Work Stealing

Vincent Danjean, UJF, University of Grenoble

LIG laboratory, Vincent.Danjean@imag.fr

November 2012, 19th

Outline

Outline

Interactive parallel computation?

- Any application is "parallel":
  - composition of several programs/library procedures (possibly concurrent)
  - each procedure written independently and also possibly parallel itself
- Example:
  - Interactive distributed simulation
    3D-reconstruction, simulation, rendering
    [B. Raffin & E. Boyer]

New parallel supports

- Parallel chips & multi-core architectures:
  - MPSoCs (Multi-Processor Systems on Chips)
  - GPU : graphics processors
  - Multi-core processors (Intel, AMD)
  - Heterogeneous multi-cores: CPUs+GPUs+DSPs+FPGAs (Cell)
- Numa machines
- Clusters
- Grids

The problem

To design a single algorithm that computes efficiently a function on an arbitrary dynamic architecture

Best existing algorithms

- sequential
- parallel, $p = 2$
- parallel, $p = 100$
- parallel, $p = \max$

How to choose the best one for:

- an heterogeneous cluster
- an multi-user SMP server
- an part (not dedicated) of an existing grid

Dynamic architecture is the key

non-fixed number of resources, variable speeds, etc.

The graal : Processor-oblivious algorithms

Non-fixed number of resources, variable speeds, etc.
motivates the design of ≪ processor-oblivious ≫ parallel algorithm that:

- is independent from the underlying architechture
  - no reference to p nor to $\Pi_i(t)$ (speed of processor i at time t) nor ...
- on a given architecture, has performance guarantees
  - behaves as well as an optimal (off-line, non-oblivious) one

In some cases, work-stealing can archive these goals

Outline

Heterogeneous processors, work and depth

Processor speeds are assumed to change arbitrarily and adversarially:

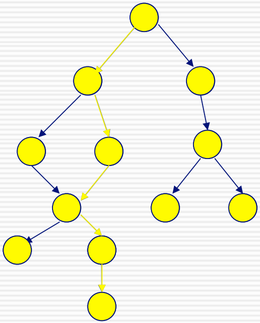
model [Bender,Rabin 02] ***$\Pi_i(t)$* = *instantaneous speed*** *of processor i at time t*

(in #unit operations per second)

Assumption : $\text{Max}_{i,t}\{\ \Pi_i(t)\ \} < constant\ .\ \text{Min}_{i,t}\{\ \Pi_i(t)\ \}$

Def: for a computation with duration T

- **total speed:** $\Pi_{tot} = (\ \Sigma_{i=0,..,P}\ \Sigma_{t=0,..,T}\ \Pi_i(t)\ )\ /\ T$
- **average speed** per processor: $\Pi_{ave} = \Pi_{tot}\ /\ P$



"Work" W = #total number operations performed

"Depth" D = #operations on a critical path

(~parallel "time" on ∞ resources)

For any greedy *maximum utilization* schedule:

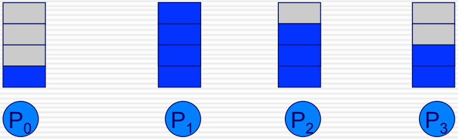
[Graham69, Jaffe80, Bender-Rabin02]

$$makespan \leq \frac{W}{p\Pi_{ave}} + \left(1 - \frac{1}{p}\right)\frac{D}{\Pi_{ave}}$$

Courtesy of Jean-Louis Roch

The work stealing algorithm

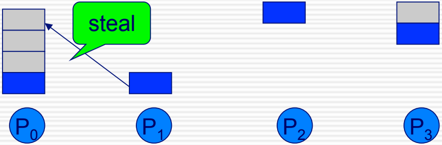
- **A distributed and randomized algorithm that computes a greedy schedule :**
  - Each processor manages a local task (depth-first execution)



Courtesy of Jean-Louis Roch

The work stealing algorithm

- **A distributed and randomized algorithm that computes a greedy schedule :**
  - Each processor manages a local stack (depth-first execution)

