Efficient Multi-core Programming

Bruno Raffin
MOAIS Team, Grenoble, France
Bruno.raffin@inria.fr

Goals

To give you the main keys of “modern” parallel multi-core programming

Not a tutorial for learning a specific programming environment

Introduction

Parallel programming has long been the domain of high performance computing (supercomputers for numerical simulations), but for some years now also of:

- Cloud Computing (Amazon Elastic Cluster)
- Big Data Analytics (Google Map Reduce)

Evolution of computer architectures has made parallelism omnipresent:

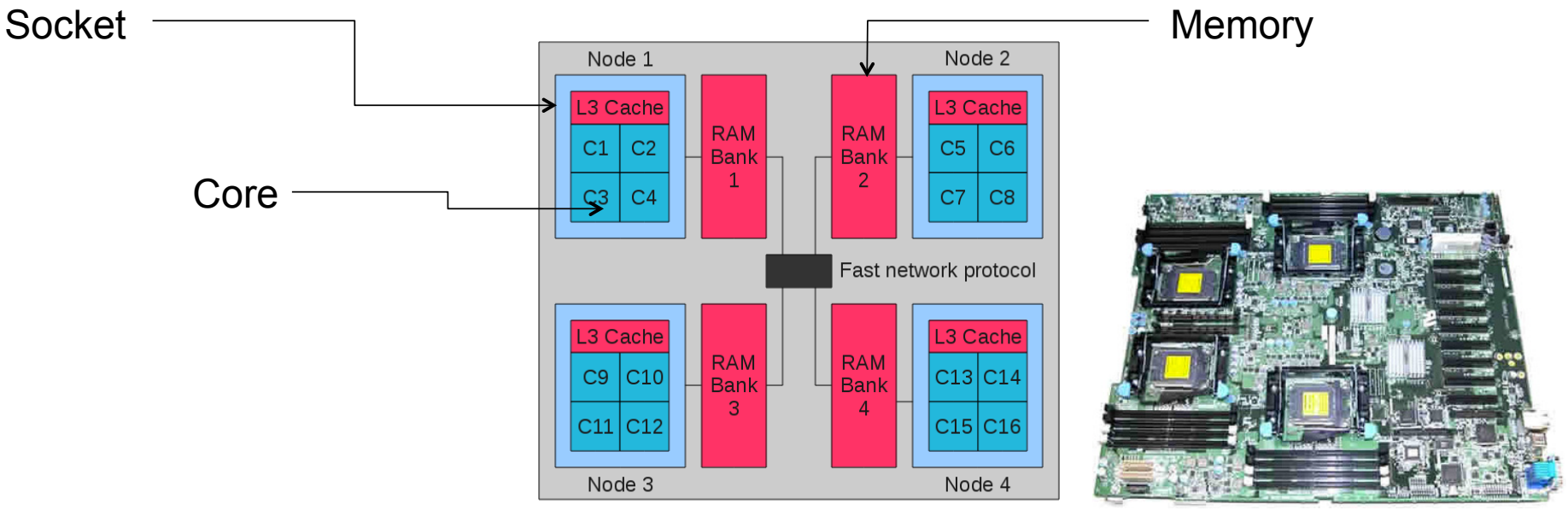
- Multi-core traditional processors
- Multi-core accelerators (GPU, Xeon Phi)
- Multi-core low power processors (ARM)

If you are somehow concerned about performance, you need to parallelize your code.

Exemple of supercomputer: SunWay TaiHuLight (China)

- #1@Top500 2016
- 10 500 000 compute cores
- 93 PetaFlops

Multi-core Architecture Overview



Memory: virtually shared memory (global address space, cache coherent), but physically distributed if more than one socket present.

- Memory access time depends on memory distance

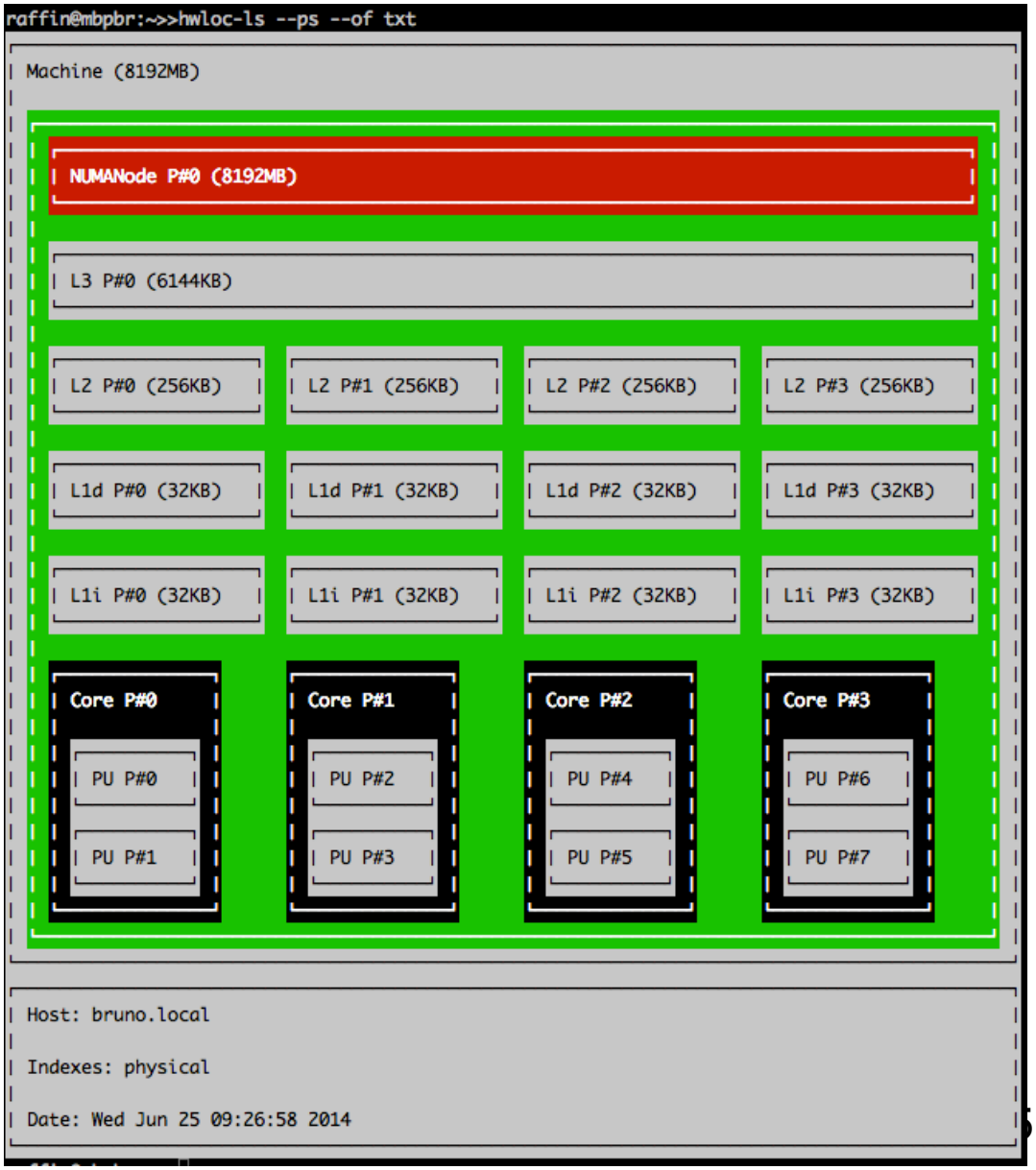
Memory hierarchy: L1 and L2 (core local) , L3 (shared in socket), memory

Multi-core Architecture Overview

How to know what you have:

hwloc-ls --of txt

hwlock tool (developed by INRIA Bordeaux Team) integrated in OpenMPI



Performance : Basics

T_1 = sequential execution time

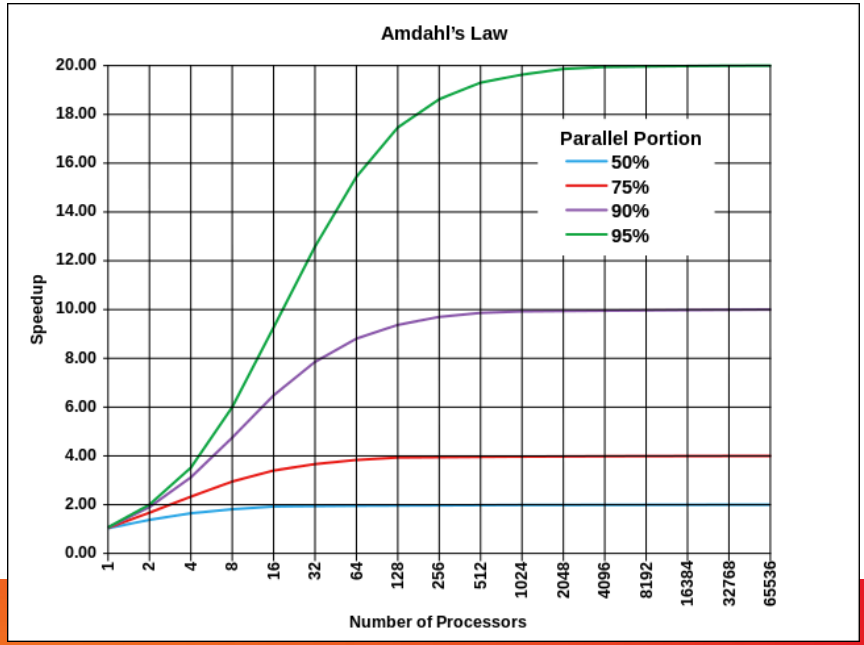
T_p = parallel execution time with p processors

Speedup: $T_1 / T_p \leq T_1 / (T_1/p) = p$

Amdahl's law: α: sequential fraction of the code

$$\lim_{P \to \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}$$

Sequential sections eventually kill the speedup



Parallel Programing Basics

Message Passing (MPI: Message passing interface)

- P processors
- Each processor owns a private memory (no shared memory)
- Processors communicate through explicit messages
  - MPI_SEND / MPI_RECV
- All processors execute the same program. Use their rank to distinguish their work.

```
if (myrank == 0)
        do something
else
        do another stuff
```

The vast majority of parallel applications are written with MPI.

