MASSIVELY PARALLEL BREADTH FIRST SEARCH USING A TREE-STRUCTURED MEMORY MODEL

by

Thomas St. John

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2013

© 2013 Thomas St. John
All Rights Reserved
MASSIVELY PARALLEL BREADTH FIRST SEARCH USING A
TREE-STRUCTURED MEMORY MODEL

by

Thomas St. John

Approved: ____________________________________________________________
Guang R. Gao, Ph.D
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____________________________________________________________
Kenneth E. Barner, Ph.D
Chair of the Department of Electrical and Computer Engineering

Approved: ____________________________________________________________
Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: ____________________________________________________________
James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education
ACKNOWLEDGMENTS

First of all, I would like to thank Professor Jack Dennis for his guidance and support during this project. He has proven to be an excellent mentor and working with him has been a truly enlightening experience. I also need to thank Dr. Xiaoxuan Meng for his assistance with this project.

I would also like to thank Stephane Zuckerman for providing feedback during the revision process of writing this thesis.

In addition, I would like to thank Professor Guang Gao for allowing me to work in his research group.

Finally, I would like to thank the National Science Foundation for supporting this research endeavor.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
</tbody>
</table>

**Chapter**

1 **INTRODUCTION** ........................................... 1

2 **PARALLEL BREADTH FIRST SEARCH** .................... 3
   2.1 Parallel BFS Algorithm .............................. 3
       2.1.1 Parallel BFS Pseudocode ..................... 4
   2.2 Motivating Example .................................. 6
   2.3 Architecture-Related Challenges .................. 8
       2.3.1 Shared Memory Systems ......................... 8
       2.3.2 Distributed Memory Systems .................... 9
   2.4 Related Work ........................................ 10
       2.4.1 Shared Memory Systems ......................... 10
       2.4.2 Distributed Memory Systems .................... 12
       2.4.3 Novel Parallel Systems ......................... 13

3 **FRESH BREEZE PROGRAM EXECUTION MODEL** .......... 14
   3.1 Memory Model ....................................... 14
   3.2 Tasking Model ...................................... 16
   3.3 Envisioned System Structure ....................... 16
   3.4 Simulation Studies .................................. 18
   3.5 An Example of Fresh Breeze Programming .......... 19
4 FRESH BREEZE IMPLEMENTATION OF PBFS

4.1 Fresh Breeze BFS Algorithm

4.1.1 Graph Representation
4.1.2 Building the Graph
4.1.3 Searching the Graph
4.1.4 Merging Sparse Vectors
4.1.5 Update Parent

4.2 Example Revisited

4.3 Optimizing Memory Access Patterns

4.4 Modifications to Fresh Breeze API

5 ANALYSIS

5.1 Chunks Needed to Represent a Sparse Vector
5.2 Merge Sparse Vectors
5.3 Memory for the Graph
5.4 Memory Bandwidth

6 EXPERIMENTS

7 CONCLUSIONS AND FUTURE WORK

BIBLIOGRAPHY
LIST OF TABLES

6.1  TEPS Per Core .................................................. 46
6.2  Task Execution Count .......................................... 46
# LIST OF FIGURES

2.1 Pseudocode for Parallel BFS Algorithm ........................................ 5
2.2 Example Graph ............................................................................. 6
2.3 Initial State of Graph Traversal ....................................................... 7
2.4 First Stage of Graph Traversal ......................................................... 8
2.5 Second Stage of Graph Traversal ...................................................... 9
2.6 Third Stage of Graph Traversal ....................................................... 10
2.7 Final State of Graph Traversal ........................................................ 11
3.1 Fresh Breeze array as tree of chunks ............................................. 15
3.2 Fresh Breeze tasking model ............................................................. 17
3.3 Modeled Fresh Breeze System ......................................................... 18
3.4 Fresh Breeze code for the counting example. .................................. 19
4.1 Dataflow diagram of parallel breadth first search. ....................... 22
4.2 Adjacency Vector Representation .................................................. 23
4.3 The BreadthFirstSearch function .................................................. 30
4.4 The CollectVectors, ForwardHandle and UpdateNull functions. .... 31
4.5 MergeSparseVectors, case of leaf chunks. ..................................... 32
4.6 MergeSparseVectors, case of non-leaf chunks. ............................... 33
4.7 Example Graph ............................................................................. 34
4.8 Initial State of Fresh Breeze Graph Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.9 First Stage of Fresh Breeze Graph Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.10 Second Stage of Fresh Breeze Graph Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.11 Third Stage of Fresh Breeze Graph Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.12 Final State of Fresh Breeze Graph Traversal . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.1 TEPS results based on simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.2 System utilization based on simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.3 Estimated TEPS for a Fresh Breeze system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
ABSTRACT

