



JavaOne™

java.sun.com/javaone

PROJECT FORTRESS: A NEW PROGRAMMING LANGUAGE FROM SUN LABS

Christine H. Flood

Programming Language Research Group

Sun Microsystems Laboratories

TS-5206



Project Fortress Background Information

Originally funded by DARPA as part of their High Productivity Computing Systems (HPCS) project .

As of March 2008 we have an open source parallel reference interpreter which is a full implementation of the Fortress 1.0 specification.

However some of the features I discuss in this talk are not part of the 1.0 Spec.

Project Fortress:

To boldly go where no programming language
has gone before.

Not Exactly...

GOAL

Project Fortress:

To seek out great programming language design ideas and make them our own.



GOAL

Java™ Programming Language's Big Ideas (In My Humble Opinion)

“Write Once Run Anywhere”

Garbage Collection

Safety

Portable multithreading

Agenda

Top Ten Big Fortress Ideas

Fortress Big Idea #10:



Contracts

Fortress Big Idea #10: Contracts

Requires

Ensures

Invariants

```
factorial(n) requires {n ≥ 0} =  
  if n = 0 then 1 else n factorial(n - 1) end
```


Fortress Big Idea #9:



Dimensions and Units

Big Idea #9: Dimensions and Units

Dimensions as types

Prevent errors such as adding kilometers to a variable in miles.

distance := 60 miles/hour (3600 seconds in hours)

See:

Object-oriented units of measurement

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Guy L. Steele

OOPSLA '04:

Fortress Big Idea #8:

A large, light blue horizontal scroll graphic with rounded ends and a vertical strip on the left side, resembling a rolled-up document. The text "Traits and Objects" is centered on the scroll.

Traits and Objects

Big Idea #8: Traits and Objects

Multiple vs. single inheritance

Single inheritance is limiting.

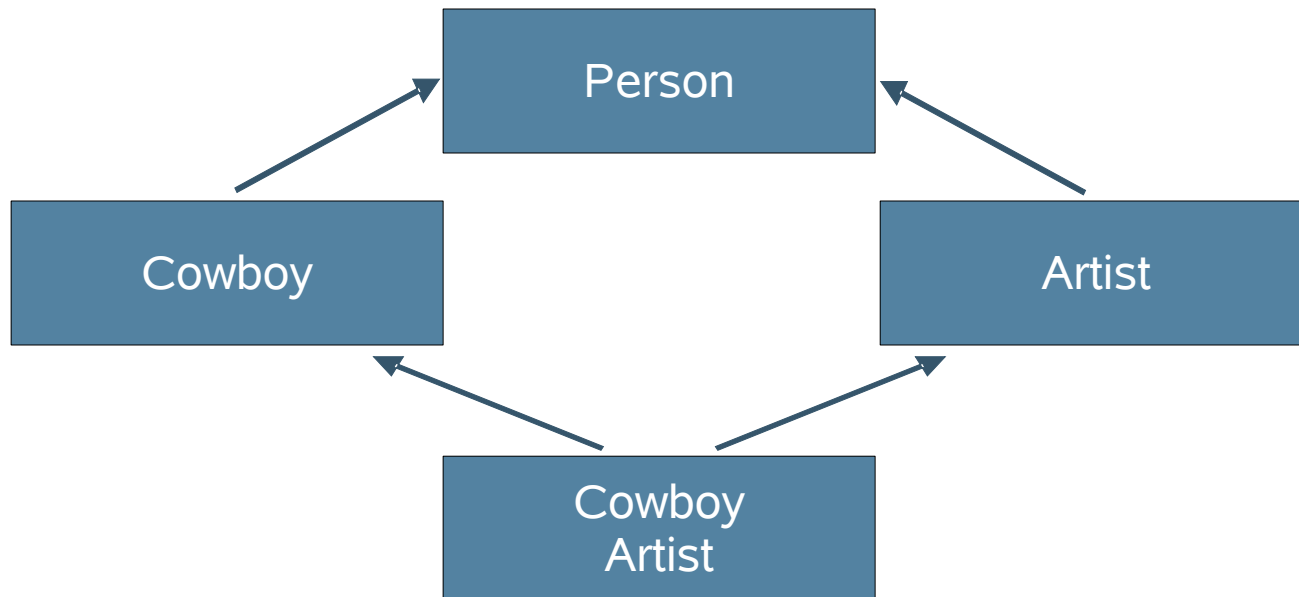
Multiple inheritance is complicated.