- Well adapted to distributed memory architectures (clusters)
- Also work well on multi-core architectures
- But no data sharing (duplicates), load balancing can be difficult, parallelization needs a deep code refactoring

Parallel Programing Basics

Shared memory parallel programming:

- Direct thread programming (Java threads, Posix threads):
  - Need to care for too many low level details
    - Error-prone
    - Difficult to scale to many threads

    Forget about it except for some very specific cases and a few threads.

- CUDA/OpenCL programming
  - CUDA: no portability (NVIDIA products only)
  - OpenCL:
    - portable (not true for performance, but make progress)
    - Programming model deeply influenced by GPU hardware
  - Require a deep code refactoring

Task Programming

Express potential parallelism. Let the runtime actually extract the required parallelism and schedule it when and where it thinks its appropriate.

Potential parallelism: **a task** (sequence of instructions)

- Dynamics and recursive: a task can create other tasks.

Shared memory model: the programmer needs to ensure concurrent accesses (R/W or W/W) are correctly managed.

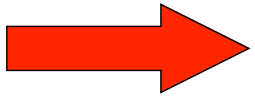
- Base synchronization primitive: **sync**

  wait for the completion of all previously -sequential order- defined tasks.

Programming with tasks: Cilk, Intel TBB, OpenMP, KAAPI, OmpSS

Task Programming: Cilk Example

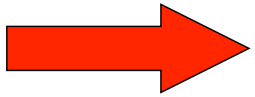
```
Int fib (int n)
{
        int x, y;
        if (n<2) return n;
        x = fib (n-1);
         y = fib (n-2);
         return x+y;
}
```



Parallelizing Fibonacci with Cilk

Task Programming: Cilk Example

```
Int fib (int n)
{
        int x, y;
        if (n<2) return n;
        x = fib (n-1);
         y = fib (n-2);
         return x+y;
}
```



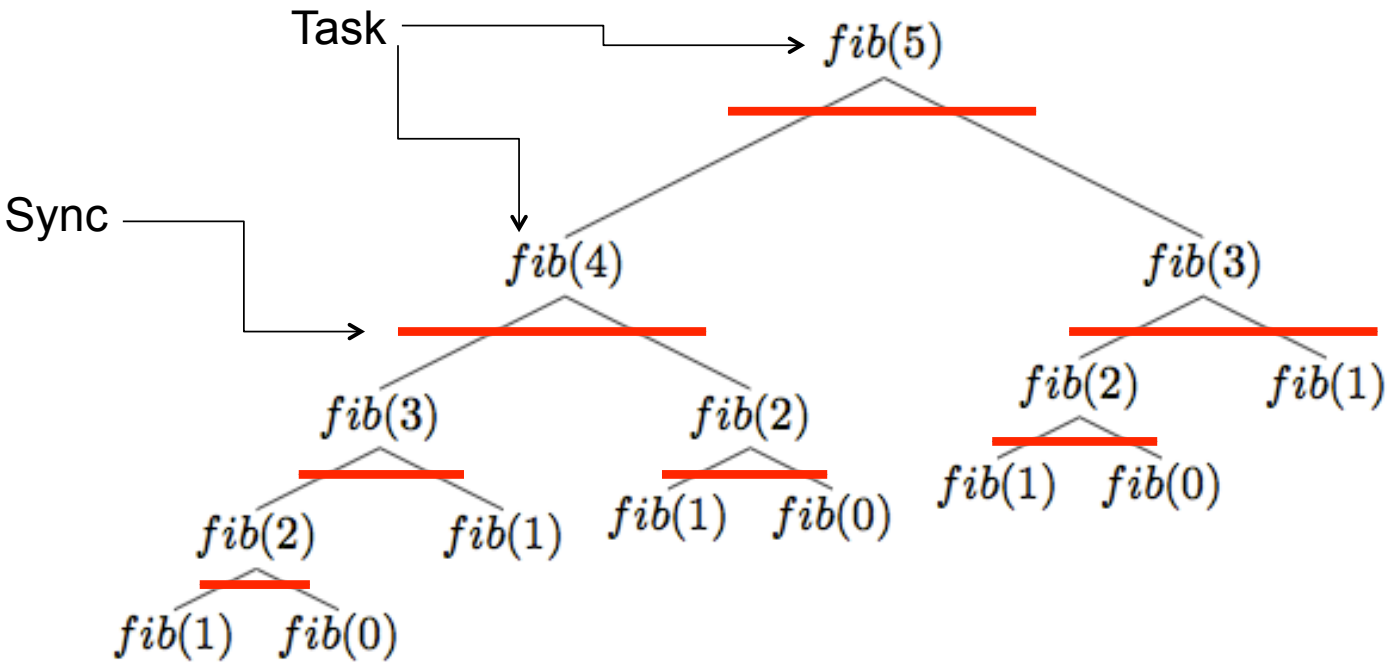
Parallelizing Fibonacci with Cilk

```
Int fib (int n)
{
        int x, y;
        if (n<2) return n;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return x+y;
}
```

Task creation

Wait the completion of all previously (sequential order) spawned tasks

Task Programming: Cilk Example



Task Scheduling

Where and when tasks are actually executed ?

Various scheduling algorithms can be used:

- List scheduling: all processors (or thread) get tasks from a centralized list

  -> Some OpenMP implementations

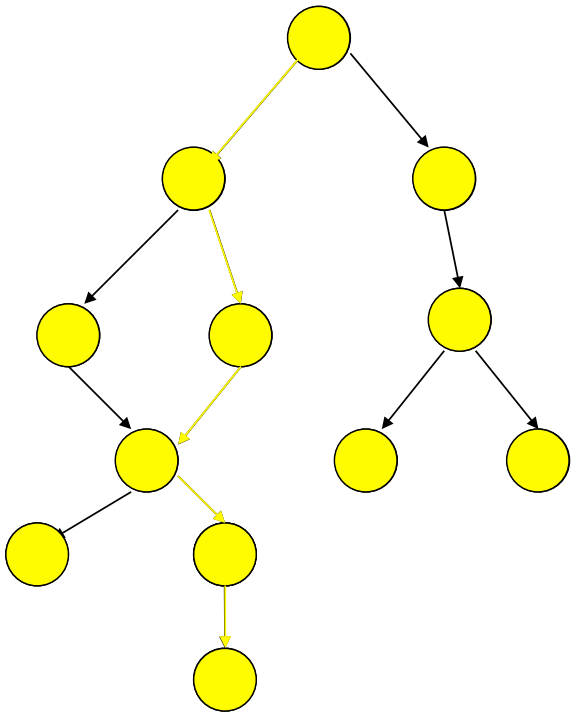
- Work stealing: distributed tasks lists. Execute local tasks first, randomly steal from others if idle.

  -> TBB, Cilk, KAAPI

List Scheduling [Graham 65]

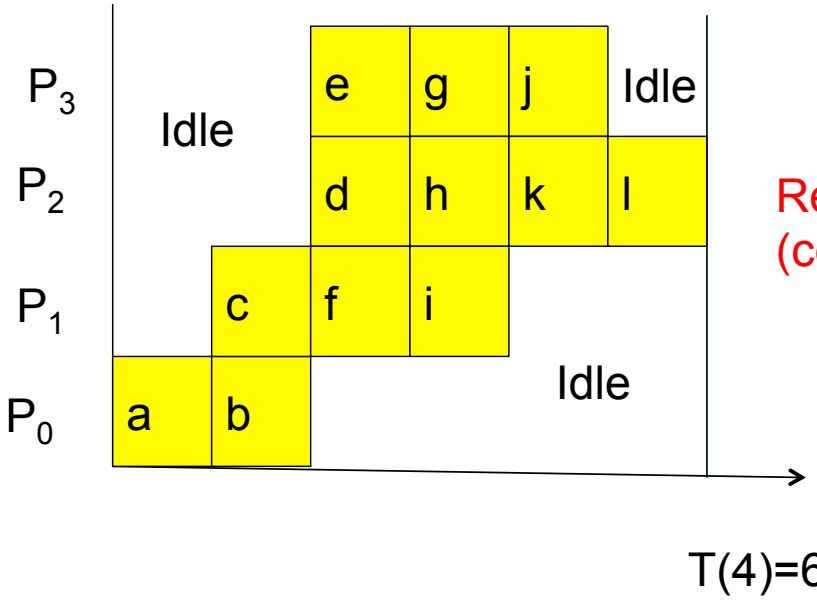
- List of tasks to be executed: dependencies define a **direct acyclic graph**
- A task is **ready** to be executed when all its dependencies are resolved.