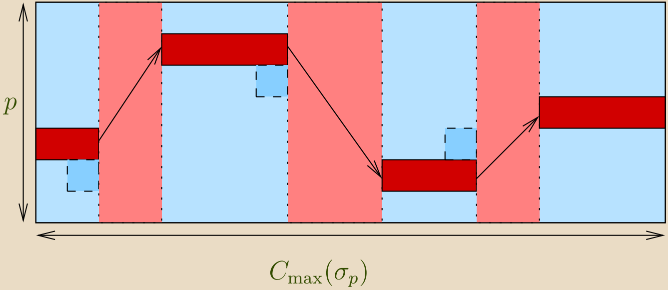


  - When idle, a processor steals the topmost task on a remote -non idle- victim processor (randomly chosen)
  - **Theorem**: With good probability, [Acar,Blelloch, Blumofe02, BenderRabin02]
    - **#steals = O(*p.D*)** **and execution time** $\leq \frac{W}{p\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$
  - **Interest**:
    if ***W* independent of *p*** and ***D* is small**, work stealing achieves **near-optimal** schedule

Courtesy of Jean-Louis Roch

Back on greedy list scheduling (Coffman result)

Proof.



Therefore, $Idle \leqslant (p-1).w(\Phi)$ for some Φ

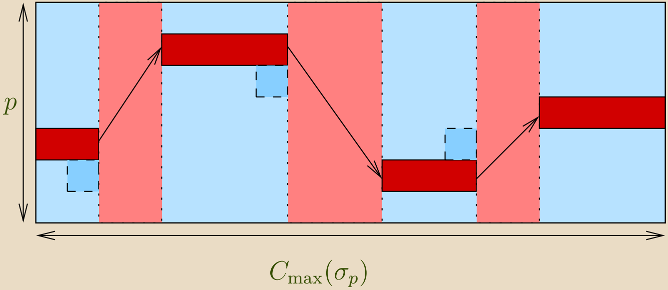
Hence,

$$
\begin{aligned}
p.C_{\max}(\sigma_p) &= Idle + Seq \leqslant (p-1)w(\Phi) + Seq \\
&\leqslant (p-1)C^*_{\max}(p) + p.C^*_{\max}(p) = (2p-1)C^*_{\max}(p)
\end{aligned}
$$

□

Back on greedy list scheduling (Coffman result)

Proof.



By definition of D, $w(\Phi) \leq D$
Hence,

$$p.C_{\max}(\sigma_p) = Idle + Seq \leqslant (p-1)D + W$$

$$T_p \leqslant \frac{W}{p} + O(D)$$

□

Warning: work-stealing is not greedy list scheduling

Even if the bound on execution time is the same, the hypothesis are not the same:

- in WS, a processor can be idle (trying to steal)
- the result for WS is "with a high probability"
- WS also gives a bound on the number of steal:

$$\#\text{Steal requests} = O\left(p.D\right) \quad \text{w.h.p.}$$

- WS works with heterogeneous processors:

$$T_p \leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$$

Outline

Work stealing implementation

Scheduling

efficient policy
(close to optimal)

control of the policy
(realisation)

Difficult in general (coarse grain) **But easy if *D* is small** [Work-stealing] Execution time $\leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$ **(fine grain)**	Expensive in general (fine grain) **But small overhead if a small number of tasks** **(coarse grain)**

*If D is small, a work stealing algorithm performs a **small number of steals***

=> ***Work-first principle***: "scheduling overheads should be borne by the critical path of the computation" [Frigo 98]

Implementation: since all tasks but a few are executed in the local stack, overhead of task creation should be as close as possible as sequential function call

At any time on any non-idle processor,
efficient local *degeneration* of the *parallel* program in a *sequential execution*

Courtesy of Jean-Louis Roch

Work-stealing implementations following the work-first principle : Cilk

- **Cilk-5** http://supertech.csail.mit.edu/cilk/ **: C extension**
  - **Spawn** f (a) ; **sync** (serie-parallel programs)
  - Requires a shared-memory machine
  - Depth-first execution with synchronization (on sync) with the end of a task :
    - Spawned tasks are pushed in double-ended queue
  - "Two-clone" compilation strategy [Frigo-Leiserson-Randall98] :
    - on a successfull steal, a thief executes the continuation on the topmost ready task ;
    - When the continuation hasn't been stolen, "sync" = nop ; else synchronization with its thief

