FreshBreeze: A Data Flow Approach for Meeting DDDAS Challenges

Xiaoming Li1, Jack B. Dennis2, GuangR. Gao1, Willie Lim2, Haitao Wei1, Chao Yang1, and Robert Pavel1

1 University of Delaware, Newark, DE, U.S.A
xli@udel.edu, ggao.capsl@gmail.com, hwei@udel.edu, chaoyang@udel.edu, rspavel@udel.edu
2 Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, U.S.A
dennis@csail.mit.edu, wlim@csail.mit.edu

Abstract

The DDDAS paradigm, unifying applications, mathematical modeling, and sensors, is now more relevant than ever with the advent of Large-Scale/Big-Data and Big-Computing. Large-Scale-Dynamic-Data (advertised as the next wave of Big Data) includes the integrated range of data from high-end systems and instruments together with the dynamic data arising from ubiquitous sensing and control in engineered, natural, and societal systems. In this paper we present Fresh Breeze, a dataflow-based execution and programming model and computer architecture and how it provides a sound basis to develop future computing systems that match the DDDAS challenges. Development of simulation models and a compiler for Fresh Breeze computer systems is discussed and initial results are reported.

Keywords: DDDAS, Dataflow, FreshBreeze, Compiler, Computer Architecture, Codelet, Big-Data, Big-Computing

1 Introduction

Applications that may be called Dynamic Data Driven Application Systems (DDDAS) involve system modeling with high performance numerical computing and real time monitoring and control of a real or simulated environment. In addition to linear algebra operations using vectors and matrices of numerical data DDDAS require means for expressing and efficiently executing asynchronous interactions of components. Moreover DDDAS require great flexibility in applying computing resources to a variety of computation methods depending on events and input or computed data.

Our research is the study and evaluation of new computing platforms that are a better match to DDDAS requirements than currently available computing systems. Because systems of the future will certainly be composed of large numbers of processing cores, a major challenge is to organize systems with many processing cores to provide the dynamic resource management and support the reactive nature of DDDAS. The basis of our work is the Fresh Breeze programming model and computer system architecture. We are close to being able to exercise a simulated
model of a Fresh Breeze system with multiple processing cores for an illustrative DDDAS. The current status of our work is reported in this paper.

The paper is organized as following. First we briefly introduce the Fresh Breeze programming model and system architecture. Then we illustrate how the parallelism in two test programs may be expressed in terms of the codelet model and report simulation results from a model of the Fresh Breeze system architecture. Next, we present the principles of the Fresh Breeze compiler, which was used to generate machine level codelets for the two test programs. We conclude with a discussion of our approach to including streams and transactions in the programming model and an illustrative example how these features apply to a representative DDDAS.

2 Fresh Breeze Project

The Fresh Breeze project[6, 5] has developed a massively parallel system architecture that directly supports a programming model designed for natural expression of all forms of parallelism present in applications. This includes expression parallelism, data parallel computation, producer-consumer parallelism and concurrent transaction processing. In addition, the programming model satisfies requirements for modular software construction, ensuring that any program module may be used without change as a component of another program module. The system architecture provides for automatic managements of memory and processor resources, as demanded by advanced applications. The next paragraphs describe data representation and tasking in Fresh Breeze systems, and give an overview of the supporting system architecture.

Data Object Representation  All data objects in a Fresh Breeze system are represented by trees of fixed-size chunks of memory. A chunk holds up to 16 items, each of which may be either a data value or the handle of another memory chunk. The handle of a memory chunk serves as a globally valid means to locate the chunk within the storage system. The collection of all memory chunks forms a multi-rooted directed acyclic graph (DAG) that is the “heap” held by the Fresh Breeze multi-level memory system. Chunks are created and filled with data, but are frozen before being shared with concurrent tasks. This write-once policy eliminates data consistency issues and simplifies memory management by precluding creation of cycles in the graph of chunks and references.