- W_1: total number of operations to perform to execute the program
- W_∞: number of operations to perform along the critical path



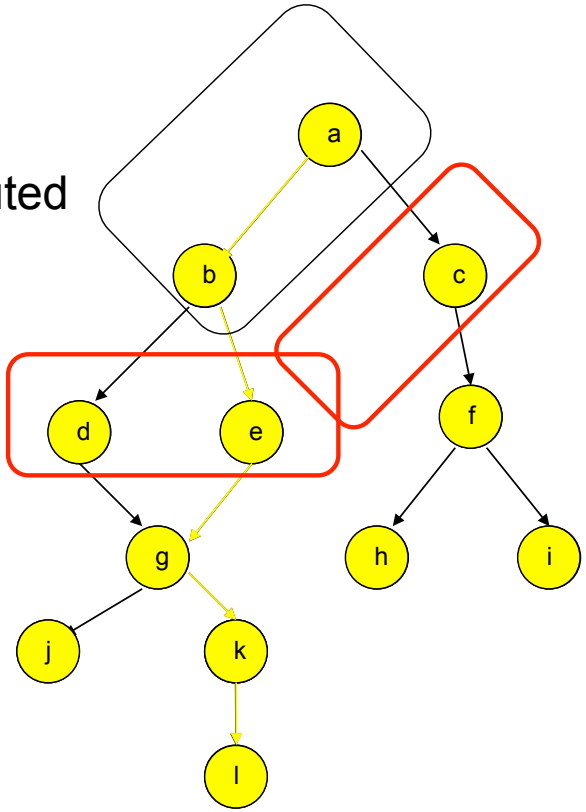
$W_1 = 12$
$W_\infty = 6$

List Scheduling [Graham 65]



Tasks
already executed

Ready Tasks
(centralized list)



List Scheduling [Graham 65]

Theorem [Graham 69]

$$T(p) \leq [W_1 + (p - 1) . W_\infty] / \pi.p$$

where:

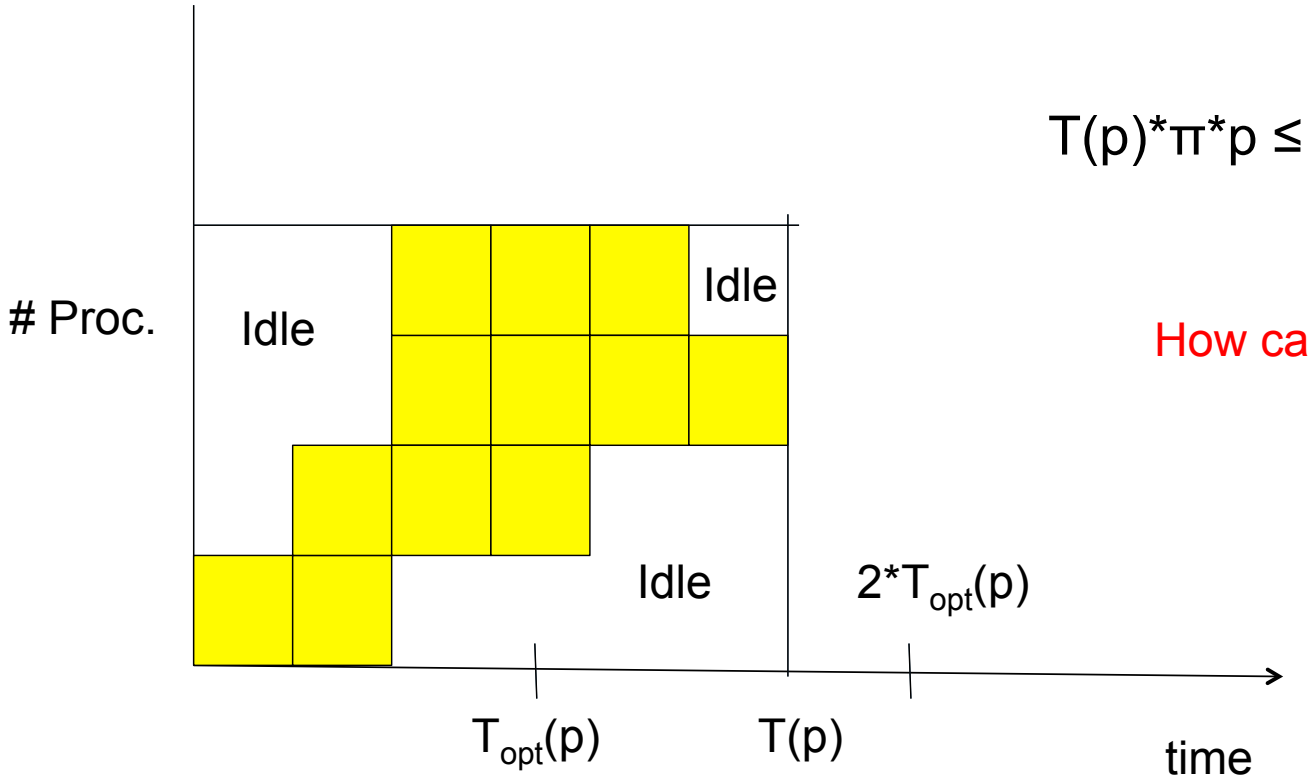
p: number of processors

π: processor speed (ops/s)

Idea of the proof:

- $T(p) \leq (W_1 + Idle) / \pi.p$
- There is at most one Idle zone per step along the critical path : $Idle \leq (p - 1) . W_\infty$

List Scheduling [Graham 65]



$$T(p)*\pi*p \leq W_1 + \text{Sum(Idle)}$$

How can we bound Sum(Idle) ?

List Scheduling [Graham 65]

As

- $T(p) \leq [W_1 + (p - 1) . W_\infty] / \pi.p$
- $W_1 / \pi.p \leq T_{optimal}(p)$
- $W_\infty / \pi \leq T_{optimal}(p)$

We have:

$$T(p) \leq 2 . T_{optimal}(p)$$

Work Stealing Algorithm

- Each processor maintains its own list of ready tasks (tasks ready to be executed)
- Each processor executes its ready tasks
- New ready tasks are added to the local list.
- When a processor becomes idle, it randomly selects a victim and tries to steal part of its work (50%). If steal fails select another victim.

Decentralized scheduling algorithm

Work Stealing Algorithm: Provable Performance

Theorem [Arora et al. 98]

Work stealing guarantees with a high probability:

$$T(p) \leq W_1 / \pi.p + 0 (W_\infty / \pi),$$

with a total number of steals of $0(p.W_\infty)$.

- If $W_1 >>> W_\infty$: $T(p) \approx W_1 / \pi.p = T_{optimal}(p)$

- Number of steals related to $p.W_\infty$

Task Programming: Performance Considerations

List scheduling or work stealing: dynamics load balancing

Task creation (even if not actually used to extract parallelism) cost some overheads:

- To few tasks: not enough potential parallelism to enable load balancing and feed all processors
- To many (small) tasks: performance may be affected by task management overheads

Need to control the amount of tasks created: the granularity of tasks.

Tasks: Grain and Overheads

Sequential loop:

```
for i:=0 to n-1 {
        a[j] = F(A[i]);
}
```

Cilk parallelization ?

Tasks: Grain and Overheads

Sequential loop:

```
for i:=0 to n-1 {
        a[j] = F(A[i]);
}
```

Cilk parallelization:

```
Range(F,x,y){
  for z:=x to y-1 { A[z]=F(A[z]);}
}
r:= n/j;
for  j:=0 to n-1 stride r {
        spawn  Range(F,i,i+r);
}
sync;
```

$W_1(n) = 0(n)$

$W_\infty(n) = 0(n/j)$

If j = n : smallest granularity, but n task creations (overheads)

If j = p and all processors available during execution, we are close to the optimal (modulo steal overheads), but no flex for load balancing.

Difficulty: task size choice

High Level Parallel Instructions

```
Cilk_for i:=0 to n-1 {
    A[i]:=F(A[i]);
}
```

Parallel loop with independent iterations

No need to explicit tasks:

- Easy to program
- No need to deal with task grain (well, some hints may be needed)

#pragma cilk grainsize = 42

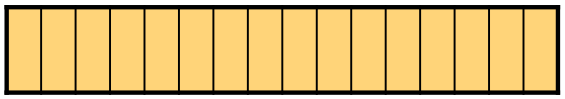
The runtime can rely on various approaches:

- Static iteration range partitioning (N/P or smaller)
- Recursive partitioning:

  Recursively create 2 tasks with 50% of the iteration domain each, down to the grain size limit
- On-demand partitioning:

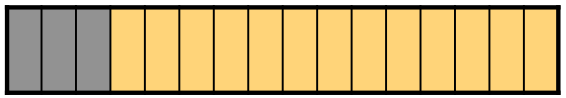
  When a victim receives a steal request, it gives half of its remaining iteration domain

On-demand Partitioning



T_1 : [0 - 15]