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
       int x, y;

       x = spawn fib (n-1);
       y = spawn fib (n-2);

       sync;

       return (x+y);
    }
}
```

```
int fib (int n)
{
    fib_frame *f;                  frame pointer
    f = alloc(sizeof(*f));         allocate frame
    f->sig = fib_sig;              initialize frame
    if (n<2) {
        free(f, sizeof(*f));       free frame
        return n;
    }
    else {
        int x, y;
        f->entry = 1;              save PC
        f->n = n;                  save live vars
        *T = f;                    store frame pointer
        push();                    push frame
        x = fib (n-1);             do C call
        if (pop(x) == FAILURE)     pop frame
            return 0;              frame stolen
        ...                        second spawn
        ;                          sync is free!
        free(f, sizeof(*f));       free frame
        return (x+y);
    }
}
```

- *won the 2006 award "Best Combination of Elegance and Performance" at HPC Challenge Class 2, SC'06, Tampa, Nov 14 2006 [Kuszmaul] on SGI ALTIX 3700 with 128 bi-Ithanium]*

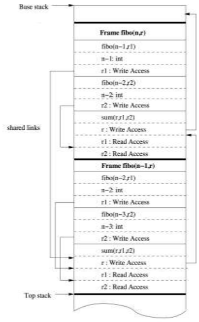
Courtesy of Jean-Louis Roch

Work-stealing implementations following the work-first principle : KAAPI

- **Kaapi / Athapascan** http://kaapi.gforge.inria.fr **:** C++ library
  - **Fork**<f>()(a, …) with **access mode** to parameters (value;read;write;r/w;cw) **specified in f prototype** (macro dataflow programs)
  - Supports distributed and shared memory machines; heterogeneous processors
  - Depth-first (*reference order*) execution with synchronization on data access :
    - Double-end queue (mutual exclusion with compare-and-swap)
    - on a successful steal, one-way data communication (write&signal)

```
struct sum {
   void operator()(Shared_r < int > a,
                   Shared_r < int > b,
                   Shared_w < int > r )
   { r.write(a.read() + b.read()); }
 } ;

