JVMCSP

Approaching Billions of Processes on a Single-Core JVM

Cabel Shrestha & Matt B. Pedersen

UNLV | Howard R. Hughes College of Engineering

Last Time … (CPA 2014)

We presented

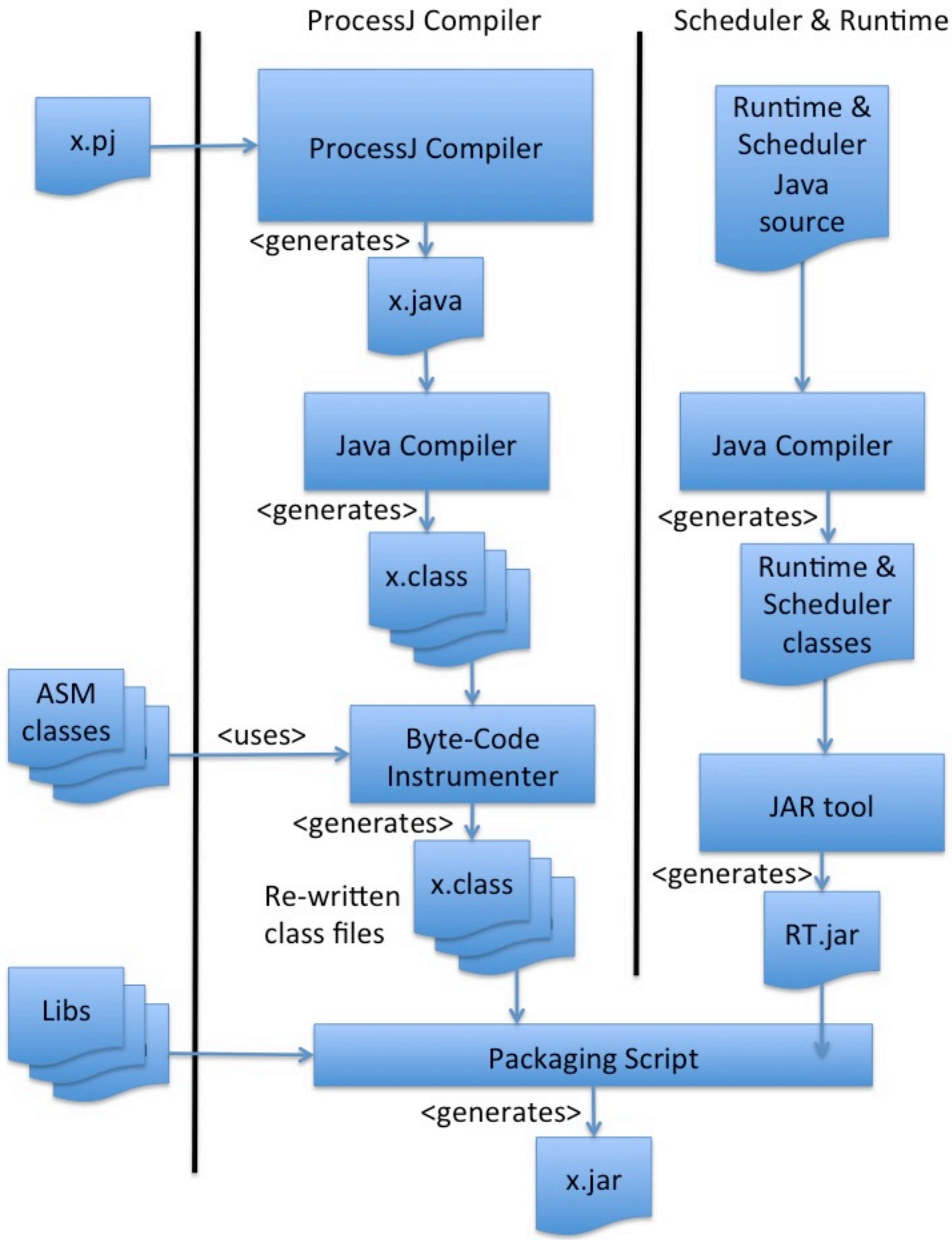
- ProcessJ (Process-Oriented Language)
- Handcrafted JVM runtime: LiteProc (proof of concept)
- ~ 95,000,000 concurrent processes