Java programming language works around this by having single inheritance augmented by interfaces.

Traits and Objects solve this problem by having multiple inheritance of methods, but not fields.

Big Idea #8: Traits and Objects

Multiple Inheritance Diamond Problem



If both Cowboy and Artist define a draw method, then this is a compile time error in Fortress unless Cowboy Artist has a more specific method that covers them both. There must be one most specific method.

Objects with fields are the leaves of the hierarchy and therefore multiple inheritance is not an issue.

Big Idea #8: Traits and Objects

Extends and comprises enhance
readability

```

trait List comprises {Cons, Empty}
    cons(x : Object) = Cons(x, self)
end

object Cons(first : Object, rest : List)
    extends List
end

object Empty extends List
end
  
```

Fortress Big Idea #7:



Functional Methods

Big Idea #7: Functional Methods

Sometimes you want to define methods which have self not be the first parameter.

This allows you to define methods in subtypes for cleaner code.

```
trait BigNum extends Number
  opr -(Number, self) : BigNum...
  ...
end
```


Fortress Big Idea #6:



Parametric Polymorphism

Big Idea #6: Parametric Polymorphism

Subtype polymorphism allows code reuse.

```
object Container(var element : Object)
  setElement(e : Object) : () = element := e
  getElement() : Object = element
```

end

(* Storing: safe upcasts *)

```
c = Container(0)
```

```
c.setElement(2)
```

(* Retrieving: potentially unsafe downcasts *)

```
x :  $\mathbb{Z}_{64}$  = c.getElement()
```

Big Idea #6: Parametric Polymorphism

Parametric polymorphism allows safe code reuse.

```
object Container[[T extends Equality[[T]]]  
  (var element:T)  
  setElement(e:T):() = element := e  
  getElement(): T = element  
end
```

```
c = Container[[Z64]](0)  
c.setElement(2)  
x:Z64 = c.getElement()
```

Fortress Big Idea #5:



Generators and Reducers

Big Idea #5: Generators and Reducers

$$y = \sum_{k \leftarrow 1:n} a_k x^k$$

$$z = \text{MAX}[(j, k) \leftarrow a.\text{indices}] |a_{jk} - b_{jk}|$$

Reducers such as Σ (or **SUM**) and **MAX** are defined by libraries.

Reducers are driven by generators.

Generators may have serial or parallel implementations.

Distribution of generator guides parallelism of reducer.

Big Idea #5: Generators and Reducers

```

database : Map[[String, Z32]] =
    BIG UNIONSUM [l ← rs.lines()](getWords(l))
invDatabase : Map[[Z32, List[[String]]]] =
    BIG UNIONUNION [(x, y) ← database](makeInv(x, y))
  
```

This is an example from our pod demo.

We are finding the most common words in a document in parallel.

The first line creates a mapping from words to occurrence counts.

The second line inverts the mapping so we can find the top n most common words in the document.

rs.lines() and (*x, y*) <- *database* are parallel generators.
The big operators are reducers.

Fortress Big Idea #4:



Mathematical Syntax

Big Idea #4: Mathematical Syntax

Fortress syntax looks more like a math text book than a traditional programming language.

