Programming in the Multicore Era

Kornilios Kourtis

kkourt@cslab.ece.ntua.gr

Computing Systems Laboratory
National Technical University of Athens

July 9, 2010

Programming language trends

make the programmer's life easier:
(and the code **less error-prone**)

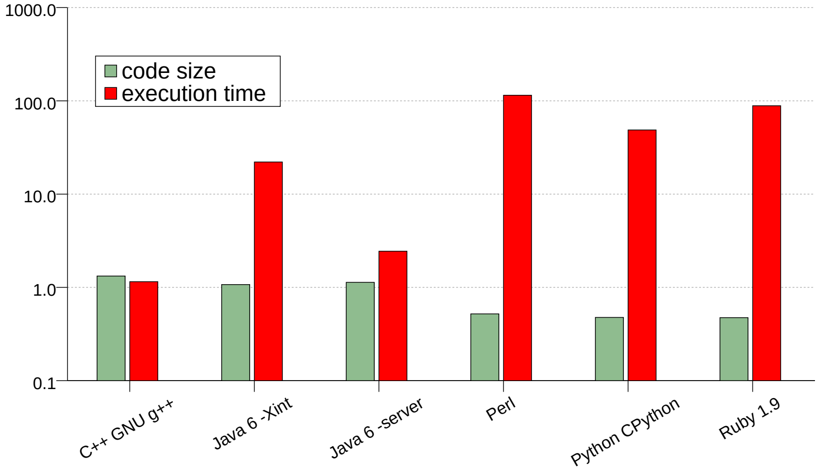
- garbage collection
- array bounds check
- everything is an object (even integers!)
- dynamic typing

The result:

+ It is easier to write programs
- significant run-time overheads
  $\Rightarrow$ performance degradation

Programming language trends

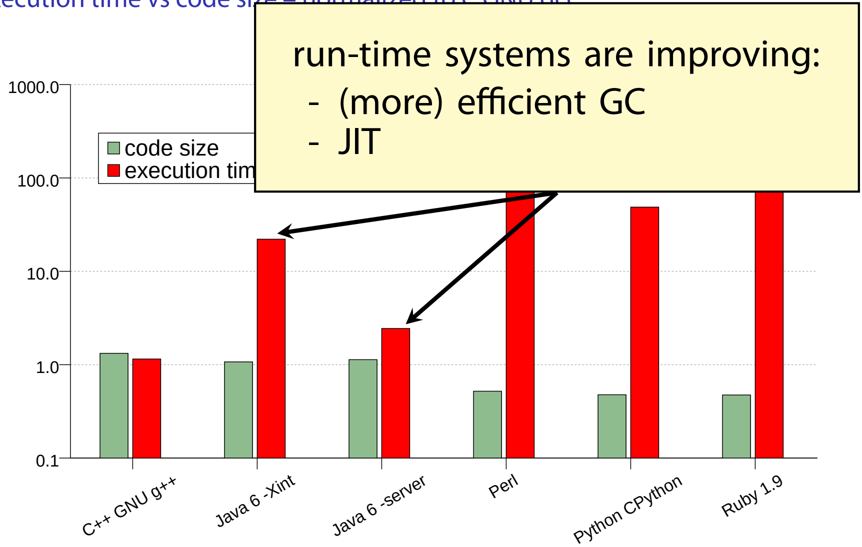
execution time vs code size – normalized to C GNU gcc



Computer language benchmarks game
(http://shootout.alioth.debian.org/, 21/06/10)

Programming language trends

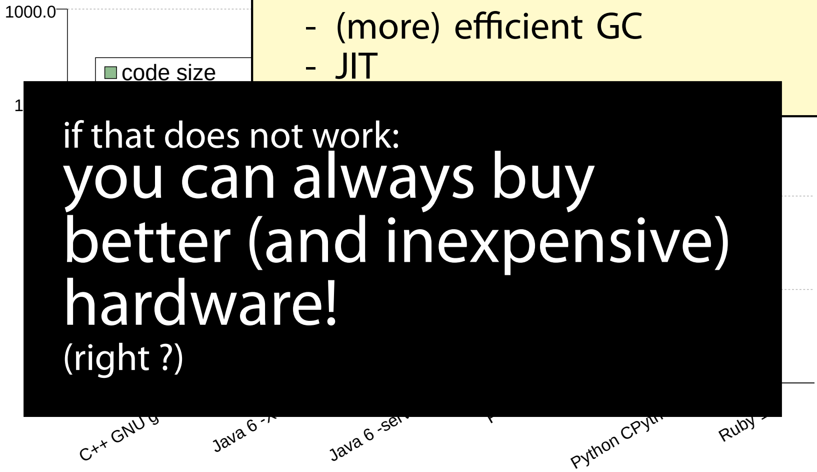
execution time vs code size – normalized to C GNU gcc



Computer language benchmarks game
(http://shootout.alioth.debian.org/, 21/06/10)

Programming language trends

execution time vs code size – normalized to C GNU gcc



run-time systems are improving:

- (more) efficient GC
- JIT

if that does not work:

you can always buy better (and inexpensive) hardware!