This Time … (CPA 2016)

- An implemented code generator in ProcessJ
- New improved runtime
  - Faster
  - Handles more processes

From ProcessJ to Java Bytecode

- ProcessJ compiler produces Java source code.
- Compiled with Javac.
- Instrumented with ASM byte code manipulation tool.
- Jar'd together with runtime


ProcessJ Compiler
Scheduler & Runtime
x.pj
ProcessJ Compiler
<generates>
x.java
Java Compiler
<generates>
x.class
ASM classes
<uses>
Byte-Code Instrumenter
<generates>
x.class
Re-written class files
Libs
Packaging Script
<generates>
x.jar
Runtime & Scheduler Java source
Java Compiler
<generates>
Runtime & Scheduler classes
JAR tool
<generates>
RT.jar

Runtime (Scheduler)

- User-level scheduler
  - Cooperative, non-preemptive.

```
Queue<Process> processQueue;
...
// enqueue one or more processes to run
while (!processQueue.isEmpty()) {
  Process p = processQueue.dequeue();
  if (p.ready())
    p.run();
  if (!p.terminated())
    processQueue.enqueue(p);
}
```

Essential Questions

- How does a procedure yield?
- When does a procedure yield and who decides?
- How is a procedure restarted after yielding?
- How is local state maintained?
- How are nested procedure calls handled when the innermost procedure yields?

How does a procedure yield? When does it yield and who decides?

CPA 2014 version

- Yields by calling return
- Procedures voluntarily give up the CPU at synchronization points

JVMCSP

- Yields by calling return
- Procedures voluntarily give up the CPU at synchronization points

Reads, writes, barrier syncs, alts, timer operations: procedure returns to scheduler (Bytecode: return)

How is a procedure restarted?

CPA 2014

- Procedure is simply recalled by scheduler

JVMCSP

- Procedure is simply recalled by scheduler

- How do we ensure that local state survives?
- How do we avoid restart from the top of the code?

Preservation of Local State

CPA 2014

- An activation record structure was used to store locals.
- Each procedure is a class with an activation stack.

JVMCSP

- All locals and fields have been converted to fields.
- Each procedure is a class.

Correct Resumption

CPA 2014

- Insert an empty switch statement at the top of the code to hold jumps.
- Instrument (by hand in decompiled bytecode) jumps to the correct resume points.

JVMCSP

- Insert an empty switch statement at the top of the generated code (source) to hold jumps.
- Instrument (by using ASM) jumps to the correct resume points.

A resume point counter (called `runlabel`) is kept for each process to remember where to continue.

Correct Resumption (Abstract)

- Each synchronization point is a yield point:

```
L1:
    .. synchronize (read, sync etc)
    if (succeeded)
      yield(L2); // return to L2 when resumed
    else
      yield(L1); // return to L1 when resumed
L2:
```

Correct Resumption (Generated Code)

- Each synchronization point is a yield point:

```
label(1);
.. synchronize (read, sync etc)
if (succeeded)
  yield(2);
else
  yield(1);
label(2);
```

`yield(i)` will set the runlabel for the process object to `i`.

Correct Resumption (ASM Instrumentation)

```
label(1);
.. synchronize
if (succeeded)
  yield(2);
else
  yield(1);
label(2);
```

```
61: aload_0
62: iconst_1
63: invokevirtual label/(I)V
66: ...
...
```

```
61: nop
62: nop
63: nop
64: nop
65: nop
66: ...
...
```

Dummy invocations are removed.

