Resource Oblivious Parallel Computing

Vijaya Ramachandran
Department of Computer Science
University of Texas at Austin

Joint work with Richard Cole

Reference. R. Cole, V. Ramachandran, “Efficient Resource Oblivious Algorithms for Multicores”. http://arxiv.org/abs/1103.4071.

THE MULTICORE ERA

- *Chip Multiprocessors (CMP)* or *Multicores*:

  Due to power consumption and other reasons, microprocessors are being built with multiple cores on a chip.

  Dual-cores are already on most desktops, and number of cores is expected to increase (dramatically) for the foreseeable future

- The multicore era represents a *paradigm shift* in general-purpose computing.

- Computer science research needs to address the multitude of challenges that come with this shift to the multicore era.

ALGORITHMS: VON NEUMANN ERA VS MULTICORE

In order to successfully move from the von Neumann era to the emerging multicore era, we need to develop methods that:

- Exploit **both** parallelism **and** cache-efficiency.

- Further, these algorithms need to be *portable* (i.e., independent of machine parameters).

  Even better would be a *resource oblivious* computation where both the algorithm and the run-time system are independent of machine parameters.

MULTICORE COMPUTATION MODEL

- We model the multicore computation with:
  - A multithreaded algorithm that generates parallel tasks ('threads').
  - A run-time scheduler that schedules parallel tasks across cores. (Our scheduler has a distributed implementation.)
  - A shared memory with caches.
  - Data organized in blocks with cache coherence to enforce data consistency across cores.
  - Communication cost in terms of cache miss costs, including costs incurred through false sharing.

Our main results are for multicores with private caches.

OUR RESULTS

- The class of *Hierarchical Balanced Parallel (HBP)* algorithms.
- HBP algorithms for scans, matrix computations, FFT, etc., building on known algorithms.
- A new HBP sorting algorithm:
  SPMS: Sample, Partition, and Merge Sort.
- Techniques to reduce the adverse effects of false sharing:
  limited access writes, $O(1)$ *block sharing,* and *gapping*.

OUR RESULTS (CONTINUED)

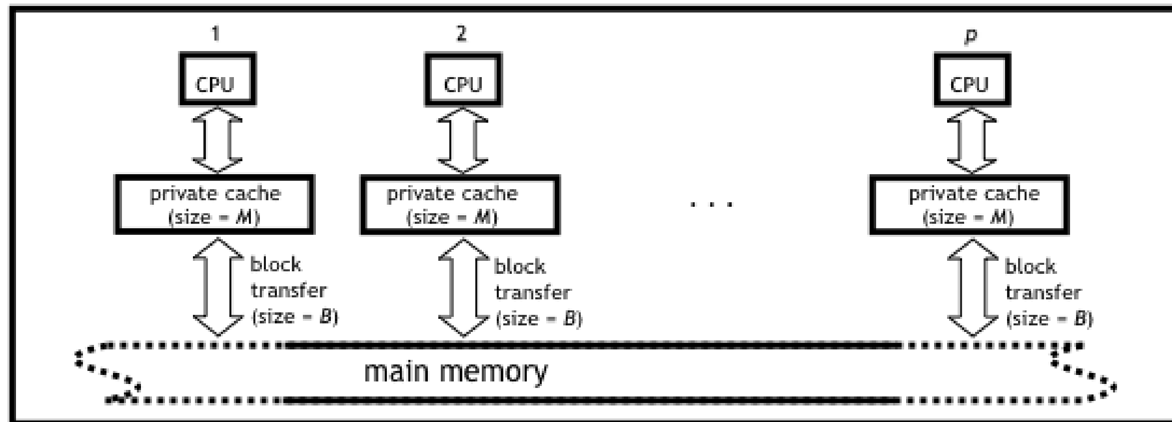
- The *Priority Work Stealing Scheduler (PWS)*.

- Cache miss overhead of the HBP algorithms, when scheduled by PWS, is bounded by the sequential cache complexity, *even when the cost of false sharing is included*, given a suitable 'tall cache'.

  (for large inputs that do not fit in the caches).

At the end of the talk, we address multi-level cache hierarchy [Chowdhury-Silvestri-B-R'10], and other parallel models.