(right ?)

Computer language benchmarks game
(http://shootout.alioth.debian.org/, 21/06/10)

The free lunch

exponential performance improvement

- Moore's law: exponentential increase in number of transistors
- Up until recently, exponential increase in CPU performance
  (frequency scaling, ILP exploitation)

The free lunch **is over!**

- Moore's law: exponentential increase in number of transistors
- Up until recently, exponential increase in CPU performance
  (frequency scaling, ILP exploitation)

but:

- architects hit hard limits (power, available ILP)
- Moore's law is inadequate for improving serial performance
- solution: **multicore CPUs**
  (use extra transitors for multiple cores)

the "Multicore Era"

where only parallel programs benefit from new hw!

difficulties:

- reasoning about parallel execution is harder
  (e.g., data races)
- parallel programming is an esoteric art
- absence of tools
  (programming languages, debuggers, profilers)

so:

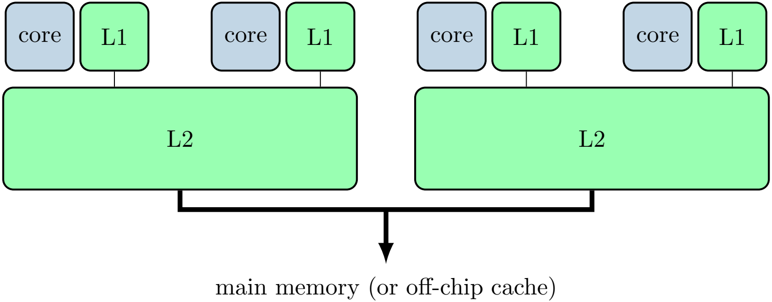
- effort to make parallel programming **easier**
  (and less **error-prone**)
- emerging parallel languages and paradigms

Outline

- **Introduction**
- Expressing parallelism
- Algorithmic concerns
- Cooperation

Multicore designs

current:

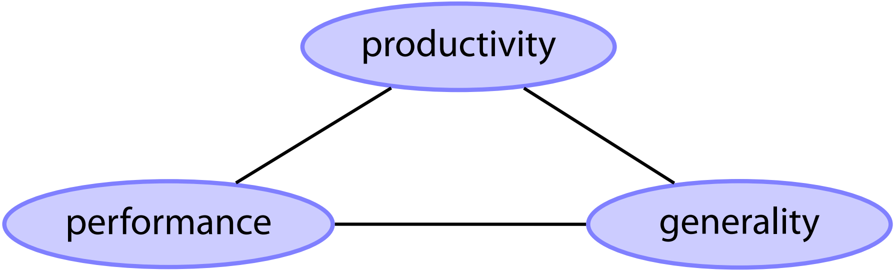


future:

- manycore
- heterogeneous

Goals of parallel programming

[McKenney et. al. '09]



- No silver bullet! (pick 2 out of 3)
- performance predictability
- language approach: give constructs for both generic and productive

Parallel languages

- Why not a library ?
  - compiler/run-time system awareness
- Parallelism
  - explicit
  - implicit
  - semi-implicit
  - retain serial semantics
- Languages
  - openmp, cilk
  - erlang, scala
  - clojure, haskell
  - chapel, fortress
  - ...

Outline

- Introduction
- **Expressing parallelism**
  - **data parallelism**
  - **task parallelism**
- Algorithmic concerns
- Cooperation

Basic concepts

expressing parallelism: partition work
work must be split in **tasks** that can execute in parallel

- **scheduling**: mapping of tasks into resources (e.g., CPUs)
  - balancing (static,dynamic)
  - run-time system
- task granularity — how much work a task performs ?
  - too fine $\longrightarrow$ large overhead
  - too coarse $\longrightarrow$ not enough **parallel slack**

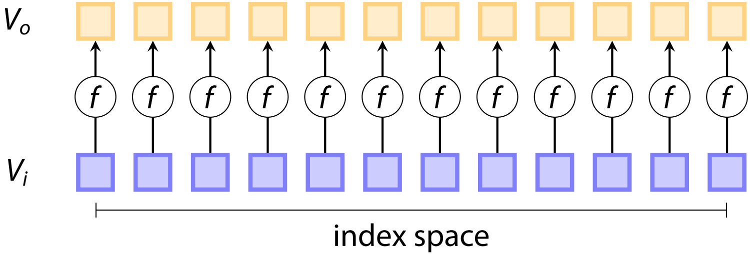
Expressing parallelism

parallel programming paradigms

- **Data parallel**
  An operation is applied simultaneously to an aggregate of individual items (e.g., arrays).