Correct Resumption (ASM Instrumentation)

- This address (61) is associated with runlabel 1.
- Upon resumption, the code must jump to address 61 if the runlabel is 1.

```
61: nop
62: nop
63: nop
64: nop
65: nop
66: ...
...
```

Correct Resumption (Generated Code)

Generated Java Code

(top of the code)

```
switch (runlabel) {
  case 0: resume(0);
          break;
  case 1: resume(1);
          break;
  ...
  case k: resume(k);
          break;
}
```

Equivalent Java Bytecode

```
0: aload 0
1: getfield runLabel
4: tableswitch  // 0 to 2
    0: 32
    1: 35
    2: 43
    default: 48
32: goto 48
35: aload 0
36: iconst 1
37: invokevirtual resume/(I)V
40: goto 48
...
```

Correct Resumption (ASM Instrumentation)

```
0: aload 0
1: getfield runLabel
4: tableswitch  // 0 to 2
    0: 32
    1: 35
    2: 43
    default: 48
32: goto 48
35: aload 0
36: iconst 1
37: invokevirtual resume/(I)V
40: goto 48
...
```

Runlabel 1

```
0: aload 0
1: getfield runLabel
4: tableswitch  // 0 to 2
    0: 32
    1: 35
    2: 43
    default: 48
32: goto 48
35: nop
36: nop
37: goto 61
40: goto 48
...
```

@ of runlabel 1

Placeholder code replaced by nop instructions and gotos adjusted to the correct label addresses

Correct Suspension

```
yield(2);
```

Becomes

```
78: aload_0
79: iconst_2
80: invokevirtual yield/(I)V
83: goto 100
...

100: return
```

Shared return point

`yield(2)` sets the `runLabel` field.

From ProcessJ to Java

```
proc void foo(pt_1 pn_1, ..., tp_n pn_n) {
   ...
   lt_1 ln_1;
   ...
   lt_m ln_m;
   ... statements ...
}
```

- Parameters
- Locals
- Code

From ProcessJ to Java

New class

```
public class A {
  public static class foo
        extends PJProcess {





                  }
                }
```

Process `foo` lives in a file called `A.pj`

From ProcessJ to Java

```
public class A {
  public static class foo
         extends PJProcess {
    pt1 pn1;
    pt2 pn2;
    ...
    lt1 ln1;
    ...
    ltm lnm;
```

Parameters

locals

```
    }
  }
```

Locals and Parameters are turned into fields

From ProcessJ to Java

```
public class A {
  public static class foo
        extends PJProcess {
    pt1 pn1;
    pt2 pn2;
    ...
    lt1 ln1;
    ...
    ltm lnm;

    public foo(pt1 pn1, ...,
               tpn pnn) {
      this.pn1 = pn1;
      ...
      this.pnn = pnn;
    }
  }
}
```

Constructor

Constructors set the parameters

From ProcessJ to Java

```
public class A {
  public static class foo
        extends PJProcess {
    pt_1 pn_1;
    pt_2 pn_2;
    ...
    lt_1 ln_1;
    ...
    lt_m ln_m;

    public foo(pt_1 pn_1, ...,
               tp_n pn_n) {
      this.pn_1 = pn_1;
      ...
      this.pn_n = pn_n;
    }
```

Run method

```
public void run() {

}
  }
}
```

run method is called by the scheduler

From ProcessJ to Java

```
public class A {
  public static class foo
        extends PJProcess {
    pt1 pn1;
    pt2 pn2;
    ...
    lt1 ln1;
    ...
    ltm lnm;

    public foo(pt1 pn1, ...,
               tpn pnn) {
      this.pn1 = pn1;
      ...
      this.pnn = pnn;
    }
```

Jump switch

```
    public void run() {
      switch (runlabel) {
        case 0: resume(0);
                break;
        case 1: resume(1);
                break;
        ...
        case k: resume(k);
                break;
      }
    }
  }
}
```