Multicore with Private Caches



- p cores
- a *private cache* of size M for each core
- an arbitrarily large *global shared memory*
- block transfer size B (between caches and memory)

ROAD MAP

- Background on multithreaded computations and work stealing.
- Cache and block misses.
- Hierarchical Balanced Parallel (HBP) computations.
- Priority Work Stealing (PWS) Scheduler.
- An example with Strassen's matrix multiplication algorithm.
- Discussion.

MULTITHREADED COMPUTATIONS

M-Sum($A[1..n], s$) % Returns $s = \sum_{i=1}^{n} A[i]$
if $n = 1$ **then return** $s := A[1]$ **end if**
fork(M-Sum($A[1..n/2], s_1$); M-Sum($A[\frac{n}{2} + 1..n], s_2$))
join: **return** $s = s_1 + s_2$

- *Sequential execution* computes recursively in a dfs traversal of this computation tree.

MULTITHREADED COMPUTATIONS

M-Sum($A[1..n], s$) % Returns $s = \sum_{i=1}^{n} A[i]$
if $n = 1$ **then return** $s := A[1]$ **end if**
fork(M-Sum($A[1..n/2], s_1$); M-Sum($A[\frac{n}{2} + 1..n], s_2$))
join: **return** $s = s_1 + s_2$

- *Sequential execution* computes recursively in a dfs traversal of this computation tree.
- Forked tasks can run in parallel.
- Runs on p cores in $O(n/p + \log p)$ parallel steps by forking $\log p$ times to generate p parallel tasks.

Work-Stealing Parallel Execution

M-Sum($A[1..n], s$) % Returns $s = \sum_{i=1}^{n} A[i]$

if $n = 1$ **then return** $s := A[1]$ **end if**

fork(M-Sum($A[1..n/2], s_1$); M-Sum($A[\frac{n}{2} + 1..n], s_2$))

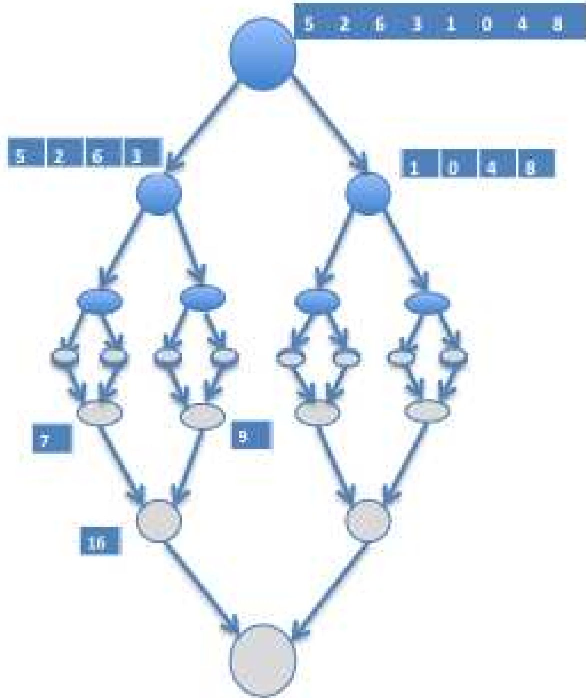
join: **return** $s = s_1 + s_2$

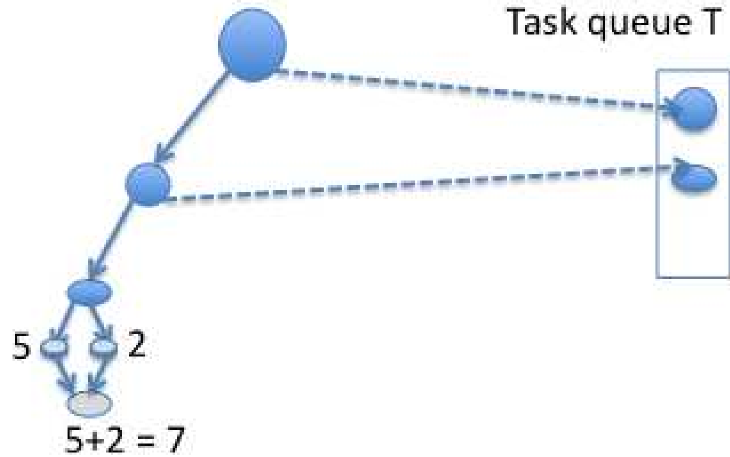
WORK-STEALING PARALLEL EXECUTION

M-Sum($A[1..n], s$) % Returns $s = \sum_{i=1}^{n} A[i]$
if $n = 1$ **then return** $s := A[1]$ **end if**
fork(M-Sum($A[1..n/2], s_1$); M-Sum($A[\frac{n}{2} + 1..n], s_2$))
join: **return** $s = s_1 + s_2$

