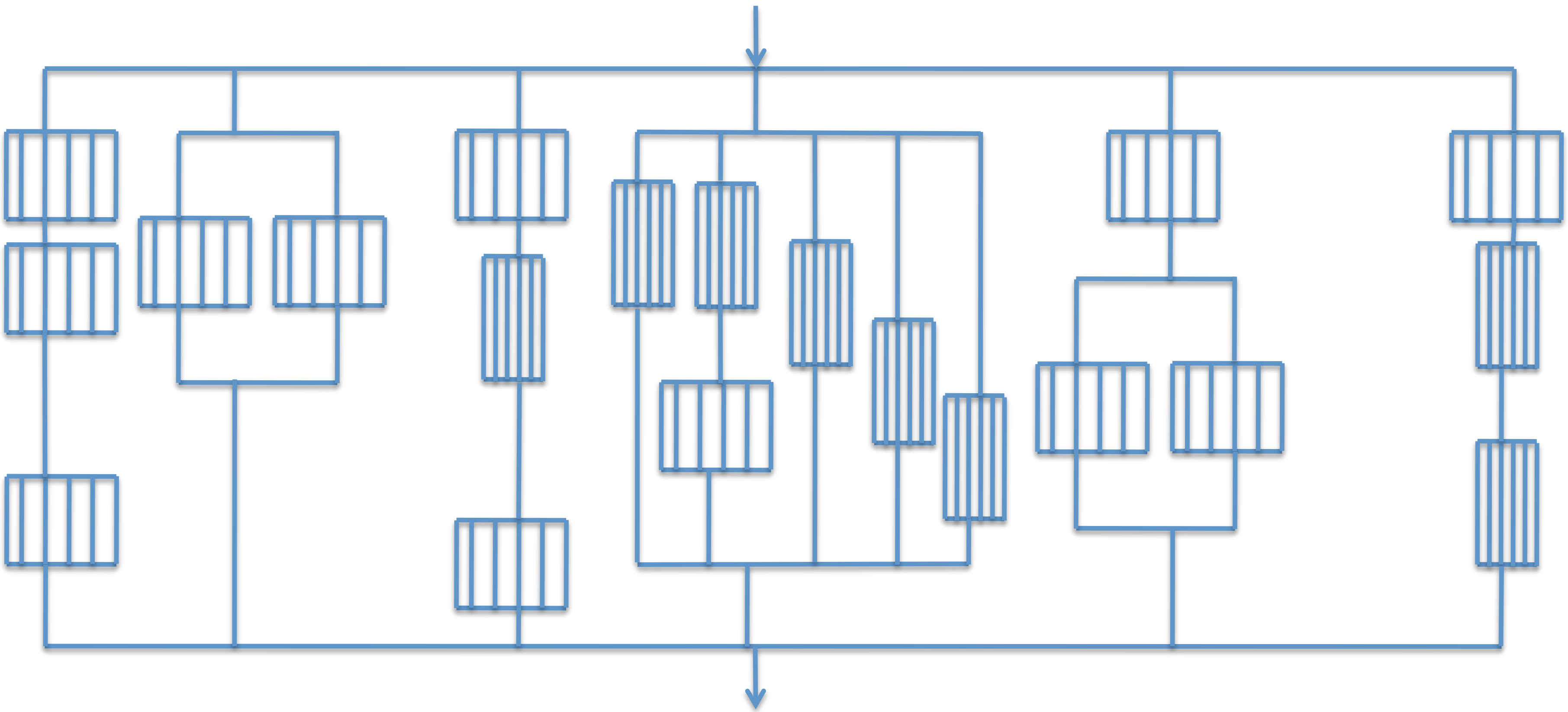
Data-Parallel Programming using SaC lecture 4

F21DP Distributed and Parallel Technology

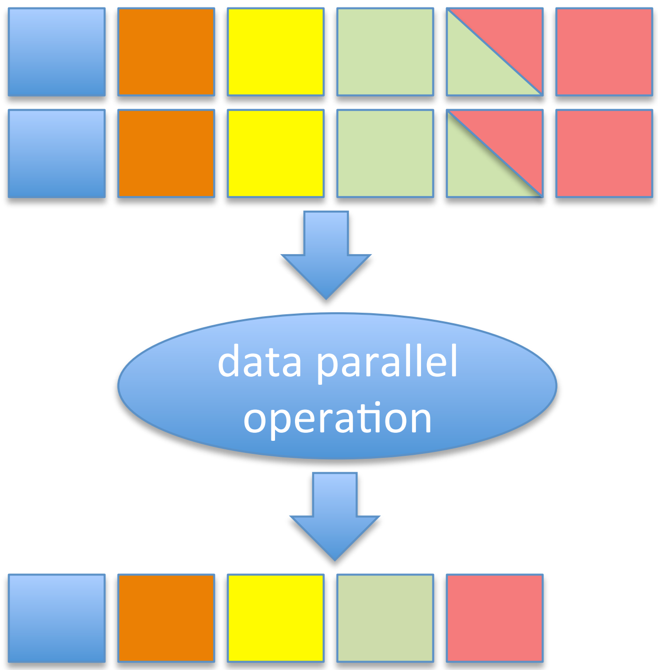
Sven-Bodo Scholz

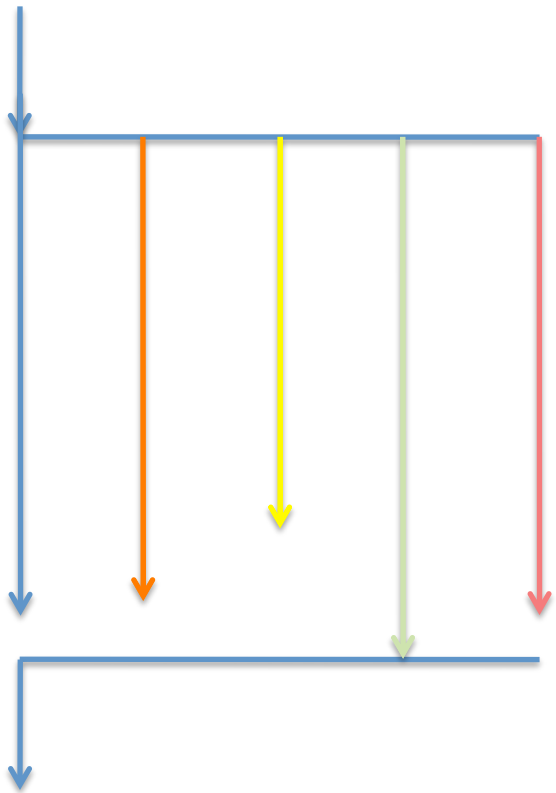
HERIOT WATT UNIVERSITY

Looking at Entire Programs:

