Programming in the Multicore Era

WISES 2010

Kornilios Kourtis

kkourt@cslab.ece.ntua.gr

Computing Systems Laboratory
National Technical University of Athens

July 9, 2010

The free lunch

The free lunch:

- exponential increase in serial CPU performance
  (frequency scaling, ILP exploitation)
- exponential increase in number of transistors
  (Moore's law)

The free lunch **is over!**

The free lunch:

- exponential increase in serial CPU performance
  (frequency scaling, ILP exploitation)
- exponential increase in number of transistors
  (Moore's law)

is over:

- architects hit hard limits (power, available ILP)
- solution: **multicore CPUs**
  (use extra transitors for multiple cores)
- Moore's law $\leftrightarrow$ exponential increase in cores

the “Multicore Era”

where only parallel programs benefit from new hw!

parallel programming is difficult:

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

so in the last years:

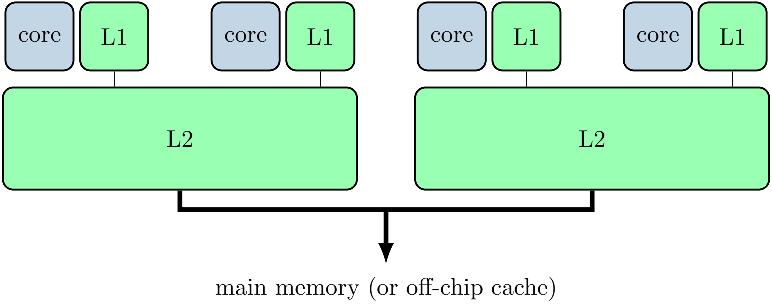
- effort to make parallel programming **easier**
  (and less **error-prone**)
- emerging parallel languages and paradigms

Outline

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

Multicore designs

current:

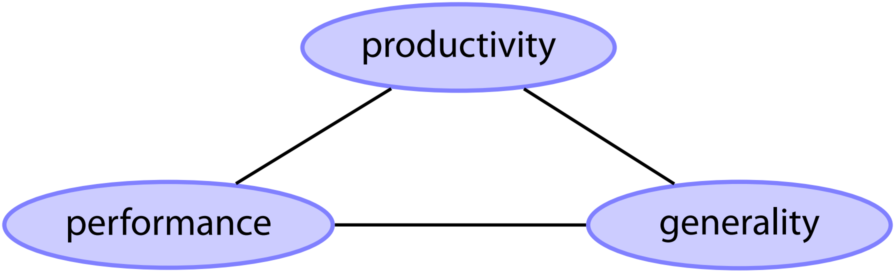


future:

- manycore
- heterogeneous (e.g., cell, GPUs)

Goals of parallel programming

[McKenney et. al. '09]



- No silver bullet! (pick 2 out of 3)
- language approach: provide constructs for generic or productive parallel programming

Parallel languages

this talk is about:

language constructs for
expressing and managing parallelism.

this talk is **not** about:

ways of automatically making a serial program parallel

- Why not a library ?
  - parallelism too pervasive to leave out of compiler/run-time system

Expressing parallelism

parallel programming paradigms

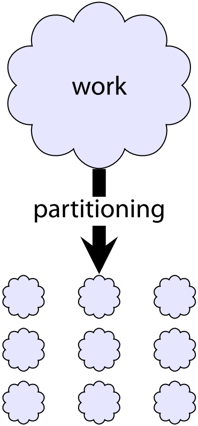
- **Data parallel**
  An operation is applied simultaneously to an aggregate of individual items (e.g., arrays).
  (productive, not general)

- **Task parallel**
  User explicitly defines parallel tasks.
  (general, not productive)

Basic concepts

work partitioning (expressing parallelism)

work must be split in **parallel tasks**



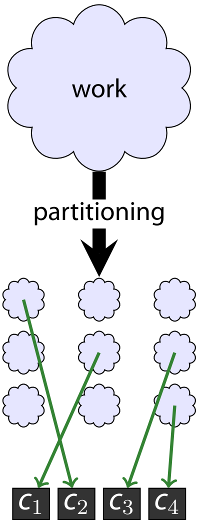
Basic concepts

work partitioning (expressing parallelism)

work must be split in **parallel tasks**

scheduling

tasks must be mapped into cores



Basic concepts

work partitioning (expressing parallelism)