Tasking Model: Codelets  The basic unit of parallelism in a Fresh Breeze system is a task, the activity of performing a single execution of a block of instructions known as a codelet. The organization of multiple tasks is expressed in a way similar to the spawn/join model for parallel programming of Cilk [8]: A master codelet may spawn one or more worker tasks to execute independent instances of the same or different codelets. Worker tasks may receive data objects as arguments provided by the master codelet, and each worker task contributes the results of its activity to a continuation task using a special type of memory chunk called a sync chunk [6]. 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. The scheme matches the data parallel features of a programming language such as Sisal [9] or NESL [1]. Through recursive use of this scheme, a program can generate an arbitrary hierarchy of concurrent tasks corresponding to available parallelism in the computation being performed.

System Architecture  A Fresh Breeze computer system will consist of many multi-core processing chips and several levels of memory split between on-chip memory for the most active data, and off-chip for less active data objects. Each many-core processing chip uses processor cores that have a simple Reduced Instruction Set Computer (RISC) style load/store instruction set and a simple execution pipeline. Although our current simulation of the Fresh Breeze
architecture only models two memory levels, we expect to extend our cycle accurate simulation
to model a four level memory hierarchy that includes local memory at each processor, shared on-
chip memory, plus off-chip Dynamic Random-Access Memory (DRAM) and Solid State Drive
(SSD) memory levels.

The unit of storage (and allocation) at each memory level is a single chunk, together with
metadata providing data type information. Single chunks are the unit of transfer between
adjacent memory levels. Chunks are moved downward only between adjacent levels. Chunks
are moved upward on demand by a processor; the referenced chunk is moved from the memory
level where it is found to all higher levels on the path to the requesting processor. Each memory
unit at the lowest (SSD) level maintains an inventory of unallocated chunks, as a bit map. The
handle of a chunk is the index of the SSD memory unit concatenated with its bit-map index.

A processor accesses data – an element of a chunk, which may be a scalar value or a handle –
by presenting the handle of the chunk and an offset in the range [0...15], both held in processor
registers. Access to local memory of the processor is simplified by loading the index of the
buffer holding the chunk into an auxiliary field of the register holding its handle. In this way
the cost of the usual tag memory of a conventional level one cache is avoided. At other levels
of the memory system, means must be provided to locate the chunk on the basis of its handle.
At the bottom level this is easy because the handle can be exactly the memory address of the
chunk. For the on-chip shared memory and the DRAM some form of associative lookup must
be used.

3 Examples of Kernel Computations

In this section we use Dot Product and Matrix Multiplication as examples to introduce how
programs are expressed by using codelets on tree-based memory model.

Dot Product  The dot product is a simple program which can be easily parallelized. A
simple way to parallelize the dot product is to divide the loop into small parts of dot product
and assign each part of the dot product to different tasks. Each task executes the small part
of dot product and does the sum reduction with other tasks to produce the final result.

To translate a dot product code into the tree-based memory model and the codelet-based
programming model, first, the vectors need to be converted to tree based memory chunks. For
current system, a memory chunk holds 16 long values therefore, vectors are divided into 16-
element segments and organized as a tree structure—the leaf chunks hold the actual values while
the non-leaf chunks store the handles of chunks that point to other chunks.

After vectors are constructed, the computation step follows as shown in figure 1(a). In the
Compute step a master codelet Traverse Vector is executed. It takes the roots of two tree-of-
chunks A and B as inputs. It then checks the depth of the tree to see if it is a leaf node. If
not, it recursively spawns Traverse Vector codelets taking the root handles of the next level as
inputs. At the leaf level, a Compute codelet is spawned to compute dot product of 16 elements
and return the result sum to the Sync chunk. The continuation codelet Reduce adds all results
in the Sync Chunk that are filled by lower level Compute/Reduce codelets and returns the
handle of the Sync Chunk at the next (upper) level Sync Chunk and so on until the root level
Sync Chunk which holds the final result is reached.