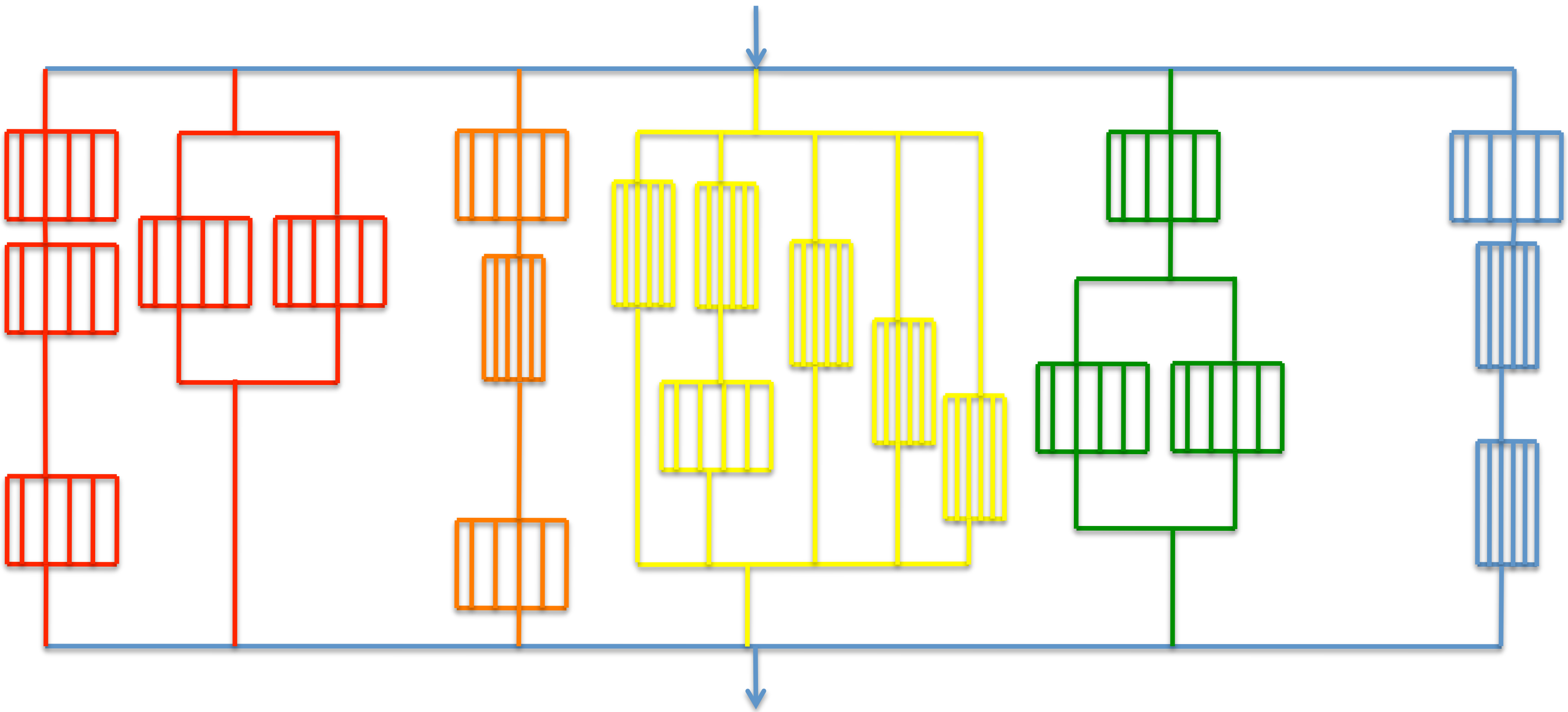


Multi-Threaded Execution on SMPs

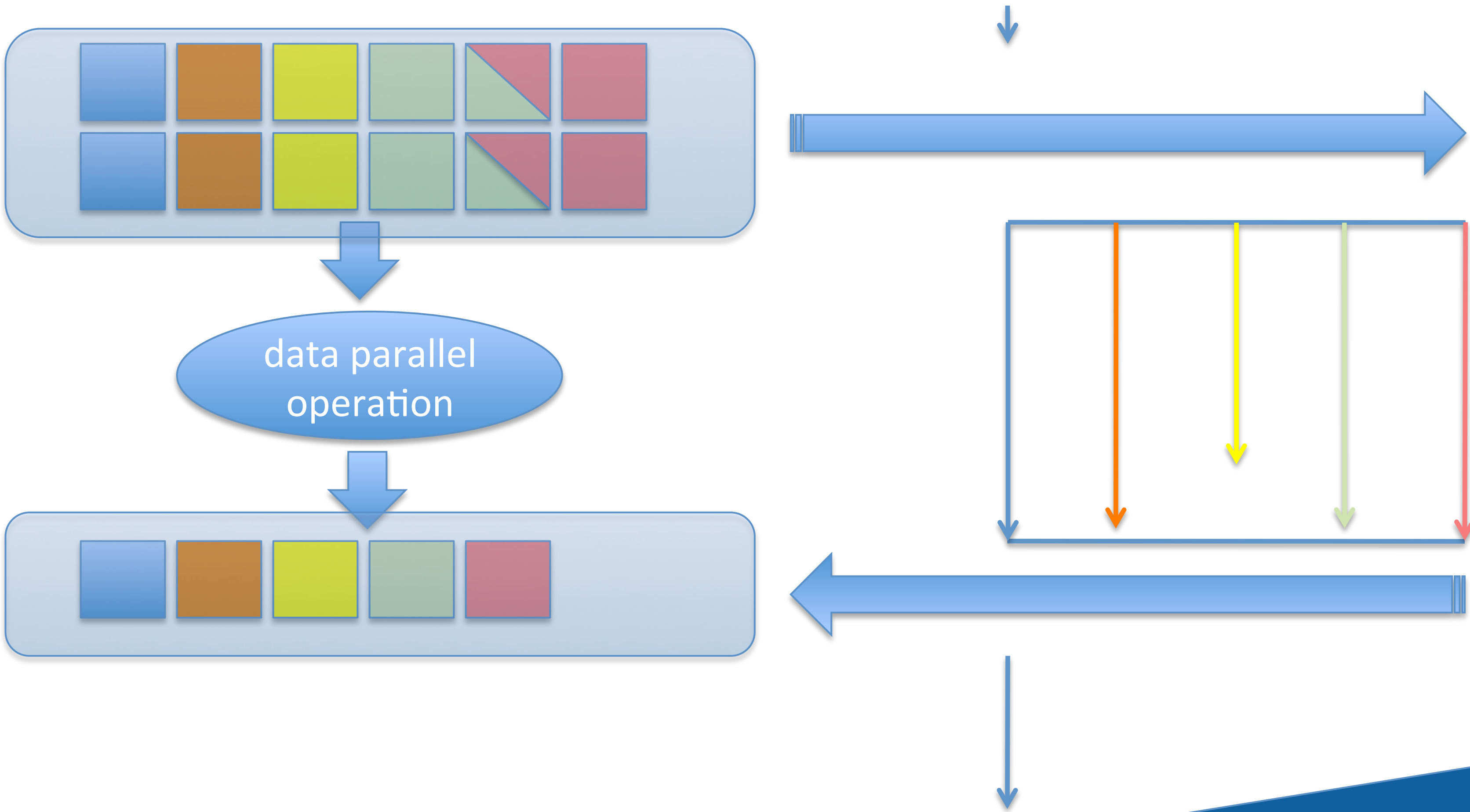




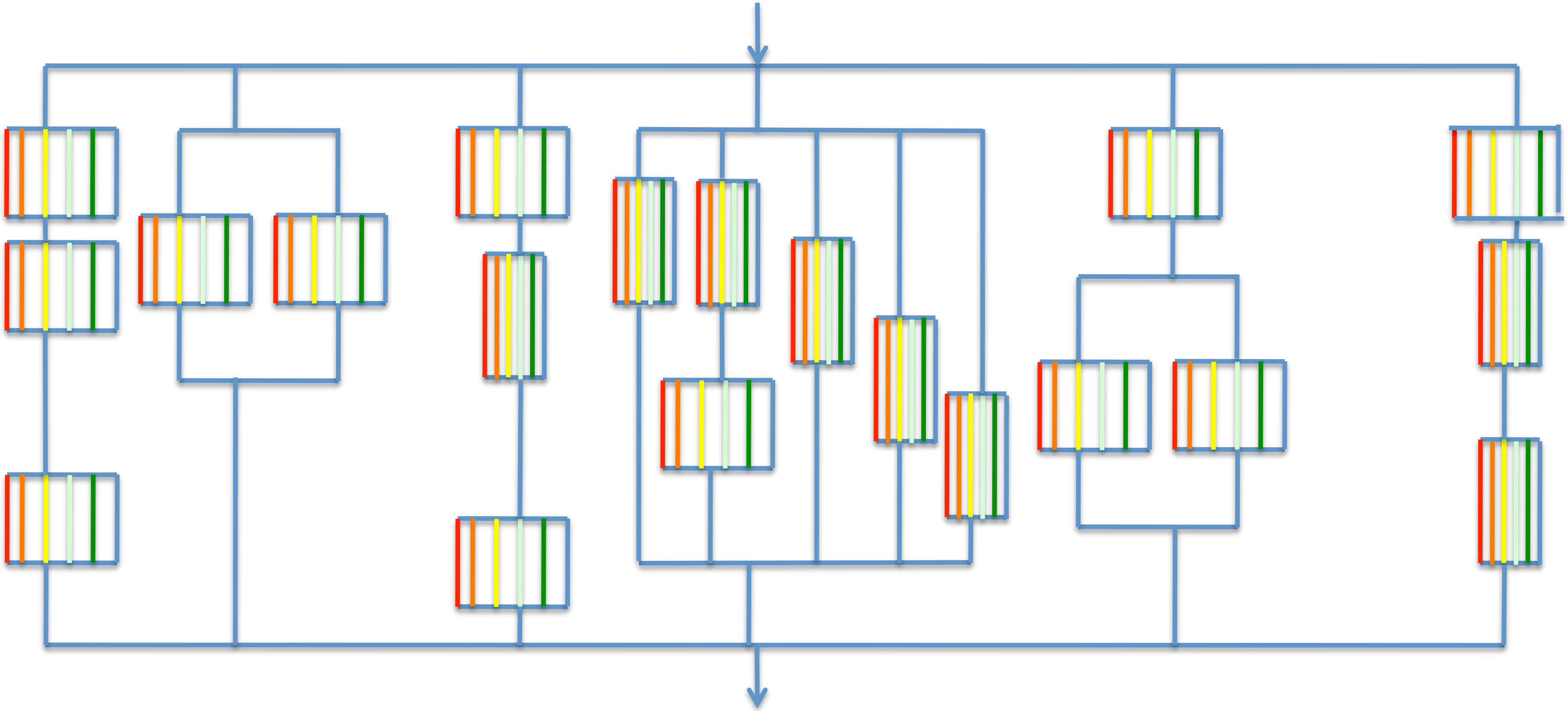
SMP Ideal: Coarse Grain!