Goal: What you write on your whiteboard works.

$$v_{\text{norm}} = v / \|v\|$$

$$\sum_{k \leftarrow 1:n} a_k x^k$$

$$C = A \cup B$$

$$y = 3x \sin x \cos 2x \log \log x$$

Big Idea #4: Mathematical Syntax (NAS CG Kernel)

Specification:

```

z = 0
r = x
ρ = rT r
p = r
DO i = 1, 25
    q = A p
    α = ρ / (pT q)
    z = z + α p
    ρ0 = ρ
    r = r - α q
    ρ = rT r
    β = ρ / ρ0
    p = r + β P
ENDDO
compute residual norm explicitly:
||r|| = ||x - Az||
  
```

Fortress Code:

```

conjGrad(A: Matrix[[R64]], x: Vector[[R64]]):
    (Vector[[R64]], R64) = do
        cgit_max = 25
        z: Vector[[R64]] := 0
        r: Vector[[R64]] := x
        p: Vector[[R64]] := r
        ρ: R64 := rT r
        for j ← seq(1: cgit_max) do
            q = A p
            α = ρ / pT q
            z := z + α p
            r := r - α q
            ρ0 = ρ
            ρ := rT r
            β = ρ / ρ0
            p := r + β p
        end
        (z, ||x - A z||)
    end
  
```

Big Idea #4: Mathematical Syntax

Comparison: NAS NPB 2.3 Serial Code

```

do j=1,naa+1
  q(j) = 0.0d0
  z(j) = 0.0d0
  r(j) = x(j)
  p(j) = r(j)
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
  do j=1,lastrow-firstrow+1
    sum = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
    enddo
    w(j) = sum
  enddo
  do j=1,lastcol-firstcol+1
    q(j) = w(j)
  enddo

```

```

do j=1,lastcol-firstcol+1
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + p(j)*q(j)
enddo
d = sum
alpha = rho / d
rho0 = rho
do j=1,lastcol-firstcol+1
  z(j) = z(j) + alpha*p(j)
  r(j) = r(j) - alpha*q(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
beta = rho / rho0
do j=1,lastcol-firstcol+1
  p(j) = r(j) + beta*p(j)
enddo
enddo

```

```

do j=1,lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j),rowstr(j+1)-1
    sum = sum + a(k)*z(colidx(k))
  enddo
  w(j) = sum
enddo
do j=1,lastcol-firstcol+1
  r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  d = x(j) - r(j)
  sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )

```

Big Idea #4: Mathematical Syntax

Which would you rather write?

```

do j=1,naa+1
  q(j) = 0.0d0
  z(j) = 0.0d0
  r(j) = x(j)
  p(j) = r(j)
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
  do j=1,lastrow-firstrow+1
    sum = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
      sum = sum +
a(k)*p(colidx(k))
    enddo
    w(j) = sum
  enddo
  do j=1,lastcol-firstcol+1
    q(j) = w(j)
  enddo
enddo

do j=1,lastcol-firstcol+1
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + p(j)*q(j)
enddo
d = sum
alpha = rho / d
rho0 = rho
do j=1,lastcol-firstcol+1
  z(j) = z(j) + alpha*p(j)
  r(j) = r(j) - alpha*q(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
beta = rho / rho0
do j=1,lastcol-firstcol+1
  p(j) = r(j) + beta*p(j)
enddo
enddo

do j=1,lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j),rowstr(j+1)
    sum = sum + a(k)*z(coli
  enddo
  w(j) = sum
enddo
do j=1,lastcol-firstcol+1
  r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  d = x(j) - r(j)
  sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )

```

```

conjGrad(A: Matrix[[Float]], x: Vector[[Float]]):
  (Vector[[Float]], Float) = do
    cgit_max = 25
    z: Vector[[Float]] := 0
    r: Vector[[Float]] := x
    p: Vector[[Float]] := r
    ρ: Float := rT r
    for j ← seq(1: cgit_max) do
      q = A p
      α = ρ/pT q
      z := z + α p
      r := r - α q
      ρ0 = ρ
      ρ := rT r
      β = ρ/ρ0
      p := r + β p
    end
    (z, ||x - A z||)
  end