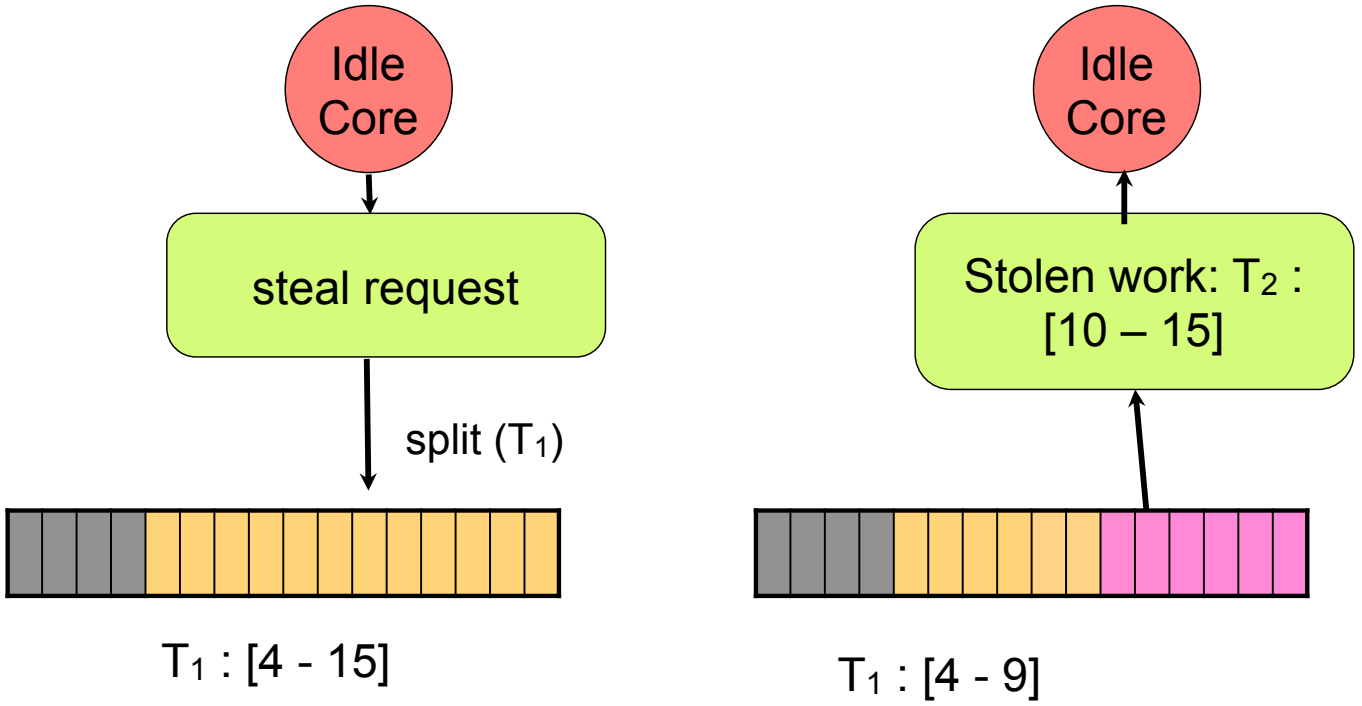
Proc 1 got all the work



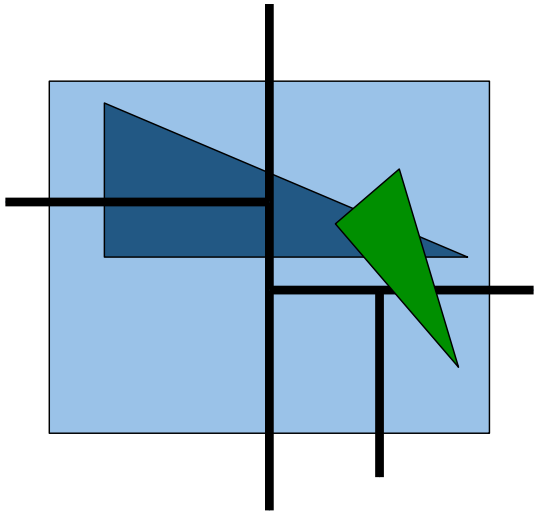
T_1 : [3 - 15]

Proc1 performed 3 iterations

On-demand Partitioning



Parallel Kd-Tree



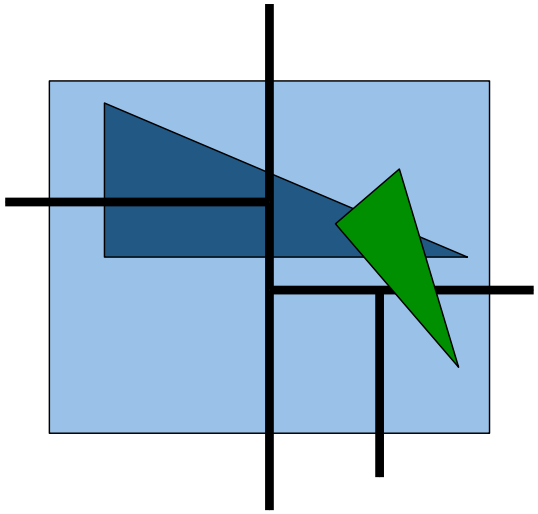
A classical acceleration data structure:

Ray tracing (ray/triangle intersection)

```
Buid_kd-tree(node)
        Select splitting plane
        Foreach triangle in node
                split triangle, put left part in node1, right part in node2
        Build_kd-tree (node1)
        Build_kd-tree (node2)
End
```

How to parallelize it ?

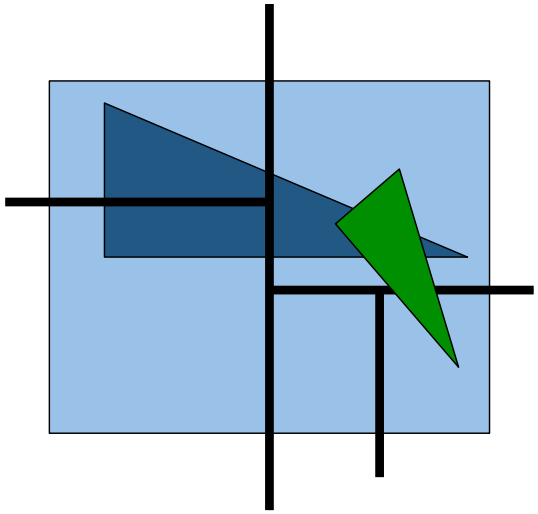
Parallel Kd-Tree



```
Buid_kd-tree(node)
        Select splitting plane
        Foreach triangle in node
                split triangle, put left part in node1, right part in node2
        Spawn Build_kd-tree (node1)
        Spawn  Build_kd-tree (node2)
End
```

Benefit of task programming ?
Benefit of work stealing ?
What should be improved ?

Parallel Kd-Tree



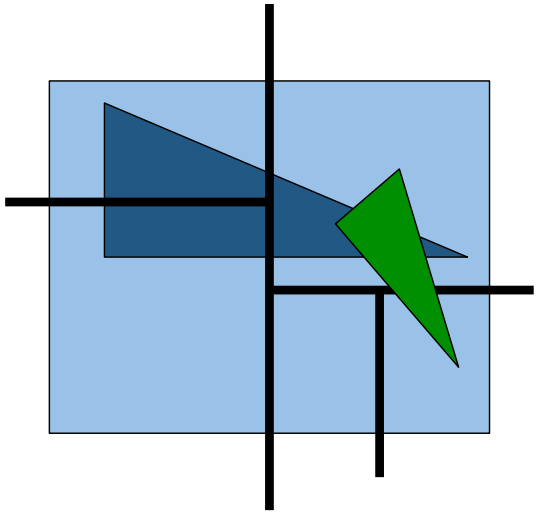
```
Buid_kd-tree(node)
        Select splitting plane
        foreach triangle in node
                split triangle, put left part in node1, right part in node2
        if (# triangle in node1 > threshold)
                Spawn Build_kd-tree (node1)
        else
                Build_kd-tree (node1)
        if (# triangle in node2 > threshold)
                Spawn Build_kd-tree (node2)
        else
                Build_kd-tree (node2)
End
```

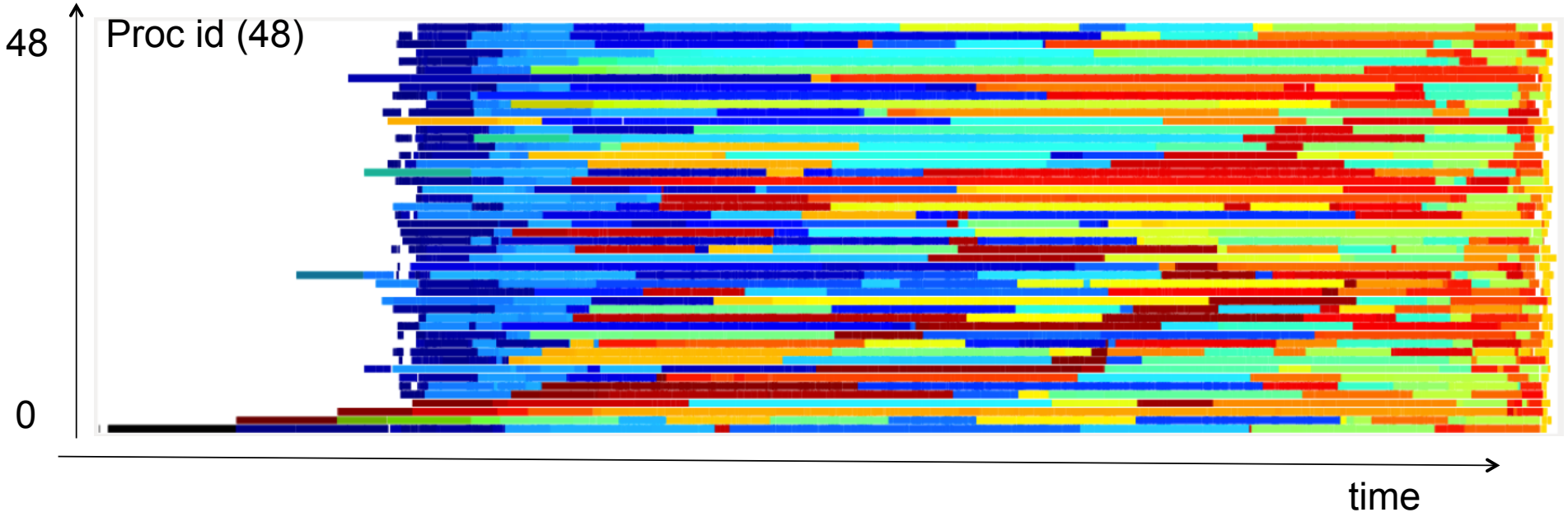
Control granularity by setting a threshold for task spawning

What about a threshold based on depth ?

What else to improve ?