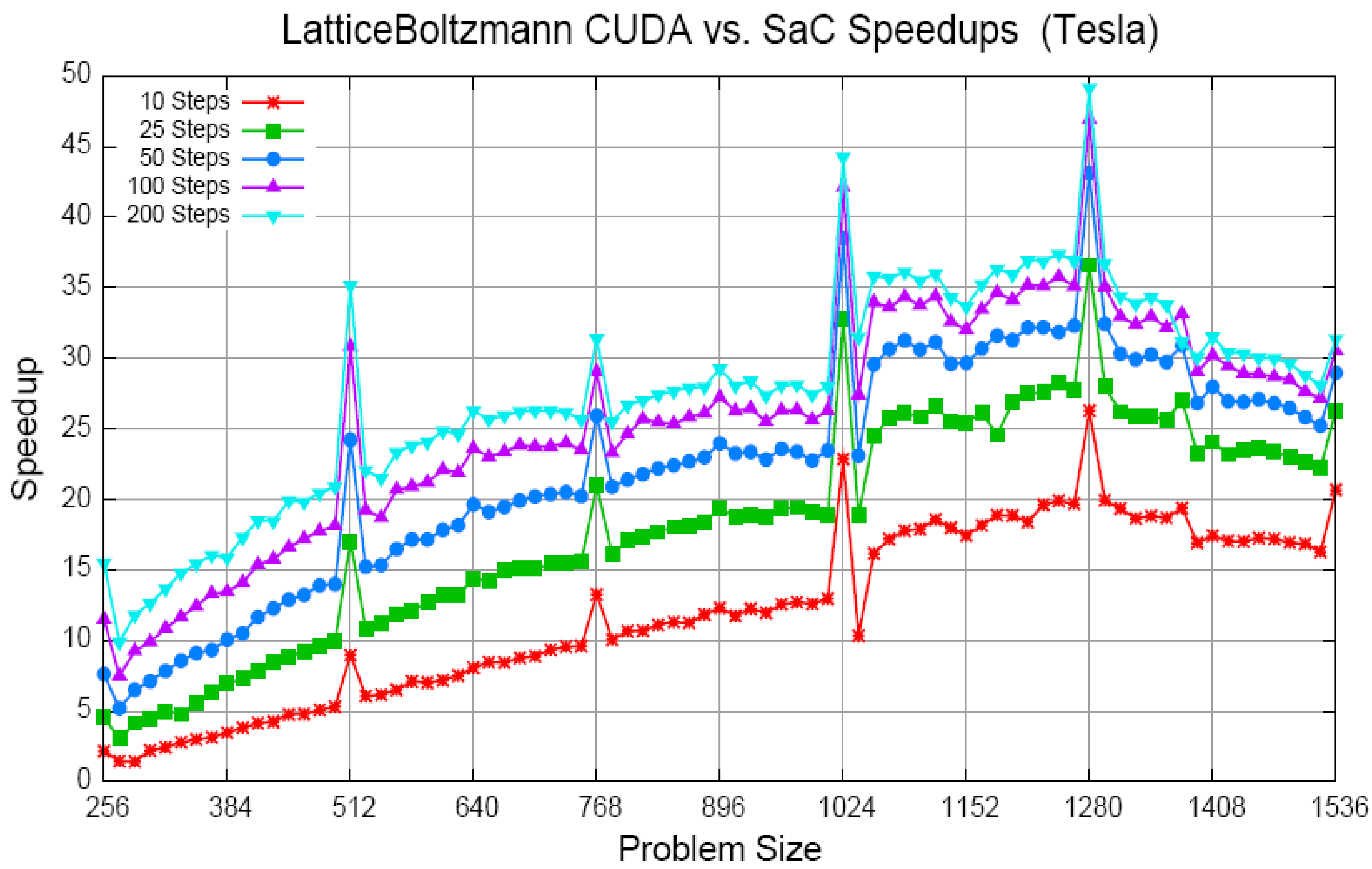
Multi-Threaded Execution on GPGPUs



GPGPU Ideal: Homogeneous Flat Coarse Grain



GPUs and synchronous memory transfers



GPGPU Naive Compilation

```
PX = { [i,j] -> PX[ [i,j] ] + sum( VY[ [.,i] ] * CX[ [i,j] ])  };
```

```
PX_d = host2dev( PX);
CX_d = host2dev( CX);
VY_d = host2dev( VY);

PX_d = { [i,j] -> PX_d[ [i,j] ] + sum( VY_d[ [.,i] ] * CX_d[ [i,j] ])  };

PX = dev2host( PX_d);
```

CUDA kernel

Hoisting memory transfers out of loops

```
for( i = 1; i< rep; i++) {
    PX_d = host2dev( PX);
    CX_d = host2dev( CX);
    VY_d = host2dev( VY);
    PX_d = { [i,j] -> PX_d[ [i,j] ] + sum( VY_d[ [.,i] ] * CX_d[ [i,j] ])  };
    PX = dev2host( PX_d);
}
```

Retaining Arrays on the Device

CX_d = { [i,j] -> ..... } ;

