Google Maps

Données cartographiques ©2016 GeoBasis-DE/BKG (©2009), Google, Inst. Geogr. Nacional, Mapa GISrael, ORION-ME   200 km

Seminar of the GPG, July 13th, 2016, University of Glasgow

How to compute on a manycore processor

Bernard Goossens

Université de Perpignan Via Domitia, DALI-LIRMM

UNIVERSITÉ PERPIGNAN VIA DOMITIA

DALI
Digits Architectures et Logiciels Informatique

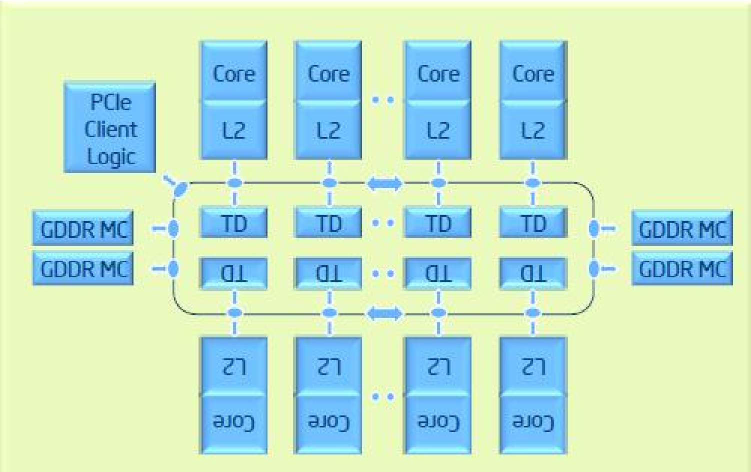
Outline.

1. Introduction : Parallelism in manycore processors.
2. A parallelizing hardware.
3. A parallel execution.
4. Implicitely parallel programs.
5. Conclusion.

Section 1

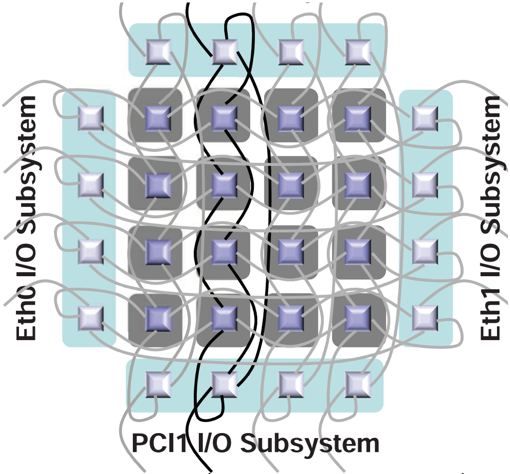
Introduction : Parallelism in manycore processors.

Xeon Phi.



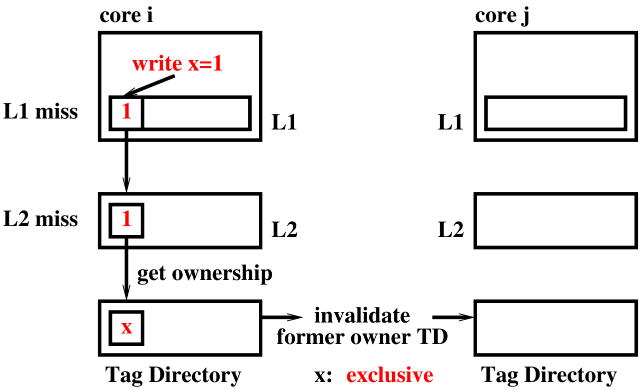
- **Shared memory** (61 cores, 30MB (0.5MB/c), L1+L2+tag+RAM), **ring**.
- **Not easily extensible**.

Kalray MPPA.



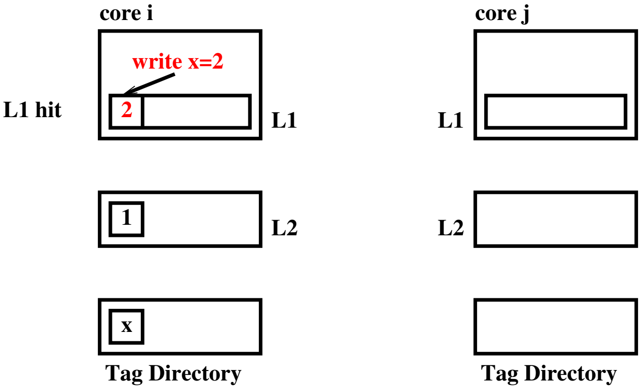
- **Interleaved tore** 2D, 256 cores (16 clusters of 16 cores).
- **Shared memory** (intra-cluster : private L1 + internal private RAM) and **Distributed memory** (inter-cluster).
- **Expensive** : 47MB of internal memory (0.18MB/c), **complex NoC**.

Shared memory coherence in the Xeon Phi : get first ownership.



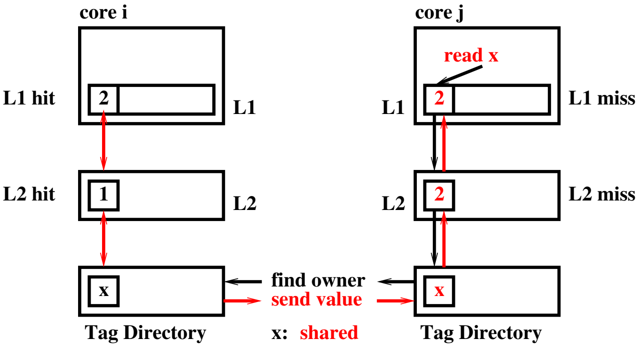
- **Write miss** : write through the local hierarchy (L1+L2+TD).
- **Broadcast ownership invalidation**.
- Tag x marked **Exclusive**.

Shared memory coherence in the Xeon Phi : exclusive data write.



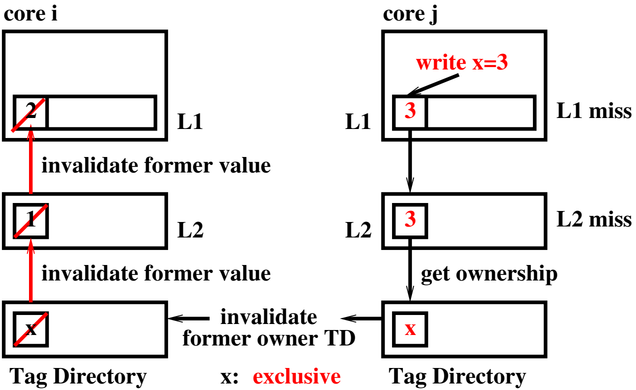
- **Write hit** : L1 local write.

Shared memory coherence in the Xeon Phi : share ownership.



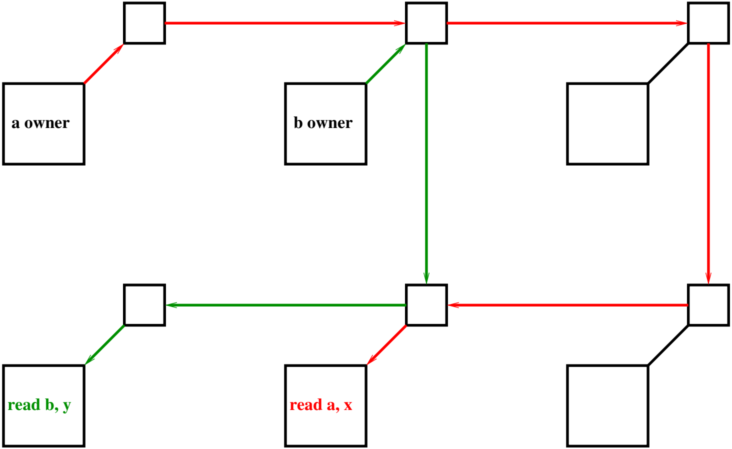
- **Read miss** : find owner in Tag Directory (search broadcast).
- First owner **provides value**.
- Update owners list (**update each owner TD**).
- Tag x marked **Shared**.

Shared memory coherence in the Xeon Phi : change ownership.



- **Write miss** : write through the local hierarchy (L1+L2+TD).
- **Broadcast ownership invalidation** to the set of owners.
- **Invalidate old value** : write up the local hierarchy (TD+L2+L1) of each owner.

Kalray MPPA distributed memory.



- **Address = core identifier**.
- **Complex routing** : Complex NoC.

Determinism and parallelism.

- OS threads : **non deterministic** parallelism.
- Computation total order : deterministic but **no parallelism**.
- Partial order respecting causalities : **deterministic parallelism**.

Determinism and parallelism.

- OS threads : **non deterministic** parallelism.
- Computation total order : deterministic but **no parallelism**.
- Partial order respecting causalities : **deterministic parallelism**.
- To have a deterministic computation, it is necessary and sufficient to :
  - Synchronize **only** the terms of causalities : the effect waits for the cause.
  - Communicate **only** from the cause to the effect.

What communication hardware is really needed ?

- Communicate from the cause to the effect : **unidirectional link**.
- If the cause is neighbouring the effect, a link **between neighbours**.

What communication hardware is really needed ?

- Communicate from the cause to the effect : **unidirectional link**.
- If the cause is neighbouring the effect, a link **between neighbours**.
- **Ring** interconnect and inter-neighbour communications : **simple, fast, extensible**.

What memorizing hardware is really needed ?

- Memorizing means **saving** an intermediary result **to reuse it** later.
- **On a single core**, memorizing is a key to **efficiency**.
- **On a manycore**, is it useful to memorize ?