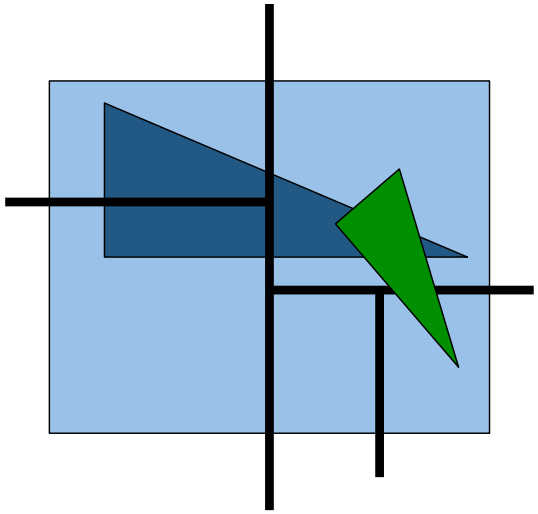
Parallel Kd-Tree





Trace profile you probably get

Parallel Kd-Tree

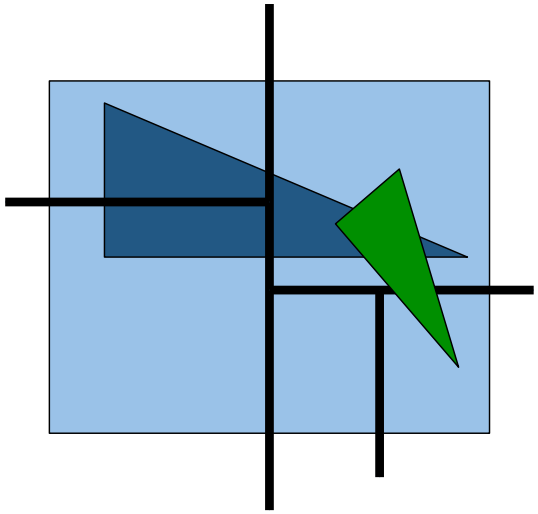


```
Buid_kd-tree(node)
        Select splitting plane
        Cilk_for each triangle in node
                split triangle, put left part in node1, right part in node2
        if (# triangle in node1 > threshold)
                Spawn Build_kd-tree (node1)
        else
                Build_kd-tree (node1)
        if (# triangle in node2 > threshold)
                Spawn Build_kd-tree (node2)
        else
                Build_kd-tree (node2)
End
```

! Concurrent write on node1 and node2

-> use concurrent data structures

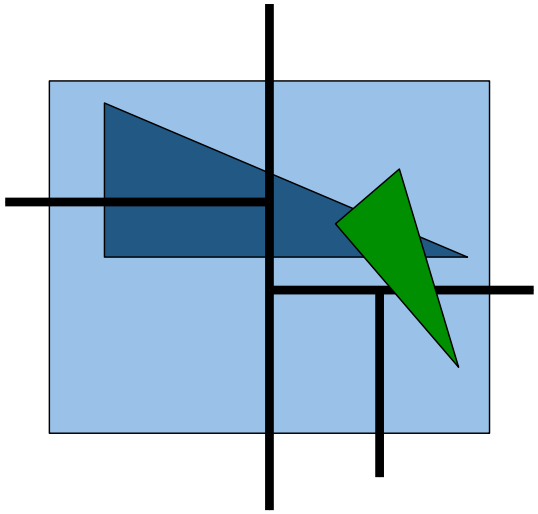
Parallel Kd-Tree



```
Buid_kd-tree(node)
        Select splitting plane
        if (#triangle in node > threshold)
                Cilk_for each triangle in node
                        split triangle, put left part in node1, right part in node2
        else
                Foreach triangle in node
                        split triangle, put left part in node1, right part in node2
        if (# triangle in node1 > threshold)
                Spawn Build_kd-tree (node1)
        else
                Build_kd-tree (node1)
        if (# triangle in node2 > threshold)
                Spawn Build_kd-tree (node2)
        else
                Build_kd-tree (node2)
End
```

Only internally parallelize big tasks
(mostly in the top of the tree)

Parallel Kd-Tree



```
Buid_kd-tree(node)
        Select splitting plane
        if (#triangle in node > threshold)
                Cilk_for each triangle in node
                        split triangle, put left part in node1, right part in node2
        else
                Foreach triangle in node
                        split triangle, put left part in node1, right part in node2
        if (# triangle in node1 > threshold)
                Spawn Build_kd-tree (node1)
        else
                Build_kd-tree (node1)
        if (# triangle in node2 > threshold)
                Spawn Build_kd-tree (node2)
        else
                Build_kd-tree (node2)
End
```

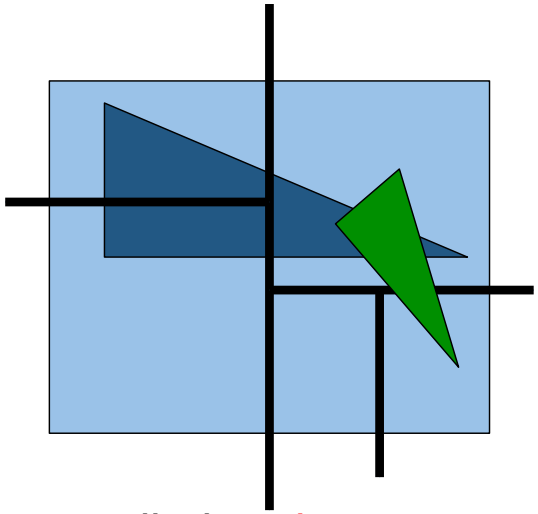
Let have a closer look at the concurrent write issue here !

Related papers

Advanced Graphics related algorithms parallelized with Work stealing:

- QuickCSG: Arbitrary and Faster Boolean Combinations of N Solids
- Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture

Reducers



Cilk provides thread safe and efficient parallel data structures called reducers

```
Cilk:reducer_list_append<triangle> node1, node2;
node_list =   node->get_value()
Cilk_for each triangle in node
        split triangle
        node1->push_back(left part)
        node2->push_back(right part)
```

Get the result of the reducer

Append data to the reducer

Append reducer: allow safe concurrent append to a list

Reducers

Other reducers exists like (and new ones can be developed - the reduction operation needs to be associative):

reducer_max_index

reducer_max

reducer_opadd

But keep in mind that for some reducers (case of append or reductions on floats) the result of reduction is often different from the one of the sequential execution.

Reducers

How it works (append reducer) ?

3 threads (workers), red, green and blue, execute the parallel cilk_for loop with work stealing.

Possible distribution of iterations amongst the threads:

it0 …. itN

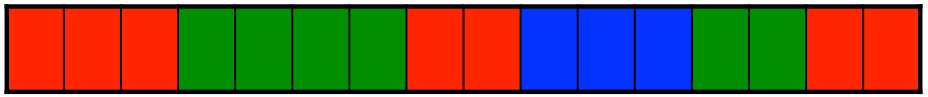
How to fill safely and efficiently (in parallel) the reducer list ?

Protect the reducer list with a lock operation ?
Safe but what about performance ?
Can be worst than a sequential execution.

Holder / Thread Local Storage

How to fill safely and efficiently (in parallel) the reducer list ?

Possible distribution of iterations amongst the red green and blue threads:



it0 …. itN

First notice that each thread executes sequentially a fraction of the iterations
Each thread can thus accumulate a partial result computed on the iterations it processes.

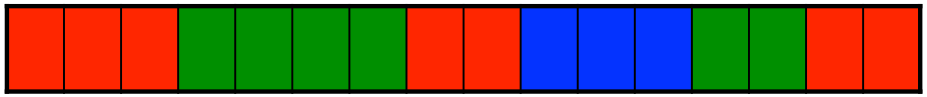
Hey, it's task based programming. I have no access to threads !!!

Well it's possible to declare a variable that has one instance per thread:

it's called a thread local storage (holders in Cilk)

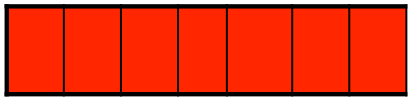
Holder / Thread Local Storage

Possible distribution of iterations amongst the red, green and blue threads:

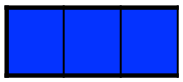


it0 …. itN

In parallel each thread accumulates a partial result computed on its iterations in its local storage:


Thread red local storage


Thread green local storage


Thread blue local storage

`

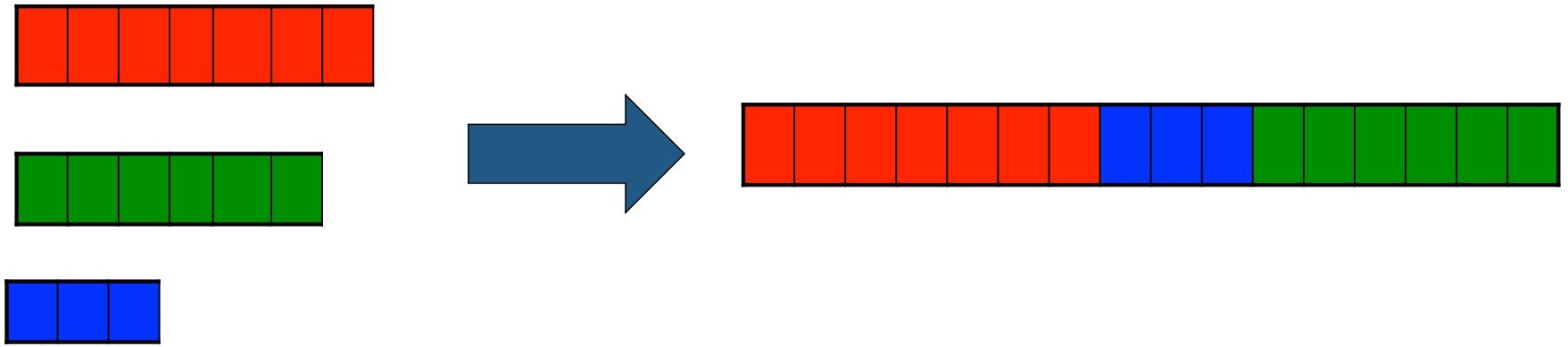
No sync between threads: very efficient

Cilk:holder<list<triangle>> holder1, holder2;

Cilk_for each triangle in node
    split triangle
    holder1->push_back(left part)
    holder2->push_back(right part)

Holder / Thread Local Storage

Now need to append the partial result: need sync between



1. Recursively by appending partial results two by two

2. Compute a parallel scan/prefix to get the offset for each thread, next threads can copy in parallel their list directly to their final destination

Scan/Prefix Computation

Spawn/Sync Considered Harmful ?