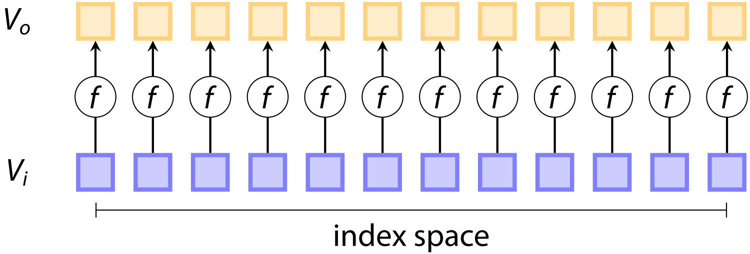
- **Task parallel**
  User explicitly defines parallel tasks.

vector map
(silly) data parallel example



- each operation (f) can be performed in parallel
- work partitioning $\leftrightarrow$ index partitioning

vector map
(silly) data parallel example

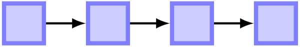


- each operation (f) can be performed in parallel
- work partitioning $\leftrightarrow$ index partitioning

- efficient parallelization requires
  efficient partitioning of aggregate structures

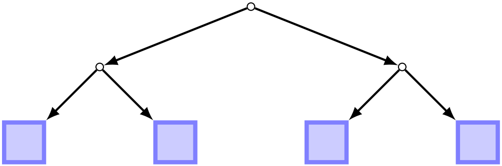
partitioning of aggregate structures

- linked lists: ☹



- arrays: ☺

- trees (if balanced): ☺



simple data parallel language constructs

... that work by partitioning the index space

- map in Data Parallel Haskell:

```
Prelude GHC.PArr> A
[:40,40,40,40:]
Prelude GHC.PArr> mapP (\x -> x + 2) A
[:42,42,42,42:]
```

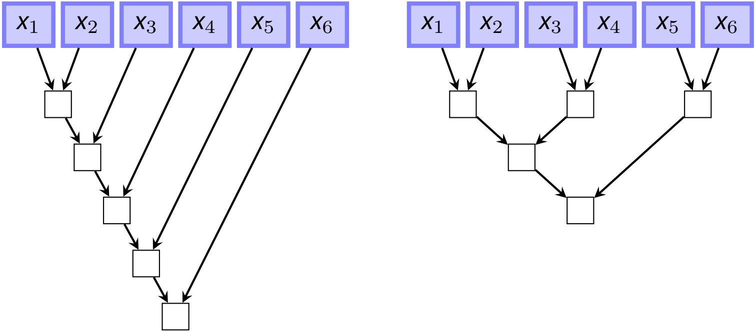
- scalar promotion in Chapel:

```
C = A + B*3
```

- comprehensions in Fortress:
  $s = \{x/2 \mid x \leftarrow t\}$

reductions

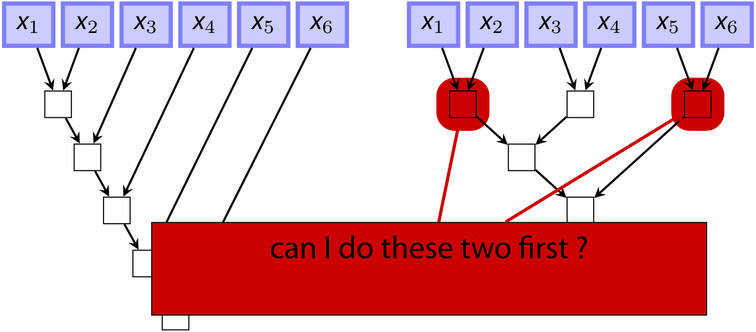
- reduction on an **associative** operation
  (e.g., $+$ for producing sums)



- based on index space partitioning

reductions

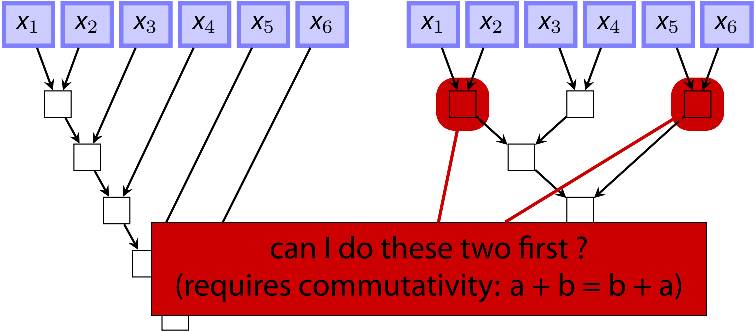
- reduction on an **associative** operation
  (e.g., $+$ for producing sums)