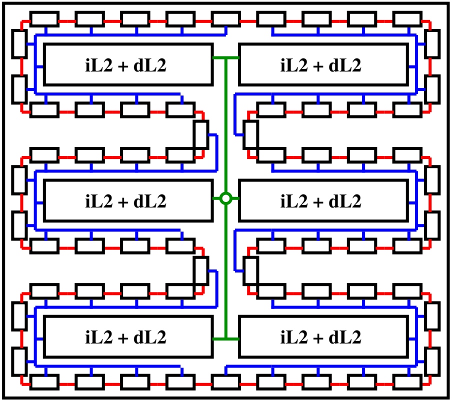
What memorizing hardware is really needed ?

- Memorizing means **saving** an intermediary result **to reuse it** later.
- **On a single core**, memorizing is a key to **efficiency**.
- **On a manycore**, is it useful to memorize ?
- On a manycore, **re-computing is more efficient** than memorizing and communicating.
- **Memory** for **code**, for **initial data** and for results (**I/O**).
- **Shared and hierarchized** memory : **spatial locality**, natural **coherency** when the writer is unique.
- If each computation consumes one or two data and produces one result the needed memory per core is small (**small caches : 0.0025MB/c** ; GPU GP100 = 0.002MB/c ; kilocore = 0.0016MB/c).

Section 2

A parallelizing hardware.

Our processor.

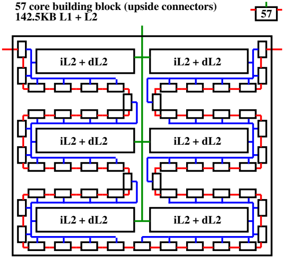


66 core processor chip

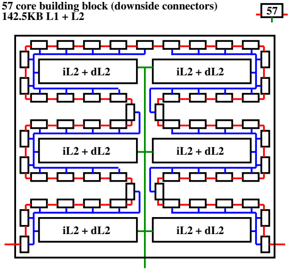
- A small low power processor : **66 cores**.
- **Unidirectional** ring.
- Hierarchized memory (L1+L2) : **code and I/O** (1.25KB L1/core, 1.25KB L2/core, 166KB L1+L2).

Extensible construct.

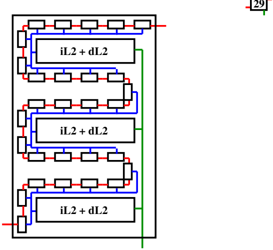
57 core building block (upside connectors)
142.5KB L1 + L2



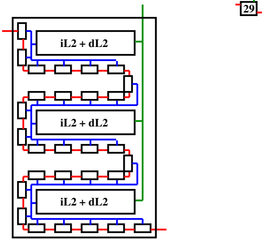
57 core building block (downside connectors)
142.5KB L1 + L2



29 core building block (leftside bottom connector)
72.5KB L1 + L2



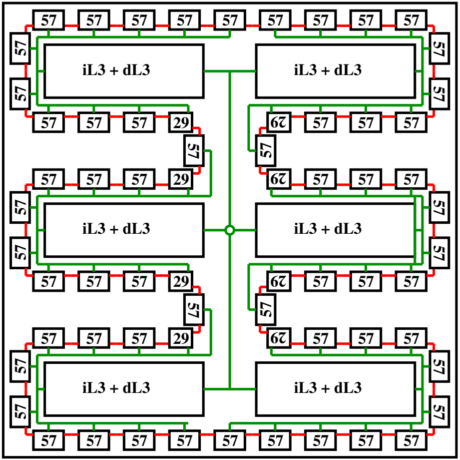
29 core building block (rightside bottom connector)
72.5KB L1 + L2



- **Uniform** ring connectors.

A 3538 core processor.

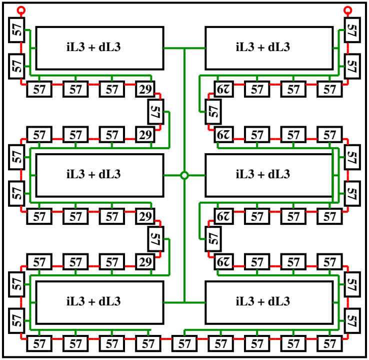
58*57 + 8*29 = 3538 core processor chip
L1+L2+L3 = 17.69MB



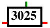
- Big manycore = **clusters on a ring**.

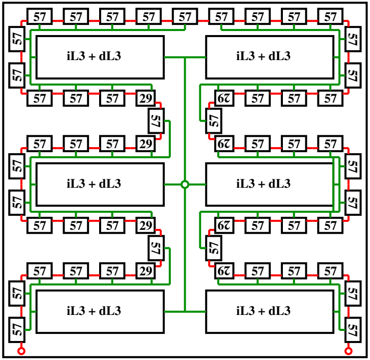
A four processor board, 12000 cores.

49*57 + 8*29 = 3025 core building block chip
upper edge connector
L1+L2+L3 = 15.125MB

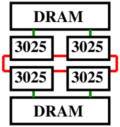


49*57 + 8*29 = 3025 core building block chip
lower edge connector
L1+L2+L3 = 15.125MB

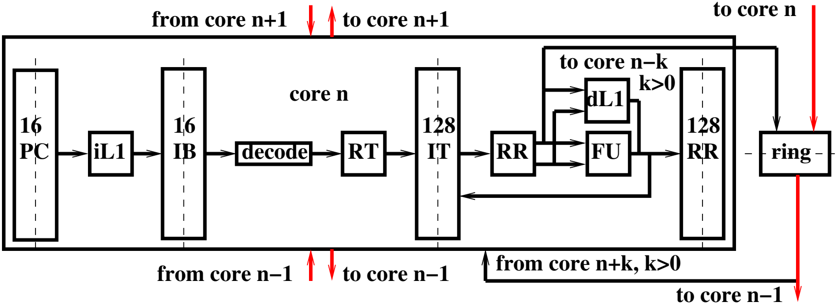




4 socket board
12100 cores
60.5MB caches



The core design.



Three stage pipeline core **1KB iL1: 64B/thread** **256B dL1: 16B/thread**

- **16 threads**, 1KB iL1, 0.25KB dL1 (multithreading to tolerate memory latency, computation organization to increase the locality).
- ISA : **register-register**, control and I/O.
- **Non speculative** execution (a branch suspends its thread).
- **Out-of-order** execution (renaming) : 1 IPC/core peak.
- Core n is linked to $n+1$, to $n-1$ and to $n-k, k>0$ through the **unidirectional ring**.

Section 3

A parallel execution.

A sum reduction.

```
#include <stdio.h>
inline int f(int i){return i;}
int sum(int i, int n){
 if (n==1) return f(i);
 if (n==2) return f(i)+f(i+1);
 return sum(i,n/2) + sum(i+n/2,n-n/2);
}
void main(){
 printf("s=%d\n",sum(0,10));
}
```

- On a classical hardware the execution is **sequential**.
- On a parallelizing hardware the execution is **parallel**.

A sum reduction in x86 assembly language.

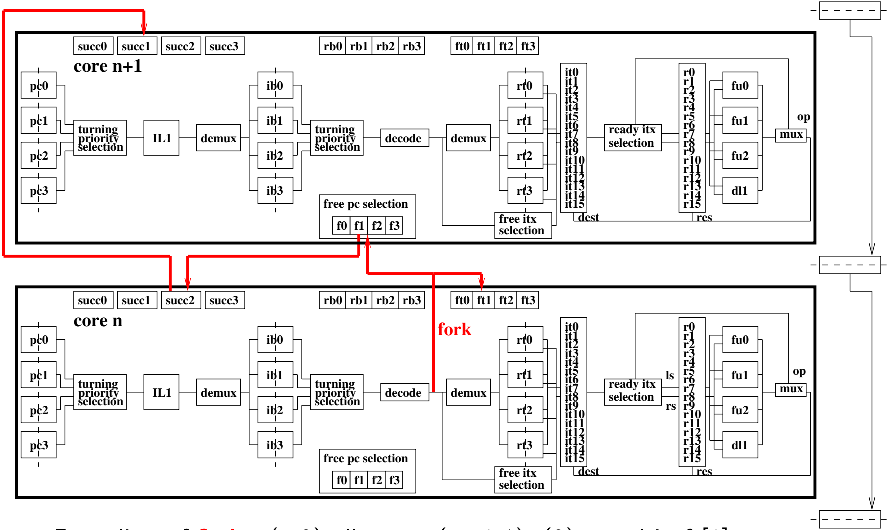
```
sum:  cmpq   $2, %rsi     ja     .L2    ;/*  if (n>2)  goto .L2      */
      movq   %rdi, %rax                 ;/*  rax  = f(i)             */
      subq   $1, %rsi     je     .L1    ;/*  if (n==1) goto .L1      */
      addq   $1, %rdi                   ;/*  rdi  = f(i+1)           */
      addq   %rdi, %rax                 ;/*  rax  = f(i)+f(i+1)      */
.L1:  retint                            ;/*  stop                    */
.L2:  movq   %rsi, %rbx   shrq   %rsi   ;/*  rbx  = n;  rsi = n/2    */
      fork   $3                         ;/*  start thread            */
      push   %rdi         push   %rsi   ;/*  send rdi, rsi           */
      push   %rbx                       ;/*  send rbx                */
      call   sum                        ;/*  rax  = sum(i,n/2)       */
      pop    %rbx                       ;/*  receive rbx             */
      pop    %rsi         pop    %rdi   ;/*  receive rsi, rdi        */
      movq   %rax, %rcx                 ;/*  rcx  = rax              */
      addq   %rsi, %rdi                 ;/*  rdi  = i + n/2          */
      subq   %rsi, %rbx                 ;/*  rbx  = n - n/2          */
      movq   %rbx, %rsi                 ;/*  n    = n - n/2          */
      fork   $1                         ;/*  start thread            */
      push   %rcx                       ;/*  send rcx                */
      call   sum                        ;/*  rax  = sum(i+n/2,n-n/2) */
      pop    %rcx                       ;/*  receive rcx             */
      addq   %rcx, %rax                 ;/*  rax += sum(i,n/2)       */
      retint                            ;/*  stop                    */
```

- New instructions : **fork**, **retint**.
- New semantic : **call**, **push**, **pop**.
- Reg-reg instructions (**no ld/st**).

