. There are several instruction cache memories, each serving a group of ten TUs. The chip incorporates a cross-bar switching network that interconnects all 160 TUs, allowing each TU to access the SRAM of any other TU. The TUs have access to 1GB of off- chip DRAM memory through four additional ports of the X-bar network. The FAST simulator implements a cycle-accurate model of a complete Cyclops chip.

A hypothetical Fresh Breeze computer system is modeled by code written in C that runs on the simulated Cyclops cores. Forty of the Cyclops thread units are used to model Fresh Breeze processing cores, including an L1 cache memory with each core and a local task scheduler. The remaining thread units are used to model a second level of memory, implemented in the off-chip DRAM, that serves chunk read requests that miss in the L1 cache. One thread unit is reserved to support the task stealing mechanism for distributing tasks among the 40 cores.

Although this Fresh Breeze simulator is not cycle-accurate, faithful modeling for earlier results [11] was achieved by "padding" the code for actions in the simulation for a uniform ratio of simulation time to expected cycle counts for the Fresh Breeze system. For the Graph500 experiments reported here, "padding" was not feasible as it would extend simulation times too much.

Test programs for graph construction and breadth first search, the two kernel algorithms of the Graph 500 benchmark, were written in C using code libraries for the special Fresh Breeze instructions for task coordination, and for memory access to read and write chunks of data.

Using our simulator, Breadth First Search computation was performed for several graph sizes. These test cases were generated according to the Graph 500 benchmark specifications [1]. Vertices of each graph are linked to randomly-chosen neighbor vertices, with an average degree of 16. The graph sizes considered are: 32, 64, 128, 256, 512 and 1024, limited by the amount of simulator memory and the duration of simulation runs.

We first present results based on raw data from one simulation run for each test case, Figure 7. TEPS (Traversed Edges Per Second) for each graph size was determined by dividing the number of graph edges contained in the connected component consisting of vertices reached during the traversal by the product of the measured number of Cyclops cycles used in performing BFS tasks and the duration of a cycle, assuming a 1GHz core. This value was reduced by processor core utilization, measured as simulation cycles executing problem tasks divided by total simulation cycles including core idle cycles and shown in Figure 8

The results show that, with a fixed set of forty cores, TEPS increases roughly linearly with graph size before reaching a peak rate of 12 million edges per second. For the small graphs of our experiments, the number of tasks available for each core to execute is small, and the amount of concurrency is limited by the fact that a "pinch" occurs between levels of search. Nevertheless, a processor utilization of 61% was measured for a graph of 1024 vertices. In addition, we observed that, on average, each processing core traverses 242,748 edges per second, which is comparable to machines ranked on the Graph500 list [1].

We have also used the experimental results to estimate performance of an actual Fresh Breeze system (although with just 40 cores) for BFS. For this calculation, we coded the four principal

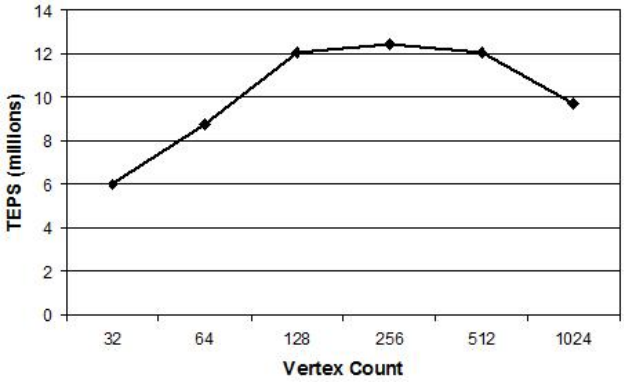

Figure 7: TEPS results based on simulation

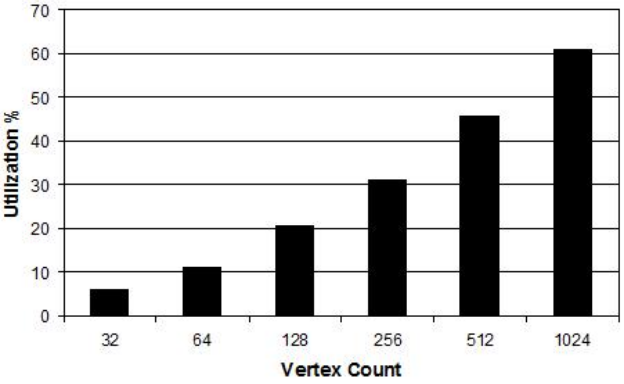

Figure 8: System utilization based on simulation

tasks from the MergeSparseVector and UpdateParent functions in a hypothetical instruction set and counted cycles for their execution. Using these numbers and the measured counts of task executions from the simulation runs, a count of total task execution cycles for BFS was obtained. Using the 1 GHz core frequency and the measured utilization values gives the results shown in Figure 9.