- Computation starts in first core C
- At each fork, *second* forked task is placed on C's *task queue* T.
- Computation continues at C (in sequential order), with tasks popped from tail of T as needed.
- Task at head of T is available to be *stolen* by other cores that are idle.

Multithreaded Computation





Input: 5 2 6 3 1 0 4 8

WORK-STEALING

- Work-stealing is a well-known method in scheduling with various heuristics used for stealing protocol.
- *Randomized* work-stealing (RWS) has provably good parallel speed-up on fairly general computation dags. [Blumofe-Leiserson 1999].
- Caching bounds for RWS are derived in [ABB02, Frigo-Strumpen10, BGN10]; more recently in [Cole-R11].

  None of these cache miss bounds are optimal.

ROAD MAP

- Background on multithreaded computations and work stealing.
- Cache and block misses.
- Hierarchical Balanced Parallel (HBP) computations.
- Priority Work Stealing (PWS) Scheduler.
- An example with Strassen's matrix multiplication algorithm.
- Discussion.

Cache Misses

Definition. Let τ be a task that accesses r data items (i.e., words) during its execution. We say that $r = |\tau|$ is the *size* of τ.

τ is f-cache friendly if these data items are contained in $O(r/B + f(r))$ blocks.

A multithreaded computation C is f-cache friendly if every task in C is f-cache friendly

CACHE MISSES

Definition. Let τ be a task that accesses r data items (i.e., words) during its execution. We say that $r = |\tau|$ is the *size* of τ.

τ is f-cache friendly if these data items are contained in $O(r/B + f(r))$ blocks.

A multithreaded computation C is f-cache friendly if every task in C is f-cache friendly

Lemma. A stolen task τ incurs an additional $O(\min\{M, |\tau|\}/B + f(\tau))$ cache misses compared to the steal-free sequential execution.

If $f(|\tau|) = O(|\tau|/B)$ and $|\tau| \geq 2M$, this is a 0 asymptotic excess,

i.e., the excess is bounded by the sequential cache miss cost.