Matrix Multiplication  Matrix multiplication $C = A \times B$ follows the same two steps as in dot
product: first construct the data chunks for matrices and then perform the computation. Matrix
multiplication has three-level nested loop with dot product as the innermost loop. We assumed
that matrix $B$ is already transposed. Therefore, the two matrices has the same structure when
represented by trees, row first and column second. Each row vector is represented as a subtree
where Each leaf node stores 16 elements of the row. The details can be found in section 4.
Since the two steps almost uses the same codelets structures, we focus on the compute step here. As figure 1(b) illustrates, the program first spawns ForAllRow codelets to traverse all the rows of matrix $A$. Each of the spawned codelet takes a subtree representing one row of matrix $A$ as input. For each row, the ForAllRow codelet spawns ForAllColumn codelet to traverse all the columns of matrix $B$, where each spawned codelet takes the subtree of a column of matrix $B$ as input. The DotProduct codelet takes one row and one column and follows the codelet structure in figure 1(a) to calculate result of dot product. CombineSum codelet returns the value. The Construct Column codelet collects the results of dot product to construct a row vector of matrix $C$. Then, the Construct Row codelet collects all rows to construct the whole matrix $C$ as the final results. More details can be found in section 4.

![Diagram](image1.png)

(a) Codelets structure of dot product

![Diagram](image2.png)

(b) Codelets structure of matrix multiplication

Figure 1: Codelets Structures of Dot Product and Matrix Multiplication.

**Simulation Results** We have obtained initial performance measurements on two simulation models of Fresh Breeze system: SystemOne consists of one Processing Unit, one Memory Unit and a local task scheduler, and is capable of running any Fresh Breeze machine level program (group of compiled codelets). SystemTwo is a model of a Fresh Breeze multi-core chip and consists of several copies of SystemOne with each Memory Unit being accessible to all Processing Units through a packet-switched network. SystemTwo also includes a load balancing feature in the form of a ring network linking all Processing Units. The ring is used to pass task records from highly loaded processors to processors that have fewer tasks pending execution.

Some parameters of the simulated system are: 1) Processor Clock Rate: 1.0 GHz. 2) Memory Network delay (each way): 10 nanosec. 3) Memory Unit Response Time: 3 nanosec. We have assumed that all register-to-register instructions, including multiply, are completed in a single processor cycle. Our system model currently does not include any model of instruction fetch from memory. The simulation tool used for our experiments is PCASim which is a general purpose simulation tool for systems that have packet communication architecture \[4, 2\]. In carrying out simulations of SystemTwo, the simulator is able to execute at least 400,000 simulated instructions executions per second.

Two linear algebra kernels are currently being used as benchmark codes for system diagnosis and evaluation. They are the dot product of two vectors, and matrix multiplication; in both cases the type of data elements is 64-bit integers. Results from running the dot product benchmark are shown in Table 1 for a system with one and two processing cores, and for vector with lengths that are powers of two: 16, 256, 4096, and 65536.
The results show the expected increase in performance with increasing numbers of processing cores. They are conservative values in relation to potential performance because we have not yet exploited several obvious optimizations that are applicable: 1) Loop unrolling: Perform two, four, eight or sixteen products per loop iteration instead of one. This will, for example, reduce the number of loop control instruction per multiply/add from \( nn \) to \( mm \) with four-fold unrolling. 2) Performing two or more 16-element dot products per codelet execution instead of just one. This will amortize the work needed to schedule and spawn codelet over a larger block of computation. 3) The present system model uses blocking memory operations for transferring chunks between memory levels. Implementing non-blocking operations would yield a major gain in performance.

Finally, it is known that two concurrent threads sharing use of a pipelined floating point function unit can achieve much better utilization of the FP unit than if each thread has a dedicated FP unit [3]. In particular, 70% of peak performance for matrix multiply has been achieved. Applying the same principle to Fresh Breeze processor architecture may be expected to yield similar results.

<table>
<thead>
<tr>
<th>Vector Length</th>
<th>Processing Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>56.9</td>
</tr>
<tr>
<td>256</td>
<td>82.6</td>
</tr>
<tr>
<td>4096</td>
<td>84.8</td>
</tr>
<tr>
<td>65536</td>
<td>81.8</td>
</tr>
</tbody>
</table>

Table 1: Performance in MFLOPS for the dot product benchmark.