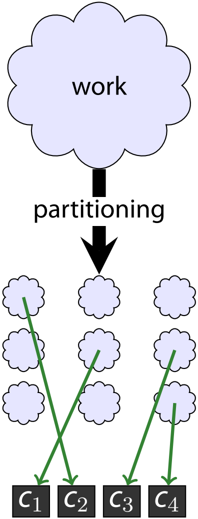
work must be split in **parallel tasks**

(data parallel: system, task parallel: user)

scheduling

tasks must be mapped into cores

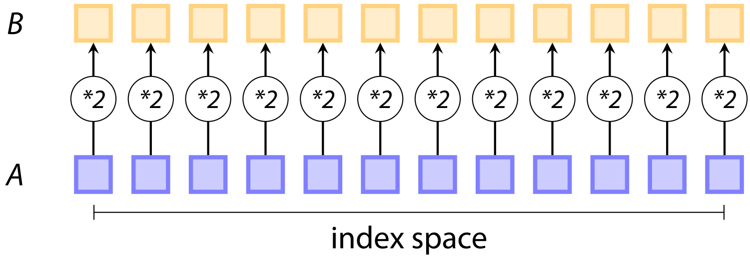
(system)



data parallel constructs

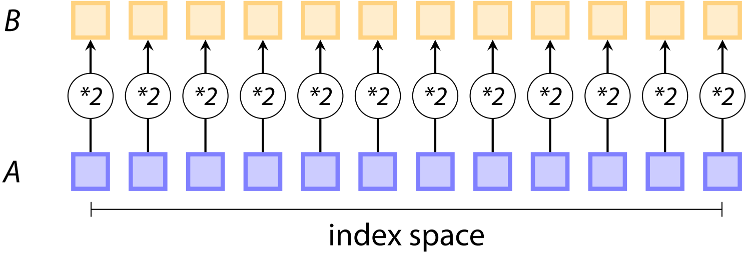
vector map
(simple) data parallel example

```
B = 2*A;
```



vector map
(simple) data parallel example

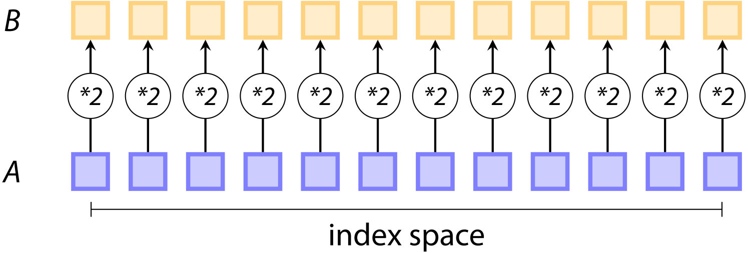
```
B = 2*A;
```



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

vector map
(simple) data parallel example

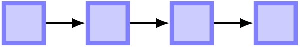
```
B = 2*A;
```



- each operation can be performed in parallel
- work partitioning $\leftrightarrow$ index partitioning
- efficient parallelization requires
  efficient partitioning of aggregate structures

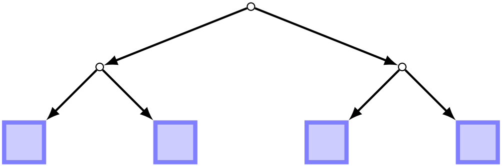
partitioning of aggregate structures

- linked lists: ☹



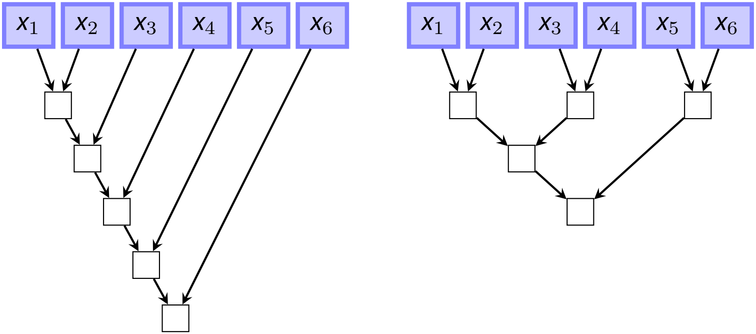
- arrays: ☺

- trees (if balanced): ☺



reductions

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



- based on index space partitioning
- some languages support user-defined reductions