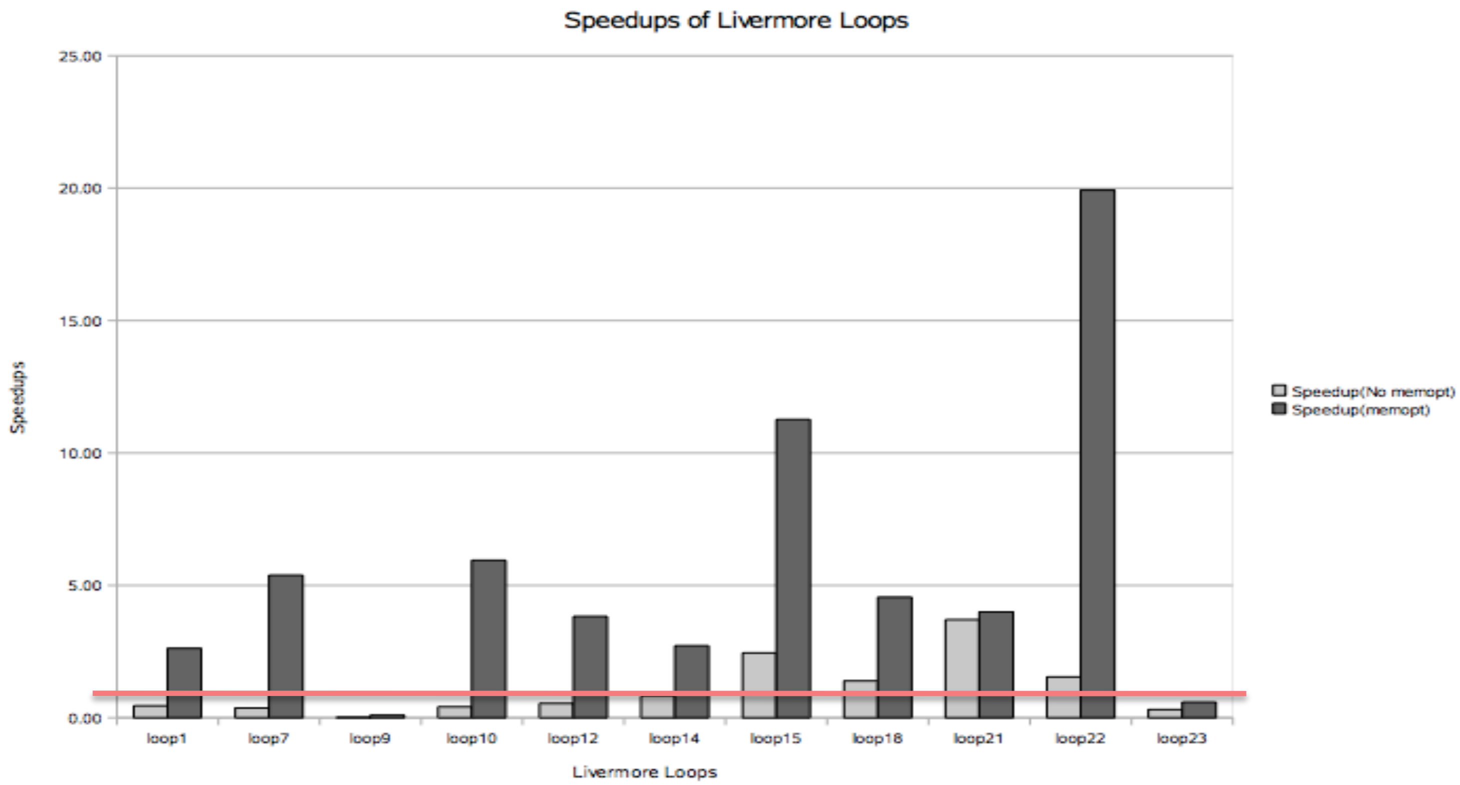
CX = dev2host(CX_d);

PX_d = host2dev(PX);

VY_d = host2dev(VY);

for(i = 1; i< rep; i++) {

The Impact of Redundant Memory Transfers



Challenge: Memory Management: What does the λ-calculus teach us?

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
   ... a..... a....b.....b....c...
}
```

How do we implement this? – the scalar case

f(a, b , c)


```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	implementation
read	read from stack
funcall	push copy on stack

How do we implement this? – the non-scalar case naive approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
    ... a..... a....b.....b....c...
}
```

operation	non-delayed copy
read	O(1) + free
update	O(1)
reuse	O(1)
funcall	O(1) / O(n) + malloc

How do we implement this? – the non-scalar case widely adopted approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
   ... a..... a....b.....b....c...
}
```

GC

operation	delayed copy + delayed GC
read	O(1)
update	O(n) + malloc
reuse	malloc
funcall	O(1)

How do we implement this? – the non-scalar case reference counting approach

conceptual copies

f(a, b , c)

```
f( a, b ,c )
{
   ... a..... a....b.....b....c...
}
```

operation	delayed copy + non-delayed GC
read	O(1) + DEC_RC_FREE
update	O(1) / O(n) + malloc
reuse	O(1) / malloc
funcall	O(1) + INC_RC

How do we implement this? – the non-scalar case a comparison of approaches

operation	non-delayed copy	delayed copy + delayed GC	delayed copy + non-delayed GC
read	O(1) + free	O(1)	O(1) + DEC_RC_FREE
update	O(1)	O(n) + malloc	O(1) / O(n) + malloc
reuse	O(1)	malloc	O(1) / malloc
funcall	O(1) / O(n) + malloc	O(1)	O(1) + INC_RC

Avoiding Reference Counting Operations

```
a = [1,2,3,4];

b = a[1];

c = f( a, 1);

d= a[2];

e = f( a, 2);
```

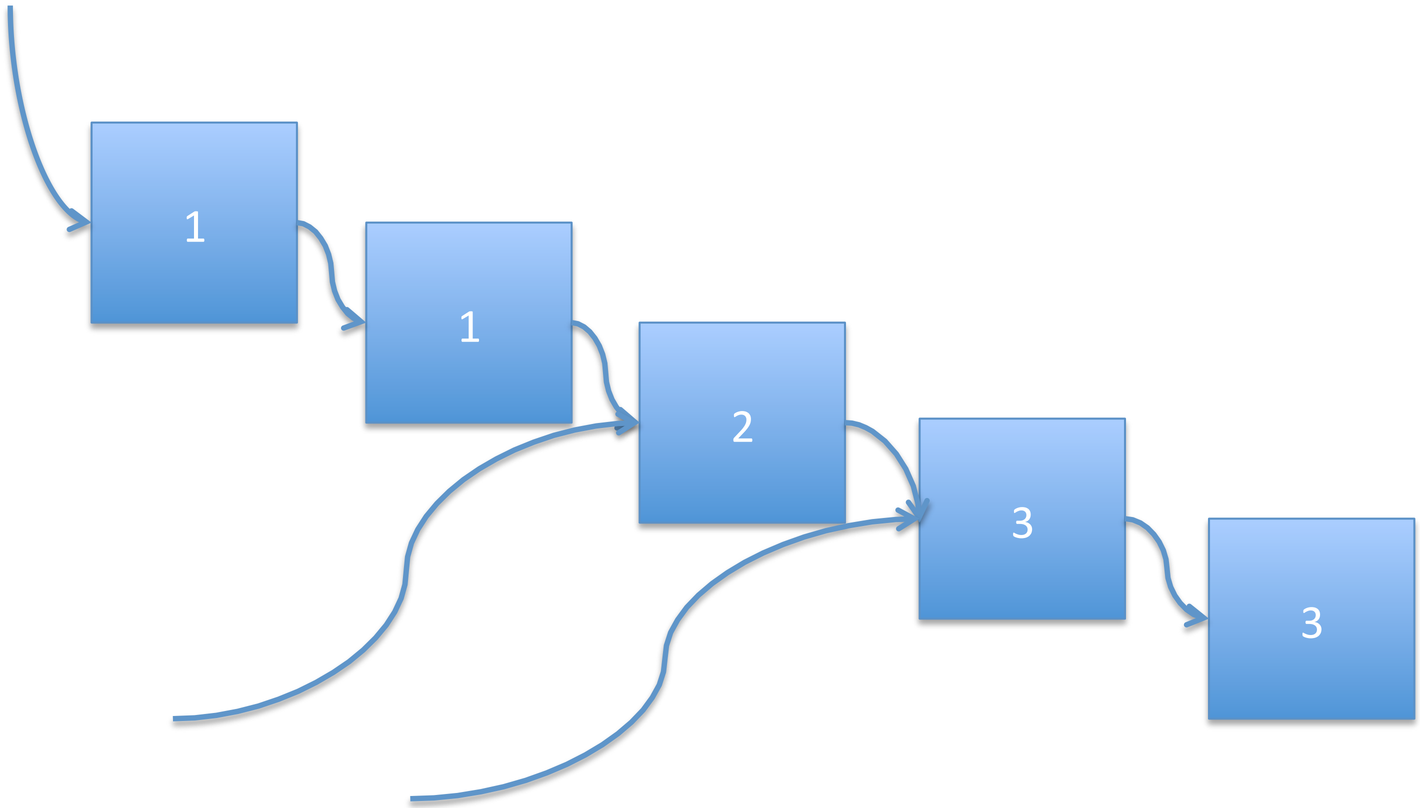
clearly, we can avoid RC here! (b = a[1];)

we would like to avoid RC here! (c = f(a, 1);)