 struct fib {
  void operator()(int n, Shared_w<int> r)
  { if (n <2) r.write( n );
    else
    { int r1, r2;
      Fork< fib >() ( n-1, r1 ) ;
      Fork< fib >() ( n-2, r2 ) ;
      Fork< sum >() ( r1, r2, r ) ;
    }
  }
} ;
```



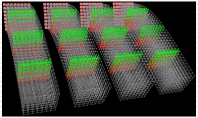
(b) Shared links

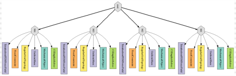
- *Kaapi won the 2006 award "Prix special du Jury" for the best performance at NQueens contest, Plugtests-Grid&Work'06, Nice, Dec.1, 2006 [Gautier-Guelton] on Grid'5000 1458 processors with different speeds.*

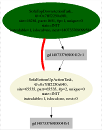
Courtesy of Jean-Louis Roch

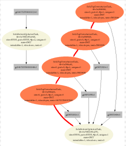
Experimental results on SOFA [CIMIT-ETZH-INRIA]

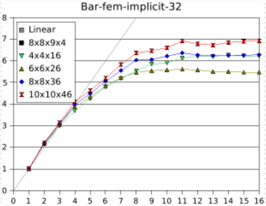
[Allard 06]

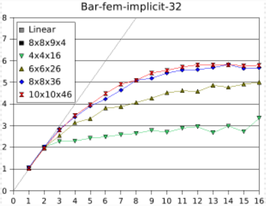










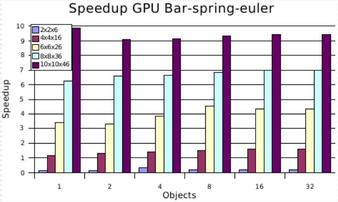


Kaapi (C++, ~500 lines)

Cilk (C, ~240 lines)

Preliminary results on GPU NVIDIA 8800 GTX

- speed-up ~9 on Bar 10x10x46 to Athlon64 2.4GHz
  - 128 "cores" in 16 groups
  - CUDA SDK : "BSP"-like, 16 X [16 .. 512] threads
  - Supports most operations available on CPU
  - ~2000 lines CPU-side + 1000 GPU-side



Courtesy of Jean-Louis Roch

Outline

Algorithm Design

$$T_p \leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$$

- from WS theorem, optimizing the execution time by building a parallel algorithm with both:
  - $W = T_{seq}$

  and
  - small depth D
- Double criteria
  - minimum work W: ideally T_{seq}
  - Small depth D: ideally polylog in the work: $D = O\left(\log^{O(1)} W\right)$

Cascading Divide & Conquer

- $W(n) \le a.W\left(\frac{n}{K}\right) + f(n)$ with $a > 1$
  - if $f(n) \ll n^{\log_k a}$ then $W(n) = O\left(n^{\log_k a}\right)$
  - if $f(n) \gg n^{\log_k a}$ then $W(n) = O\left(f(n)\right)$
  - if $f(n) = \Theta\left(n^{\log_k a}\right)$ then $W(n) = O\left(f(n)\log n\right)$
- $D(n) = D\left(\frac{n}{K}\right) + f(n)$
  - if $f(n) = O\left(\log^i n\right)$ then $D(n) = O\left(\log^{i+1} n\right)$

Cascading Divide & Conquer

- $W(n) \leq a.W\left(\frac{n}{K}\right) + f(n)$ with $a > 1$
  - if $f(n) \ll n^{\log_k a}$ then $W(n) = O\left(n^{\log_k a}\right)$
  - if $f(n) \gg n^{\log_k a}$ then $W(n) = O\left(f(n)\right)$
  - if $f(n) = \Theta\left(n^{\log_k a}\right)$ then $W(n) = O\left(f(n) \log n\right)$
- $D(n) = D\left(\frac{n}{K}\right) + f(n)$
  - if $f(n) = O\left(\log^i n\right)$ then $D(n) = O\left(\log^{i+1} n\right)$
- $D(n) = D\left(\sqrt{n}\right) + f(n)$
  - if $f(n) = O(1)$ then $D(n) = O\left(\log \log n\right)$
  - if $f(n) = O\left(\log n\right)$ then $D(n) = O\left(\log n\right)$

function $\textsc{MergeSort}(A, i, j)$
    if $i < j$ **then**
        $k \leftarrow \frac{i+j}{2}$
        spawn $\textsc{MergeSort}(A, i, k)$
        $\textsc{MergeSort}(A, k+1, j)$
        sync
        $\textsc{Merge}(A, i, k, j)$
    end if
end function

- $W(n) =$
- $D(n) =$
- $T_p(n) =$

Example: MergeSort with Cilk

```
function MergeSort(A,i,j)
    if i < j then
        k ← (i+j)/2
        spawn MergeSort(A,i,k)
        MergeSort(A,k+1,j)
        sync
        Merge(A,i,k,j)
    end if
end function
```

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta\left(n\right) =$
- $D(n) =$
- $T_p(n) =$

Example: MergeSort with Cilk

function MERGESORT(A,i,j)
    if $i < j$ **then**
        $k \leftarrow \frac{i+j}{2}$
        spawn MERGESORT(A,i,k)
        MERGESORT(A,$k+1$,j)
        sync
        MERGE(A,i,k,j)
    end if
end function

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$
- $D(n) =$
- $T_p(n) =$

Example: MergeSort with Cilk

function $\textsc{MergeSort}(A,i,j)$
    if $i < j$ **then**
        $k \leftarrow \frac{i+j}{2}$
        spawn $\textsc{MergeSort}(A,i,k)$
        $\textsc{MergeSort}(A,k+1,j)$
        sync
        $\textsc{Merge}(A,i,k,j)$
    end if
end function

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n \log n\right)$
- $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(n\right) =$
- $T_p(n) =$

Example: MergeSort with Cilk