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
- work partitioning $\rightarrow$ partition iteration space
- more flexibility on expressing an algorithm

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
- work partitioning $\rightarrow$ partition iteration space
- more flexibility on expressing an algorithm
- programmer must avoid data races

Data parallelism

Advanced issues:

- locality concerns
- heterogeneity in hardware

In conclusion:

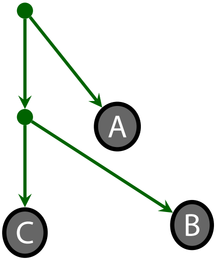
\+ performance, productivity

\- not general

Task parallelism

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

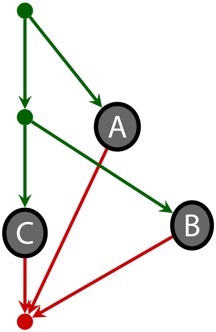
```
/* Cilk example */
x = spawn A();
y = spawn B();
z = C();
```



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 */
```



divide & conquer is easily parallelized

```
Divide and Conquer:
    if cant divide:
        return unitary solution  (stop recursion)
    divide problem in two
    solve first   (recursively)
    solve second  (recursively)
    combine solutions
```

- solve first/second can be performed in parallel
- recursive splitting
- example: quicksort

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"
  - incrementaly update solution

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)

incrementaly update serial solution:

```
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_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)
```

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_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)

rle_conc: combine 2 partial rle solutions
if last(rle1), first(rle2) have the same symbol:
    merge them
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

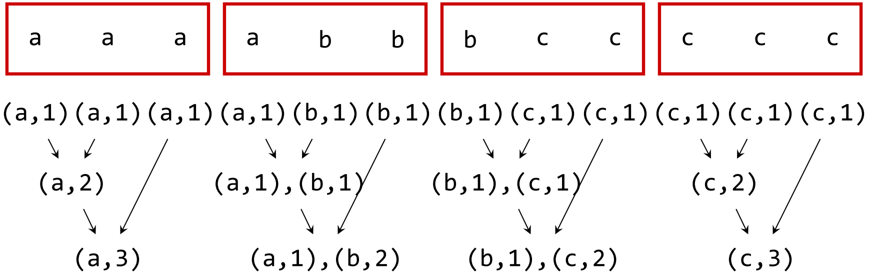
a	a	a	a	b	b	b	c	c	c	c	c
(a,1)	(a,1)	(a,1)	(a,1)	(b,1)	(b,1)	(b,1)	(c,1)	(c,1)	(c,1)	(c,1)	(c,1)

Example: RLE recursive splitting

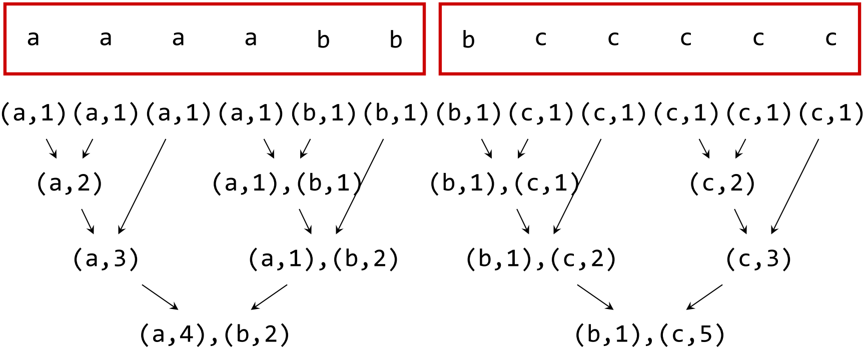
a a	a	a b	b	b c	c	c c	c

```
(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



Example: RLE recursive splitting



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)

```
data parallel solution:
  map all inputs to unitary solution
  reduce on rle_conc
```