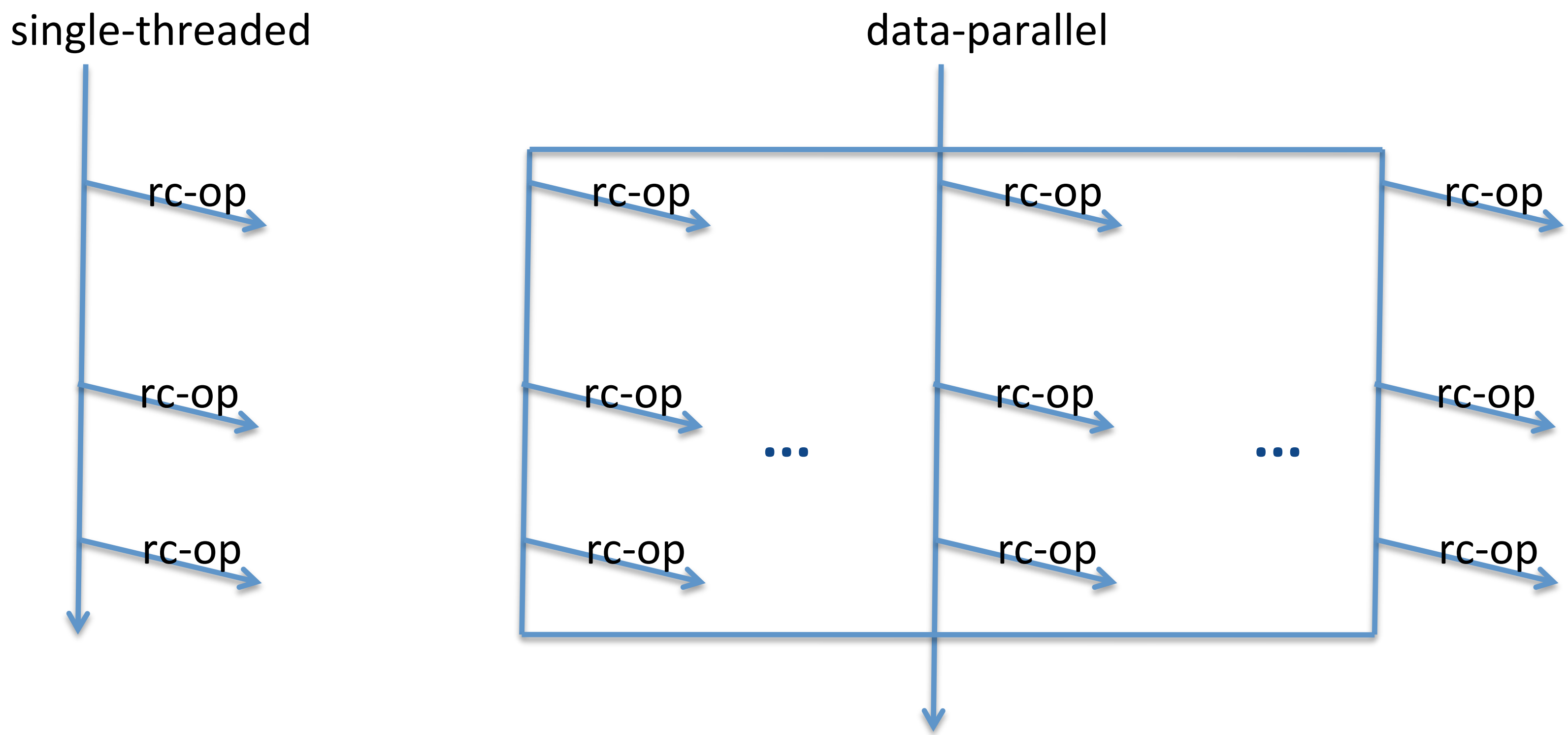
and here! (d= a[2];)

BUT, we cannot avoid RC here! (e = f(a, 2);)

NB: Why don't we have RC-world-domination?



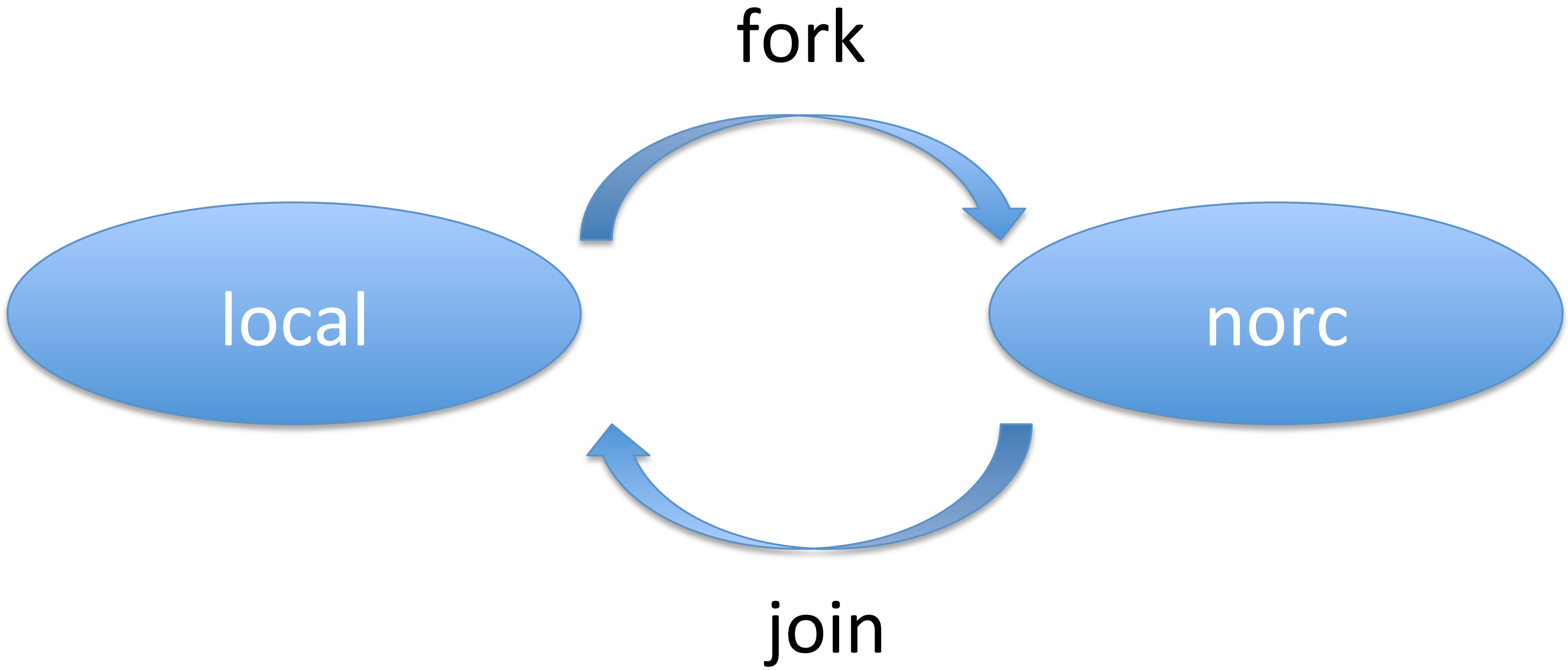
Going Multi-Core



local variables do not escape!
relatively free variables can only benefit from reuse in 1/n cases!

