Architecture and Programming Model for High Performance Interactive Computation

—Based on “Title of TM127”, CAPSL TM-127
By Jack B. Dennis, Arvind, Guang R. Gao, Xiaoming Li and Lian-Ping Wang
April 3rd, 2014

Haitao Wei
CAPSL at UDEL
Outline

• Introduction to DDDAS/Interaction Computation
  – An Example and Problems
• Fresh Breeze Execution Model and Architecture
  – Execution Model
  – Memory Model
  – Task Model
• Streaming and Transactions
  – Stream Type and Operations
  – Concurrency Operations of Transaction Style
• Conclusion
Outline

• Introduction to DDDAS/Interaction Computation
  – An Example and Problems
• Fresh Breeze Execution Model and Architecture
  – Execution Model
  – Memory Model
  – Task Model
• Streaming and Transactions
  – Stream Type and Operations
  – Concurrency Operations of Transaction Style
• Conclusion
An Example of DDDAS/Interaction Computation — Radio Astronomy

- Antenna Array
- Local Processor
- Observer

Receipt signals
Filter signals and control
Analyze signals
Data Analysis
Make decision and change parameters
Dynamic Data Driven Application System (DDDAS)—Challenges

• real time interaction with parts of the physical environment.

• management of processing and memory resources according to dynamic needs generated by local events

• input and output devices process streams of data items

• make decisions about the work using transaction processing
Our Solutions: Programming Model and Architecture Support

• Fresh Breeze Execution Model and Architecture
  – based on codelet execution model
  – support fine-grained execution and memory management

• Streaming
  – support streaming data expression and operations

• Transaction
  – support concurrency operations of *transaction* style
Outline

• Introduction to DDDAS/Interaction Computation
  – An Example and Problems

• Fresh Breeze Execution Model and Architecture
  – Execution Model
  – Memory Model
  – Task Model

• Streaming and Transactions
  – Stream Type and Operations
  – Concurrency Operations of Transaction Style

• Conclusion
Coarse-Grain vs. Fine-Grain Multithreading

Coarse-Grain thread-
The family home model

Fine-Grain *non-preemptive* thread-
The “hotel” model
Case Studies of Fine-Gran Execution Models

• Dataflow Model (1970s - )
• EARTH Model (1993 - 2006)
• HTVM Model (2000 - 2010)
• Fresh Breeze Model (2000 - )
• Runnemede Model (2010 - )
Fresh Breeze Execution Model

**Task Model**
A set of rules for creating, destroying and managing threads

**Memory Model**
Dictate the ordering of memory operations

**Synchronization Model**
Provide a set of mechanisms to protect from data races

**The Abstract Machine**
Fresh Breeze Memory Model
-- Main Features and Vision

- Global shared name space with “one-level store”
- A single-update storage model to eliminate the cache-coherence problem
- Concept of “sealed” memory chunks/sections with single assigned property
- Trees of fixed-sized chunks
- Fine-Grain memory management support
- Memory allocation and data transfer is performed entirely by architecture/hardware mechanisms
Fresh Breeze Memory Model

- Write Once then Read only
- Fix chunk size: 128 Bytes: 16 doubles, 32 integers,…
- Chunk handle: 64 bits unique identifier
- Arrays: Three levels yields 4096 elements(longs)
Task/Concurrency Model

- Asynchronous tasking
- Continuation Task receives children’s results
- Non-blocking continuation
- Light-Weight Tasks
Example—Dot Product

```c
sum = 0;
for (i = 0; i < 16*16*16; i++)
    sum += A[i] * B[i];
```

**Step 1: Build Vector**
Example—Dot Product

sum=0;
for(i=0;i<16*16*16;i++)
    sum+=A[i]*B[i];

Step 2: Compute
Outline

• Introduction to DDDAS/Interaction Computation
  – An Example and Problems
• Fresh Breeze Execution Model and Architecture
  – Execution Model
  – Memory Model
  – Task Model
• Streaming and Transactions
  – Stream Type and Operations
  – Concurrency Operations of Transaction Style
• Conclusion
What We Will Learn