Analysis of massive graphs has emerged as an important area for massively parallel computation. In this thesis, 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 thesis 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.
- We present a novel parallel breadth-first search algorithm which is fully determinate.
- 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.
Chapter 1

INTRODUCTION

Analysis of massive graphs has emerged as an important area for massively parallel computation. However, the existing tools developed for scientific computing applications are poorly suited for large-scale graph analysis. This is because the application characteristics observed in graph analysis vary greatly from those seen when performing dense linear algebra or similar applications. The following properties seen in graph applications make them particularly difficult to efficiently execute on a large-scale system [33]:

- The execution pattern observed when performing graph analysis is typically determined entirely by the structure of the input graph. Because of this, it is extremely difficult to explicitly partition the computation because the application behavior cannot be statically determined.

- Just as it is difficult to partition the computation of graph applications due to their irregular nature, it is also challenging to statically determine how to best partition the data across a large-scale system. Unlike more traditional HPC applications which have regular memory access patterns which can be easily predicted, graph computations typically suffer from poor locality due to the lack of structure present in the graph data.

- Because graph computations are usually focused on exploring the structure of the graph data, they exhibit a much higher ratio of data accesses to computations than is seen in other high-performance computing domains.

In this thesis, 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 thesis 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 [41].
• 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 [33].

• 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.

Chapter 2 introduces the graph search problem and breadth-first search. The memory model and tasking scheme of the Fresh Breeze program execution model (PXM) are reviewed in Chapter 3. Our implementation strategy for graph search is presented in Section 4. We analyze memory usage patterns in Chapter 5 and the performance on an envisioned massively parallel system architecture is estimated in Chapter 6. Experimental results from simulation runs are reported in Chapter 6. Conclusions and future work are reported in Chapter 7.
Chapter 2
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 [6, 19, 25, 31, 35, 37, 44].

2.1 Parallel BFS Algorithm

We can express the problem using the following formulation [41]. 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 V × V, containing pairs of vertices. A spanning tree of G may be represented by a function P : V → 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 [16] proceeds in stages that correspond to levels of the spanning tree being constructed. The algorithm uses a search set S, the vertices to be examined in a stage, and the partial function P_v : V → 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 S' and parent assignment P':
1. For each vertex $v$ in $S$, define the partial update function $U_v$ that maps each neighbor of $v$ to a candidate parent $u$.

2. Combine the maps $U_v$ for all vertices in $S$ to obtain $U$ defined on $S$, the update function for all neighbors of vertices in $S$. Conflict occurs if two or more vertices in $S$ 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 $U$), excluding vertices that have already been assigned as parents (the range of function $P$). The new parent function $P'$ is the union of $P$ and $U$.

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.

### 2.1.1 Parallel BFS Pseudocode

We now present pseudocode for parallel breadth-first search in Figure 2.1. The algorithm accepts a graph $G(V,E)$ and a source vertex $r$ as its inputs and generates an array $P$ containing $n$ elements as its output, where $n$ is the number of vertices in the graph and $P[v]$ contains the index corresponding to the parent of vertex $v$ in the graph traversal. We also use two queues called $CQ$ and $NQ$ to store the sets of vertices belonging to the search frontier for each stage of the traversal. The set of vertices which remain to be searched in the current stage is stored in $CQ$ and the set of vertices which are to be searched in the next level is stored in $NQ$.