=> use thread-local heaps
=> inhibit rc-ops on rel-free vars

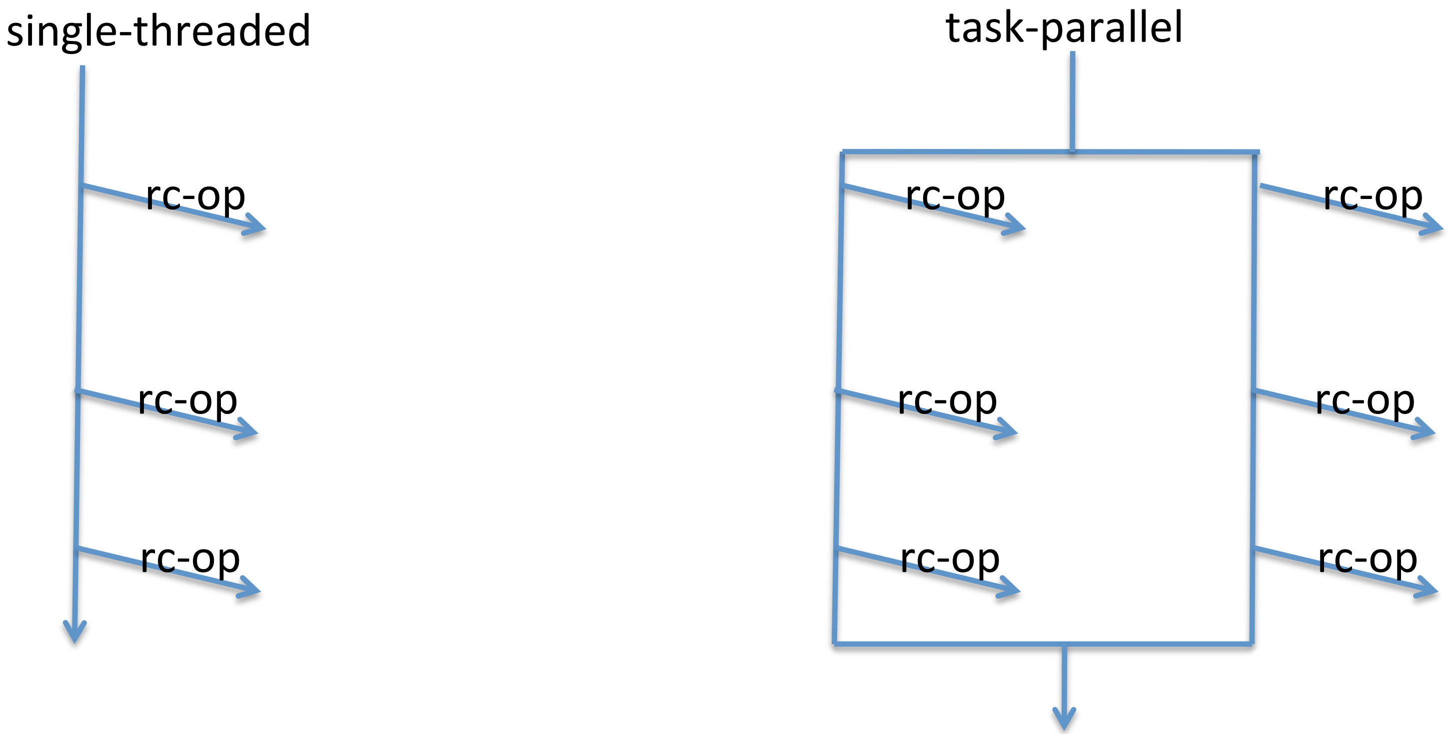
Bi-Modal RC:



Conclusions

- There are still many challenges ahead, e.g.
  - Non-array data structures
  - Arrays on clusters
  - Joining data and task parallelism
  - Better memory management
  - Application studies
- If you are interested in joining the team:
  - talk to me ☺

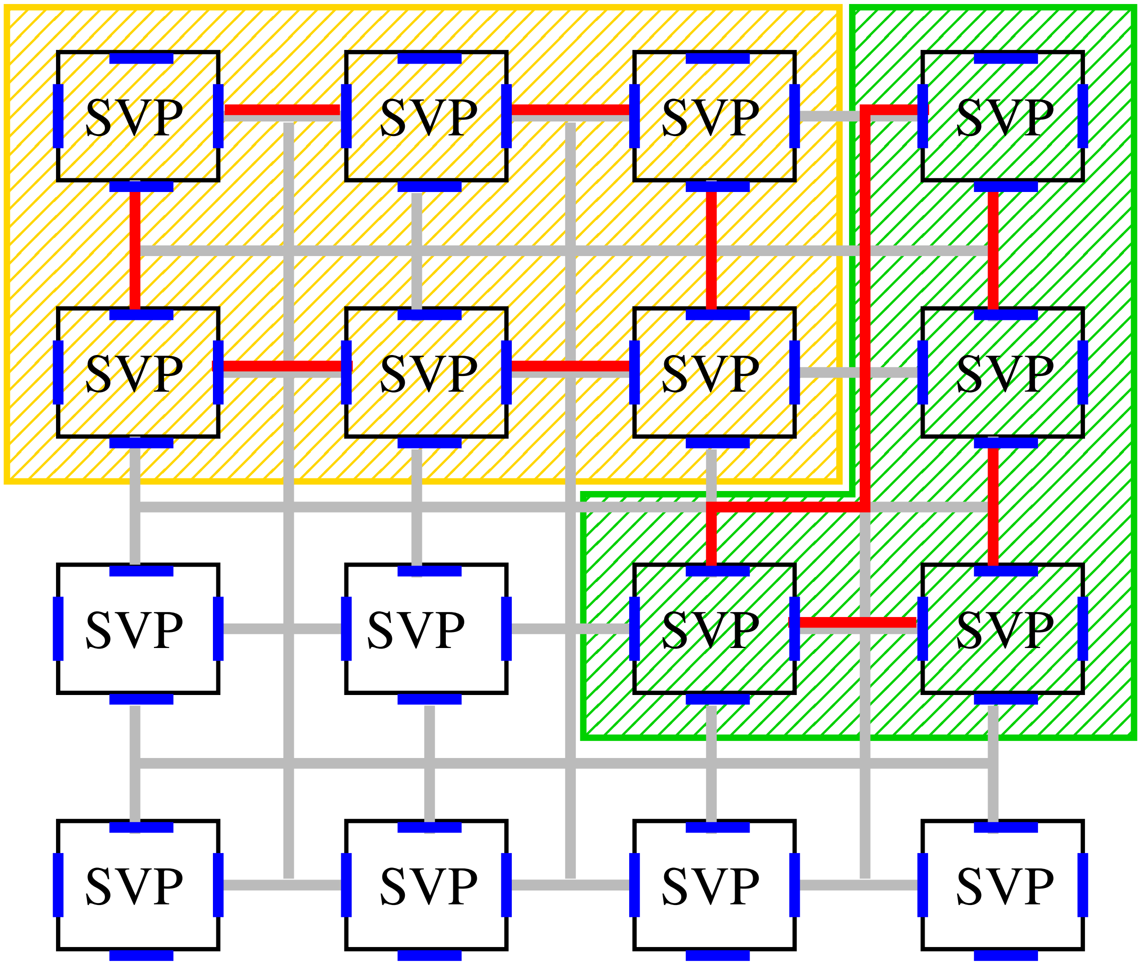
Going Multi-Core II



local variables **do** escape!
relatively free variables **can** benefit from reuse in 1/2 cases!

=> use locking....

Going Many-Core

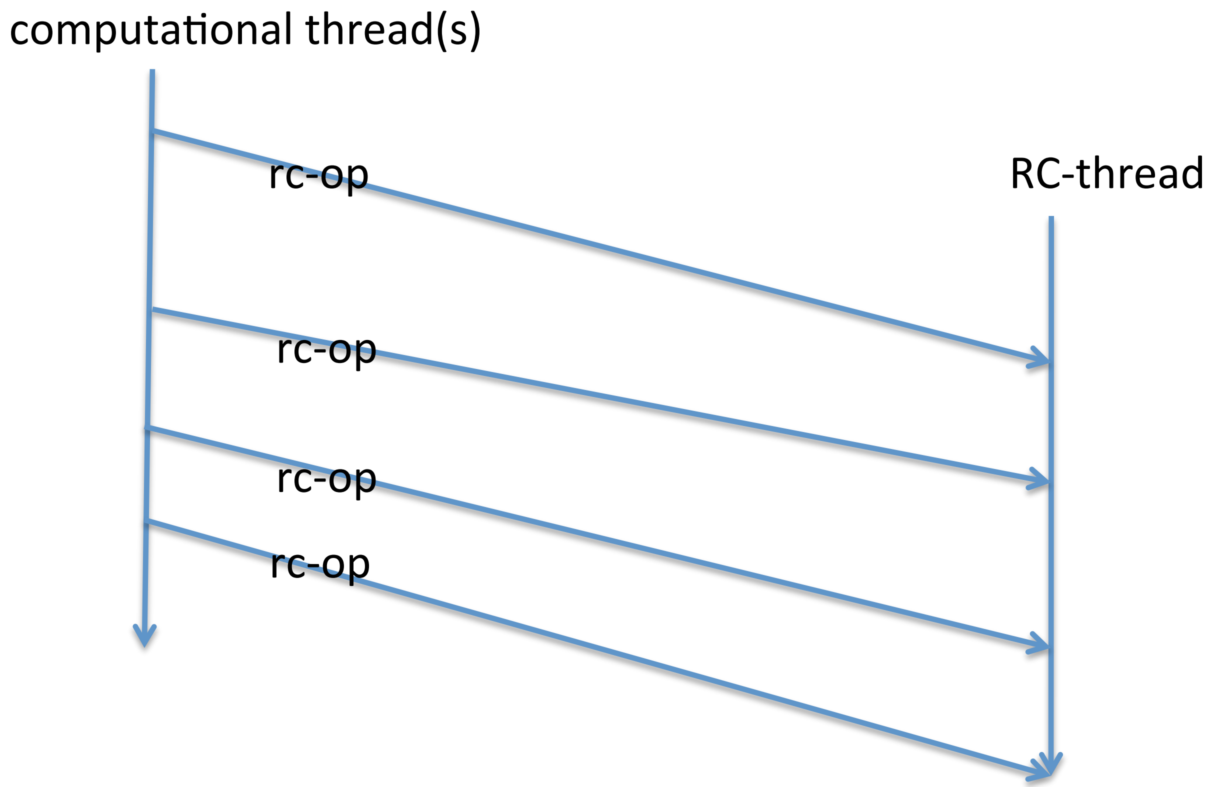


256 cores
500 threads in HW each

functional programmers paradise, no?!