• Streaming
  – stream data type and operations
  – “Future” structure to implement streaming
  – streaming on dataflow model: static dataflow and synchronous dataflow

• Transactions
  – Determinate and Repeatable
  – Using “Future” and “Guard” to implement concurrency operations of transaction style
Stream Type and Operations

• Stream: A sequence of values of type, maybe infinite

• Define a stream
  – Stream <DataItem> inStream = new Stream <DataItem>();
  DateItem can be any data type

• Concatenate two streams
  – Stream <DataItem> strm1 = strm0 + new Stream <DataItem>{i0, i1, ... }

• Get first element in stream
  – strm.first();
Stream Type and Operations (cont’d)

- Remove the first element in stream
  - Stream <DataItem> strm1 = strm0.rest ()
  - Stream <DataItem> strm = strm.first () + strm.rest ()

- Append an data item to stream
  - strm.append(item) ;

- It is the end of data stream
  - if ( strm.moreData ()) { statement }
An Stream Example—Average Pairs

Stream <int> sourceStream = new Stream <int> ( ) ;

while ( true ) {
    int itm = geneDataItem ( );
    sourceStream.append ( itm );
}

result sourceStream ;

Stream <int> result = sourceStream ;

while(inStream.moreData()) {
    int itm0 = inStream.first ( );
    inStream = inStream.rest ( );
    int itm1 = inStream.first ( );
    inStream = inStream.rest ( );
    outStream.append (( itm0 + itm1 ) / 2);
}

result outStream ;

while ( inStream .moreData ( ) )
{
    achive.put ( inStream.first ( ) );
    inStream = inStream.rest ( );
}

result achive.getAchive ( );
Stream Implementation in FreshBreeze

• Stream representation
  – a linear chain of chunks, each chunk holds data items and a reference to the next chunk

• Stream operations
  – FIFO queue operations on chain of chunks
  – read from the head of the chain of chunks, write to the tail of the chain of chunks

• Synchronization between Producer and Consumer
  – Special Object: Future
Future

- A future is a memory cell with a state waiting to receive a data value: status: undefined, defined, waiting
- Future Read and Future Write are Atomic

1. create future
2. T1 write future
3. T2 read future

Read After Write
Future (Cont’d)

- A future is a memory cell with a state waiting to receive a data value: status: undefined, defined, waiting
- Future Read and Future Write are Atomic

1. create future
2. T1 read future
3. T2 read future
4. T3 write future

Write After Read
Stream Operation Based on Future

- Fresh Breeze Instruction Set Support 4 stream operations
  - New, Append, First and Rest

1. new stream

```
 undef
```
Stream Operation Based on Future

- Fresh Breeze Instruction Set Support 4 stream operations
  - New, Append, First and Rest

1. new stream  
2. append
Stream Operation Based on Future

- Fresh Breeze Instruction Set Support 4 stream operations
  - New, Append, First and Rest

1. new stream  
2. append  
3. first

Data1 defined  
Data1
Stream Operation Based on Future

- Fresh Breeze Instruction Set Support 4 stream operations
  - New, Append, First and Rest

1. new stream  2. append  3. first  4. rest
Related Works on Streaming Based on Dataflow Model

- Streaming based on Static Dataflow Model
- Streaming based Synchronized Dataflow Model
Translate Average Pairs to Dataflow Graph—Tail Recursive

AveragePairs

D

reset

AveragePairs

Extract

[1]

[2]

Compute

+ /2

concat

R
Translate Average Pairs to Dataflow Graph—General

AveragePairs

D → T → I → + → /2 → R

*TTF
Translate Average Pairs to Dataflow Graph—General

AveragePairs

...4321

...TTF

Compute

I

+ /2

R
Translate Average Pairs to Dataflow Graph—General

AveragePairs

\[
\text{AveragePairs} = \frac{I + T}{2}
\]

\[
\ldots 432 \xrightarrow{1} I \xrightarrow{1} \text{T}
\]

\[
\text{T} \xrightarrow{F} \ldots \text{TT}
\]

\[
\text{Compute} \quad + \quad /2 \quad R
\]