- based on index space partitioning

reductions

- reduction on an **associative** operation
  (e.g., $+$ for producing sums)



- based on index space partitioning

reduction support on languages

- OpenMP:
  - reduction over a list of specific operators
- fortress, chapel:
  - (will) support reductions on user-defined operators
  - must be associative to allow parallelization
  - different operator types
    (e.g., better parallelization with commutativity)

- similar operation: prefix scans

parallel for construct

parallelization of iteration space

```
#pragma omp parallel for /* OpenMP parallel for */
for (i=1; i<N; i++){
    B[i] = (A[i] + A[i-1])/2.0;
}
```

- **parallel for**: iterations can be executed in parallel
  (*forall* in chapel, *for* in fortress, ...)
- work partion $\rightarrow$ partition iteration space
- more flexibility on expressing an algorithm

parallel for construct

parallelization of iteration space

```
forall (i,j,k) in [1..n,1..n,1..n] do
    C[i][j] += A[i][k] * B[k][j];
```

- **parallel for**: iterations can be executed in parallel (*forall* in chapel, *for* in fortress, ...)
- work partion $\rightarrow$ partition iteration space
- more flexibility on expressing an algorithm
- iteration space can have > 1 dimensions

parallel for construct

parallelization of iteration space

```
#pragma omp parallel for /* OpenMP parallel for */
for (i=2; i<N; i++){
    factorial[i] = i*factorial[i-1];
}
```

- **parallel for**: iterations can be executed in parallel (*forall* in chapel, *for* in fortress, ...)
- work partion $\rightarrow$ partition iteration space
- more flexibility on expressing an algorithm
- iteration space can have > 1 dimensions
- programmer must avoid data races

Data parallelism

Advanced issues:

- index space not necessary regular
  (e.g., associative arrays)
- nested data parallel structures
  (NESL, DP Haskell)
- locality concerns

In conclusion:

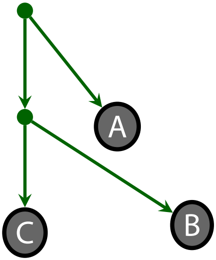
\+ performance, productivity

\- not general

Task parallelism

- user explicitly defines parallel tasks (task graph)
- generic (but not always productive)
- user defines:
  - task creation points

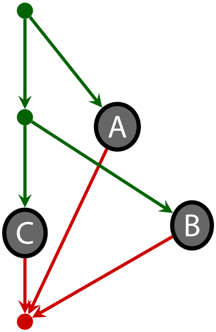
```
/* Cilk example */
x = spawn A();
y = spawn B();
z = C();
```



Task parallelism

- user explicitly defines parallel tasks (task graph)
- generic (but not always productive)
- user defines:
  - task creation points
  - task synchronization points

```
/* Cilk example */
x = spawn A();
y = spawn B();
z = C();
sync;
/* x,y are available */
```



The task graph unfolds dynamically

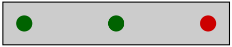
(in the general case ...)

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

The task graph unfolds dynamically

(in the general case ...)

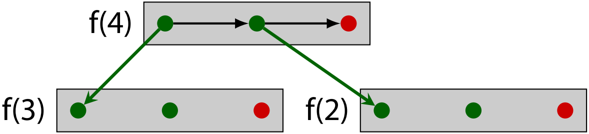
```
cilk int fib(int n) {
    if (n < 2) return (n);
    x = spawn fib(n - 1); y = spawn fib(n - 2);
    sync;
    return (x + y);
}
```

f(4) 

The task graph unfolds dynamically

(in the general case ...)

