Can Software Transactional Memory make Parallel Programs Simple and Safe?

Ketil Malde

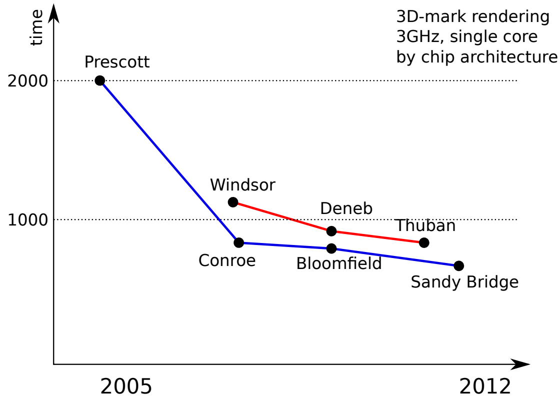
Institute of Marine Research

February 11, 2013

Outline

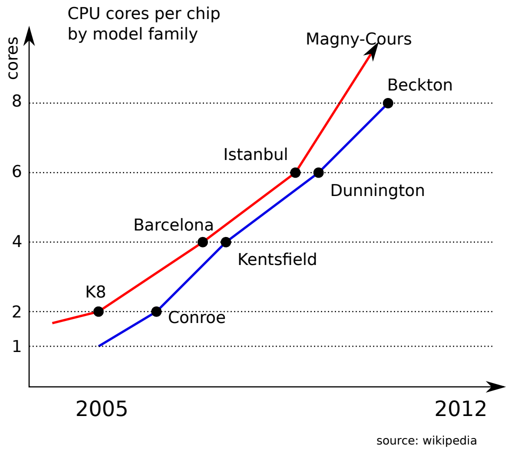
Outline

CPU performance

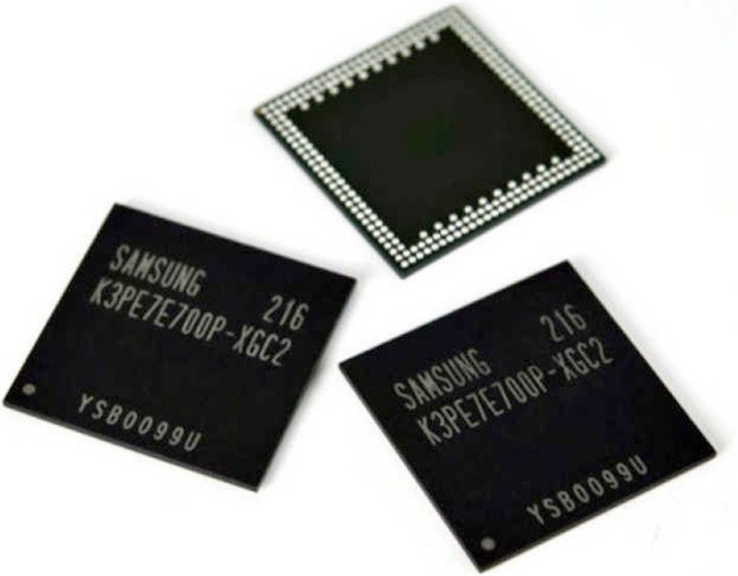


source: tomshardware.com

Increasing number of cores



2013: Octo-core phones?



Outline

Bank account example

```
deposit(acct, val) =
   x <- readAcct(acct)
   y = x+val
   writeAcct(acct, y)
```

Bank account example

A: $100

```
deposit( A , 10 ) =
  x <- readAcct( A )
  y = x+10
  writeAcct( A , y)
```

Bank account example

A: $100

```
deposit( A  ,10 ) =
  x <- readAcct( A  )
  y = x+10
  writeAcct( A  , y)
```

x=100

Bank account example

A: $100

```
deposit( A ,10 ) =
  x <- readAcct( A )
  y = x+10
  writeAcct( A , y)
```

x=100 y=110

Banc account example

A: $110

```
deposit( A ,10 ) =
  x <- readAcct( A )
  y = x+10
  writeAcct( A , y)
```

x=100 y=110

Race condition!

A: $???

```
deposit( A ,10 ) =
  x <- readAcct( A )
  y = x+10
  writeAcct( A , y)
```

```
deposit( A ,20 ) =
  x <- readAcct( A )
  y = x+20
  writeAcct( A , y)
```

x=100 y=110

x=100 y=120

Solution: Locking

```
deposit( A , 10 ) =
  take_lock(A)
  x <- readAcct( A )
  y = x+ 10
  writeAcct( A , y)
  release_lock(A)
```

```
deposit( A , 20 ) =
  take_lock(A)
  x <- readAcct( A )
  y = x+ 20
  writeAcct( A , y)
  release_lock(A)
```

Solution: Locking?