`resume()` calls replaced by jumps to `label()`s

From ProcessJ to Java

```
public class A {
  public static class foo
        extends PJProcess {
    pt_1 pn_1;
    pt_2 pn_2;
    ...
    lt_1 ln_1;
    ...
    lt_m ln_m;

    public foo(pt_1 pn_1, ...,
               tp_n pn_n) {
      this.pn_1 = pn_1;
      ...
      this.pn_n = pn_n;
    }

    public void run() {
      switch (runlabel) {
        case 0: resume(0);
                break;
        case 1: resume(1);
                break;
        ...
        case k: resume(k);
                break;
      }
      ... Statements
    }
  }
}
```

Code

Code is translated ProcessJ + generated primitives

Yielding in Nested Calls

CPA 2014

- Maintain a complex activation stack.
  - Constant creation and destruction of activation records.
  - Resumptions started from the outermost procedure and worked its way in

JVMCSP

- Calls of procedures that may yield

`f(x)` becomes

```
par {
   f(x)
}
```

JVMCSP Runtime Componenets

- `PJProcess` represents a process.
- PJPar represents a par block.
- `PJChannel` represetns a channel.
  - `PJOne2OneChannel`, `PJOne2ManyChannel`, `PJMany2OneChannel`, `PJMany2ManyChannel`
- `PJBarrier` represents a barrier.
- `PJTimer` represents a timer.
- `PJAlt` represents an alt.

Par Blocks

```
par {
   f(8);
   g(9);
}
```

becomes

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Par Blocks

Create new PJPar object with 2 processes

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Par Blocks

Instantiate an f process

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Par Blocks

Decrement process count of par when done

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Par Blocks

Schedule the process

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Par Blocks

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Set process with par not ready to run

Par Blocks

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

Yield to the scheduler
(this is just a return)

Par Blocks

```
final PJPar par1 = new PJPar(2, this);
(new A.f(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
(new A.g(8) {
  public void finalize() {
    par1.decrement();
  }
}).schedule();
setNotReady();
yield(1);
label(1);
```

When ready again continue here

Channel Read

```
x = in.read();
```

Becomes

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

Return here if channel is not ready for read

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

If the channel is ready (data present)

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

Read

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

Yield and return at label 3

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

If no, set this process not read to run

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

Add the reader to the channel

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

Channel Read

Yield and return
at label 2 next time

```
...
label(2);
if (in.isReadyToRead(this)) {
  x = in.read();
  yield(3);
} else {
  setNotReady();
  in.addReader(this);
  yield(2);
}
label(3);
```