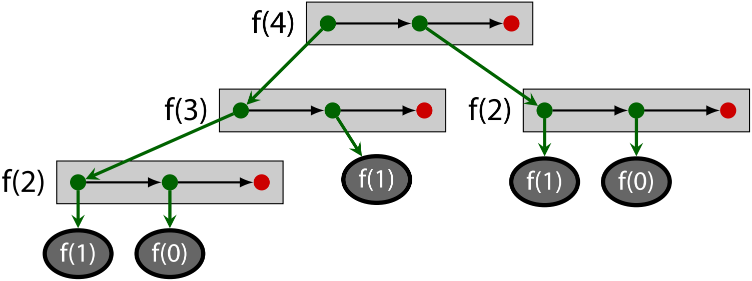
```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1); y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```



The task graph unfolds dynamically

(in the general case ...)

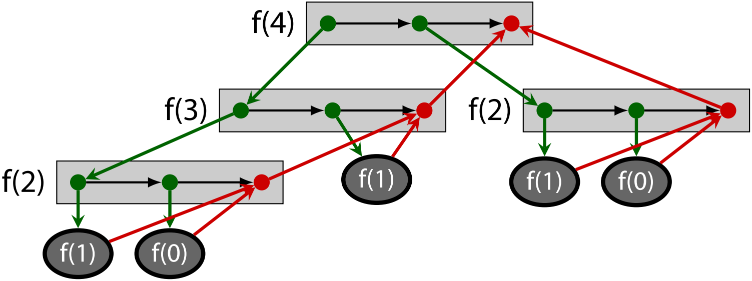
```
cilk int fib(int n) {
     if (n < 2) return (n);
     x = spawn fib(n - 1); y = spawn fib(n - 2);
     sync;
     return (x + y);
}
```



The task graph unfolds dynamically

(in the general case ...)

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



parallel quicksort

divide & conquer algorithms can be easily parallelized

```
def qsort(arr, low, high){
    if high == low
        return;
    pivotVal = findPivot();
    pivotLoc = partition(pivotVal);
    qsort(arr, low, pivotLoc-1);
    qsort(arr, pivotLoc+1, high);

}
```

parallel quicksort

divide & conquer algorithms can be easily parallelized

```
def qsort(arr, low, high){
    if high == low
        return;
    pivotVal = findPivot();
    pivotLoc = partition(pivotVal);
    spawn qsort(arr, low, pivotLoc-1);
    qsort(arr, pivotLoc+1, high);
    sync;
}
```

- recursive splitting

D&C vs accumulators

(conclusion points from Guy Steele's talk at ICFP '09)

DONTs:

- use linked lists (even arrays are suspect)
- use accumulators
  - split a problem into the "first" and the "rest"

DOs:

- use trees
- use D&C:
  - split a problem
  - recursively solve sub-problems
  - combine solutions *

D&C vs accumulators

(conclusion points from Guy Steele's talk at ICFP '09)

DONTs:

- use linked lists (even arrays are suspect)
- use accumulators
  - split a problem into the "first" and the "rest"

DOs:

- use trees
- use D&C:
  - split a problem
  - recursively solve sub-problems
  - combine solutions *

* usually trickier than incremental update of a single solution

Example: Run-length encoding

a,a,a,a,b,b,b,c,c,c,c,c $\rightarrow$ (a,4), (b,3), (c,5)

```
def rle(xs):
  ret,curr,freq = ([],xs[0],1)
  for item in xs[1:]:
    if item == curr:
     freq += 1
    else:
      ret.append((curr,freq))
      curr,freq = (item,1)
  ret.append((curr,freq))
  return ret
```

Example: Run-length encoding

a,a,a,a,b,b,b,c,c,c,c,c $\rightarrow$ (a,4), (b,3), (c,5)

```
def rle(xs):
  ret,curr,freq = ([],xs[0],1)
  for item in xs[1:]:
    if item == curr:
     freq += 1
    else:
      ret.append((curr,freq))
      curr,freq = (item,1)
  ret.append((curr,freq))
  return ret
```

```
def rle_rec(xs):
  if len(xs) <= 1:
    return [(xs[0], 1)]
  mid = len(xs) // 2
  rle1 = rle_rec(xs[:mid])
  rle2 = rle_rec(xs[mid:])
  return rle_conc(rle1, rle2)
```

```
def rle_conc(rle1,rle2):
 if rle1[-1][0] == rle2[0][0]:
   r1, rle1 = rle1[-1], rle1[:-1]
   r2, rle2 = rle2[0], rle2[1:]
   rle1.append((r1[0],r1[1] + r2[1]))
 return rle1 + rle2
```

Example: RLE recursive splitting

a	a	a	a	b	b	b	c	c	c	c	c

Example: RLE recursive splitting

a	a	a	a	b	b

b	c	c	c	c	c

Example: RLE recursive splitting

a a a	a b b	b c c	c c c

Example: RLE recursive splitting

a	a	a	a	b	b	b	c	c	c	c	c
(a,1)	(a,1)	(a,1)	(a,1)	(b,1)	(b,1)	(b,1)	(c,1)	(c,1)	(c,1)	(c,1)	(c,1)

Example: RLE recursive splitting

a a	a	a b	b	b c	c	c c	c

```
(a,1)(a,1)(a,1)(a,1)(b,1)(b,1)(b,1)(c,1)(c,1)(c,1)(c,1)(c,1)
   ↘ ↙         ↘ ↙         ↘ ↙         ↘ ↙
  (a,2)    (a,1),(b,1)  (b,1),(c,1)   (c,2)
```

Example: RLE recursive splitting

a a a	a b b	b c c	c c c