It is difficult to get right.

- Need to protect all shared data
- Deadlocks
- Livelocks
- *Locks prevent composability*

Composability

transfer(src,dst,amount) =
  withdraw(src,amount)
  deposit(dst,amount)

Composability

```
transfer(src,dst,amount) =
  withdraw(src,amount)
  deposit(dst,amount)
```

← Inconsistent state!

Composability

```
transfer(src,dst,amount) =
   take_lock(src)
   take_lock(dst)
   withdraw(src,amount)
   deposit(dst,amount)
   release_lock(dst)
   release_lock(src)
```

Outline

Software Transactional Memory

```
deposit( A , 10 ) =
  x <- readAcct( A )
  y = x+ 10
  writeAcct( A , y)
```

```
deposit( A , 20 ) =
  x <- readAcct( A )
  y = x+ 20
  writeAcct( A , y)
```

Software Transactional Memory


```
deposit( A , 10 ) =
  BEGIN
  x <- readAcct( A )
  y = x+ 10
  writeAcct( A , y)
  COMMIT
```

```
deposit( A , 20 ) =
  BEGIN
  x <- readAcct( A )
  y = x+ 20
  writeAcct( A , y)
  COMMIT
```

Software Transactional Memory

A: $100

deposit(A , 10) = BEGIN x <- **readAcct**(A) y = x+ 10 **writeAcct**(A , y) COMMIT x=100 y=110	**deposit**(A , 20) = BEGIN x <- **readAcct**(A) y = x+ 20 **writeAcct**(A , y) COMMIT x=100 y=120

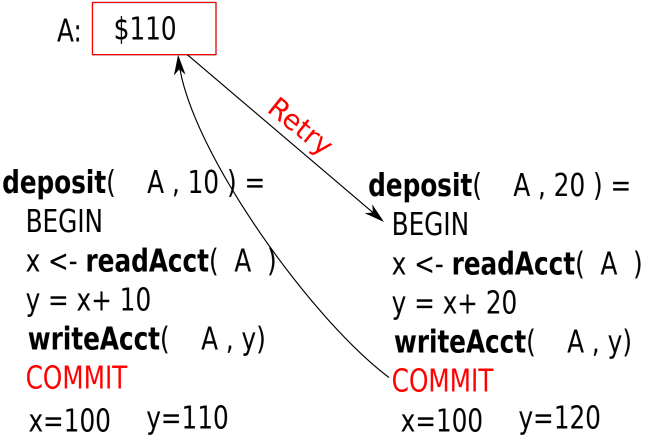
Software Transactional Memory

A: $110

deposit(A , 10) =
BEGIN
x <- **readAcct**(A)
y = x+ 10
writeAcct(A , y)
COMMIT
x=100 y=110

deposit(A , 20) =
BEGIN
x <- **readAcct**(A)
y = x+ 20
writeAcct(A , y)
COMMIT
x=100 y=120

Software Transactional Memory



STM Performance Impact

- Logging data access
- Retrying transaction

Outline

Scaffolding

Given a set of *contigs*
And a set of *links*
Order and orient the contigs,
satisfying as many links as possible.

A graph problem:

- Contigs are nodes
- Link information make edges

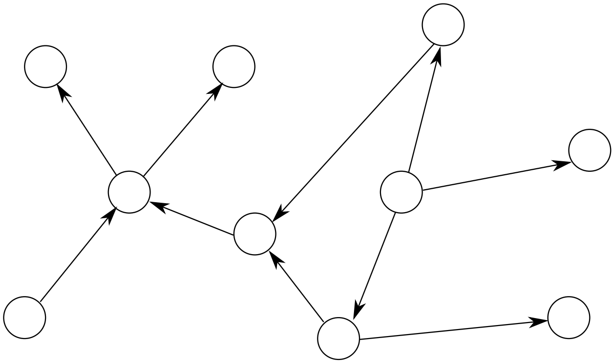
Scaffolding

Given a set of *contigs*
And a set of *links*
Order and orient the contigs,
satisfying as many links as possible.