```

Big Idea #4: Mathematical Syntax

NAS CG Kernel (ASCII)

```

conjGrad(A: Matrix[\Float\], x: Vector[\Float\]):
  (Vector[\Float\], Float) = do
    cgit_max = 25
    z: Vector[\Float\] := 0
    r: Vector[\Float\] := x
    p: Vector[\Float\] := r
    rho: Float := r^T r
    for j <- seq(1:cgit_max) do
      q = A p
      alpha = rho / p^T q
      z := z + alpha p
      r := r - alpha q
      rho0 = rho
      rho := r^T r
      beta = rho / rho0
      p := r + beta p
    end
    (z, ||x - A z||)
  end

```

Big Idea #4: Mathematical Syntax

Why don't all programming languages look like math?

- Parsing
- Type inference
- Overloading

Big Idea #4: Mathematical Syntax

Parsing

Unicode enabled

Requires a PackRat parser. We use Rats!

Whitespace-sensitive grammar

Example: juxtaposition is an operator

a b

Means a times b if a and b are numeric types. It's perfectly natural to mathematicians, revolutionary to computer programmers.

See:

Better Extensibility through Modular Syntax

Robert Grimm

PLDI 2006

Fortress Big Idea #3:



Transactional Memory

Big Idea #3: Transactional Memory

Programming with locks is hard, often inefficient, and error prone.

Transactions are simple and easy to reason about.

As with GC, let the run time system do the heavy lifting, not the application programmer or library writer.

Big Idea #3: Transactional Memory

```
for  $(i, j) \leftarrow a.indices()$  do
  atomic do
     $hist[a_{ij}] += 1$ 
  end
end
```

Big Idea #3: Transactional Memory

Fortress requires:

Software Transactional Memory

Hardware transactional memory takes advantage of a processors cache to keep track of accesses.

We may have multiple threads cooperating in a single transaction.

Nested Transactions

Mixing atomic and non-atomic accesses to the same data.

Big Idea #3: Transactional Memory

Implementation

Built on Top of DSTM2.

All mutable values are represented by Reference Cells and may be a part of a transaction.

See:

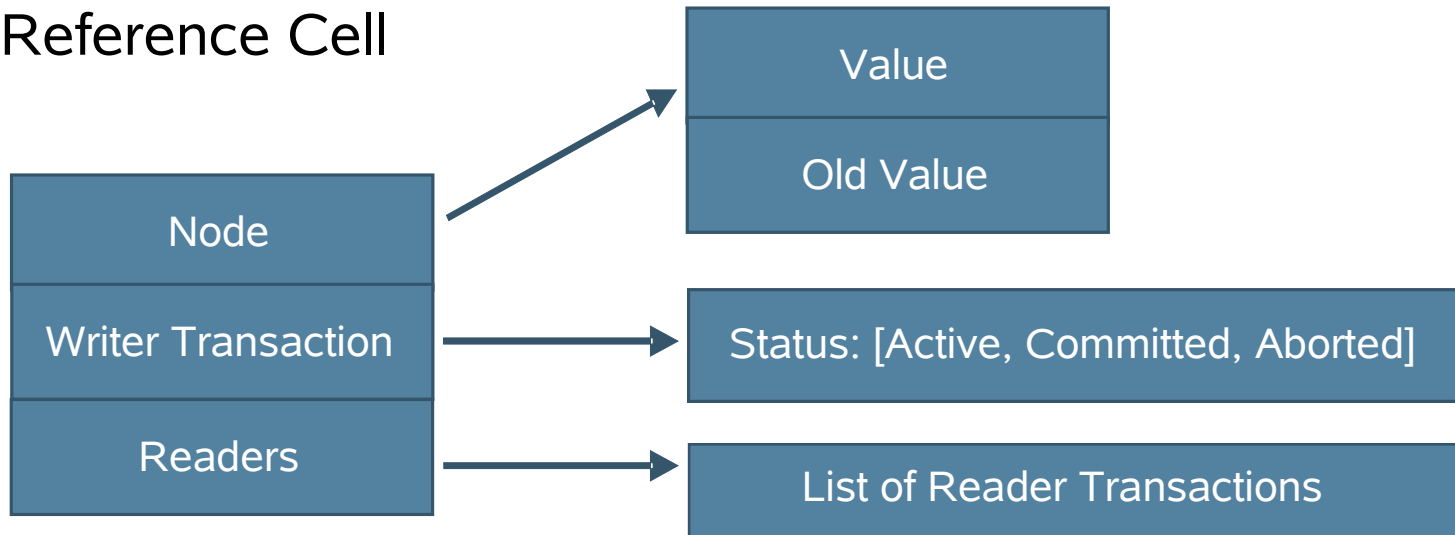
A flexible framework for implementing software transactional memory
Maurice Herlihy, Victor Luchangco, Mark Moir
OOPSLA 2006