As you can see in lines 1 and 2, we first initialize all elements of $P$ to -1, which is the value assigned to represent an unknown or nonexistent parent vertex. We then update the values related to vertex $r$ since it is known that $r$ is the root vertex in the graph traversal. In line 3, we set $P[r]$ to be $r$ and then in line 4, we initialize $CQ$ by inserting $r$ as the sole element in the queue.
1 for all $v$ in $V$ parallel do
2 \hspace{1em} P[v] = -1;
3 \hspace{2em} P[r] = r
4 \hspace{2em} CQ = Enqueue r;
5 \hspace{1em} \textbf{while} \ CQ \neq \text{null} \ \textbf{do}
6 \hspace{2em} \textbf{for} all elements $u$ in $CQ$ parallel dc
7 \hspace{3em} u = Dequeue CQ;
8 \hspace{2em} \hspace{2em} \textbf{for} each vertex $v$ adjacent to $u$ parallel dc
9 \hspace{3em} \hspace{2em} \hspace{2em} \textbf{if} \ P[v] = -1 \ \textbf{then}
10 \hspace{3em} \hspace{3em} \hspace{3em} P[v] = u;
11 \hspace{3em} \hspace{3em} \hspace{3em} NQ = Enqueue $v$;
12 \hspace{3em} \hspace{2em} \hspace{2em} \textbf{endif}
13 \hspace{3em} \hspace{2em} \textbf{endfor}
14 \hspace{2em} \textbf{endfor}
15 \hspace{1em} \textbf{Swap}(CQ, NQ);
16 \hspace{1em} \textbf{endwhile}

\textbf{Figure 2.1:} Pseudocode for Parallel BFS Algorithm

Each iteration of the \textbf{while} loop that spans from line 5 to line 17 corresponds to a single stage of the graph traversal. In line 5, the first step is to verify that $CQ$ is not empty. If the queue is empty, this implies that all reachable vertices have already been encountered and the traversal can terminate.

In the \textbf{parallel for} loop containing lines 6 through 14, each vertex $u$ contained in $CQ$ is removed and each of its neighbors $v$ is examined during the inner \textit{parallel for} loop containing lines 8 through 13. In line 9, we check if this is the first time encountering $v$ during the traversal. If so, then $u$ is assigned as the parent of $v$ in line 11 and $v$ is added to $NQ$ to be searched in the next traversal stage in line 11.

After all vertices belonging to the current search frontier have been processed, the statement in line 15 allows us to prepare for the next stage of the traversal by
redefining $CQ$ to be the set of vertices that were inserted into $NQ$ during the just-completed stage. If no vertices were added to $NQ$ during the most recent stage, then all reachable vertices have already been encountered and the algorithm can terminate.

Note that the enqueue/dequeue operations in lines 7 and 11, along with the comparison in line 9, must be performed atomically in order to enforce correctness.

2.2 Motivating Example

We will now present an example to illustrate how the graph traversal unfolds. In this case, we will traverse the graph shown in Figure 2.2 using vertex 0 as the root vertex for the traversal. Since the queue $NQ$ is only used as a temporary buffer, it will be omitted from the example.

![Figure 2.2: Example Graph](image)

In Figure 2.3, we see the initial state of the graph traversal. Initially, the search vector only contains the root vertex and the parent of each non-root vertex is undefined. The root vertex is assigned as its own parent.
Figure 2.3: Initial State of Graph Traversal

In the first stage of the traversal (Figure 2.4), all vertices adjacent to vertex 0 are examined and since they have not yet been encountered during a prior traversal stage, the parent of each examined vertex is defined as vertex 0 and each vertex is added to the search set.

In the next stage of the traversal (Figure 2.5), we see that vertex 6 has already been visited in a prior stage, so its parent is not redefined and it is not added to the search set. In addition, since vertex 4 is adjacent to both vertex 3 and vertex 6, either vertex is a potential parent of vertex 4.

In the third stage of the traversal (Figure 2.6), we see that vertex 7 is the only vertex which is both adjacent to at least one vertex in the search set and has not been previously visited, so it is the only vertex which is added to the search set.