Forking protocol.

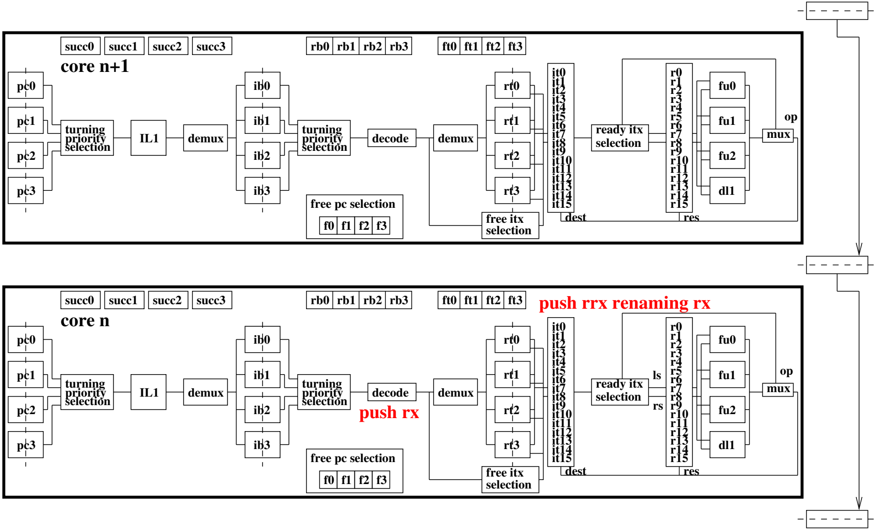
- Instruction **fork k** : allocates a thread in core **n+1**, *k* values to be sent.
- Instruction **push r** : sends register *r* to core **n+1**.
- Instruction **call** : sends next PC to core **n+1**.
- Instruction **pop r** : receives into register *r* from core **n-1**.
- Instruction **ret** : ends the current thread.
- Instruction **retint** : ends the current thread and sends register **rax** to the core holding the succeeding thread (core **n-k, k**$\geq$**-1**).

Fork instruction.



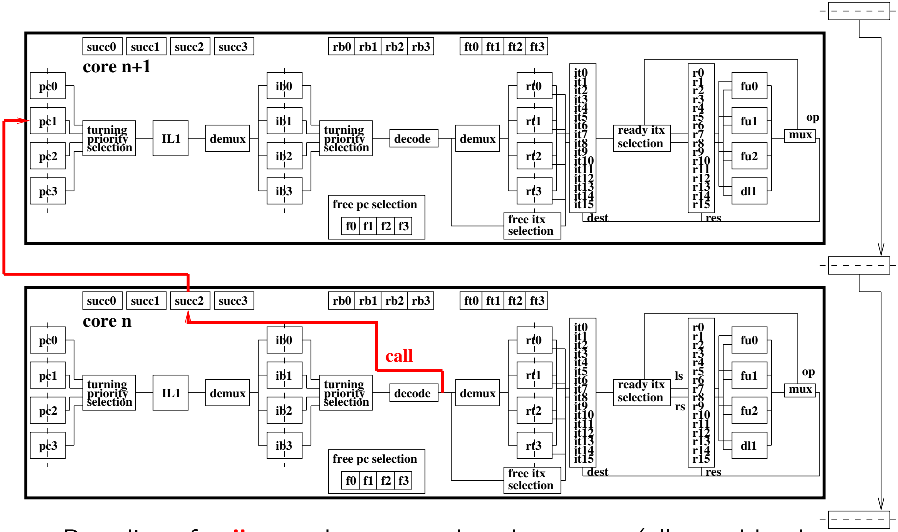
- Decoding of **fork** : (n,2) allocates (n+1,1), (2) saved in ft[1].
- The **next thread link** is updated (the new thread is inserted between the creator and its former successor).

Push instruction decoding.



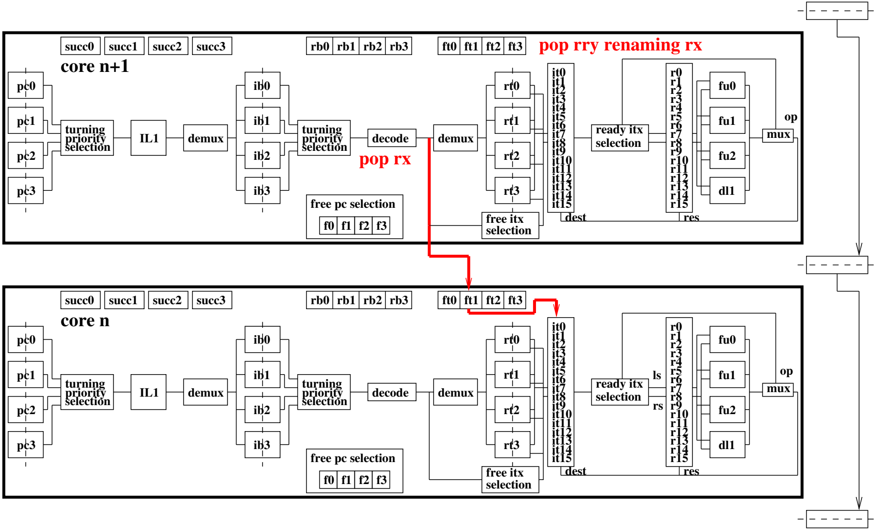
- Decoding of **push** : numbered ; wait for **rrx** value.
- Decoding of **push** : wait for **pop rx** signal.

Call instruction.



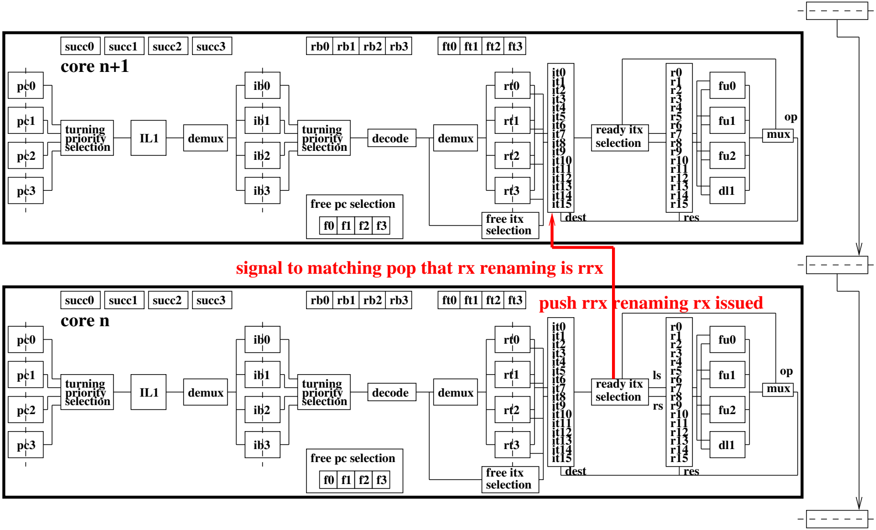
- Decoding of **call** : get the current thread successor (allocated by the preceding fork).
- Decoding of **call** : send the return PC to the created thread.

Pop instruction decoding.



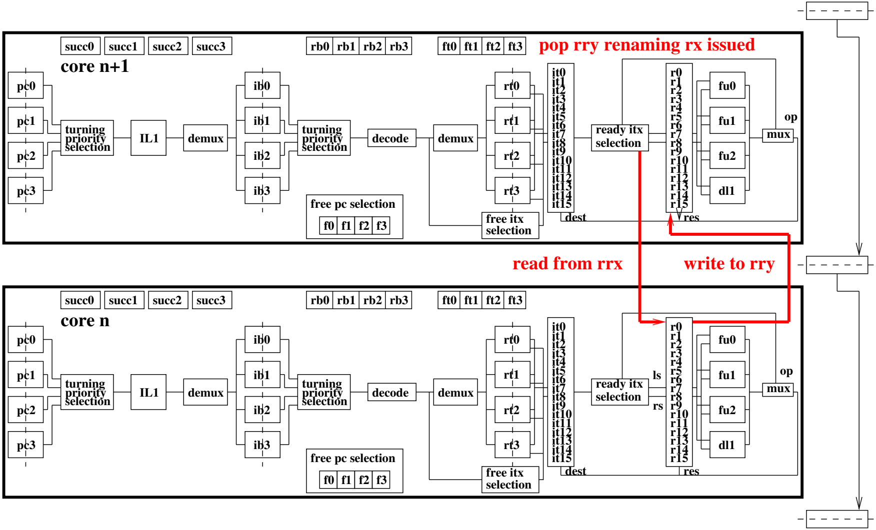
- Decoding of **pop** : send **pop rx** signal to the creating thread.
- Decoding of **pop** : wait for **push rx** signal, same number.

Push instruction execution.