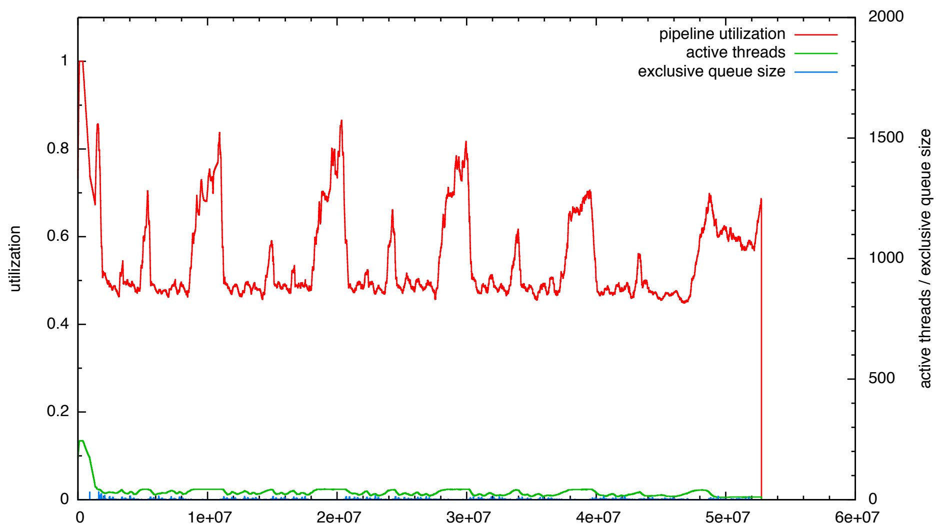
nested DP and TP parallelism

RC in Many-Core Times

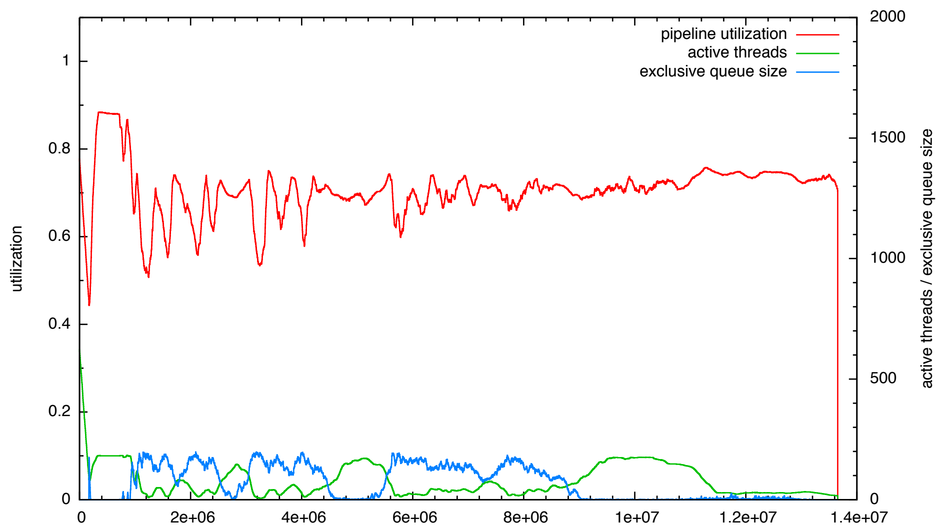


and here the runtimes

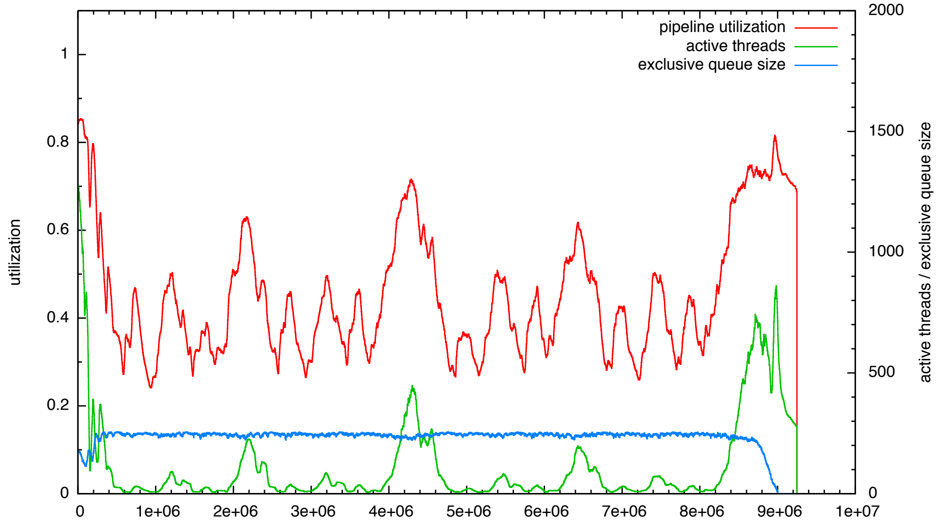
Execution on 1 cores



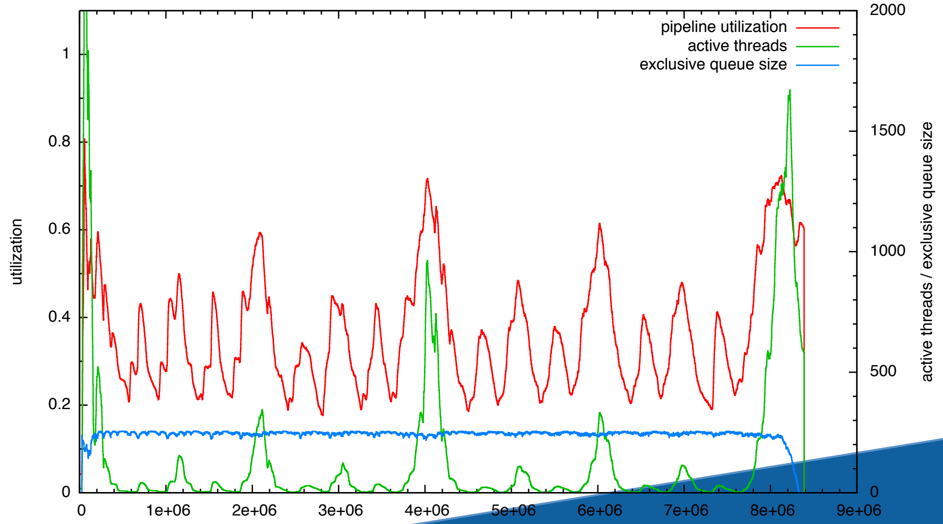
Execution on 4 cores



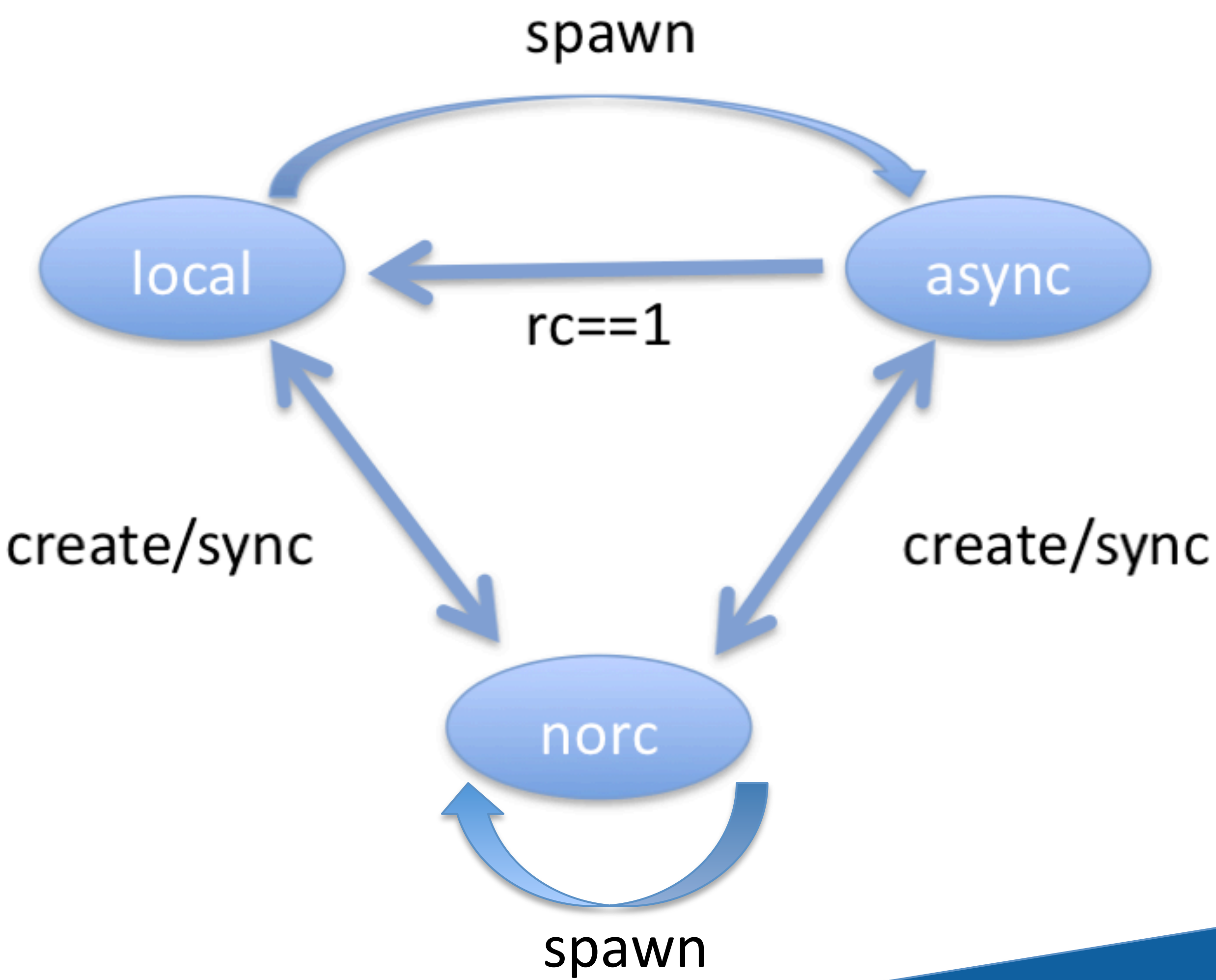