In the last stage of the traversal (Figure 2.7), we discover that there are no unvisited vertices adjacent to vertex 7, which means that the search set is now empty. This implies that there are no remaining vertices to examine and the traversal terminates.
2.3 Architecture-Related Challenges

We now provide an overview of both shared and distributed memory computer systems and describe the challenges that arise when executing parallel breadth first search on each class of architecture.

2.3.1 Shared Memory Systems

Shared memory systems contain a single address space which is globally visible to all processors. When using shared memory machines, individual vertices belonging to the set $S$ are processed in parallel by the system’s processing elements. Since it is possible that two vertices found in set $S$ share a common neighbor, updates to the set $P$ must be performed atomically. This introduces a source of contention between processing elements and can create bottlenecks during execution[32, 33]. 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.
2.3.2 Distributed Memory Systems

In a distributed memory system, each processor has its own local memory which cannot be accessed by other processors. When implementing breadth-first search on a distributed memory architecture [45], 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 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 multiprocessors, this is frequently accomplished by interprocessor messaging, often using MPI. Because many messages arrive at each processor asynchronously, there are possibilities for non-repeatable behavior without careful design [33].

The problem of delivering candidate parents to the appropriate processing domain is a sorting problem – one for which an interconnection network is effectively a Batcher sorting network [8]. 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" [16] with a separate bucket for each key.

2.4 Related Work

Before describing our implementation in detail, we first provide an overview of the existing work in this area for a range of parallel systems.

2.4.1 Shared Memory Systems

One of the more well-known efforts to parallelize the traditional parallel BFS algorithm on a multicore system was performed by Agarwal et al. [3] using the Intel Nehalem family of processors. In order to lessen the previously mentioned bottleneck due to atomic updates, they partition the graph among the sockets and only allow processors to operate on the portion of the graph which they "own". While these optimizations significantly improved their performance, they also introduce possible load-imbalance and additional cross-socket communication.
Figure 2.7: Final State of Graph Traversal

Hong et al. [30] expand upon this work by utilizing a hybrid algorithm which dynamically selects an optimized CPU implementation at each stage of the graph traversal based on the number of vertices which are currently contained in the search frontier.

A similar technique has been developed by Chhugani et al. [14] to also include dynamic load balancing of the graph vertices among the CPU sockets. In addition, their implementation guarantees correctness even in the presence of race conditions. This property allows them to use atomic-free updates at the cost of a small number of redundant updates. Satish et al. [39] extended this work to multi-node systems by including numerous optimizations to reduce the volume of inter-node communication.

Leiserson and Schardl[32] have also attempted to optimize this queue-based algorithm by replacing the queue with a data structure known as a ”bag”, which acts as an unordered set as opposed to the strict behavior of a FIFO queue. By eliminating these unnecessary ordering constraints, they create an implementation which is much better suited for fine-grained parallelization.

Hassaan et al. [29] provided a study which compared the amount of available
parallelism available in a traditional ordered BFS algorithm compared to an unordered (but not necessarily work-efficient) alternative. Although the unordered algorithm performed some amount of wasted work, it demonstrated higher levels of parallelism and shorter critical path length than the ordered BFS algorithm.

Recently, Beamer et al. [9] proposed a novel approach which combines a traditional top-down algorithm with a new bottom-up technique. This technique can significantly reduce the total number of edges which must be examined throughout the traversal, resulting in a more efficient execution.

### 2.4.2 Distributed Memory Systems

The first noteworthy large-scale BFS implementation on a distributed memory system was developed by Yoo et al. [45] using the BlueGene/L system. The authors use a 2D graph partitioning scheme which reduces the number of processors involved in collective communications from $O(P)$ to $O(\sqrt{P})$. This decrease in communication overhead allows the implementation to scale up to 32,768 processors. However, this work only examines Poisson random graphs, which have a random degree distribution and provide a more regular distribution of work than the scale-free graphs which we are studying. Since then, implementations have also been developed for the BlueGene/P and BlueGene/Q systems [13].

More recently, Buluc and Madduri [11] developed a distributed parallel BFS implementation based on sparse matrix operations [10, 27] and implemented it on a Cray XE6 system. This implementation achieves performance of 17.8 billion traversed edges per second (TEPS) when running on 40,000 cores.