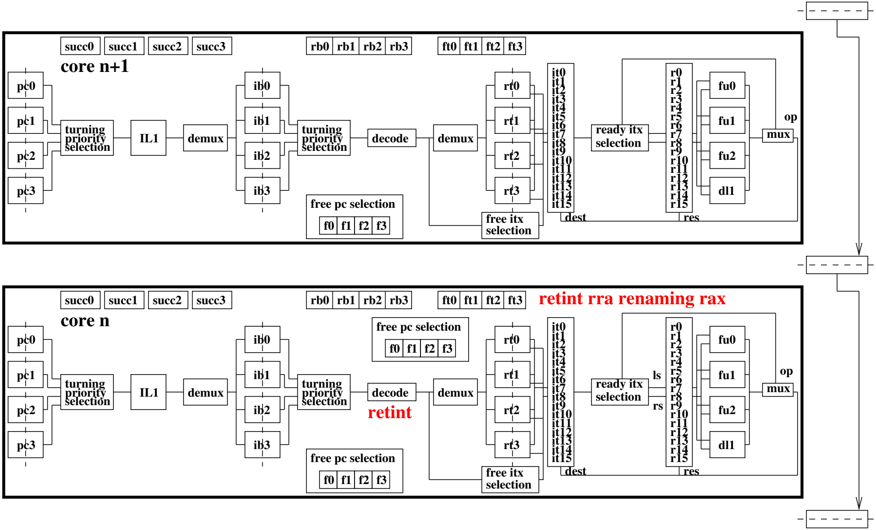
- **Push** execution : when **rrx** is full and **pop rx** has been renamed.
- **Push** execution : sends **rrx** name to **pop rx**.

Pop instruction execution.



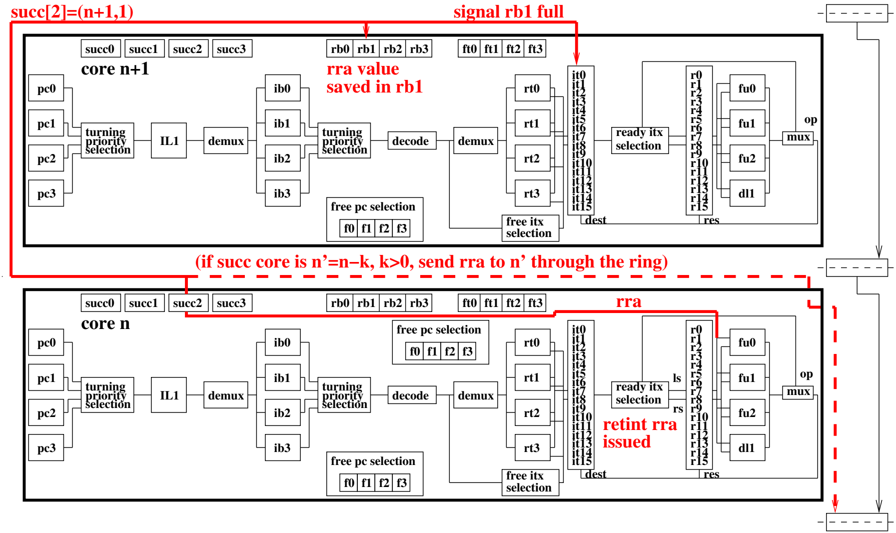
- **Pop** execution : when **push** has been issued.
- **Pop** execution : **rrx** read (special read port), **rry** write.

Retint instruction decoding.



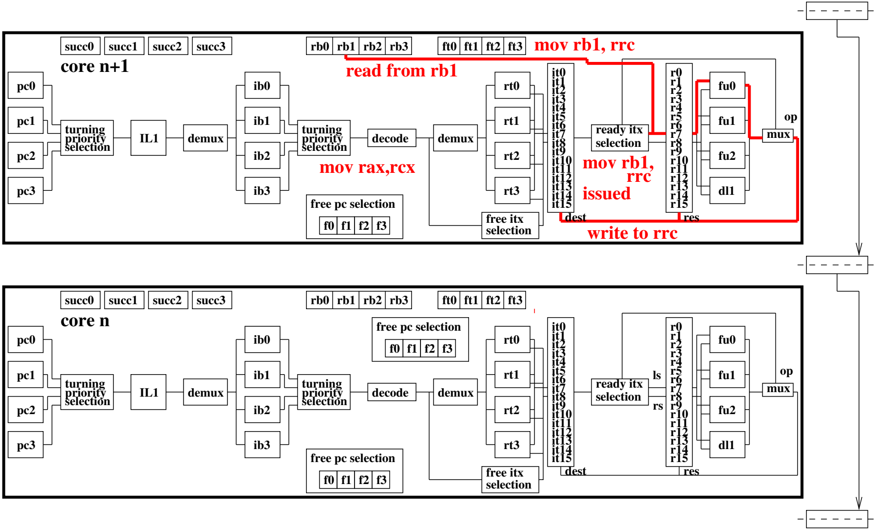
- Decoding of **retint** : **rax** renamed **rra**.
- Decoding of **retint** : wait for **rra** value.

Retint instruction execution.



- **Retint** execution : read from **rra**.
- **Retint** execution : write to **rb** buffer of the successor thread (rb1).

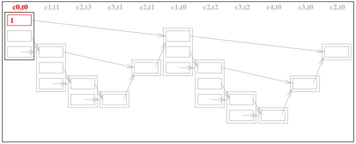
Result consuming instruction execution.

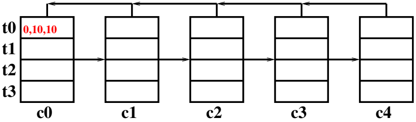


- Renaming of **mov rax, rcx** : **rax** renamed **rb1**, **rcx** renamed **rrc**.
- **mov rax, rcx** is issued when **rb1** is full : read from **rb1**, write to **rrc**.

Sum reduction execution.

```
1   sum:    cmpq $2, %rsi
            ja .L2

2           movq %rdi, %rax
            subq $1, %rsi
            je .L1

3           addq $1, %rdi
            addq %rdi, %rax

4   .L1:    retint

5   .L2:    movq %rsi, %rbx
            shrq %rsi
            fork $3
            push %rdi
            push %rsi
            push %rbx
            call sum

6           pop %rbx
            pop %rsi
            pop %rdi
            movq %rax, %rcx
            addq %rsi, %rdi
            subq %rsi, %rbx
            movq %rbx, %rsi
            fork $1
            push %rcx
            call sum

7           pop %rcx
            addq %rcx, %rax
            retint
```

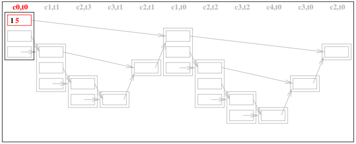


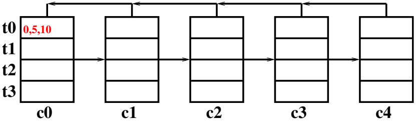


- Code cut into **basic blocks**.
- **One active thread** on one core (c0, t0).
- rdi=0 (**i=0**), rsi=10 (**n=N**), rbx=10 (oldn).

Sum reduction execution.

1
```
sum:    cmpq $2, %rsi
        ja .L2
```
2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```
3
```
        addq $1, %rdi
        addq %rdi, %rax
```
4
```
.L1:    retint
```
5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```





6
```
        pop %rbx
        pop %rsi
        pop %rdi
        movq %rax, %rcx
        addq %rsi, %rdi
        subq %rsi, %rbx
        movq %rbx, %rsi
        fork $1
        push %rcx
        call sum
```
7
```
        pop %rcx
        addq %rcx, %rax
        retint
```

	c0	c1	c2	c3	c4
t0	1 5				
t1					
t2					
t3					

- Block 5 read (**at least 7 cycles**).
- **One active thread** on one core (c0, t0).
- Rdi=0 (**i=0**), rsi=5 (**n=N/2**), rbx=10 (oldn=N).

Sum reduction execution.

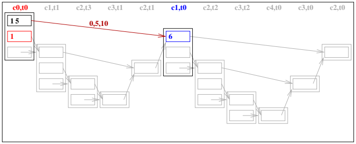
1
```
sum:    cmpq $2, %rsi
        ja .L2
```

2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```

3
```
        addq $1, %rdi
        addq %rdi, %rax
```

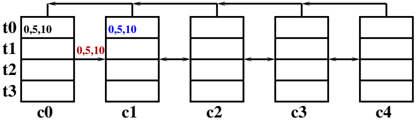
4
```
.L1:    retint
```

5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```



6
```
        pop %rbx
        pop %rsi
        pop %rdi
        movq %rax, %rcx
        addq %rsi, %rdi
        subq %rsi, %rbx
        movq %rbx, %rsi
        fork $1
        push %rcx
        call sum
```

7
```
        pop %rcx
        addq %rcx, %rax
        retint
```

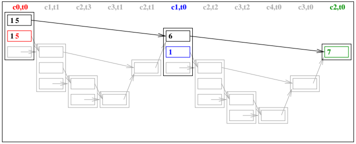


	c0	c1	c2	c3	c4
t0	1 5 1	6			
t1					
t2					
t3					

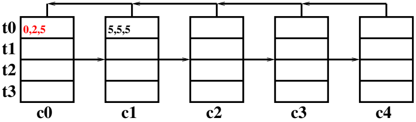
- **Two active threads** on two cores (c0, t0 : block 1 and c1, t0 : block 6).
- Transmission of the **continuation context** (push : send, pop : receive).
- Construction of the **threads tree**.