4 Fresh Breeze Compiler

**Fresh Breeze Compiler Framework** Fresh Breeze compiler takes a Java source file as input and generates Fresh Breeze machine code. Figure 2 shows the overall work-flow of the compiler. It starts by using `javac` to convert the Java source code to Java bytecode. The *Class File Reader* translates the stack-oriented Java bytecode into a flat internal representation and constructs an intermediate representation—*data flow graph (DFG)*—for each method of the source program. The *Transform* component operates separately on the DFG of each method and analyses loops to extract data parallelism. If a loop is eligible for parallelization, the DFG of this sequential loop is converted into a combination of codelets based on the FreshBreeze tasking model and tree structured memory model. The *Construct Code* component takes each codelet DFG produced by transformation of a method DFG and generates the Fresh Breeze machine instructions that define a machine-code codelet ready for execution. More details can be found in our previous paper [7].

**Array Representation in FreshBreeze** Array is represented as “tree-of-chunks” which naturally supports the divide and conquer algorithm. We begin to introduce how one dimensional array is represented by tree structure, and then extend it to multidimensional array. The data values are only stored in the leaf nodes of the tree. With the limitation of chunk size, each leaf node only holds up to 16 64-bit elements in row-major order. Instead of carrying data values, the non-leaf node holds the handles of the leaf node or the non-leaf child node. Multidimensional array is organized as the “tree-of-trees” structure. Each tree represents one dimension of the array. The leaf node of the \( i \)th dimension contains the handles pointing to the root of the \( i+1 \)th dimension tree. Similar to one dimensional array, data values are only
kept in the leaf node of the last dimension. Especially in the for-all parallel loop, the tree structure of array is naturally divided into sub trees and data chunks are accessed in parallel. In such case, processing the entire array takes O(n) runtime, equal to the number of elements in the tree. Comparing with linear address space, the tree structured array uses a low space cost to naturally support the divide-and-conquer algorithm.

**Data Flow Graph Templates** The transform component converts a data flow graph of loops with traditional linear-array accessing into a data flow graph with tree-based array accessing. The different loops have relative fixed structures, thus they are transformed into data flow graphs with similar structures. Therefore, we extract this common DFG structures as *templates* with configurable parameters and use templates to describe similar data flow graphs instead of repeating them. Following this, the main ideal of transform is to replace the loop nodes in data flow graph with parameter configurable DFG templates.

**Loop Transformation** The loop transformation component includes two steps: analyzing dependencies of loops and transforming parallelizable loops using FreshBreeze task model and tree-structured memory. By analyzing loops in each method of DFG, loop nodes are classified as *array constructions* or *reductions* or *irregular loops*. We currently focus on former two types of loops that can benefit from Fresh Breeze architecture naturally. Loop attributes are recorded for transformation, including loop classes and an list holding the reference of array to be traversed in the loop body.

The transformation part selects eligible loops, replaces these loops with predefined templates to automatically exploit the parallelism. Figure 1(a) shows the template’s calling relation for one dimensional reduction operation. The original DFG only has one while node with reduction over the whole vector, and is executed in sequence. The transformed DFG divides the array into chunks, different chunks are processed in parallel. In the following we sketch the operation of the codelets that constitute the pattern or template for a reduction/construction data parallel loops:

- **DfgVectorForAll** is the entry of templates, it executes the code of the original DFG except the *while* node. The *while* node is replaced by the DfgApply node which will spawn the template: **DfgTraverseVectors**.

- **DfgTraverseVectors** spawns 16 worker tasks, each worker recursively performs **DfgTraverseVectors** on a 1/16 segment of arrays. This process continues recursively until get to the last 16 elements in a segment, at which point a set of Compute codelet is spawned.
• **DfgCompute** executes the computation of while loop body of the segment. It computes and returns the reduction value of this segment.

• **DfgReduceDone** continuation codelet collects the reduction values from a group of Compute or lower (child) DeduceDone codlets, combines these values with reduction operator and returns the resulting scalar value.