Execution on 16 cores



Execution on 64 cores

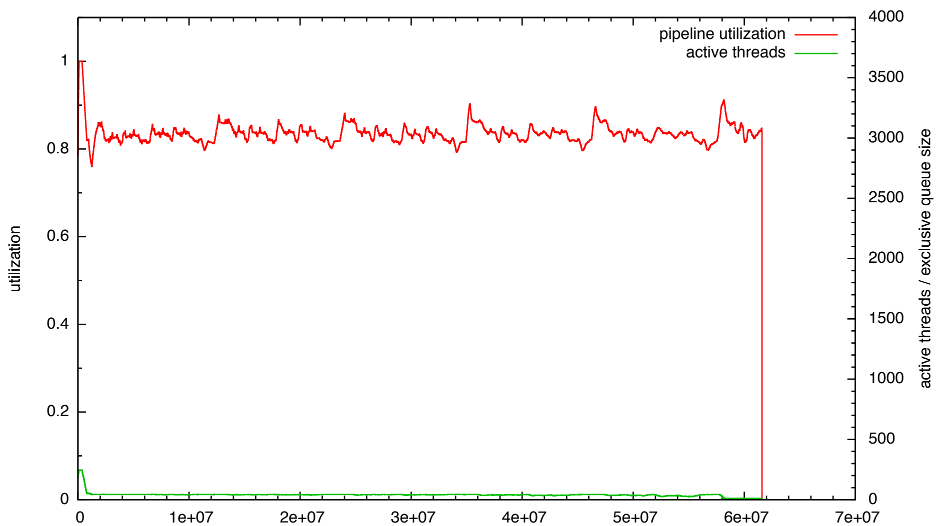


Multi-Modal RC:

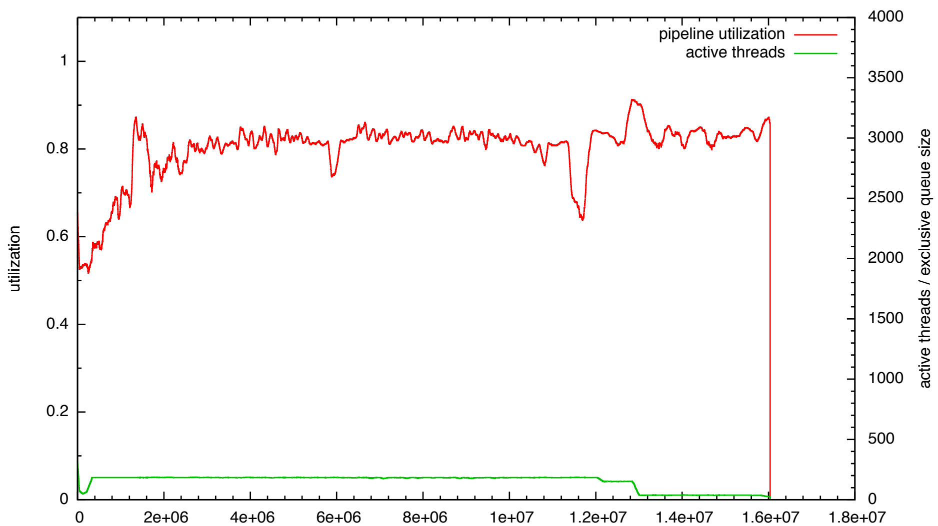


new runtimes:

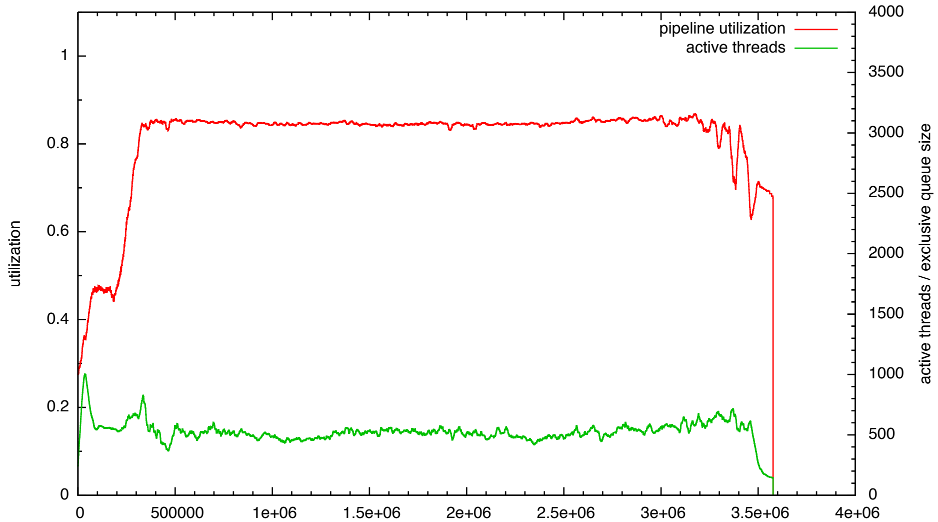
Execution on 1 cores



Execution on 4 cores



Execution on 16 cores



Execution on 64 cores