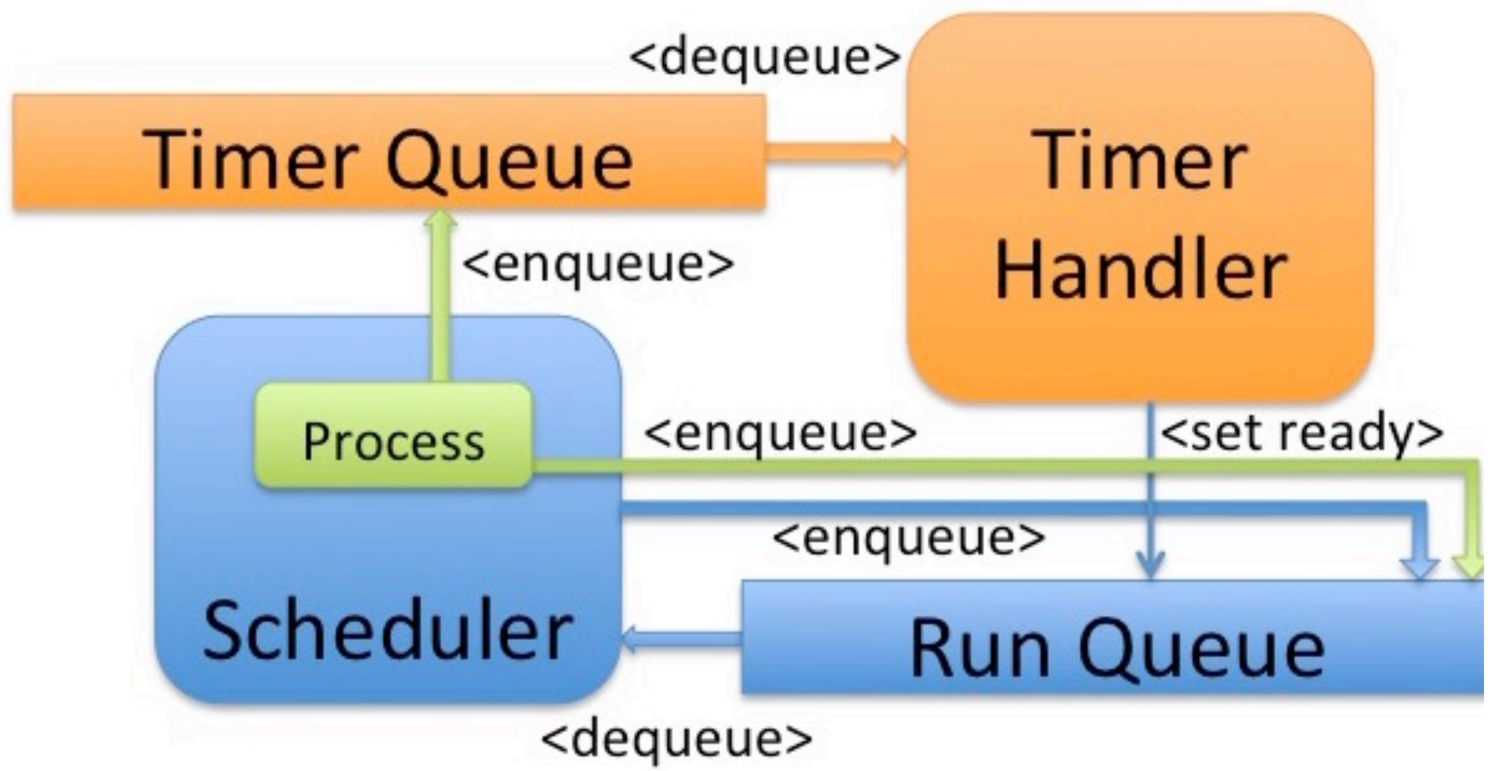
Other Channel Operations

- Channel writes are similar to reads.
- Channels with shared ends must be claimed.
  - Functionality to claim and unclaim is included in PJ...2... channel classes.

Timers and the Timer Queue

- Timers are handled by a TimerQueue and a TimerHandler.
  - The TimerQueue is a delay-queue.
  - Timeout calls cause insertions into TimerQueue
- TimerHandler dequeues expired timers from the TimerQueue.
  - Sets corresponding processes ready to run.

Timers and the Timer Queue



Timers

```
t.timeout(100);
```

Becomes

```
t.start(100);
setNotReady();
yield(1);
label(1);
```

`t.start(100)` will insert a new timer object into the TimerQueue.

Barriers

```
sync(b);
```

Becomes

```
b.sync(this);
yield(1);
label(1);
```

`b.sync(this)` will

* decrement the barrier's process counter
* enqueue the process in the barrier's process list
* set itself not ready

When counter reaches 0 all processes are set ready.

Alts

- We probably do not have time for this… but they are cool.

Results

- Timing
- Context switching
- Max process count
- Overhead (we will skip this one too)

Timings and Context Switches

Mandelbrot fractal image 4,000 x 3,000 (12,000,000 pixels)

Version	Time (Sec.)	# Processes	# Context Switches
Java Sequential	6.24	1	0
ProcessJ Sequential	6.21	1	0
ProcessJ row parallel	6.05	3,001	3,001
ProcessJ pixel parallel	31.98	12,000,001	12,003,001

Context Switching Time

CommsTime

	Mac / OS X			AMD / Linux		
	CPA'14	JCSP	JVMCSP	CPA'14	JCSP	JVMCSP
µs/iteration	9.26	27.00	8.30	13.56	136.00	7.52
µs/communication	2.31	6.00	2.08	3.90	35.00	1.88
µs/context switch	1.32	3.00	0.69	1.94	17.00	0.63