Download the source code for DSTM2:

<http://www.sun.com/download/products.xml?id=453fb28e>

Big Idea #3: Transactional Memory

Reference Cell



Status update via compare and set

Big Idea #3: Transactional Memory Example

Thread 1

```
atomic do
  x := 3
  y := 7
end
```

Thread 2

```
atomic do
  x := 30
  y := 70
end
```

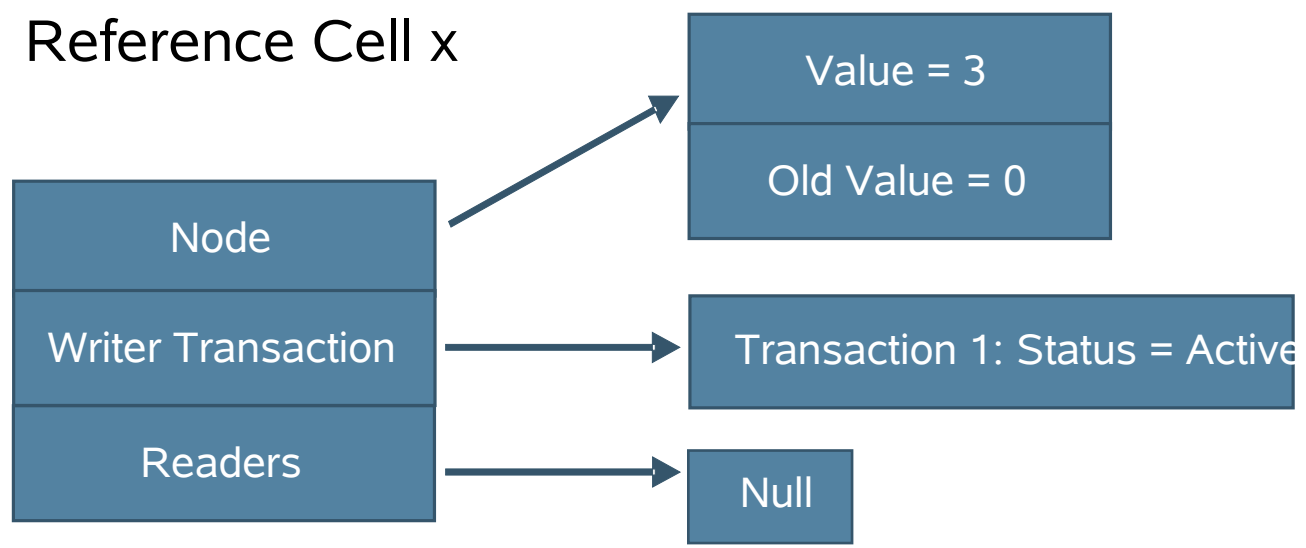
After these two threads run, the values of (x,y) are either (3,7) or (30,70).

Big Idea #3: Transactional Memory Example

```
atomic do
  x := 3
  y := 7 ← Thread 1
end
```

```
atomic do
  x := 30 ← Thread 2 discovers conflict with Thread 1
  y := 70
end
```

Thread 2 discovers conflict with Thread 1



Big Idea #3: Transactional Memory

Contention Managers

Current strategy

Transaction created by the lowest numbered thread wins.