```
function MergeSort(A,i,j)
    if i < j then
        k ← (i+j)/2
        spawn MergeSort(A,i,k)
        MergeSort(A,k+1,j)
        sync
        Merge(A,i,k,j)
    end if
end function
```

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n \log n\right)$
- $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n\right)$
- $T_p(n) =$

Example: MergeSort with Cilk

```
function MergeSort(A,i,j)
    if i < j then
        k ← (i+j)/2
        spawn MergeSort(A,i,k)
        MergeSort(A,k+1,j)
        sync
        Merge(A,i,k,j)
    end if
end function
```

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n \log n\right)$
- $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n\right)$
- $T_p(n) = \Theta\left(\frac{n \log n}{p}\right) + \Theta\left(n\right)$

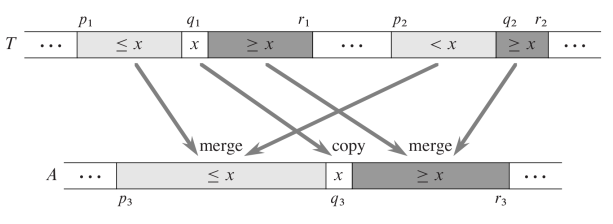
Example: MergeSort with Cilk

```
function MergeSort(A,i,j)
    if i < j then
        k ← (i+j)/2
        spawn MergeSort(A,i,k)
        MergeSort(A,k+1,j)
        sync
        Merge(A,i,k,j)
    end if
end function
```

- $W(n) = 2W\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n \log n\right)$
- $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(n\right) = \Theta\left(n\right)$
- $T_p(n) = \Theta\left(\frac{n \log n}{p}\right) + \Theta\left(n\right)$

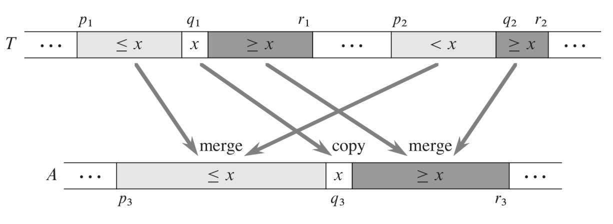
If $m > \log n$, T_p is lead by the last merge in $\Theta\left(n\right)$

MergeSort with Parallel Merge



- more parallelism required (in Merge)
  - we take the median element of the first array
  - we look its position by dichotomy in the second array
  - we merge in parallel the four sub-arrays (two by two)

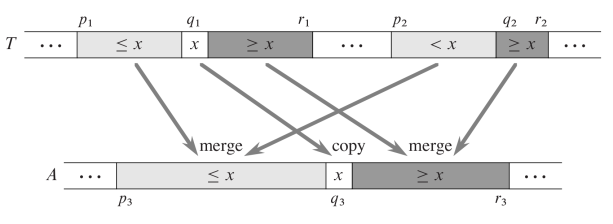
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) =$
  - $D(n) =$

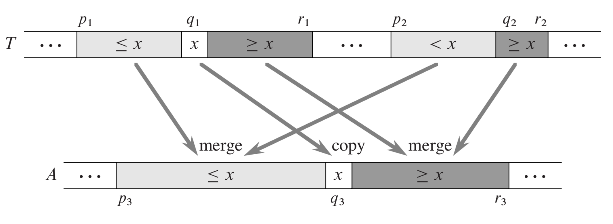
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) =$
  - $D(n) =$

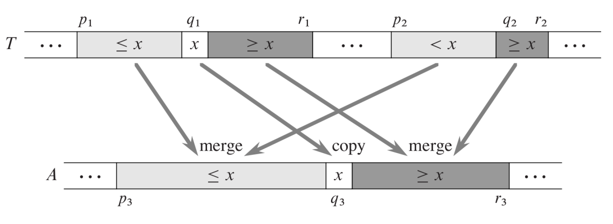
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) =$

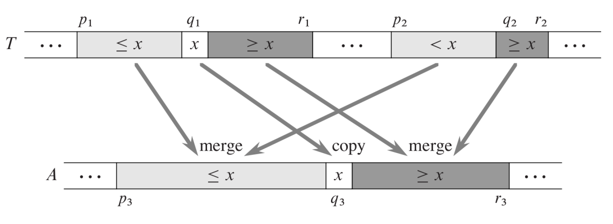
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) =$