This level of performance will only be achieved in a massively parallel Fresh Breeze system if

Table 1: TEPS Per Core

Vertex Count	TEPS / core
32	150399
64	218644
128	300782
256	310549
512	300980
1024	242748

Table 2: Task Execution Count

Vertex Count	MergeSparseVectors	UpdateParent
32	18	5
64	49	12
128	147	26
256	426	42
512	1417	97
1024	4106	167

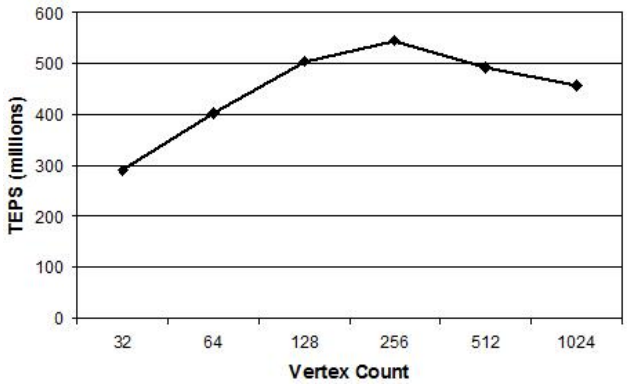

Figure 9: Estimated TEPS for a Fresh Breeze system

the task-stealing mechanism for load distribution is effective for very large numbers of processing cores, and if the memory system can support sufficiently high access bandwidth to remote memory units. However, it should work well if there are large numbers of tasks to be performed at each node, which will be the case for graphs that are very large compared to the number of processing cores.

The analysis in Section 5 indicates that the number of task executions needed for BFS grows linearly with the sizes of the graph time a log factor of the depth of trees of chunks. So long as sufficient processor cores are available and an effective load distribution mechanism is used, it is expected that competitive performance would be achieved for very large graphs on a massively parallel Fresh Breeze system.

7 Conclusions

We have shown that parallel breadth-first search can be implemented using the Fresh Breeze execution model without the use of locks or atomic operations which have been thought to be necessary when using a shared memory system. Earlier simulations demonstrating the merit of the Fresh Breeze PXM for standard linear algebra computations, dot product, matrix multiply and the Fast Fourier Transform, have been reported in recent publications [10, 11], showing effectiveness in exploiting fine-grain concurrency in these kernels. The work reported here adds

to our confidence that the Fresh Breeze PXM is a worthy candidate for guiding the architecture of future massively parallel computer systems.

8 Acknowledgments

The work reported here was done under NSF Collaborative Research Grant CCF-0937907 to MIT, University of Delaware and Rice University. The simulator used in the cited performance studies was implemented by Dr. Xiao X. Meng, post doctoral researcher with the Computer Architecture and Parallel Systems Laboratory at University of Delaware.

References

[1] The graph500 list. http://www.graph500.org.

[2] Arvind and R. S. Nikhil. I-structures: Data structures for parallel computing. Technical report, MIT, 1989.

[3] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops 64 cellular architecture. Technical report, University of Delaware, 2005.

[6] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Tiny threads: A thread virtual machine for the Cyclops 64 cellular architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.

[7] J. B. Dennis. Packet communication architecture. In *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, pages 224–229. IEEE, 1975.

[8] J. B. Dennis. Fresh Breeze: a multiprocessor chip architecture guided by modular programming principles. *ACM SIGARCH Computer Architecture News*, 31(1):7–15, 2003.

[9] J. B. Dennis. The Fresh Breeze model of thread execution. In *Workshop on Programming Models for Ubiquitous Parallelism*. IEEE, 2006. Published with PACT-2006.

[10] J. B. Dennis, G. R. Gao, and X. X. Meng. Experiments with the Fresh Breeze tree-based memory model. In *International Symposium on Supercomputing, Hamburg*, June 2011.

[11] J. B. Dennis, G. R. Gao, X. X. Meng, B. Lucas, and J. Slucom. The fresh breeze program execution model. In *Parallel Computing, Ghent, Belgium*, August 2011.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33:212–223, May 1998.

[13] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *International Parallel and Distributed Processing Symposium*. IEEE, 2009.

[14] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2010.

[15] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17:5–20, 2007.

[16] E. I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[17] J. Tou and P. Wegner. Data structures in programming languages. *ACM SIGPLAN Notices*, 6:171–190, Feb 1971.

[18] V.-Y. Vee and W.-J. Hsu. Applying Cilk in provably efficient task scheduling. *The Computer Journal*, 42:699–712, 1999.

[19] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. V. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2005.