```
(a,1)(a,1)(a,1)(a,1)(b,1)(b,1)(b,1)(c,1)(c,1)(c,1)(c,1)(c,1)

   (a,2)        (a,1),(b,1)    (b,1),(c,1)        (c,2)

      (a,3)        (a,1),(b,2)    (b,1),(c,2)         (c,3)
```

Example: RLE recursive splitting

a	a	a	a	b	b

b	c	c	c	c	c

```
(a,1)(a,1)(a,1)(a,1)(b,1)(b,1)(b,1)(c,1)(c,1)(c,1)(c,1)(c,1)

(a,2)   (a,1),(b,1)   (b,1),(c,1)   (c,2)

(a,3)   (a,1),(b,2)   (b,1),(c,2)   (c,3)

(a,4),(b,2)           (b,1),(c,5)
```

Example: RLE recursive splitting

a a a a b b b c c c c c

(a,1) (a,1) (a,1) (a,1) (b,1) (b,1) (b,1) (c,1) (c,1) (c,1) (c,1) (c,1)

(a,2) (a,1),(b,1) (b,1),(c,1) (c,2)

(a,3) (a,1),(b,2) (b,1),(c,2) (c,3)

(a,4),(b,2) (b,1),(c,5)

(a,4),(b,3),(c,5)

Example: RLE recursive splitting

a	a	a	a	b	b	b	c	c	c	c	c

```
(a,1)(a,1)(a,1)(a,1)(b,1)(b,1)(b,1)(c,1)(c,1)(c,1)(c,1)(c,1)

  (a,2)        (a,1),(b,1)      (b,1),(c,1)          (c,2)

     (a,3)         (a,1),(b,2)      (b,1),(c,2)          (c,3)

         (a,4),(b,2)                        (b,1),(c,5)
```

(a,4),(b,3),(c,5)

- data structure for (efficient) rle concatenation

Example: RLE recursive splitting

a	a	a	a	b	b	b	c	c	c	c	c

```
(a,1)(a,1)(a,1)(a,1)(b,1)(b,1)(b,1)(c,1)(c,1)(c,1)(c,1)(c,1)

   (a,2)           (a,1),(b,1)     (b,1),(c,1)         (c,2)

      (a,3)           (a,1),(b,2)     (b,1),(c,2)         (c,3)