Sum reduction execution.

```
1  sum:   cmpq $2, %rsi
          ja .L2

2         movq %rdi, %rax
          subq $1, %rsi
          je .L1

3         addq $1, %rdi
          addq %rdi, %rax

4  .L1:   retint

5  .L2:   movq %rsi, %rbx
          shrq %rsi
          fork $3
          push %rdi
          push %rsi
          push %rbx
          call sum
```




```
6         pop %rbx
          pop %rsi
          pop %rdi
          movq %rax, %rcx
          addq %rsi, %rdi
          subq %rsi, %rbx
          movq %rbx, %rsi
          fork $1
          push %rcx
          call sum

7         pop %rcx
          addq %rcx, %rax
          retint
```



	c0	c1	c2	c3	c4
t0	1 5 1 5	6 1	7		
t1					
t2					
t3					

- **Three active threads** on three cores (c0, t0 : b5 ; c1, t0 : b1 ; c2, t0 : b7).
- (c0, t0) –> (c1, t0) **–> (c2, t0)**.
- **Waiting** instructions : (c1, t0, **b6**), **rcx=rax** ; (c2, t0, **b7**), **rax+=rcx**.

Sum reduction execution.

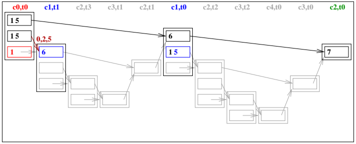
1
```
sum:    cmpq $2, %rsi
        ja .L2
```

2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```

3
```
        addq $1, %rdi
        addq %rdi, %rax
```

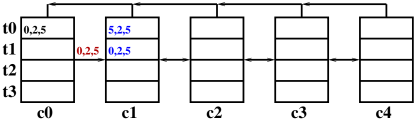
4
```
.L1:    retint
```

5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```



6
```
        pop %rbx
        pop %rsi
        pop %rdi
        movq %rax, %rcx
        addq %rsi, %rdi
        subq %rsi, %rbx
        movq %rbx, %rsi
        fork $1
        push %rcx
        call sum
```

7
```
        pop %rcx
        addq %rcx, %rax
        retint
```

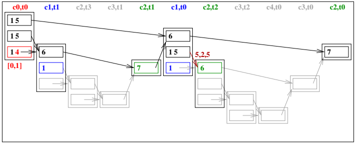


	c0	c1	c2	c3	c4
t0	1 5 1 5 1	6 1 5	7		
t1		6			
t2					
t3					

- **4 threads** on 3 cores (c0, t0, b1), (c1, t0, b5), (c2, t0, b7), (c1, t1, b6).
- (c0, t0) –> **(c1, t1)** –> (c1, t0) –> (c2, t0).
- Transmission of the **continuation context** (rdi, rsi, rbx : c0 –> c1).

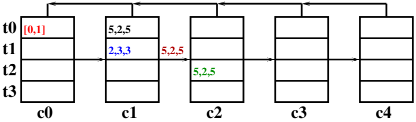
Sum reduction execution.

1
```
sum:    cmpq $2, %rsi
        ja .L2
```
2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```
3
```
        addq $1, %rdi
        addq %rdi, %rax
```
4
```
.L1:    retint
```
5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```




```
        pop %rbx
        pop %rsi
        pop %rdi
        movq %rax, %rcx
        addq %rsi, %rdi
        subq %rsi, %rbx
        movq %rbx, %rsi
        fork $1
        push %rcx
        call sum
```
6

```
        pop %rcx
        addq %rcx, %rax
        retint
```
7



	c0	c1	c2	c3	c4
t0	1 5 1 5 1 2 3 4	6 1 5 1	7		
t1		6 1	7		
t2			6		
t3					

- **6 threads** on 3 cores (s0+s1 in (c0, t0)).
- (c0, t0) –> (c1, t1) **–> (c2, t1)** –> (c1, t0) **–> (c2, t2)** –> (c2, t0).
- Transmission of the continuation context (**3 sendings from c1 to c2**).

Sum reduction execution.

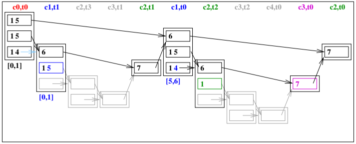
1
```
sum:    cmpq $2, %rsi
        ja .L2
```

2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```

3
```
        addq $1, %rdi
        addq %rdi, %rax
```

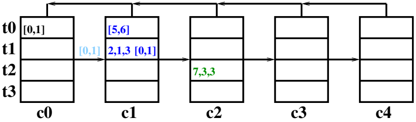
4
```
.L1:    retint
```

5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```




```
        pop %rbx
        pop %rsi
        pop %rdi
        movq %rax, %rcx
        addq %rsi, %rdi
        subq %rsi, %rbx
        movq %rbx, %rsi
        fork $1
        push %rcx
        call sum
```
6

```
        pop %rcx
        addq %rcx, %rax
        retint
```
7



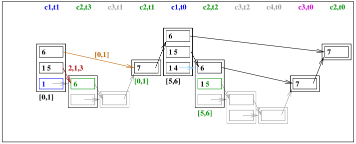
	c0	c1	c2	c3	c4
t0	1 5 1 5 1 2 3 4	6 1 5 1 2 3 4	7	7	
t1		6 1 5	7		
t2			6 1		
t3					

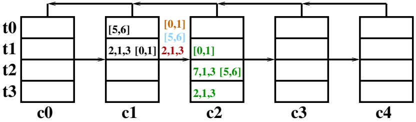
- **7 threads** on 4 cores (s5+s6 in (c1, t0)).
- (c0, t0) –> ... –> (c2, t2) **–> (c3, t0)** –> (c2, t0).
- Transmission of **result rax=s0+s1** (c0 to c1, retint –> rcx=rax).

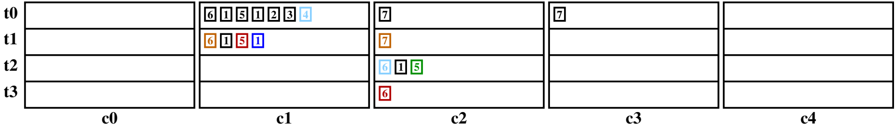
Sum reduction execution.

```
1  sum:   cmpq $2, %rsi
          ja .L2
2         movq %rdi, %rax
          subq $1, %rsi
          je .L1
3         addq $1, %rdi
          addq %rdi, %rax
4  .L1:   retint
5  .L2:   movq %rsi, %rbx
          shrq %rsi
          fork $3
          push %rdi
          push %rsi
          push %rbx
          call sum
```

```
   pop %rbx
   pop %rsi
   pop %rdi
   movq %rax, %rcx
   addq %rsi, %rdi
   subq %rsi, %rbx
   movq %rbx, %rsi
   fork $1
   push %rcx
   call sum                6
   pop %rcx
   addq %rcx, %rax
   retint                  7
```



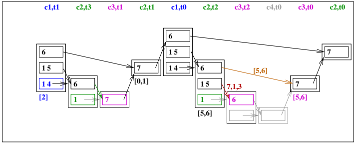




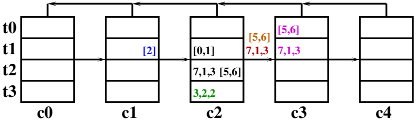
- **(c0, t0) ended**, t0 freed. 5 transmissions from c1 to c2 (5 cycles).
- Transmission of the continuation context (**rcx=s0+s1 from c1 to c2**).
- Transmission of **result rax=s5+s6** (c1 to c2, retint –> rcx=rax).

Sum reduction execution.

1
```
sum:   cmpq $2, %rsi
       ja .L2
```
2
```
       movq %rdi, %rax
       subq $1, %rsi
       je .L1
```
3
```
       addq $1, %rdi
       addq %rdi, %rax
```
4
```
.L1:   retint
```
5
```
.L2:   movq %rsi, %rbx
       shrq %rsi
       fork $3
       push %rdi
       push %rsi
       push %rbx
       call sum
```



6
```
       pop %rbx
       pop %rsi
       pop %rdi
       movq %rax, %rcx
       addq %rsi, %rdi
       subq %rsi, %rbx
       movq %rbx, %rsi
       fork $1
       push %rcx
       call sum
```
7
```
       pop %rcx
       addq %rcx, %rax
       retint
```

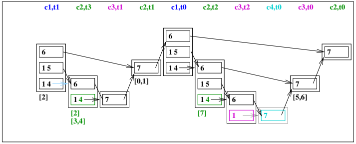


	c0	c1	c2	c3	c4
t0		6 1 5 1 2 3 4	7	7	
t1		6 1 5 1 2 4	7	7	
t2			6 1 5 1	6	
t3			6 1		

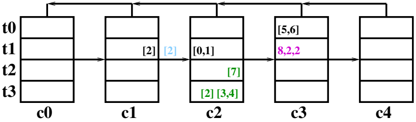
- **9 threads** on 3 cores. **Four transmissions** from c2 to c3 (one per cycle).
- Transmission of the **continuation context** ((c2, t2, b5) and (c3, t2, b6)).
- Transmission of the **continuation context** ((c2, t2, b6) and (c3, t0, b7)).