Diagram of the dataflow graph with nodes and connections.
Translate Average Pairs to Dataflow Graph—General

AveragePairs

...432

...TT

Compute

R

+ /2

I

T

1

1
Translate Average Pairs to Dataflow Graph—General

AveragePairs

...43

...T

Compute

R

\[ \text{Compute} \]

\[ \frac{1}{2} \]
Translate Average Pairs to Dataflow Graph—General

AveragePairs

\[ \frac{I + T}{2} \]

\[ \frac{T_1 + T_2}{2} \]
Translate Average Pairs to Dataflow Graph—General

AveragePairs

...43
...T
...T

Compute

2

1

I

+  3  /2

R
Translate Average Pairs to Dataflow Graph—General

AveragePairs

Compute

\[ \text{AveragePairs} = \frac{\text{\ldots} + \text{T} + \text{T} + \text{\ldots}}{2} \]

\[ \text{Compute} = \frac{\text{\ldots} + \text{T} + \text{T} + \text{\ldots}}{2} \]
Translate Average Pairs to Dataflow Graph—General

Average Pairs

\[ \text{AveragePairs} \]

\[ \text{Compute} \]

\[ \frac{I + \frac{T \cdot T}{2}}{2} \]
Translate Average Pairs to Dataflow Graph—General

AveragePairs

\[ \text{AveragePairs} = \frac{I + \frac{R}{2}}{2} \]

\[ \begin{align*}
...4 & \quad \rightarrow \quad T & \quad \rightarrow \quad 1 \\
...T & \quad \rightarrow \quad 3 & \quad \rightarrow \quad + & \quad 5 & \quad \rightarrow \quad \frac{1}{2} & \quad \rightarrow \quad R_1
\end{align*} \]
Translate Average Pairs to Dataflow Graph—General

AveragePairs

...5 → 4

...T → 4 → 1 → 3 → Compute

...T → T

R → 1

\[
\bar{I} = \frac{\sum_{i=1}^{n} I_i}{n}
\]

\[
\bar{R} = \frac{\bar{I}}{2}
\]
Can Any Streaming Program Work?

- Translate **any streaming program** into dataflow graph---Recursive and General

- Answer is YES, see Jack 94’s Paper [2]
Related Works on Streaming Based on Dataflow Model

• Streaming based on Static Dataflow Model

• Streaming based Synchronized Dataflow Model
Synchronous Dataflow Model

- Synchronous Dataflow Graph (SDF)
  - Node (actor) represents computation
  - Edge is FIFO queue representing data communication
  - Weights on Edge presents producing rate and consuming rate

Each time firing actor A produces $p$ data items and actor B consumes $q$ data items.
Average Pairs to Synchronous Dataflow Graph

Extend the SDF to support “peek” semantic
Average Pairs to Synchronous Dataflow Graph

Source → … → Average Pairs → Sink
Average Pairs to Synchronous Dataflow Graph
Average Pairs to Synchronous Dataflow Graph
Average Pairs to Synchronous Dataflow Graph
Average Pairs to Synchronous Dataflow Graph

Details about SDF model, see Lee 87’s Paper [8]
Some Projects Based on SDF model

• Early Ptolemy Project at UC Berkeley
  – Software Synthesis for Embedded system
• StreamIt at MIT
  – streaming program language and compiler
• InforStream and SPL
  – IBM streaming computing product
• Our work COStream
  – hierarchical data flow programming language and compiler
• OpenStream
  – language and compiler support for streaming in OpenMP
Resources

- Early Ptolemy Project at UC Berkeley
  - [http://ptolemy.eecs.berkeley.edu/projects/index.htm](http://ptolemy.eecs.berkeley.edu/projects/index.htm)
- StreamIt at MIT
  - [http://groups.csail.mit.edu/cag/streamit/](http://groups.csail.mit.edu/cag/streamit/)
- InforStream and SPL
- COStream
- OpenStream
  - [http://openstream.info/](http://openstream.info/)
Outline