            (a,4),(b,2)                        (b,1),(c,5)
                      (a,4),(b,3),(c,5)
```

- data structure for (efficient) rle concatenation
- rle concatenation is associative $\rightarrow$ reduction

Outline

- Expressing parallelism
  - data parallel
    - parallel for
    - reductions
  - task parallel
    - recursive splitting
- Algorithmic concerns
  - Divide and conquer
- **Cooperation of tasks**
  - support for generic parallelization
  - sharing data
  - message passing

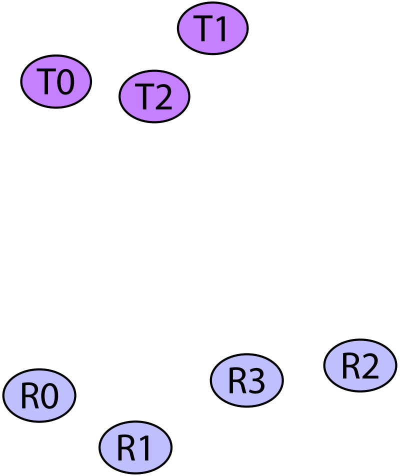
Data sharing

- shared memory architectures allow *data sharing*.
- applications can utilize it
  examples:
  - one task per request on a network server
  - tasks implementing different functionalities
    (e.g., workers, logger, balancer, I/O)
  - parallel tasks that operate on irregular data structures
- **but:** concurrent accesses may lead to inconsistencies
  (e.g., concurrent updates on a linked list)
- **solution:** mutual exclusion (locks).

Locks

mutual exclusion

- Model:
  - T: Tasks
  - R: Resources

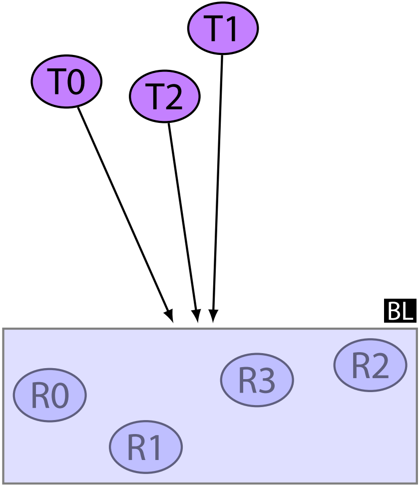


Locks

mutual exclusion

- Model:
  - T: Tasks
  - R: Resources

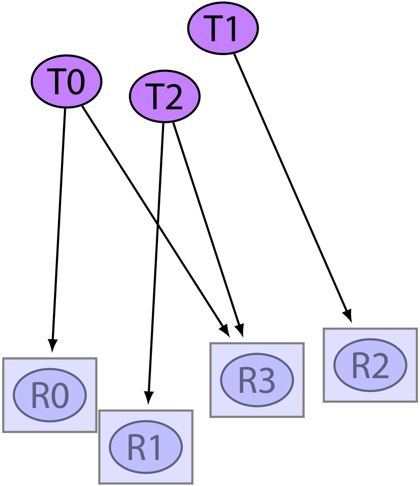
- Big Lock:
  - one lock for all
  - poor scalability



Locks

mutual exclusion

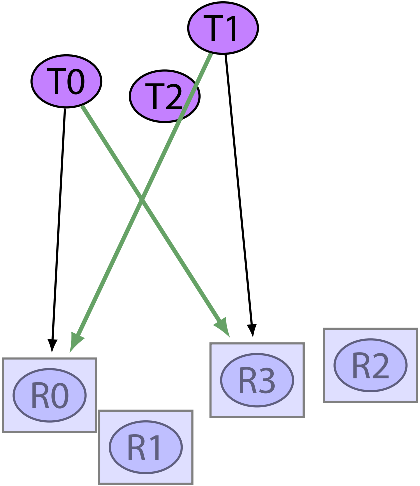
- Model:
  - T: Tasks
  - R: Resources
- Big Lock:
  - one lock for all
  - poor scalability
- Fine-grain locking:
  - one lock per R
  - possible deadlock
  - global order of Rs



Locks

mutual exclusion

- Model:
  - T: Tasks
  - R: Resources
- Big Lock:
  - one lock for all
  - poor scalability
- Fine-grain locking:
  - one lock per R
  - possible deadlock
  - global order of Rs



Locks are ~~hard~~ impossible

(...for application programmers)

- Ensuring ordering (and correctness) is **really hard** (even for advanced programmers).
  - rules are ad-hoc, and not part of the program (documented in comments at best-case scenario)
- Locks are not composable
  - how *n* thread-safe operations are combined ?
  - internal details about locking are required
  - hard for self-contained systems (e.g., OS kernel)
  - almost impossible for application programmers
- moreover, locks are pessimistic
  - worst is assumed
  - performance overhead paid every time

Composition example

atomic transfer of an element from queue to another

- lock solution:
  - ugly

    (intention of progammer is hidden)
  - internals exposed
  - broken (deadlock)

```
qXfer(q1, q2) {
  q1.lock()
  q2.lock()
  v = q1.dequeue()
  q2.enqueue(v)
  q2.unlock()
  q1.unlock()
}
```

Composition example

atomic transfer of an element from queue to another

- lock solution:
  - ugly

    (intention of progammer is hidden)
  - internals exposed
  - broken (deadlock)

- what the programmer **really** meant to say: *do this attomically*