It has also been shown by Cong et al. [15] that shared memory graph algorithms can be implemented on distributed memory architectures using a partitioned global address space (PGAS) programming model, such as UPC. However, in order to achieve speedup on a distributed system, the initial algorithms must be extended to improve communication efficiency and cache performance.
2.4.3 Novel Parallel Systems

In the implementation developed by Scarpazza et al. [40] using the Cell/B.E. architecture, each synergistic processing element (SPE) copies a portion of the search frontier to its local memory and then explicitly sends its results to other processing elements, much like a distributed implementation.

Bader and Madduri [7] implemented the level-synchronous queue-based parallel BFS algorithm using the Cray MTA-2 architecture. Since the architecture includes hardware support for fine-grained synchronization and load balancing [5], the user is able to avoid many of the low-level details typically associated with parallel programming. This work was extended by Mizell and Maschhoff [36] using the Cray XMT architecture.
Chapter 3

FRESH BREEZE PROGRAM EXECUTION MODEL

Precise models of the programmer’s view of a computer system have a history dating back to the 1970s[42]. Moreover, there is at least one computer system, the Burroughs B6700 [38], 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 [20]. 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.

3.1 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, as shown in Figure 3.1. 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.
3.2 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 [26]. As demonstrated in Figure 3.2, 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 [22]. 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.

3.3 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 [21]. 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 [4] mechanism is expected to be competitive in performance.
**Figure 3.2:** Fresh Breeze tasking model

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 [43, 28] for load distribution.
3.4 Simulation Studies

A simulator of the envisioned Fresh Breeze system (Figure 3.3) 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 [23, 24]. Chapter 6 of this thesis reports on simulation experiments using the breadth-first search algorithm discussed in Chapter 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.
long CountDefined (Handle vectHandle, 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 dataHandle = JoinFetch ();
    long sum = 0;
    for (int i = 0; i < 16; i++)
        sum += dataHandle[i];
    JoinUpdate (sum);
    // a JoinUpdate at the top level is the same as a return
}

Figure 3.4: Fresh Breeze code for the counting example.

3.5 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 3.4 for counting the number
of defined elements in a vector. This simple example is included only to illustrate 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 specifies the number of worker tasks and creates
a place (a special *join chunk*) to collect the results of workers. It also identifies 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. For comparison, thread creation using POSIX threads takes approximately 50,000 cycles [2].

This illustrates the methodology used to estimate performance of search in Chapter 6.
Chapter 4

FRESH BREEZE IMPLEMENTATION OF PBFS

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 thesis 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.

4.1 Fresh Breeze BFS Algorithm

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. In practice, the search vector is implicitly generated by taking the set of elements contained in the update vector $U$ and removing the set of elements which are also contained in the parent vector $P$. In the first stage of the traversal, $U$ contains
only the root vertex and in subsequent traversal stages, it contains the neighbors of
the vertices searched in the previous stage of graph traversal. The parent vector \( P \) 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 4.1, the search uses the graph \( G \) 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, MergeS-
sparseVectors and UpdateParent, which are discussed in Sections 4.1.3, 4.1.4 and 4.1.5.
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 Breadth-
FirstSearch function. Further, because the updated parent function is not needed for
the next search phase, this processing may proceed in parallel with search computation.
4.1.1 Graph Representation

In our implementation, a graph of size \( n \) is represented as a Fresh Breeze vector \( G \) containing \( n \) elements, where \( G_v \) 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 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.

In Figure 4.2, we see the sparse vector representation for vertex 1, which is adjacent to vertices 2, 6, 43 and 47. Although each Fresh Breeze chunk contains 16 elements in practice, we assume a chunk size of 8 in this case to simplify the example.

![Adjacency Vector Representation](image)

**Figure 4.2:** Adjacency Vector Representation

The amount of memory needed for a graph represented this way is discussed in Chapter 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.