Sum reduction execution.

```
1   sum:   cmpq $2, %rsi
           ja .L2

2          movq %rdi, %rax
           subq $1, %rsi
           je .L1

3          addq $1, %rdi
           addq %rdi, %rax

4   .L1:   retint

5   .L2:   movq %rsi, %rbx
           shrq %rsi
           fork $3
           push %rdi
           push %rsi
           push %rbx
           call sum
```




```
6          pop %rbx
           pop %rsi
           pop %rdi
           movq %rax, %rcx
           addq %rsi, %rdi
           subq %rsi, %rbx
           movq %rbx, %rsi
           fork $1
           push %rcx
           call sum

7          pop %rcx
           addq %rcx, %rax
           retint
```

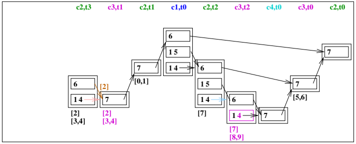


	c0	c1	c2	c3	c4
t0		6 1 5 1 2 3 4	7	7	7
t1		6 1 5 1 2 4	7	7	
t2			6 1 5 1 2 4	6 1	
t3			6 1 2 3 4		

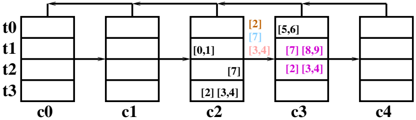
- **10 threads** on 4 cores (s3+s4 in (c2, t3)). **Result s2** from c1 to c2.
- **rax+=rdi** (c1t0b3) <– **rax+=rcx** (c2t1b7) <– **rax+=rcx** (c3t2b7).
- **rax+=rcx** (c3t2b7) <– **rax+=rdi** (c2t3b3) <– **retint** (c1t1b4).

Sum reduction execution.

1
```
sum:    cmpq $2, %rsi
        ja .L2
```
2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```
3
```
        addq $1, %rdi
        addq %rdi, %rax
```
4
```
.L1:    ret int
```
5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```



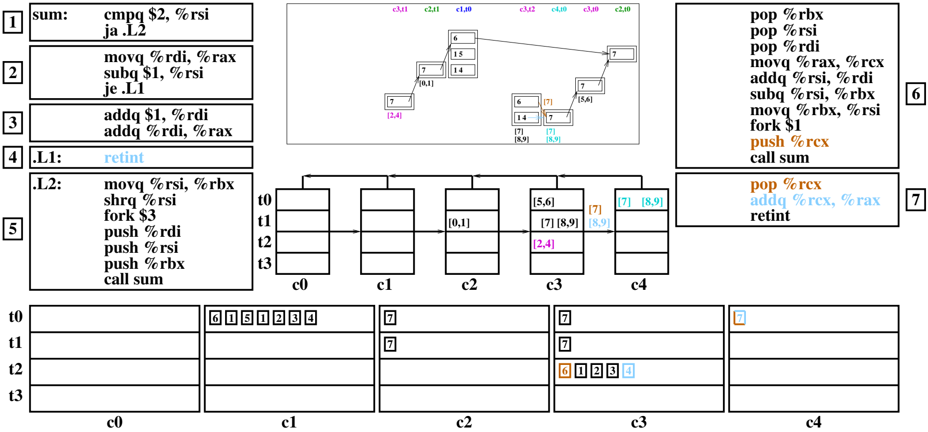
6
```
pop %rbx
pop %rsi
pop %rdi
movq %rax, %rcx
addq %rsi, %rdi
subq %rsi, %rbx
movq %rbx, %rsi
fork $1
push %rcx
call sum
```
7
```
pop %rcx
addq %rcx, %rax
retint
```



	c0	c1	c2	c3	c4
t0		6 1 5 1 2 3 4	7	7	7
t1			7	7	
t2			6 1 5 1 2 4	6 1 2 3 4	
t3			6 1 2 3 4		

- **(c1, t1) ended**, t1 freed.
- **Transmission of rax** from c2 to c3 (t2 –> t2 (s7) and t3 –> t1 (s3+s4)).
- **Transmission of rcx** from c2 to c3 (t3 –> t1 (s2)).

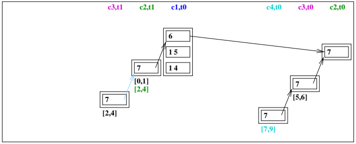
Sum reduction execution.



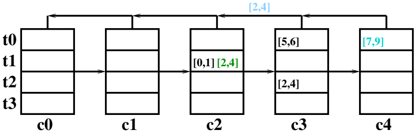
- **(c2, t2) and (c2, t3) ended**, t2 and t3 freed.
- **Transmission of rax** from c3 to c4 (t2 –> t0 (s8+s9)).
- **Transmission of rcx** from c3 to c4 (t2 –> t0 (s7)).

Sum reduction execution.

1
```
sum:    cmpq $2, %rsi
        ja .L2
```
2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```
3
```
        addq $1, %rdi
        addq %rdi, %rax
```
4
```
.L1:    retint
```
5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```



6
```
pop %rbx
pop %rsi
pop %rdi
movq %rax, %rcx
addq %rsi, %rdi
subq %rsi, %rbx
movq %rbx, %rsi
fork $1
push %rcx
call sum
```
7
```
pop %rcx
addq %rcx, %rax
retint
```

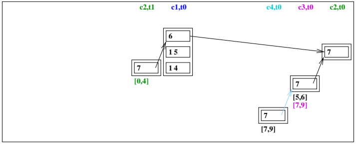


	c0	c1	c2	c3	c4
t0		6 1 5 1 2 3 4	7	7	7
t1			7	7	
t2					
t3					

- **(c3, t2) ended**, t2 freed.
- **Transmission of rax** from c3 to c2 (t1 –> t1 (s2+s3+s4)).
- **Execution of rax+=rcx** (c4, t0, b7) (rax=s7+s8+s9).

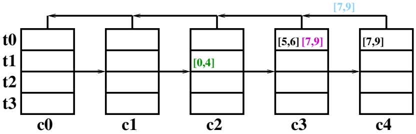
Sum reduction execution.

1
```
sum:    cmpq $2, %rsi
        ja .L2
```
2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```
3
```
        addq $1, %rdi
        addq %rdi, %rax
```
4
```
.L1:    retint
```
5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```




```
        pop %rbx
        pop %rsi
        pop %rdi
        movq %rax, %rcx
        addq %rsi, %rdi
        subq %rsi, %rbx
        movq %rbx, %rsi
        fork $1
        push %rcx
        call sum
```
6

```
        pop %rcx
        addq %rcx, %rax
        retint
```
7

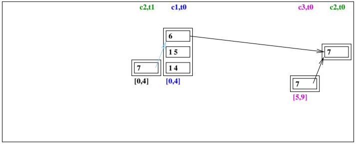


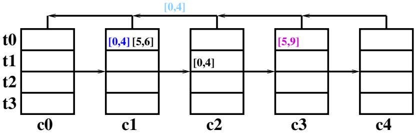
	c0	c1	c2	c3	c4
t0		6 1 5 1 2 3 4	7	7	7
t1			7		
t2					
t3					

- **(c3, t1) ended**, t1 freed.
- **Transmission of rax** from c4 to c3 (t0 –> t0 (s7+s8+s9)).
- **Execution of rax+=rcx** (c2, t1, b7) (rax=s0+...+s4).

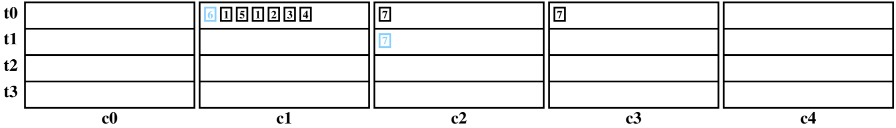
Sum reduction execution.

```
1  sum:   cmpq $2, %rsi
          ja .L2
2         movq %rdi, %rax
          subq $1, %rsi
          je .L1
3         addq $1, %rdi
          addq %rdi, %rax
4  .L1:   retint
5  .L2:   movq %rsi, %rbx
          shrq %rsi
          fork $3
          push %rdi
          push %rsi
          push %rbx
          call sum
```






```
6         pop %rbx
          pop %rsi
          pop %rdi
          movq %rax, %rcx
          addq %rsi, %rdi
          subq %rsi, %rbx
          movq %rbx, %rsi
          fork $1
          push %rcx
          call sum
7         pop %rcx
          addq %rcx, %rax
          retint
```



- **(c4, t0) ended**, t0 freed.
- **Transmission of rax** from c2 to c1 (t1 −> t0 (s0+...+s4)).
- **Execution of rax+=rcx** (c3, t0, b7) (rax=s5+...+s9).

Sum reduction execution.

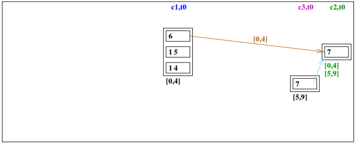
1
```
sum:    cmpq $2, %rsi
        ja .L2
```

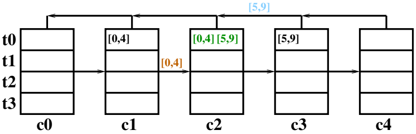
2
```
        movq %rdi, %rax
        subq $1, %rsi
        je .L1
```

3
```
        addq $1, %rdi
        addq %rdi, %rax
```

4
```
.L1:    retint
```

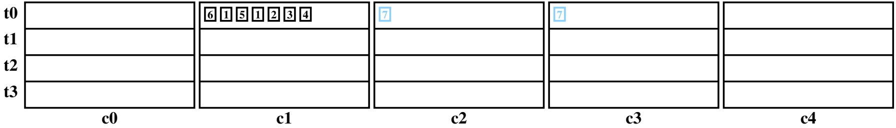
5
```
.L2:    movq %rsi, %rbx
        shrq %rsi
        fork $3
        push %rdi
        push %rsi
        push %rbx
        call sum
```