Get the OS Out of the Way:
Memory allocator

Classical allocator are made thread safe by using locks

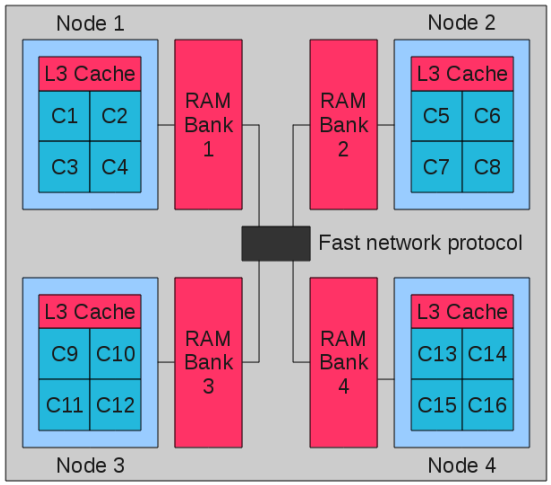
-> high performance penalty

Scalable allocators (tbbmalloc or gperftools) are designed to be efficient in a multi-threaded context.

Each thread provisions some memory space to be used when it needs to allocate memory (no lock). Provisioning memory require locks, but performed at lower frequency (amortized cost)

- Easy to use: simply link to the scalable allocator library
- Performance improvement can be significant (at no effort !)

Get the OS Out of the Way: Processor Binding



Classical scheduler can move a process or thread to a different core during the execution

- Need to repopulate the cache: more cache misses
- Data may be farther (memory bank): take longer to load

Explicitly bind each thread to a given core (and forbid migration):

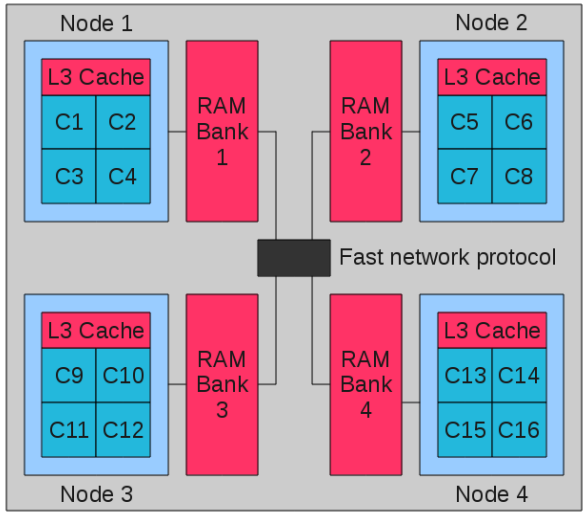
```
hwloc-bind  socket:1/core.1 socket:2.core:1 a.out
```

Get the OS Out of the Way:
Memory Binding

By default memory pages are locates in the memory bank of the first thread that touches it (first-touch policy). Can be bad.

Example:

- Data are read from a file by a single thread (usually the case)
- All pages containing these data are on the memory bank attached to the socket that ran that thread.
- In parallel sections all threads will read these data from this very same memory bank -> bottleneck



Get the OS Out of the Way:
Memory Binding

By default memory pages are locates in the memory bank of the first thread that touches it (first-touch policy). Can be bad.

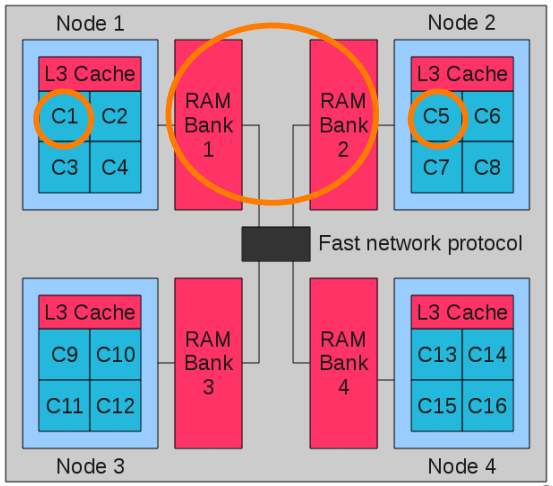
Alternative:

- Request that pages be cyclically distributed (interleave policy)
- On average threads will have their memory accesses evenly distributed on all memory banks

Example:

hwloc-bind -membind socket:1-2 --mempolicy interleave
–cpubind socket:1/core.1 socket:2.core:1 a.out

Effect: bind threads to first core of socket 1 and 2
and interleave pages on the associated memory banks



And much more can be done (see hwloc) !

Performance Monitoring

Get the execution time first

How to get lower level details (cache misses, page defaults, etc.):

Likwid : instrument the code to get some low level counters. Beware that it impacts the execution time.

Vtune: integrated performance monitoring and analysis tool from Intel. Need experience but can be very efficient once mastered.

Cilk, TBB and OpenMP

Intel Cilk (plus):

- a language (developed at MIT, transferred to Intel)
- very few constructions (cilk_spawn, cilk_sync, cilk_for, cilk_reducer, cilk_holder)
- integrated in 2 C++ compilers (ICC, GCC)

Intel TBB:

- A C++ library
- Not compiler dependent
- Verbose (c++)
- Expose more low level aspects (3 different ways to spawn tasks for instance)

OpenMP:

- A standard maintained by a consortium
- Pragmas (compilation directives)
- Various implementations (Intel, Gnu, IBM, Oracle, Portland Group …)
- Rich set of pragmas (too many ?) offering a lot of flex, but available implementations not always very efficient (on all aspects at least)
- Come just after MPI as the most used parallel programming environment

Cilk, TBB and OpenMP

Intel Cilk (plus):

- a language (developed at MIT, transferred to Intel)
- very few constructions (cilk_spawn, cilk_sync, cilk_for, cilk_reducer, cilk_holder)
- integrated in 2 C++ compilers (ICC, GCC)

Intel TBB:

- A C++ library
- Not compiler dependent
- Verbose (c++)
- Expose more low level aspects (3 different ways to spawn tasks for instance)

OpenMP:

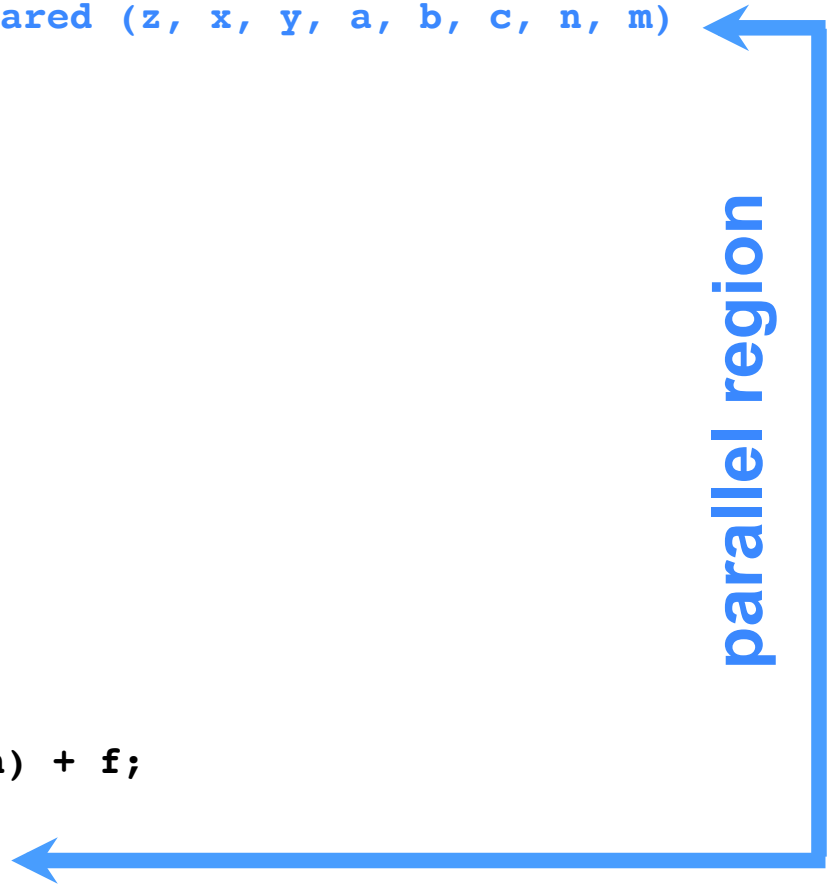
- A standard maintained by a consortium
- Pragmas (compilation directives)
- Various implementations (Intel, Gnu, IBM, Oracle, Portland Group …)
- Rich set of pragmas (too many ?) offering a lot of flex, but available implementations not always very efficient (on all aspects at least)
- Come just after MPI as the most used parallel programming environment

OpenMP: A Simple Example

```
f = 1.0

for (i = 0; i < N; i++)
  z[i] = x[i] + y[i];

for (i = 0; i < M; i++)
  a[i] = b[i] + c[i];

...

scale = sum (a, 0, m) + sum (z, 0, n) + f;
...
```

OpenMP: A Simple Example

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
   f = 1.0


   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];


   for (i = 0; i < m; i++)
      a[i] = b[i] + c[i];

   ...