```
qXfer(q1, q2) {
  atomic {
    v = q1.dequeue()
    q2.enqueue(v)
  }
}
```

Transactional Memory

User explicitly defines atomic code sections

- easier and less **error-prone**
- higher semantics
- composable
- analogy to garbage collection
  [Grossman 2007]
- optimistic

Transactional Memory approaches

- Hardware TM

  (currently, no wide-available hw implementation)

- Software TM
  - imperative (e.g., fortress, chapel):
    definition of atomic blocks
  - functional (e.g., Haskell, Clojure):
    Special types for shared variables, that can be accessed **only** via transactions.

- Hybrid TM

Transactional memory conclusion

When sharing data accross different parallel tasks:

- locks are unusable for application writers
- TM the best solution at the moment
    - yet, still a long way to go

Transactional memory conclusion

When sharing data accross different parallel tasks:

- locks are unusable for application writers
- TM the best solution at the moment
  - yet, still a long way to go

but: why share data ?

Message passing

- No data sharing!

- Parallel tasks exchange messages to cooperate.

  Usage example:

  - one task per external request (e.g., in a server)
  - on task per shared resource (e.g., cache)

Message passing approaches

- Erlang
  - Actor model
  - asynchronous messages to tasks
    (less prone to deadlocks)
  - pattern matching
  - registration
- Scala
  - similar to erlang
  - supports synchronous messages

Message passing approaches

- Erlang
  - Actor model
  - asynchronous messages to tasks
    (less prone to deadlocks)
  - pattern matching
  - registration
- Scala
  - similar to erlang
  - supports synchronous messages
- google Go
  - Communicating Sequential Processes (CSP)
  - explicit channels
  - type-safe (type determined at creation)
  - unbuffered / buffered (asynchronous)

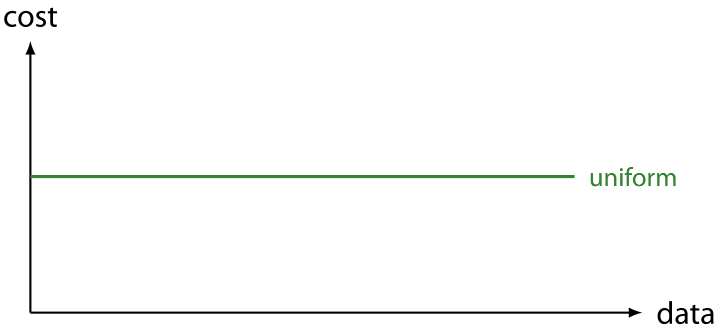
Summary

- multicore era
- Expressing parallelism
  - data parallel: maps, reductions, parallel for
  - task parallel: recursive splitting, generic model
- Algorithmic concerns:
  - D&C vs accumulators
- Cooperation
  - sharing state: TM vs locks
  - message passing

EOF!

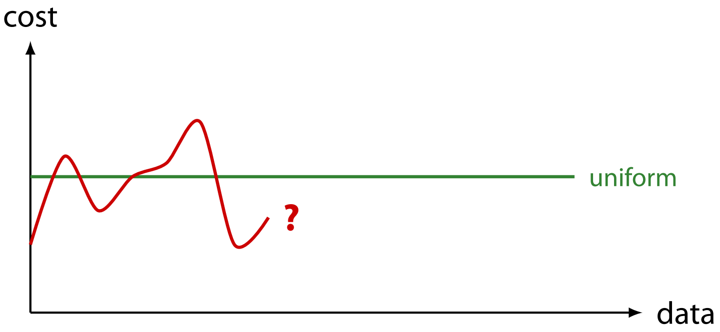
☺

Load balancing



- uniform computation cost (same for all data items):
  - divide data by the number of processors

Load balancing



- uniform computation cost (same for all data items):
  - divide data by the number of processors
- general case: unknown cost for each data item:
  - divide data in *chunks*
  - assign chunks in processors dynamically

User-space scheduling of parallel tasks

informal problem description:

- A set of parallel tasks T
- P processors, where tasks execute
  (actually, they are kernel threads)
- Tasks may spawn other tasks dynamically
- Tasks may wait for childern to finish

goals:

- execution time efficiency (load-balancing)
- space efficiency
- small overhead (independent of T)

scheduling approaches

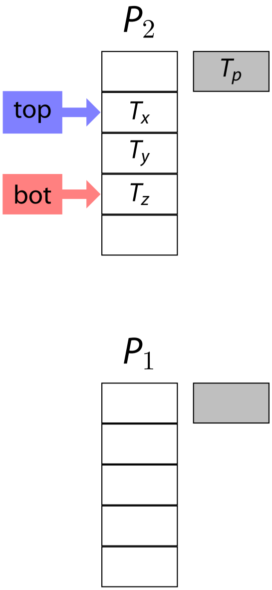
- **work sharing:** when new tasks are created, scheduler tries to migrate them to other underutilized processors
- **work stealing:** idle processors attempt to “steal” tasks.

work stealing is usually selected:

- better locality
- less synchronization overhead
- optimal theoritical bounds (time, space) [Blumofe and Leiserson ’99]

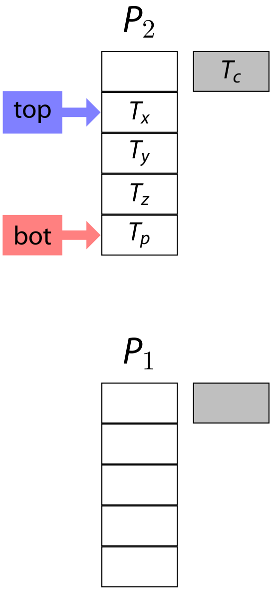
work stealing

- a *deque* (double-ended queue) per P:
  - `pushBot`
  - `popBot`
  - `popTop`



work stealing

- a *deque* (double-ended queue) per P:
  - `pushBot`
  - `popBot`
  - `popTop`
- task T_c is spawned from T_p:
  - `pushBot(`T_p`)`
  - `execute(`T_c`)`



work stealing

- a *deque* (double-ended queue) per P:
  - `pushBot`
  - `popBot`
  - `popTop`
- task T_c is spawned from T_p:
  - `pushBot(`T_p`)`
  - `execute(`T_c`)`
- P_1 is idle:
  - select random processor p
  - p`->popTop()`
  - execute result