• **DfgDummyJob** pairing with **DfgVectorForAll**, simply returns the final results to continue task.

For the array construct operation, because a chunk is immutable once it is created by a codelet, we have to build the tree structured array in a bottom-up approach. The templates are almost the same as those used by the reduction operation, except the **DfgReduceDone**. Instead, **DfgVectorDone** is used to collects a group of handles of the data chunks to create the tree structured array and updates the handles into up-level sync chunk recursively until reaching the root node.

Figure 1(b) illustrates multidimensional array construction and nested loop transformation using three groups of templates. Each group is an one dimensional array transformation. Although the basic traversal template is same as one dimensional case, additional templates **DfgForAllCompute** and **DfgTraverseLoopBody** are introduced. **DfgForAllCompute** substitutes the **DfgCompute**, spawns a set of tasks **DfgTraverseLoopBody**, which is similar to **DfgVectorForAll** to perform vector traversing to the next dimension.

For example, in matrix multiplication, the innermost loop is a dot product routine producing one element of the results. Therefore, the matrix multiplication is transformed into 3 groups of templates: **row group**, **column group**, and **dot product group**. The row group traverses matrix A’s row dimension and creates C’s row dimension; the column group traverses B’s row dimension and builds C’s column dimension. Dot product group traverses both A’s and B’s column until to the leaf data chunks, and then spawns **DfgCompute** which applies the reduction operations on A’s and B’s data chunks and computes the partial sum value. **DfgReduceDone** continues applying reduction on sum which produces the final result of $C[i][j]$.

5 Streams and Transactions

We have noted some distinctive features of DDDAS: a computation component performing system modeling operates with and is responsive to changing conditions in its environment. These characteristics make two demands for expressiveness on system infrastructure designed to support DDDAS.

Firstly, the system must provide means of communication between the computation component and controllers that monitor events and conditions in the environment, direct the progress of computation, and deliver action commands to environmental effectors, for example to adjust fuel flow in an aircraft engine. In the Fresh Breeze programming model these functions are supported by stream data types. Secondly, the system must be able to support resolution of conflicting demands for resources arising from unanticipated events: For example, failure of a critical system component will require a change of computational strategy. In the Fresh Breeze programming model this need is supported by a transaction processing pattern: Resource demands are treated as requests from agents for access to a shared data object. These two features of the Fresh Breeze programming model are briefly described below; followed by discussion of their use in supporting a DDDAS that we have been studying: simulation of turbulence in flows of particulate-laden fluids.

---

FreshBreeze: A Data Flow Approach for Meeting DDDAS Challenges  
Li, Dennis, Gao, et Al.  
2579
**Fresh Breeze Streams**  To illustrate computation with data streams, a **Source** of data items might be processed by a **Clipper** module to remove out-of-range data values, and the stream of remaining data items absorbed by an abstract **Sink**. In a DDDAS the computation expressed by this configuration of modules might be the filtering of the stream of items arriving from a sensor to remove uninteresting data, or filtering the data to reduce its volume – or both by operating two modules in cascade.

To express computations involving streams of data items the Fresh Breeze user programming language funJava includes a **Stream** class with the method signatures of the following Java interface declaration. The meta variable T denotes an arbitrary type for the data items of the declared stream type.

```java
interface Stream {
    Stream<T> create();
    <T> first (Stream<T> strm);
    Stream <t. rest (Stream<T> strm);
    Stream<T> append (Stream<T> strm, <T> item);
}
```

The method **create** returns an empty stream; the method **first** returns the head element of the stream; **rest** returns the stream containing all items of the given stream except the head item, and **append** constructs a stream by adding an item to the tail of the given stream.