Losers backoff via spin and retry.

One transaction always makes progress.

Big Idea #3: Transactional Memory

Why not have per transaction read sets instead of per object read sets?

A transaction would keep track of every value it read and then prior to committing updates it would validate that the read values haven't changed.

Validating the reads before a commit may take a long time; we can't block other threads for that long.

Fortress Big Idea #2:



Implicit Parallelism

Big Idea #2: Implicit Parallelism

[As multicore processor chips become ubiquitous,]
“what we are seeing is not a gradual shift but a
cataclysmic shift from the sequential world to one in
which every processor is parallel. In a small number of
years, if your language does not support parallelism,
that language will just wither and die.”

—John Mellor-Crummey, Rice University
(*Computerworld*, March 12, 2007)

Big Idea #2: Implicit Parallelism

```
doSomething(foo(a), foo(b))
```

tuples

```
for i ← 1 # 10 do  
    doSomething(i)  
end
```

for loops

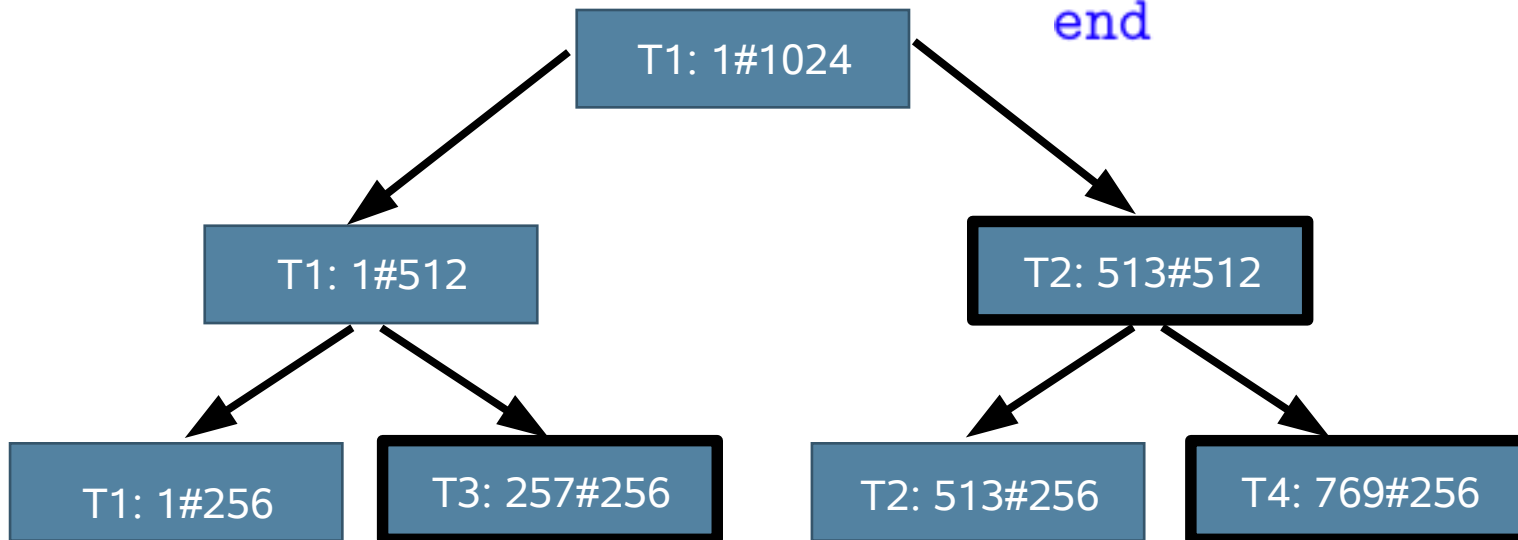
```
do  
    foo(a)  
also do  
    foo(b)  
end
```

also do

Big Idea #2: Implicit Parallelism Work Stealing

```

for i ← 1 # 1024 do
  doSomething(i)
end
  
```



The work is quickly distributed among threads (T1, T2, T3, and T4)

The darker boxes represent work that was stolen by idle threads

If machine is busy work stays local.