4.1.2 Building the Graph

Once the edge list is converted to a Fresh Breeze vector, we begin constructing the graph. For each vertex in the graph, we examine each leaf chunk in the edge list
vector and, if the chunk contains edges which are incident to that vertex, generate a partial adjacency vector containing the relevant edges contained in that chunk. These partial adjacency vectors are then merged hierarchically until we are left with a single adjacency vector representing all edges incident to that vertex. The process of merging vectors is described in detail in Section 4.1.4.

The vast majority of execution time is spent building the graph rather than traversing the graph, which is consistent with previous work. Whenever the number of vertices in the graph increases by 2, we expect to see the execution time for graph construction to increase by a factor of 4. This is because each vertex requires its own traversal of the edge list to build its corresponding adjacency vector, therefore doubling the number of traversals performed whenever the number of vertices doubles. In addition, the number of elements contained in the edge list also doubles when the number of vertices doubles, so the time spent performing each traversal increases by a factor of 2. The execution times observed during the experimental process are consistent with this hypothesis.

4.1.3 Searching the Graph

The search process at each level of graph traversal is guided by a function called BreadthFirstSearch (Figure 4.3). 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 4.4), 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.

4.1.4 Merging Sparse Vectors

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

Input to this function is vecHandle, the handle of a chunk that contains 16 handles of chunks 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 4.5 and 4.6: 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 index from 0 to 15, the first value from among the 16 input chunks which contains a defined element at that index (I). Defined values found at higher indices are ignored. We also maintain a counter which keeps track of the number of defined elements currently contained in the output chunk. Once this counter reaches the total number of elements contained in the leaf chunk, all remaining input chunks are ignored. This optimization allows us to eliminate unnecessary read operations and reduce contention through the memory subsystem.

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.
We should also note that since each Fresh Breeze task can only have up to 8 arguments, we have two versions of the \texttt{MergeSparseVectors} function. In the general version, which can accommodate up to 16 input chunks, the handles of the inputs chunks are copied into a single chunk which is then written to memory. This allows up to 16 inputs to be represented as a single argument, albeit at the cost of additional overhead to write and then read the chunk containing the inputs. We also have a specialized version, which can accommodate up to 6 input chunks. In this version, the handles of the input chunks can be directly passed as arguments which eliminates the extra memory operations needed for the general version. In our largest test case, 83\% of the \texttt{MergeSparseVectors} instances used the specialized version.

4.1.5 Update Parent

At each level, a new parent vector is generated by using the \texttt{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.

4.2 Example Revisited

We will now revisit the example graph presented in Section 2.2 now using the algorithm developed for Fresh Breeze.

The initial state of the graph traversal is shown in Figure 4.8. We first initialize the \texttt{update} vector to be the adjacency vector of the root vertex (vertex 0). The \texttt{parent} vector is initialized as null, but it is created during the first stage of the traversal by taking the adjacency vector of the root vertex and inserting the index of the root vertex as the value stored in the vector element corresponding to the root vertex. While the \texttt{parent} vector is being constructed, we also begin the first stage of the graph traversal by spawning the \texttt{BreadthFirstSearch} task using the initialized \texttt{update} and \texttt{parent} vectors.
Since the parent vector is currently null, determining whether a vertex belongs to the current search frontier is performed solely by examining the corresponding element in the update vector (line 6). In this case, the search frontier consists of vertices 1, 3 and 6. Upon the discovery of each vertex belonging to the search frontier, the handle of the vertex's adjacency vector is added to the set of vectors to be merged (lines 8 - 9). After all vertices have been checked, we see that the number of vertices in the search frontier is greater than 1 and spawn a MergeSparseVectors task to create a single adjacency vector for the entire search frontier (lines 10 - 13).

The MergeSparseVectors task then performs a linear scan of each vector in the search frontier (lines 49 - 54). Whenever a defined element is encountered in one of the adjacency vectors, that value is copied into the same position of the output vector, provided that the output vector does not already contain a defined element in that position (lines 53 - 54). After all adjacency vectors have been scanned, the task returns the handle of the output vector (line 56). Note that since the order in which the adjacency vectors are examined is clearly defined, only a single result is possible for each unique input set, unlike the example previously described in Section 2.2.