The following code illustrate how the **Clipper** could be written in funJava. It replaces out-of-range data values with the most recent valid value.

```java
Stream <int> dataClipperModule (Stream <int> inStream) {
    final int MAX = ??;
    int saveData = 0;
    Stream <int> outStream = new Stream <int> ();
    while(inStream.moreData()) {
        int itm = inStream.first();
        if (abs(itm) > MAX) {
            nextData = saveData;
        } else {
            nextData = itm;
            saveData = itm;
        }
        outStream.append (nextData);
    }
    result outStream;
}
```

In general, processing each input data item may produce no output, or any number of data items. The implementation of streams in a Fresh Breeze computer system represents a stream by a chain of chunks. Each chunk holds several data items and a reference to the next chunk in the chain. Stream elements are removed from the head chunk of the chain, and new elements are added to the tail chunk, adding a new chunk to the chain if the tail chunk is full. The four operations on streams: **new**, **first**, **rest**, and **append**, are supported by instructions of the Fresh Breeze ISA.

Before continuing, we note that programs written using the Stream class are guaranteed to be repeatable, just as other programs in a purely functional programming language. There is no possibility of writing programs containing concurrency hazards or data races.
**Fresh Breeze Transactions**  Not all computations of interest can be implemented as program modules that have repeatable behavior in execution. To illustrate, consider a directory. A directory is a key-value store that might be a directory of files shared by users of a DDDAS. We consider a key-value store shared by two concurrent users. Either user is permitted to search the directory to find the value corresponding to a key, and either user may add or delete entries.

In our user language funJava, a class for the directory could be declared as:

```java
class Directory {
    HashMap<String, Object> table;
    Directory () {
        table = new HashMap<String, Object>();
    }
    Object search (String key) {
        return table.get(key);
    }
    void insert (String key, Object value) {
        table.insert(key, value);
    }
    void delete (String key) {
        return table.delete(key);
    }
}
```

We cannot permit the two agents to have direct access to methods of the Directory class: some means is needed to ensure that the insert and delete methods are executed atomically, as they change the state of a shared object.

In the scheme we use to ensure correct behavior each agent may invoke a Request module to enter a request for service. Requests from either user are appended to a Request Queue. A Process module removes entries from the Request Queue and processes them serially using methods of the Directory class. The functions of entering transaction requests in a request queue and processing them serially are implemented by a transaction Manager, which uses a special type of memory chunk known as a guard to implement the merging of transaction requests into a stream. The crucial step is appending a request to the queue of requests, which must be done atomically. We have shown how this can be done [6] using a single special GuardSwap instruction similar to the “compare-and-swap” instruction used to construct lock-free implementations of concurrent data structures.

Use of the guard object and the GuardSwap instruction makes a computation non-repeatable because one cannot tell which of two requests arriving at nearly the same time will become the earlier entry in the request queue. The Fresh Breeze programming model ensures repeatable computation unless the GuardSwap instruction is used.

**Turbulent Flows containing Particulates**  A problem of great interest is accurate simulation of turbulence in flows of particulate-laden fluids. Efficient solution of such problems has applications to aircraft engine performance, nuclear weapon inventory maintenance and other topics of interest to AFOSR.

These computations may be performed using known methods, but, due to their multi-scale nature, make widely varying demands on the processing and memory resources of computer systems. Existing high performance systems are massively parallel computers that are best suited to the Bulk Synchronous Parallel model of computing and programming tools such as MPI, but present a formidable challenge for the implementation of DDDAS.

Let’s consider an application using three specialized simulation modules: a laminar fluid flow model, a model of boundary layer turbulence, and a model of disturbance of the flow field
by particles. As the simulation computation progresses, decisions must be made about portions of the flow field for which the specialized modules should be applied. Controller modules receive stream of observations from simulation modules indicating need to apply a different simulation method to regions of the flow field. These controllers are agents that make competing requests for use of a shared resource – each specialized simulation module. This is a perfect scenario for use of the Fresh Breeze transaction processing.

Our previous works on stream are mainly focus on software support [16, 13, 10, 14, 12]. In these work, a synchronized buffer is used [11, 15, 17]. In the next phase of our research program, we expect to implement support for streams and transaction in our compiler and Fresh Breeze simulation model, and develop an illustrative implementation of turbulent flow simulation.

Acknowledgments
The work is supported by U.S Air Force Office of Scientific Research under Grant No. FA9550-13-1-0213.

References