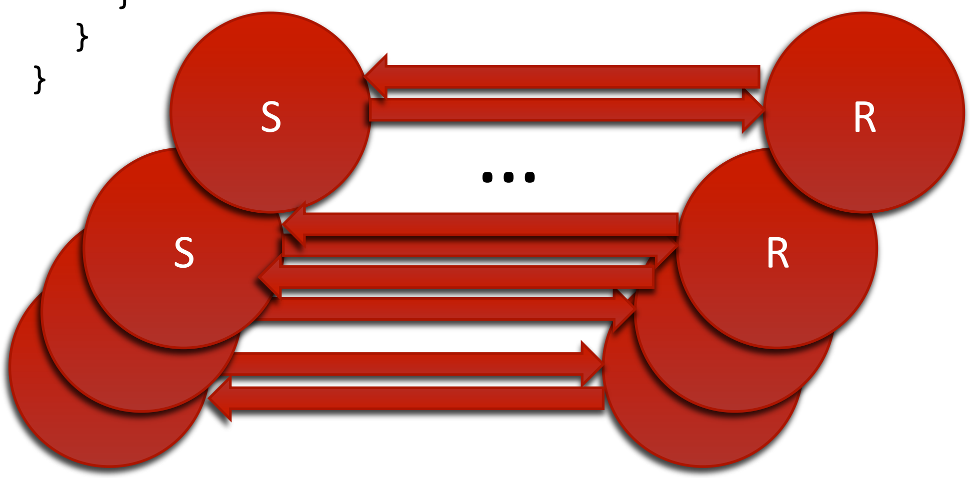
Max Process Count

```
import std.strings;

proc void foo(chan<int>.read c1r,
              chan<int>.write c2w) {
  int x;
  par {
    x = c1r.read();
    c2w.write(10);
  }
}

proc void bar(chan<int>.write c1w,
              chan<int>.read c2r) {
  int y;
  par {
    y = c2r.read();
    c1w.write(20);
  }
}
```

```
proc void main(string[] args) {
  par for (int i=0;
           i<string2int(args[1]);
           i++) {
    chan<int> c1, c2;
    par {
      foo(c1.read, c2.write);
      bar(c1.write, c2.read);
    }
  }
}
```



Max Process Count

# Processes	# Context Switches	Execution Time (Secs.)	Memory Usage (GB)
7,000,001	15,000,002	7.53	1.79
10,000,001	22,500,002	16.03	3.02
14,000,001	30,000,002	25.86	4.10