Since the update vector created during the first traversal stage is not null (Figure 4.9), the execution continues by spawning BreadthFirstSearch and UpdateParent tasks which accept the update and parent vectors as inputs and will generate more up-to-date update and parent vectors as outputs.

Now that the parent vector is not null, the condition for determining whether a vertex belongs in the current search frontier is that both the update vector contains a defined element in its position corresponding to the vertex and the parent vector does not contain a defined element in its matching position (line 6). In this case, the search frontier only contains vertices 2, 4 and 5. Despite having a defined element in the update vector, vertex 6 is not included in the search frontier because there is also a defined element in its position in the parent vector. Intuitively, we can see that this is correct because vertex 6 was already examined in the previous stage of the graph traversal. Once again, we see that the search frontier contains multiple vertices and
spawn a `MergeSparseVectors` task (lines 10 - 13) which executes as before.

Once again, we see that the newly-created `update` vector is not null (Figure 4.10), so we begin a new graph traversal stage by spawning `BreadthFirstSearch` and `UpdateParent` tasks.

At this stage, the `BreadthFirstSearch` task determines that vertex 7 is the only element in the search frontier. Since the search frontier only contains a single element, the task simply returns the handle of vertex 7’s adjacency vector (lines 14 - 15).

After verifying that the `update` vector is non-null (Figure 4.11), another pair of `BreadthFirstSearch` and `UpdateParent` tasks is spawned. The `BreadthFirstSearch` task then determines that there are no vertices belonging to the current search frontier, so it returns a null value (lines 16 - 17).

After receiving the new `update` and `parent` vectors generated during this stage of the graph traversal (Figure 4.12), we observe that the `update` vector is null, so there are no remaining vertices to examine and we return the `parent` vector as our final output.

### 4.3 Optimizing Memory Access Patterns

In order to reduce the contention between processors due to simultaneous memory requests, I added support to interleave up to 8 loop iterations between the `ChunkRead` operations in the `MergeSparseVectors` function. Since there is only enough stack space to operate on 17 chunks at any given time, the number of iterations is calculated by subtracting the number of input chunks from 17. This optimization allows for a flatter distribution of the memory operations and consequently eliminates or reduces performance bottlenecks caused by spikes in the burden on the memory system.

### 4.4 Modifications to Fresh Breeze API

I added a new function to the Fresh Breeze API which is called `JoinForward[34]`. This function allows the master task to directly send data values to the join ticket,
rather than spawning "dummy tasks" which merely contain a JoinUpdate statement.

This functionality is especially useful in the larger test cases where it is possible for the task queues to overflow. First of all, these updates are no longer considered to be separate tasks, so the total number of tasks created during program execution decreases. Second, this delays all subsequent spawn instances until after the update completes, which creates a window for other processors to steal work from the local task queue and increases the likelihood that the local task queue will have space available to accommodate the spawn operations when they do occur.
Chunk BreadthFirstSearch(Handle graph, Handle update, Handle parent, long size) {

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

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

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

6      if (((IsDefined(update[i])) && !IsDefined(parent[i]))) {
7         subtree = graph[i];
8         vectors[count] = subtree;
9         count++;
10    }
11   if (count > 1) {
12      CreateJoin(i, ForwardHandle());
13      Spawn(0, MergeSparseVectors(vectors));
14      quit();
15    }
16   if (count == 1)
17      JoinUpdate(defVector);
18   if (count == 0)
19      JoinUpdate(UnDef);
20 }
21 else {
   // The given handles represent non leaf chunks of the respective vectors

22   CreateJoin(16, CollectVectors());

23   for (int i = 0; i < 16; i++) {
24      if (IsDefined(update[i]))
25         Spawn(i, BreadthFirstSearch(graph[i], update[i], parent[i]));
26      else
27         Spawn(i, UpdateNull());
28      quit();
29   }
30 }
31

Figure 4.3: The BreadthFirstSearch function.
void CollectVectors() {
    Handle vecHandle = 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(vecHandle[i])) {
            defVector = vecHandle[i];
            new_vectors[count]=defVector;
            count++;
        }
    }