Big Idea #2: Implicit Parallelism

Implementation

Work Stealing Queues

Built on top of Doug Lea's jsr166y forkjoin library.

Work is pushed onto a per thread deque.

Unsynchronized local pushes and pops most of the time.

Idle threads may steal work from the top of another thread's deque.

The cost of packaging up work and making it available to steal is minimal.

See:

Thread Scheduling for Multiprogrammed Multiprocessors

Arora, Blumofe, Plaxton

SPAA 1998

Big Idea #2: Implicit Parallelism

Tasks are units of interpreter work which are put on deques

We have three types:

- EvaluatorTask primordial task
- TupleTasks tuples and desugaring for loops
- SpawnTasks fair threads

Fair threads are for when you really want a separate OS level thread.

Big Idea #2: Implicit Parallelism

When you write:

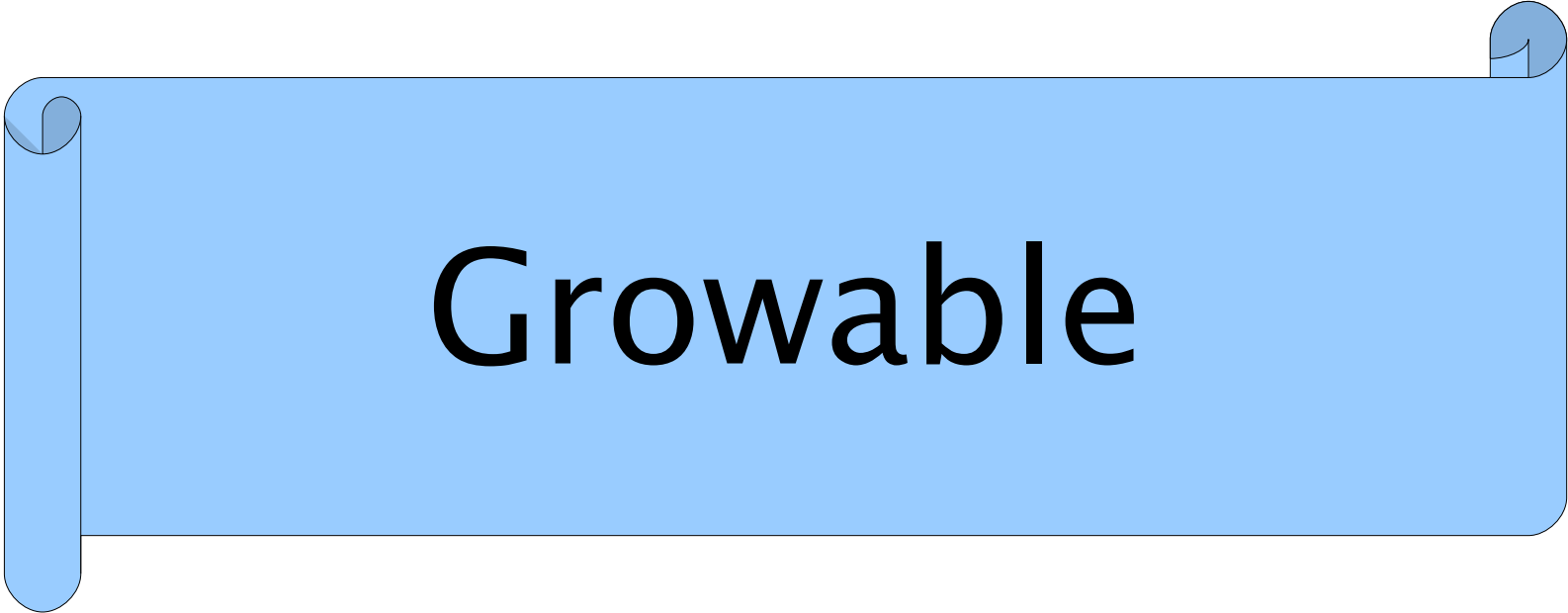
```
doSomething(foo(a), foo(b))
```

We generate an array of TupleTasks which we pass to the current FortressTaskRunner.

```
tupleTask.forkJoin(tasks);
```