• Introduction to DDDAS/Interaction Computation
  – An Example and Problems
• Fresh Breeze Execution Model and Architecture
  – Execution Model
  – Memory Model
  – Task Model
• Streaming and Transactions
  – Stream Type and Operations
  – Transactions
• Conclusion
Concurrent Transactions

• Scenario: A Simple Shared Hash Table
  – Shared by two concurrent users. Either user may search the value corresponding to a key, and either user may add or delete entries
  – Using concurrent shared queue
Determinate and Repeatable

• A system is *determinate* if the same ultimate output is produced for every run of the system for a presented input, and this is true for *all* choices of input and for *all interpretations* of operators.

• A parallel program schema is said to be *repeatable* if for a *specific interpretation* of the operators of the schema, the output for all runs will be strictly a function of the input.
Determinate

Input: x

Output: f(x), g(x)

g(x), f(x)

for all interpretation

Non-Determinate

Observe the input and output

• Black Box Principle
  – Determinant requires that a program schema produce the same results for given input regardless of the specific functions assigned to its operators
Repeatable

• **White Box Principle**
  – A program, with specified operations, is repeatable if any run of the program for a given input produces the same result

Input: x

\[ f(x), f(x) \]

for specific interpretation
\[ f() = g() \]

Repeatable
Determinate and Repeatable

Determinate  ➔  Repeatable

X
Revisit the Concurrent Request Example

• **Non-determinate**
  – The result order of request A and request B can not be fixed

• **Non-Repeateable**
  – Enter requests is interpreted as a non-determinate merge
Support Transaction Using Guard In FreshBreeze

- **Guard object**
  - special data object which can only be accessed by `GuardSwap` instruction

- **GuardSwap**
  - atomic instruction
  - put the new data object into guard, and return the old data object in guard

- **For the Concurrent Request Example**
  - using a guard to “lock” the tail of the queue
  - each request needs to get the guard before be added to the tail of the queue
Concurrent Writes to Queue

Two requests arrive

RA defined -> undef

guard

head

RA defined -> undef

RB defined -> undef

Request A

Request B
Concurrent Writes to Queue

1. Contend the guard
2. guardSwap (atomic)
   - RA defined → undef
   - RB defined → undef
3. Request A
4. Request B
Concurrent Writes to Queue

Request A gets the guard and old tail

guardSwap (atomic)

Request A

RA defined -> undef

guard

Request B

RB defined -> undef
Concurrent Writes to Queue

Request A substitute the old tail with the new request

WriteFuture (atomic)

Request A

RA defined → undef

guard

RB defined → undef

Request B

head

RA defined
Concurrent Writes to Queue

Request B gets guard and add to the tail

Head

RA defined

RB defined

Request A

Request B

Guard

Homework: Support concurrent reads?
// Init
queueHead, queueTail;
TransactionManager () {
    queueHead = queueTail new Future <QueueEntry> ();
    Guard managerGuard = new Guard (queueTail);
    processQueue ( queueHead );
}

enterRequest ( RQ request ) {
    QueueEntry newEntry = new QueueEntry ( request );
    Future <QueueEntry> oldTail =
    managerGuard.guardSwap (newEntry.next);
    FutureWrite ( oldTail , newEntry );
}

processQueue( Future <QueueEntry> reqQueue ) {
    QueueEntry entry = reqQueue.futureRead ();
    <RQ> request = entry . request ;
    request. processRequest ();
    Future <QueueEntry> newHead = entry.next ;
    processQueue ( newHead );
}
Conclusion

• Codelet based fine-grain execution model is promising for DDDAS

• Streaming and Transaction are two important features that support Interactive computing

• Future and Guard are two special mechanisms to support synchronization and concurrency.
Reference


Concurrent Writes to Queue

guardSwap (atomic)

Request A

RA defined

Request B

RB defined

guardSwap (atomic)

head

RA defined

RA undefined

guard

guard

RA undefined

RB undefined

guard