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

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

void UpdateNull () {
    JoinUpdate (Undefined);
}

Figure 4.4: The CollectVectors, ForwardHandle and UpdateNull functions.
Chunk MergeSparseVectors (Handle vecHandl, long size) {
    Chunk vecChunk = ChunkRead (vecHandl);
    // Read the chunk containing handles of trees to be merged

    if ( size < 16 ) {
        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)) && !(IsDefined(leaf[i])))
                    leaf[i] = element;
            }
        }

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

Figure 4.5: MergeSparseVectors, case of leaf chunks
else {
    // 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 4.6: MergeSparseVectors, case of non-leaf chunks.
Figure 4.7: Example Graph

Figure 4.8: Initial State of Fresh Breeze Graph Traversal
**Figure 4.9:** First Stage of Fresh Breeze Graph Traversal

**Figure 4.10:** Second Stage of Fresh Breeze Graph Traversal
Figure 4.11: Third Stage of Fresh Breeze Graph Traversal

Figure 4.12: Final State of Fresh Breeze Graph Traversal
Chapter 5

ANALYSIS

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

5.1 Chunks Needed to Represent a Sparse Vector

We consider a sparse vector $V$ of size $n$ with $x$ defined elements. The 16-ary tree of chunks representing $V$ 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 $V$ 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

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

37
Using $b = \lfloor \log_{16}(x) \rfloor$ 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:

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

For simplicity, in the remaining discussion we will use

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

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 $\lfloor x/16 \rfloor$ 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 $b = d - \lfloor \log_{16}(x) \rfloor$. 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

$$e(s) / 15$$

39
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[12], the number of edges in the graph is at most \( 16 \cdot n \), 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.
Chapter 6

EXPERIMENTS

For simulation with a model of a Fresh Breeze system, the FAST simulation tool [17] 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 [18]. 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 [24] 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 6.1. 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 6.2.

The results show that, with a fixed set of forty cores, TEPS increases roughly linearly with graph size before reaching a peak rate of 15 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 77% was measured for a graph of 1024 vertices, which is approximately equal to a 40-processor Cray MTA-2 system [7]. In addition, we observed that, on average, each processing core traverses 312,104 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
Figure 6.1: TEPS results based on simulation

coded the four principal 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 6.3.

This level of performance will only be achieved in a massively parallel Fresh Breeze system if 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 Chapter 5 indicates that the number of task executions needed for BFS grows linearly with the size of the graph multiplied by a log factor of the depth
Figure 6.2: System utilization based on simulation

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.
### Table 6.1: TEPS Per Core

<table>
<thead>
<tr>
<th>Vertex Count</th>
<th>TEPS / core</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>156503</td>
</tr>
<tr>
<td>64</td>
<td>268618</td>
</tr>
<tr>
<td>128</td>
<td>337104</td>
</tr>
<tr>
<td>256</td>
<td>380500</td>
</tr>
<tr>
<td>512</td>
<td>330572</td>
</tr>
<tr>
<td>1024</td>
<td>330637</td>
</tr>
</tbody>
</table>

### Table 6.2: Task Execution Count

<table>
<thead>
<tr>
<th>Vertex Count</th>
<th>MergeSparseVectors</th>
<th>UpdateParent</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>64</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>128</td>
<td>113</td>
<td>24</td>
</tr>
<tr>
<td>256</td>
<td>379</td>
<td>43</td>
</tr>
<tr>
<td>512</td>
<td>1575</td>
<td>90</td>
</tr>
<tr>
<td>1024</td>
<td>4816</td>
<td>162</td>
</tr>
</tbody>
</table>
Figure 6.3: Estimated TEPS for a Fresh Breeze system
Chapter 7

CONCLUSIONS AND FUTURE WORK

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 [23, 24], 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.

In the future, we intend to extend the Fresh Breeze model model to a larger-scale, which would be distributed across multiple hardware nodes. We also plan to implement an adaptive scheduling policy so that users will not have manually decide which scheduling technique to use when creating worker tasks. We also believe that the work performed in this study can be further leveraged to implement single source shortest path (SSSP) computation on weighted graphs.
BIBLIOGRAPHY