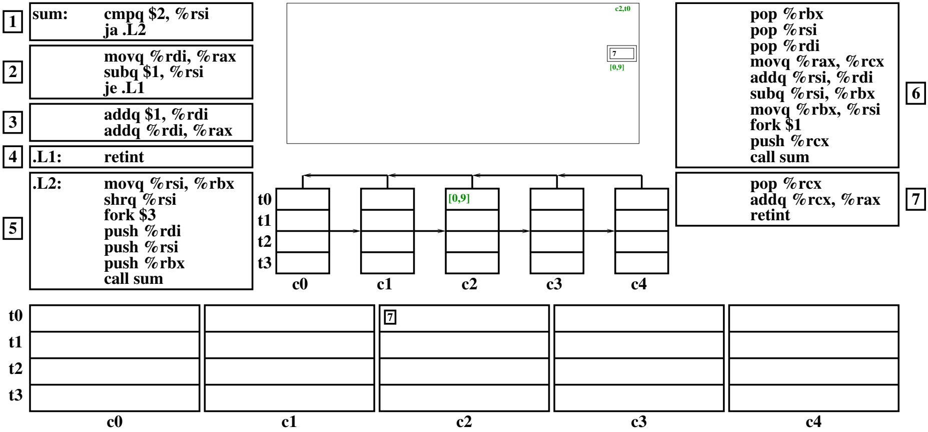
6
```
pop %rbx
pop %rsi
pop %rdi
movq %rax, %rcx
addq %rsi, %rdi
subq %rsi, %rbx
movq %rbx, %rsi
fork $1
push %rcx
call sum
```

7
```
pop %rcx
addq %rcx, %rax
retint
```



- **(c2, t1) ended**, t1 freed.
- **Transmission of rax** from c3 to c2 (t0 –> t0 (s5+...+s9)).
- **Transmission of rax** from c1 to c2 (t0 –> t0 (s0+...+s4)).

Sum reduction execution.



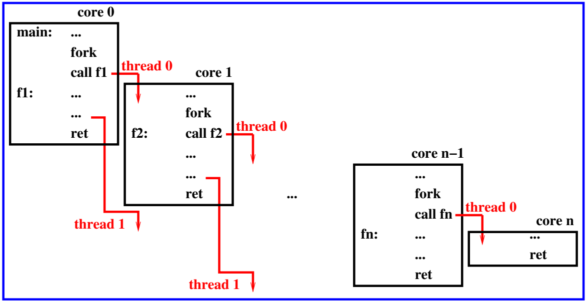
- **(c1, t0) and (c3, t0) ended**, t0 freed.
- **Execution of rax+=rcx** (c2, t0, b7) (rax=s0+...+s9).
- Instruction *retint* to **transmit the sum to *main***.

Section 4

Implicitely parallel programs.

Implicitely parallel code general organization.

```
void main(){
 //core 0 thread 0
 f1(...);
 //core 1 thread 0
 ...
 //core n-1 thread 0
 fn(...);
 //core n thread 0
}
```



- **Functions**.
- **Breadth first** calls populate **cores**.
- **Depth first** calls populate **threads**.
- **Breadth parallelizes**, **depth hides latencies**.
- When no free thread slot : local **sequential call**.

Data organization.

```
//char s[]="hello world" ;
//the array is replaced by an access function
//the compiler can automize the transformation
char s(int i){
 switch(i){
  case 0  : return 'h'; case 1   : return 'e';
  case 2  : return 'l'; case 3   : return 'l';
  case 4  : return 'o'; case 5   : return ' ';
  case 6  : return 'w'; case 7   : return 'o';
  case 8  : return 'r'; case 9   : return 'l';
  case 10 : return 'd'; default  : return '\0';
 }
}
```

- **No structured data**.
- No **array**, **structure**, **pointer**.
- Only **scalars** (integers or floating points).
- **Functions** to manipulate groups of scalars **one element at a time** (vector, matrix, structure).
- A data structure is naturally **centralized**.
- Parallelism requires **distributed** data.

Parallel I/O.

```
#include <sys/types.h>
#include <unistd.h>
//basic functions for I/O files accesses
int par_getchar_i(int i){
  int n; char c;
  lseek(0,i,SEEK_SET); n=read(0,&c,1);
  if (n==0) return -1; else return (int) c;
}
void par_putchar_i(int i, char c){
  lseek(1,i,SEEK_SET); write(1,&c,1);
}
void par_putchar_next(char c){
  lseek(1,1,SEEK_CUR); write(1,&c,1);
}
```

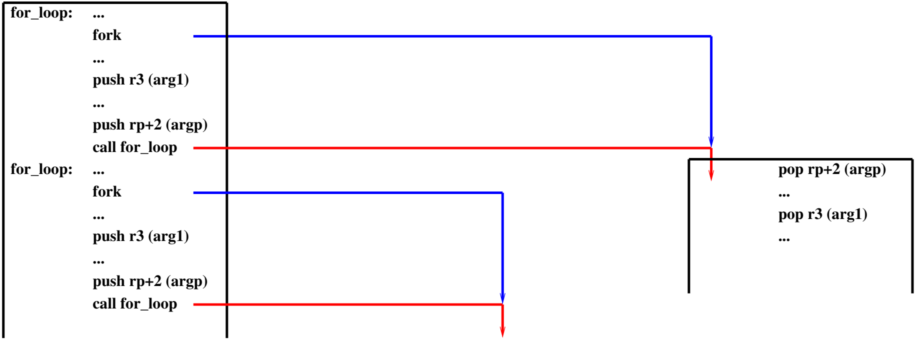
- The OS must provide an **efficient access** to external data.
- The OS must allow multiple cores to **access** a file **in parallel**.
- The OS must allow reading and writing a file **simultaneously in multiple different locations**.
- The OS must allow reading and writing a file **in partial order**.
- Every thread **reads what it needs from the file**, regardless of other threads.
- Every thread **writes at its own place in the file**, regardless of other threads.

Parallelizing *for* loops.

```
//to parallelize for (i=lower; i<upper; i++) body(i, arg);
//call for_loop(lower, upper-lower, body, arg);
//for_loop runs n iterations, starting from iteration i
//and applying body(i,arg_body)
void for_loop(int i, int n, void (*body)(), void *arg_body){
 if (n==1){body(i, arg_body); return;}
 if (n==2){body(i, arg_body); body(i+1, arg_body); return;}
 for_loop(i, n/2, body, arg_body);
 for_loop(i+n/2, n-n/2, body, arg_body);
}
```

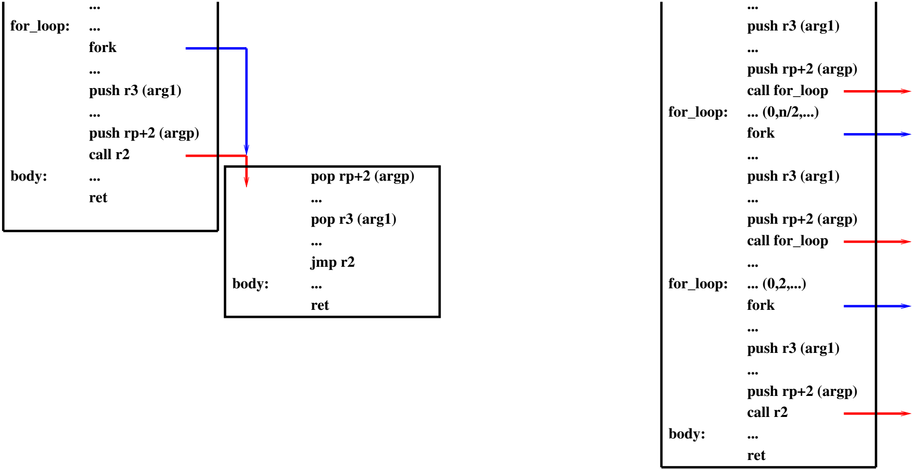
- **Divide-and-conquer spreading** : 2^n iterations launched in parallel in n steps.
- 2^n iterations spreaded on $n+1$ cores and 2^n **threads**.
- A c core processor with t threads per core may spread $c*t$ **iterations**, from which c are run in parallel.
- The *arg_body* argument is **a list of scalars**, bounded by the number of registers defined in the ISA.

Parallel execution of a *for* loop.



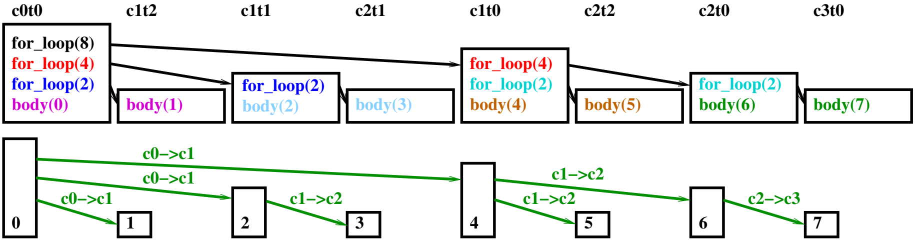
- **for_loop(i,n,body,arg)** : creates a thread on the next core.
- **push r0,...,rp+2** : transmits *i*, *n*, *body*, *arg* continuation values to the created thread.
- **call for_loop** : transmits the continuation PC to the next core.
- The **second fork** creates a second thread on the next core.

Parallel execution of a *for* loop.



- **The recursive descent** continues until n==2, then two calls of *body*.

Parallel execution of a *for* loop.



- 2^n **iterations** : $n+1$ cores, 2^n threads, at most $n!/(\lceil n/2 \rceil! * \lfloor n/2 \rfloor!)$ threads on a core.
- When each core can host up to 16 threads, **32 iterations on 6 cores** (1,5,10,10,5,1).
- With 64 iterations, **60 threads on 7 cores** (among which one runs 5 iterations sequentially) (1,6,15,20/16,15,6,1).
- With 128 iterations, **80 threads on 8 cores** (among which 4 run 6, 20, 20 and 6 iterations sequentially) (1,7,21/16,35/16,35/16,21/16,7,1).