   scale = sum (a, 0, m) + sum (z, 0, n) + f;
   ...
} /* End of OpenMP parallel region */
```



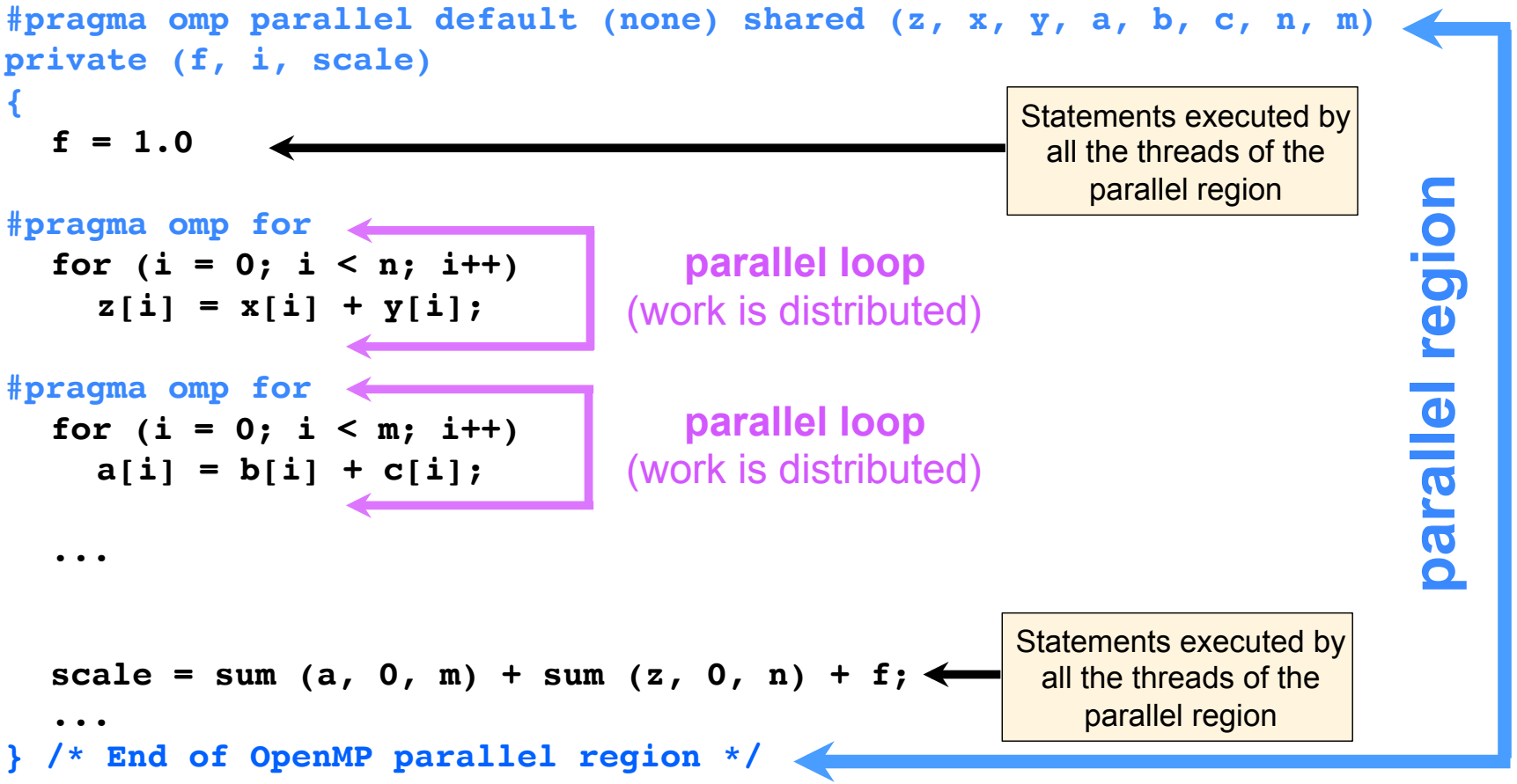
OpenMP: A Simple Example

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
   f = 1.0

   for (i = 0; i < n; i++)
     z[i] = x[i] + y[i];

   for (i = 0; i < m; i++)
     a[i] = b[i] + c[i];

   ...

   scale = sum (a, 0, m) + sum (z, 0, n) + f;
   ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region !

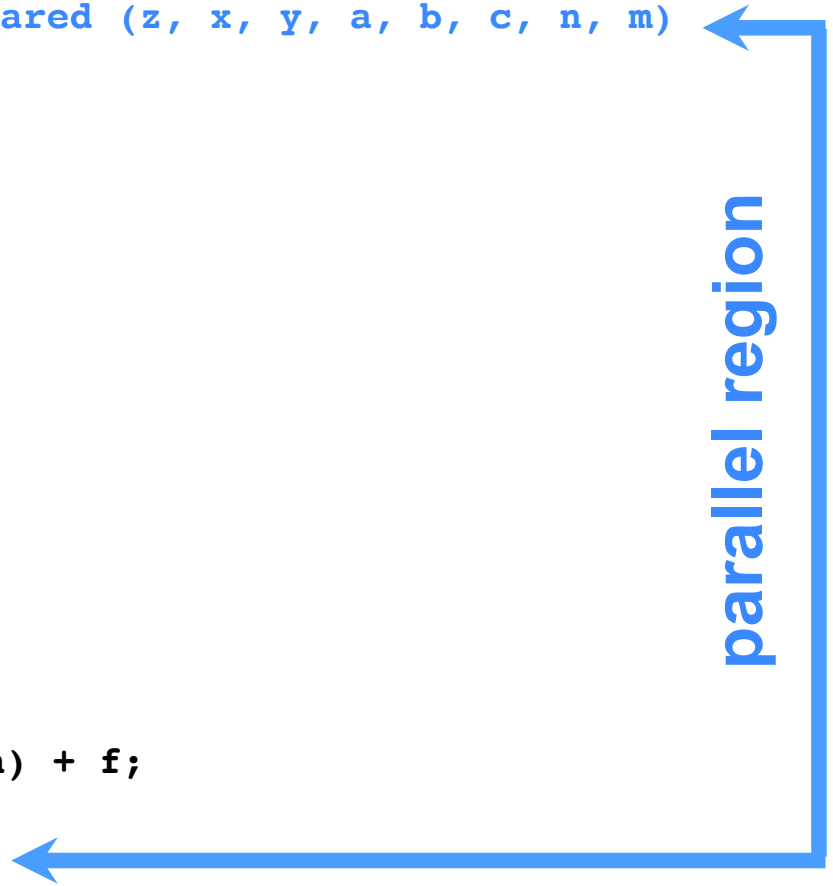
parallel region

OpenMP: A Simple Example



OpenMP: A Simple Example

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
   f = 1.0

#pragma omp for nowait
   for (i = 0; i < n; i++)
     z[i] = x[i] + y[i];

#pragma omp for nowait
   for (i = 0; i < m; i++)
     a[i] = b[i] + c[i];

   ...

#pragma omp barrier
   scale = sum (a, 0, m) + sum (z, 0, n) + f;
   ...
} /* End of OpenMP parallel region */
```



OpenMP: A Simple Example

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale) if (n > some_threshold && m > some_threshold)
{
  f = 1.0

#pragma omp for nowait
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for nowait
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...

#pragma omp barrier
  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

OpenMP: Fibonacci Example

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
       #pragma omp task shared(i)
firstprivate(n)
       i=fib(n-1);