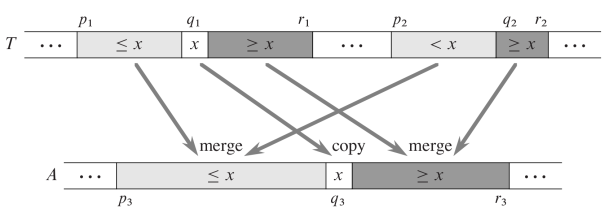
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) = \Theta(\log^2 n)$

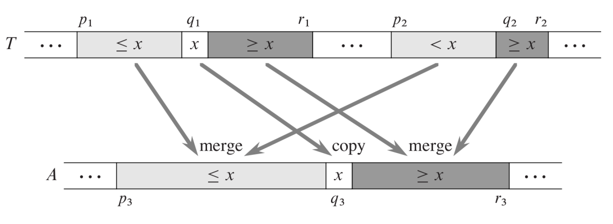
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) = \Theta(\log^2 n)$
- Back in MergeSort
  - $D(n) =$
  - $T_p(n) =$

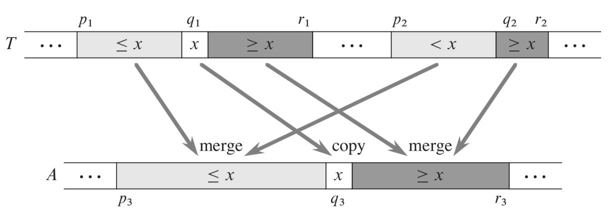
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) = \Theta\left(\log^2 n\right)$
- Back in MergeSort
  - $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) =$
  - $T_p(n) =$

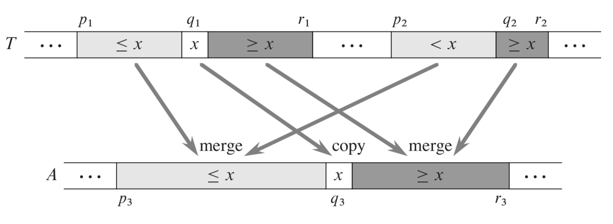
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) = \Theta(\log^2 n)$
- Back in MergeSort
  - $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) = \Theta\left(\log^3 n\right)$
  - $T_p(n) =$

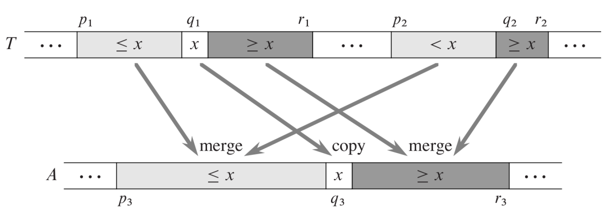
MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) = \Theta(\log^2 n)$
- Back in MergeSort
  - $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) = \Theta\left(\log^3 n\right)$
  - $T_p(n) = \Theta\left(\frac{n \log n}{p}\right) + \Theta\left(\log^3 n\right)$

MergeSort with Parallel Merge



- For the parallel merge
  Let n_1 and n_2 the number of elements $< x$ and $> x$

  $n = n_1 + n_2 + 1$ and $n_1 \geq n/4$ and $n_2 \geq n/4$
  - $W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$
  - $D(n) = \max(D(n_1), D(n_2)) + \Theta(\log n) = \Theta(\log^2 n)$
- Back in MergeSort
  - $D(n) = D\left(\frac{n}{2}\right) + \Theta\left(\log^2 n\right) = \Theta\left(\log^3 n\right)$
  - $T_p(n) = \Theta\left(\frac{n \log n}{p}\right) + \Theta\left(\log^3 n\right)$
- Can be improved ($D(n) = \Theta(\log n)$)

Outline

Conclusion

- Work Stealing concerns a wide-range of algorithms
- WS has some proven performances with weak hypothesis
  - heterogeneous processors
    but related speeds (WS model not valid for CPU/GPU)
  - etc.
- Still, algorithms must be carefully designed
  - how to split the work ?
  - in how many parts (fraction ?, root square ?, etc.)
- Efficient implementation of WS is not trivial