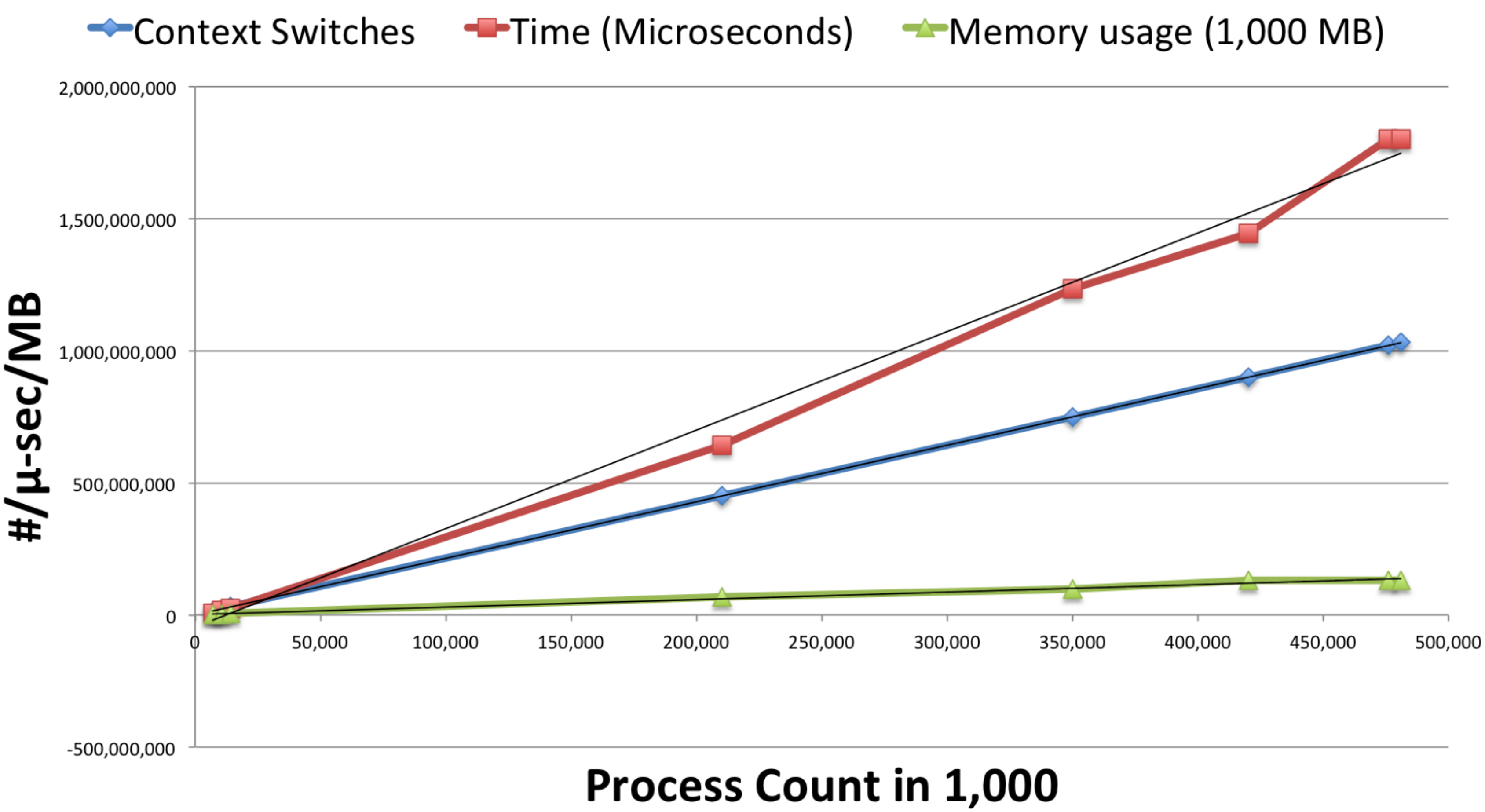
Max Process Count

# Processes	# Context Switches	Execution Time (Secs.)	Memory Usage (GB)
7,000,001	15,000,002	7.53	1.79
10,000,001	22,500,002	16.03	3.02
14,000,001	30,000,002	25.86	4.10
210,000,001	450,000,002	642.80	63.91
350,000,001	750,000,002	1,235.12	94.50
420,000,001	900,000,002	1,443.40	125.82

Max Process Count

# Processes	# Context Switches	Execution Time (Secs.)	Memory Usage (GB)
7,000,001	15,000,002	7.53	1.79
10,000,001	22,500,002	16.03	3.02
14,000,001	30,000,002	25.86	4.10
210,000,001	450,000,002	642.80	63.91
350,000,001	750,000,002	1,235.12	94.50
420,000,001	900,000,002	1,443.40	125.82
476,000,001	1,020,000,002	1,800.79	126.11
480,900,001	1,030,500,002	1,801.40	126.20

Max Process Count



Conclusion

- ProcessJ code generator that produces Java source.
- JVMCSP runtime implemented.
- ASM bytecode instrumentation.
- Performs better than CPA’14 and JCSP.
- Can handle approximately **half a billion** processes in 128GB.

Future Work

- Multi-core Scheduler
- Network distribution
- Libraries
- Mobile processes
- Alts & claims are `busy waits' (remain ready to run and cycle through the run queue)
- More back ends

Other Back Ends

- Omar and Austin won gold in the UNLV College of Engineering Senior Design Competition for a CCSP code generator for ProcessJ