This will push one on the queue, and work on the other one. Once these two tasks are completed the task runner may move on to the next statement.

Fortress Big Idea #1:

A large, light blue horizontal scroll graphic with rounded ends and a vertical strip on the left side, resembling a rolled-up document. The word "Growable" is centered on the scroll.

Growable

Big Idea #1: **Growable**

Rome wasn't built in a day.

Modern languages are huge and need to evolve over time.

Fortress was designed from the beginning to have a small fixed core with strong support for library writers.

See:

Steele, “Growing a Language” keynote talk, OOPSLA 1998;
Higher-Order and Symbolic Computation **12**, 221–236 (1999)

Big Idea #1: Growable

```
for  $i \leftarrow 1$  # 1024 do  
    doSomething(i)  
end
```

Gets desugared into a library call with

1. A generator clause 1#1024
2. A loop body doSomething(i)

The library code is responsible for recursively subdividing the loop iterations and generating tuple tasks.

This means that if you want to change the generator to exploit new hardware, you can do it at the Fortress language level in a library.

Big Idea #1: Growable

- Also Defined in Libraries:
 - Almost all types:
 - Booleans
 - Arrays
 - Lists
 - Matrices
 - Sets
 - Operators
 - + - < >
 - Generators and Reducers

Open Source Community

Source code available online
projectfortress.sun.com

Come take it for a spin, or pitch in and help us grow.

Project Fortress Contributors:

Eric Allen

David Chase

Joao Dias

Carl Eastlund

Christine Flood

Joe Hallett

Yuto Hayamizu

Scott Kilpatrick

Yossi Lev

Victor Luchangco

Jan-Willem Maessen

Cheryl McCosh

Janus Dam Nielsen

Andrew Pitonyak

Sukyoung Ryu

Dan Smith

Michael Spiegel

Guy L. Steele Jr.

Sam Tobin-Hochstadt

<Your Name Here>

Summary

Project Fortress is a new Open Source High Productivity Programming Language aimed at multi-processors with the following features:

1. Growable
2. Implicit Parallelism
3. Transactional Memory
4. Mathematical Syntax
5. Generators and Reducers
6. Parametric Polymorphism
7. Functional Methods
8. Traits and Objects
9. Dimensions and Units
10. Contracts

References

A flexible framework for implementing software transactional memory

Maurice Herlihy, Victor Luchangco, Mark Moir
OOPSLA 2006

Better Extensibility through Modular Syntax

Robert Grimm
PLDI 2006

Thread Scheduling for Multiprogrammed Multiprocessors

Arora, Blumofe, Plaxton
SPAA 1998

Steele, “Growing a Language” keynote talk, OOPSLA 1998;
Higher-Order and Symbolic Computation **12**, 221–236 (1999)

Object-oriented units of measurement

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Guy L. Steele

OOPSLA '04:

For More Information

- Fortress Language Specification
 - research.sun.com/projects/plrg/
- Reference Implementation
 - projectfortress.sun.com
 - Note not www.projectfortress.sun.com.
- Come see our Demo Pod on the Pavilion floor
- Other related JavaOne Talks:
 - TS-6316 Transactional Memory in Java Technology-Based Systems
 - TS-6206 JVM Machine Challenges and Directions in the Multicore Era
 - TS-6256 Toward a Coding Style for Scalable Nonblocking Data Structures

THANK YOU



Christine H. Flood
Project Fortress: A New Programming
Language from Sun Labs

TS-5206