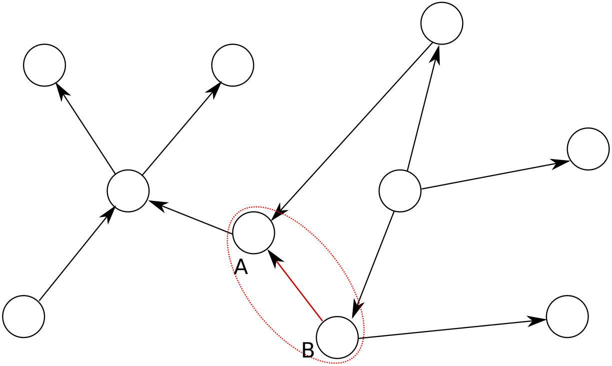
A graph problem:

- Contigs are nodes
- Link information make edges

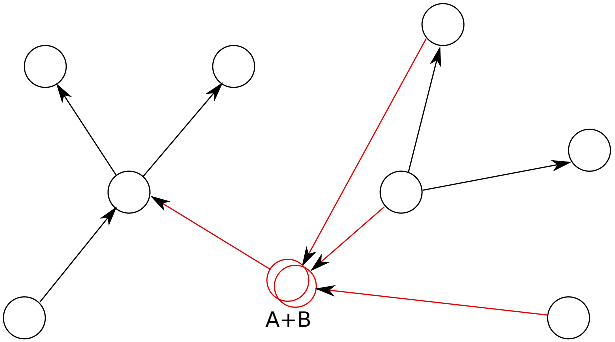
The scaffolding graph



Identify edge



Update graph



- Changes are local
- ..but unpredictable

$\rightarrow$ Ideal for transactions?

Outline

Simple Implementation

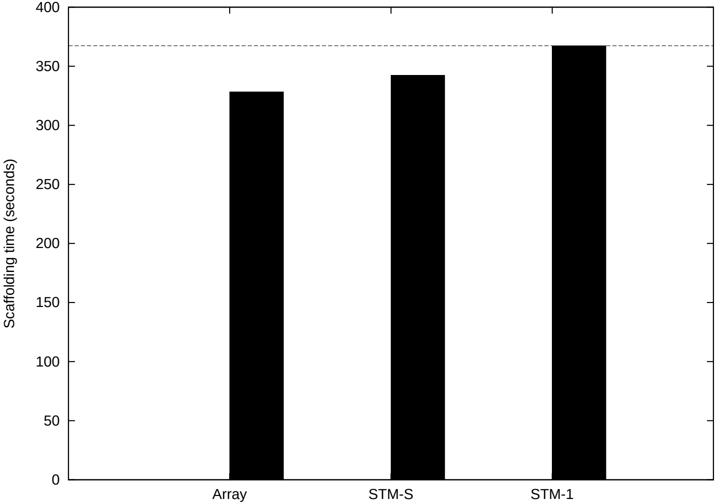
Join contigs A and B iff:

B is the contig with most links from A

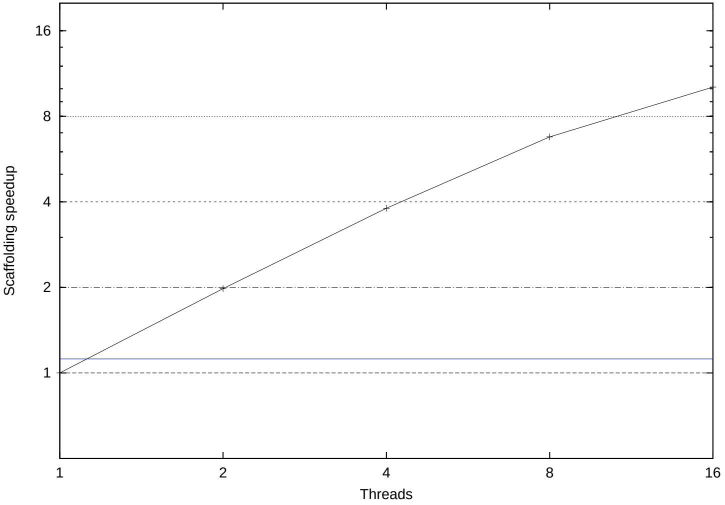
and

A is the contig with most links from B

Transactional Overhead



Speedup



Caveats

Performance will be hurt by retried transactions.

Affected by:

- Number of threads/size of graph
- Size of subgraph affected by each transaction

Outline

Conclusions

CPU time is precious — don't waste it on single-thread applications!

Programmer time is even more precious — don't waste it on primitive concurrency primitives!

Conclusions

CPU time is precious — don't waste it on single-thread applications!

Programmer time is even more precious — don't waste it on primitive concurrency primitives!

Conclusions

CPU time is precious — don't waste it on single-thread applications!

Programmer time is even more precious — don't waste it on primitive concurrency primitives!

The End

Thanks