       #pragma omp task shared(j)
firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
    }
}
```

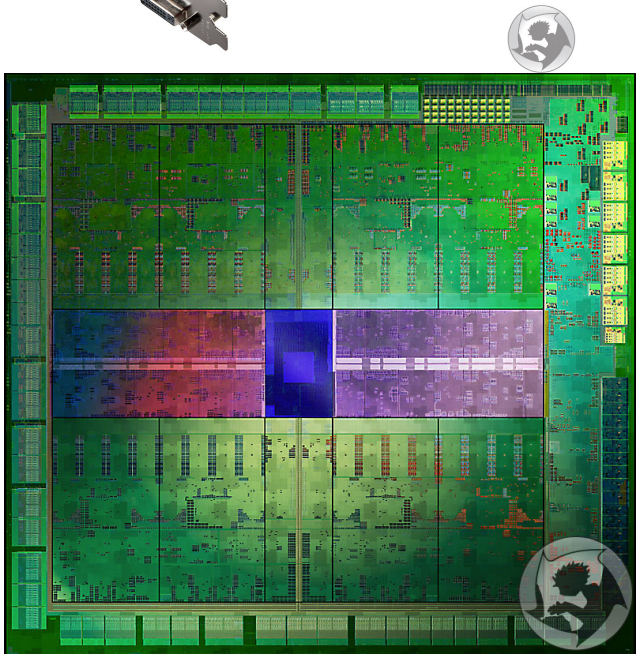
```
int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n,
fib(n));
  }
}
```

End of single (no implicit barrier)

End of parallel (implicit barrier)

A few words About Accelerators

2013 GTX Titan

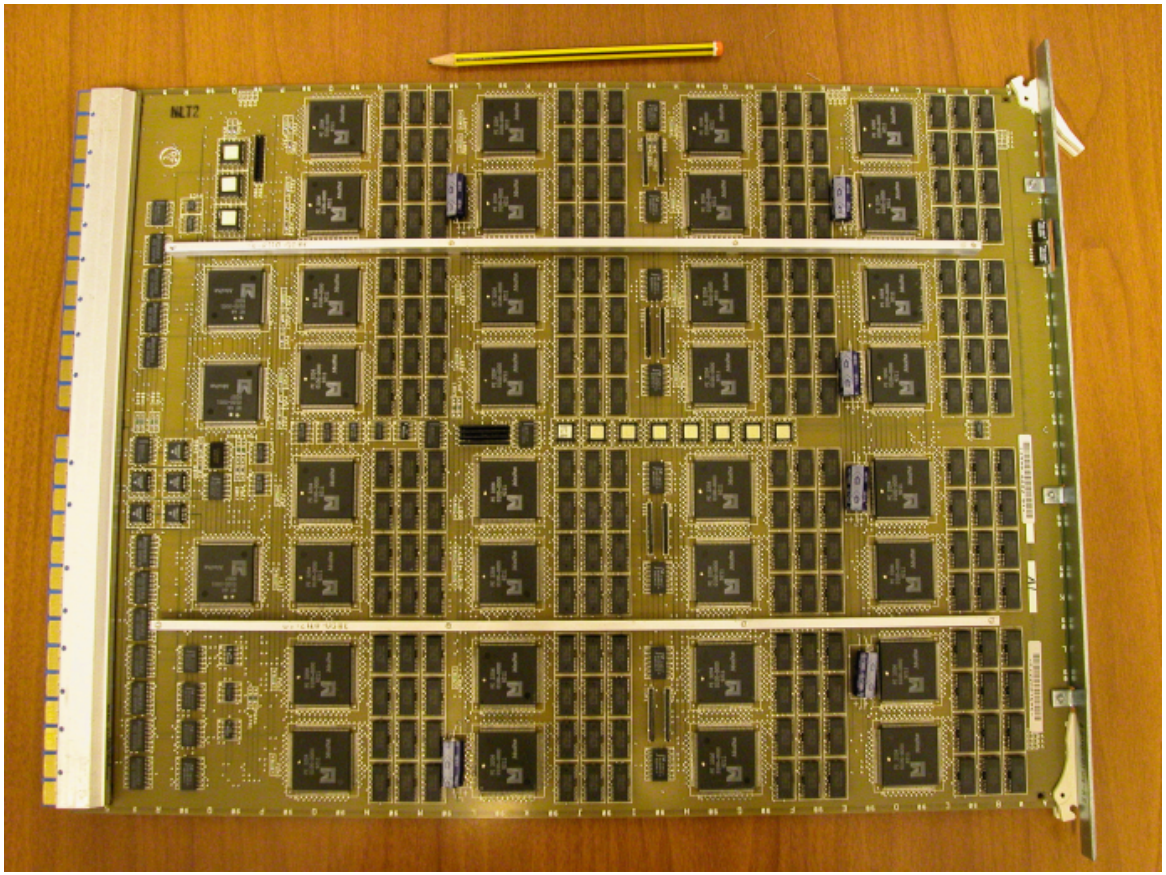


1.5 TFlops (fp64)
265 Watts

2688 cores
7.1 B transistors

6G GB of memory

1992 Maspar SIMD Machine

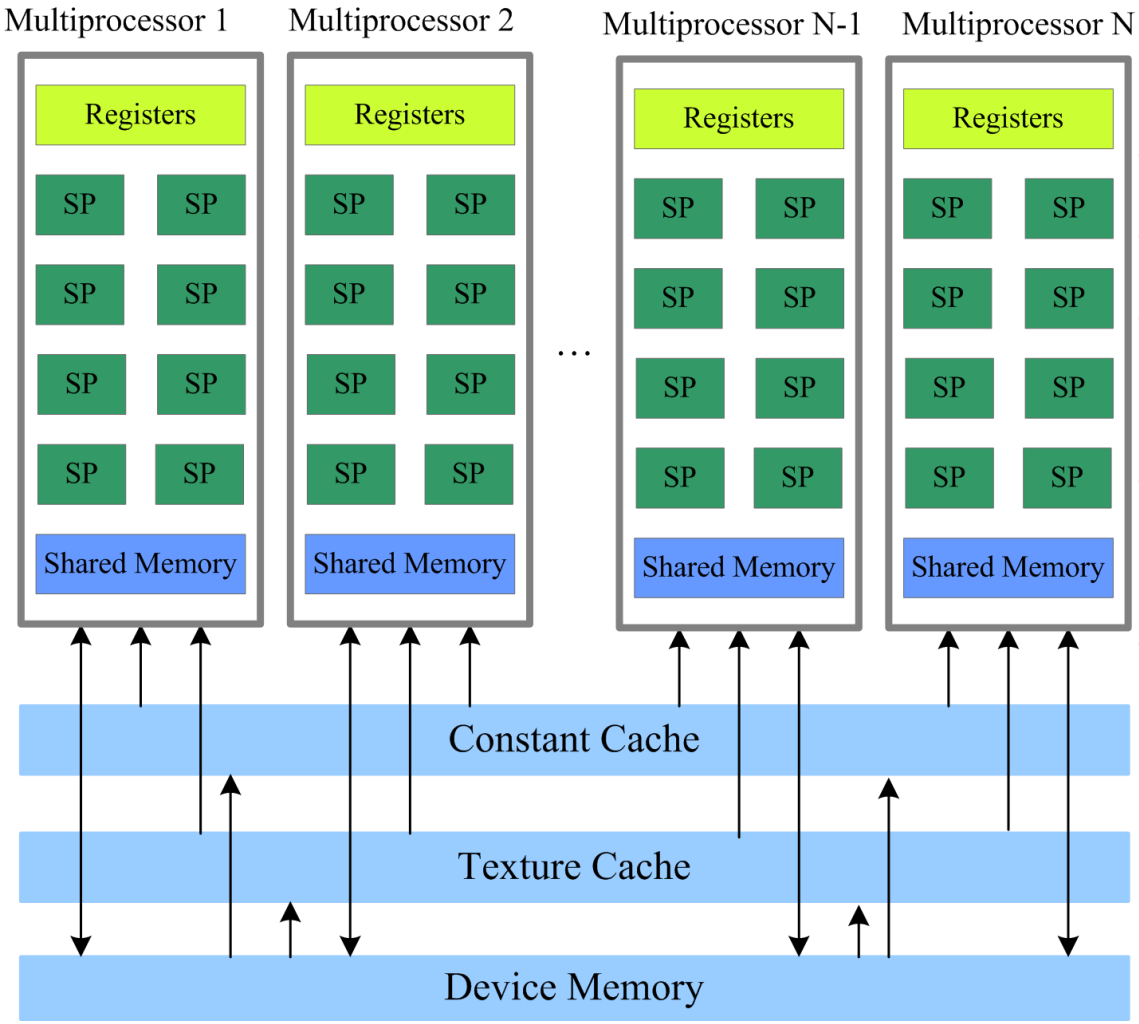


16 384 processors

2.4 Gflops (double)

1GB of memory

Are GPU really SIMD machines ?



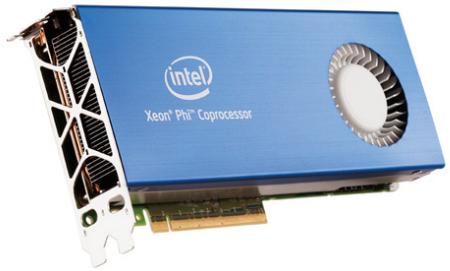
- Data parallel per multiprocessor
- Explicit memory management
- More threads than SIMD units (SP) to overlap memory accesses
- Coalesced accesses to gather global memory accesses
- Limited number of registers per Multiprocessors can limit data parallelism

Load balancing with some kind of list scheduling possible on *[Toss & al. Europar 2012]*

Intel Xeon Phi Accelerator

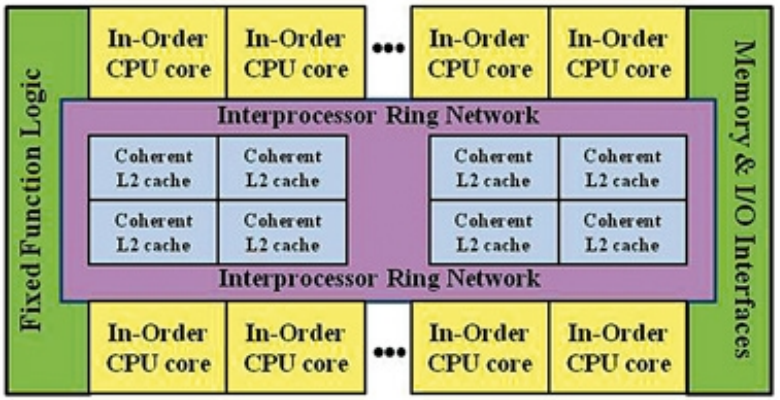
Intel Xeon Phi co-processor:

- 60 X86 cores (4 Hyperthreads per core)
- A wide vector processing engine per core
- One global memory
- Cache coherent architecture
- Connect on the PCI bus



Supported programming environments:

- MPI, OpenMP, TBB, Cilk



Today about 3 times slower than Nvidia Tesla Card, but porting an existing code is a way faster.

MIC, Larrabbe, Xeon Phi and ???

The MIC was initially introduced as a GPU called Larrabbe [Siggraph 2008], with real time ray tracing capabilities.

SC'09 prototype demo was a fiasco

Renamed Xeon Phi, it became available in 2013 as an accelerator (no video output)

Since 2016, available as a standalone processor that can be plugged in a standard Xeon socket

No more need to transfer data over the (slow) PCI Express

The first many-core processor

Conclusion

Task based programming + work stealing scheduling:

- It's becoming a standard (Intel cilk, Intel TBB, OpenMP)
- The way to go for "easy", portable and efficient parallelizations

TBB versus Cilk versus OpenMP

Make your choice. Similar base concepts but very different instantiations (we could expect Intel to unify them all at least at the runtime level)

Conclusion

Task programming versus Nvidia/OpenMP

More progressive transition from sequential to parallel code

Nvidia/OpenCL: bottom-up approach (a prog. env adapted to the hardware)

Task programming: top-dowm (an algorithm with provable performance to efficient implementations)

Xeon Phi versus GPU:

Smoother learning curve for the Phi

Higher performance for NVIDIA GPUs

Watch the Phi evolution (will fit on standard socket by 2016)

Conclusion

Get used to parallel thinking and design your algorithms with parallelization in mind. You will save a lot of time when actually parallelizing your code

Remember. If performance matters, today you cannot escape parallelization

One more thing,…….. I am not sponsored by Intel ☺

Thanks!