(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
  - data sharing
  - message passing

Data sharing

- shared memory architectures allow *data sharing*.
- applications can utilize it

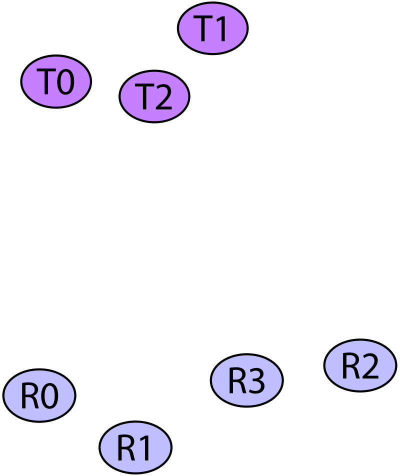
- **but:** concurrent accesses may lead to inconsistencies

  (e.g., concurrent updates on a linked list)
- **solution:** mutual exclusion (locks).

Locks

mutual exclusion

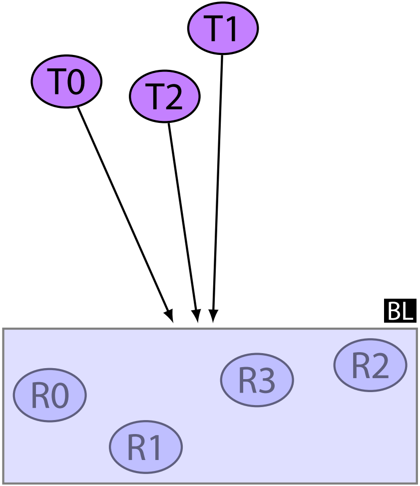
- Model:
  - T: Tasks
  - R: Resources



Locks

mutual exclusion

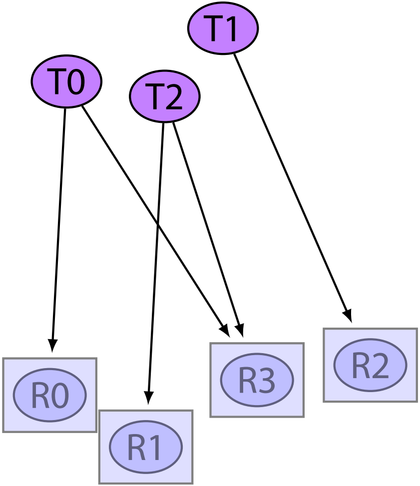
- Model:
  - T: Tasks
  - R: Resources
- Big Lock:
  - one lock for all
  - poor scalability



Locks

mutual exclusion

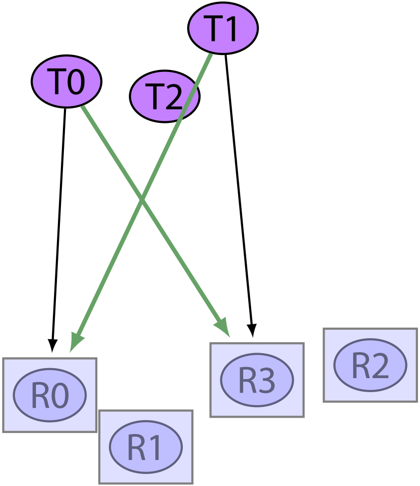
- Model:
  - T: Tasks
  - R: Resources
- Big Lock:
  - one lock for all
  - poor scalability
- Fine-grain locking:
  - one lock per R
  - possible deadlock
  - global order of Rs



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 too hard!

- 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
- 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 by design
  (e.g., does not require mutual exclusion)

Transactional Memory conclusion

When sharing data accross different parallel tasks:

- locks are hard (almost unusable)
- 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 hard (almost unusable)
- 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

- Actor model
  - erlang, scala
  - messages to tasks
- Communicating Sequential Processes (CSP)
  - google Go
  - explicitly create communication channels

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

What parallel programming languages can do for embedded systems ?

- multicore trend
- popularized embedded systems development
  (e.g., iPhone development)
- hide details from programmer
- adapt to different architectures

Thank you!

☺

Questions ?