An example of a parallelized *for* loop.

```
#include <stdio.h>
#include "for.h"
void print_fahr2cel(int i, void *arg){
 //it is assumed that printf is parallelized
 printf("%3d %6.1f\n",i,(5.0/9.0)*(i-32));
}
void print_table_fahr2cel_0_to_50(){
 void for_loop(0,51,print_fahr2cel,NULL);
}
main(){
 print_table_fahr2cel_0_to_50();
}
```

- 51 iterations spreaded in **6 cycles**.
- 7 cores, 51 threads (**1,5,11,14,13,6,1**).
- 7 instructions from **7 iterations run per cycle**.

Placing functions to increase the parallelism.

```
#include <stdio.h>
#include "for.h"
void empty(){}
void print_fahr2cel(int i, void *arg){
 //it is assumed that printf is parallelized
 printf("%3d %6.1f\n",i,(5.0/9.0)*(i-32));
}
void print_table_fahr2cel_0_to_100(){
 for_loop(0,33,print_fahr2cel,NULL);
 empty();
 for_loop(33,34,print_fahr2cel,NULL);
 empty();
 for_loop(67,34,print_fahr2cel,NULL);
}
main(){
 print_table_fahr2cel_0_to_100();
}
```

- 101 iterations spreaded in **11 cycles**.
- **11 cores**, 101 threads (1,5,11,15,16,16,16,12,6,2,1).
- Up to **11 iterations** in parallel.
- **Thread placement** is easy to control.

Parallelizing iterations.

```
#include <stdio.h>
void print_fahr2cel(int i, void *arg){
 //it is assumed that printf is parallelized
 printf("%3d %6.1f\n",i,(5.0/9.0)*(i-32));
}
void print_table_fahr2cel_0_to_100(){
 int i;
 for (i=0; i<101; i++)
  print_fahr2cel(i,NULL);
}
main(){
 print_table_fahr2cel_0_to_100();
}
```

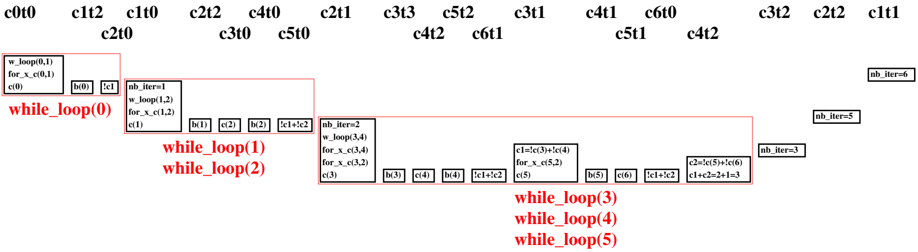
- 101 iterations spreaded in **101 cycles**.
- 101 cores, 101 threads (**1 per core**).
- **101 iterations** in parallel.
- **Maximal latency**, maximal parallelism.
- To minimize the latency, one must use ... **a GPU** (in our design, a special core accessed through a special control instruction).

While loops.

```
//while_loop launches iterations until cond (cond is the loop exit condition)
//while_loop assumes that when cond is true for i, it is true for any j>i
//while_loop returns the number of iterations in the loop
//for each iteration in the loop, body(i,arg_body) is applied
//while_loop returns the number of non excluded iterations
int while_loop(int i, int n, int (*cond)(int, void *), void *arg_cond,
                              void (*body)(int, void *), void *arg_body){
 int nb_iter=for_ex_cond(i,n,cond,arg_cond,body,arg_body);
 if (nb_iter==n)
   nb_iter+=while_loop(i+n,2*n,cond,arg_cond,body,arg_body);
 return nb_iter;
}
```

- To parallelize "i=lower ; while (!cond(i, argc)) body(i, argb) ;" call "**n=while_loop(lower,1,cond,argc,body,argb) ;**".
- Launches **1, 2, 4, 8, ... iterations** until cond(i, argc) is true.
- *n* is the number of iterations.
- 2*n* iterations are launched **only if** the *n* preceding iterations had all false conditions (*nb_iter* is *n*).
- **A parallel** *while (1)* can be implemented.
- We may be **less aggressive** (e.g. launching groups of *k* iterations).

Parallel execution of a *while* loop.



- We assume that cond(0)=...=cond(5)=**true** and cond(6)=**false**.
- We execute **body(0)**, ..., **body(5)**.
- The return value is **6**.

An example of a parallelized *while* loop.

```
#include <stdio.h>
#include "while.h"
//to compute strlen(char *s)
typedef struct {int (*f)();} ArgC;
int s(int i){
 switch(i){
  case 0: return 'h'; case 1: return 'e';
  case 2: return 'l'; case 3: return 'l';
  case 4: return 'o'; case 5: return ' ';
  case 6: return 'w'; case 7: return 'o';
  case 8: return 'r'; case 9: return 'l';
  case 10: return 'd'; default: return '\0';
 }
}
int is_eos(int i, void *arg){
 ArgC *a=(ArgC *)arg; return ((a->f)(i)=='\0');
}
void empty(int i, void *arg){}
int par_strlen(int (*f)()){
 ArgC a; a.f=f;
 return while_loop(0,1,is_eos,(void*)&a,empty,NULL);
}
void main(){printf("%d\n",par_strlen(s));}
```

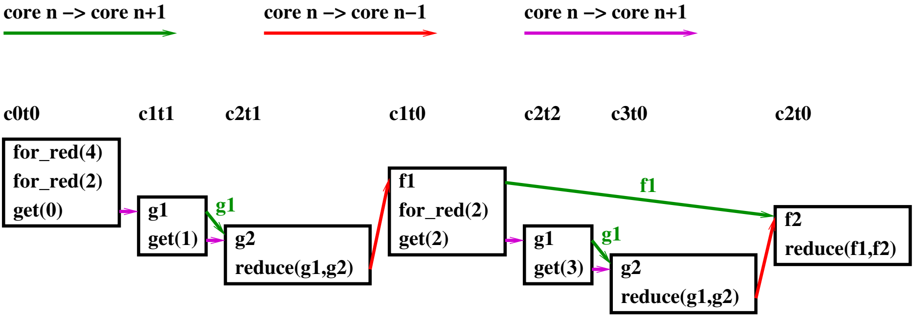
- Launches **1, 2, 4 and 8 iterations**.
- The call to *for_ex_cond*(7, 8, ...) returns 4. Function *par_strlen* **returns 11**.

Parallelizing reductions.

```
//for_reduce runs n iterations
//each iteration produces a scalar value by application of get(i,arg_get)
//the binary tree of iterations reduces the produced values
//according to the reduce function; for_reduce returns the reduced value
int for_reduce(int i, int n, int rnv, int (*get)(int, void *), void *arg_get,
               int (*reduce)(int, int, int, int, void *), void *arg_reduce){
 int g1, g2, f1, f2;
 if (n==1){return reduce(i,n,get(i,arg_get),rnv,arg_reduce);}
 if (n==2){
  g1=get(i,arg_get); g2=get(i+1,arg_get);
  return reduce(i,n,g1,g2,arg_reduce);
 }
 f1=for_reduce(i, n/2, rnv, get, arg_get, reduce, arg_reduce);
 f2=for_reduce(i+n/2, n-n/2, rnv, get, arg_get, reduce, arg_reduce);
 return reduce(i,n,f1,f2,arg_reduce);
}
```

- **Sum reduction** : int reduce_sum(int i, int n, int a, int b, void *arg){return a+b;}.
- **Maximum reduction** : int reduce_max(int i, int n, int a, int b, void *arg){return (a>b)?a:b;}.
- Arguments *i* and *n* serve to implement a **reduction on the iteration numbers**.
- Argument *rnv* is the **reduction neutral value**.

Parallel execution of a *for_reduce*.



- The return value is **reduce(reduce(get(0),get(1)),reduce(get(2),get(3)))**.

Example of a parallelized reduction.

```
//par_filechr(i,c) returns the position of first c after i
#include <stdio.h>
#include <sys/stat.h>
#include "for.h"
#include "getput_i.h"
typedef struct {char c; int l; int (*f)();} Arg;
int min(int i, int n, int a, int b, void *arg){
 if (a<b) return a; else return b;
}
int found(int i, void *arg){
 Arg *a=(Arg *)arg;
 if (a->f(i)!=a->c) return a->l; else return i;
}
int par_filechr(int i, char c){
 struct stat st;
 Arg a; int l;
 fstat(0,&st); l=st.st_size; a.c=c; a.l=l; a.f=getchar_i;
 return for_reduce(i, l, l+1, found, (void *)&a, min, NULL);
}
main(){printf("first { at %d\n",par_filechr(0,'{'));}
```

- ./par_filechr < par_filechr.c **returns "first { at 154"**.
- We look at **all the characters in the file in parallel**.
- The result is **the least index** returned by "found".
- By using a "while" loop, we avoid **useless iterations**.

Section 5

Conclusion.

Conclusion.

- Parallelize a run **from the hardware** rather than from the OS.
- Keep parallelism **implicit**.
- Follow a referential **deterministic order**.
- **Avoid memory data structures**.
- Communicate only between **neighbours**, from cause to effect.

Conclusion.

- Parallelize a run **from the hardware** rather than from the OS.
- Keep parallelism **implicit**.
- Follow a referential **deterministic order**.
- **Avoid memory data structures**.
- Communicate only between **neighbours**, from cause to effect.
- Our design illustrates that **automatic parallelization** is possible.
- **Functional programming paradigm** seems better **suited to parallel computation** than imperative programming paradigm : in parallel computers, memory is more a burden than a helper.