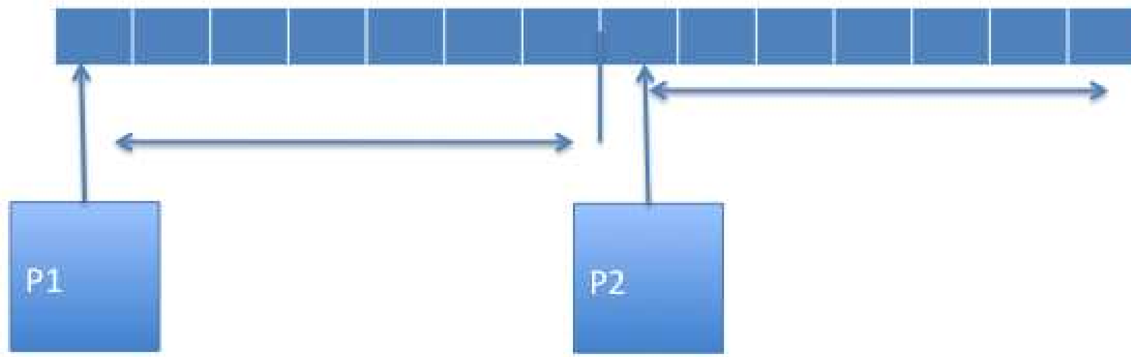
False Sharing

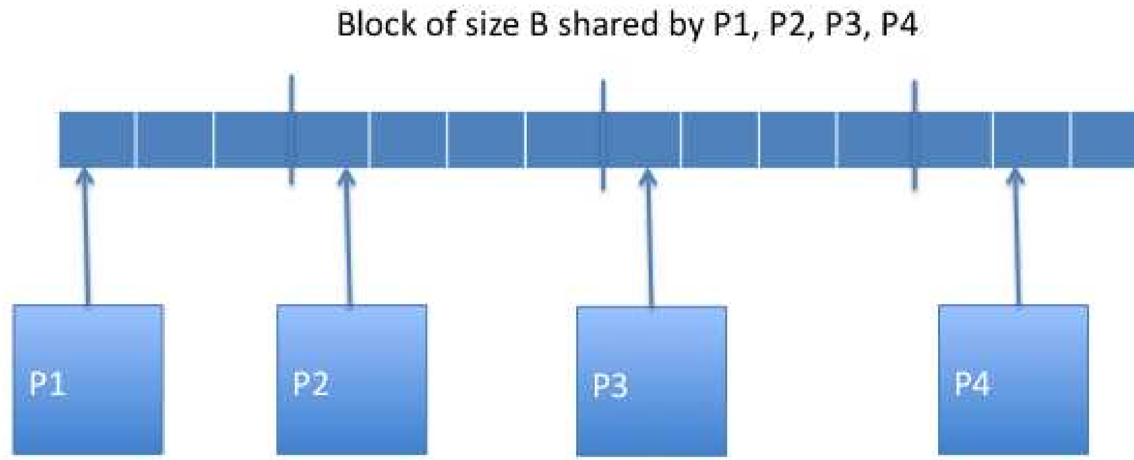
- *False sharing*, and more generally *block misses*, occur when there is at least one write to a shared block.

- In such shared block accesses, delay is incurred by participating cores when control of the block is given to a writing core, and the other cores wait for the block to be updated with the value of the write.

- A typical *cache coherence* protocol *invalidates* the copy of the block at the remaining cores when it transfers control of the block to the writing core.

  The delay at the remaining cores is *at least* that of one cache miss, and could be more.


Block of size B shared by P1 and P2
P1
P2
False Sharing: B/2 cache misses incurred by P1 and by P2



Block Misses: P1 – P4 execute small tasks whose outputs lie in a single block.

While one core writes to one location in the block, the remaining cores will wait for their turn.

Block Miss Cost Measure

Definition. Suppose that block β is moved m times from one cache to another *(*due to cache or block misses*)* during a time interval $T = [t_1, t_2]$. Then m is defined to be the *block delay* incurred by β during T.

The *block wait cost* incurred by a task τ on a block β is the delay incurred during the execution of τ due to block misses when accessing β, measured in units of cache misses.

Block Miss Cost Measure

Definition. Suppose that block β is moved m times from one cache to another *(*due to cache or block misses*)* during a time interval $T = [t_1, t_2]$. Then m is defined to be the *block delay* incurred by β during T.

The *block wait cost* incurred by a task τ on a block β is the delay incurred during the execution of τ due to block misses when accessing β, measured in units of cache misses.

The block wait cost could be much larger than B if multiple writes to the same location are allowed.

In most of our analysis, we will use the block delay of β within a time interval T as the block wait cost of *every* task that accesses β during T.

BLOCK MISS COST MEASURE

Definition. Suppose that block β is moved m times from one cache to another *(*due to cache or block misses*)* during a time interval $T = [t_1, t_2]$. Then m is defined to be the *block delay* incurred by β during T.

The *block wait cost* incurred by a task τ on a block β is the delay incurred during the execution of τ due to block misses when accessing β, measured in units of cache misses.

The block wait cost could be much larger than B if multiple writes to the same location are allowed.

In most of our analysis, we will use the block delay of β within a time interval T as the block wait cost of *every* task that accesses β during T.

This cost measure is highly pessimistic, hence upper bounds obtained using it are likely to hold for other cost measures for block misses.

Reducing Block Miss Costs: Algorithmic Techniques

1. We enforce *limited access writes*: An algorithm is limited access if each of its writable variables is accessed $O(1)$ times.

Reducing Block Miss Costs: Algorithmic Techniques

1. We enforce *limited access writes*: An algorithm is limited access if each of its writable variables is accessed $O(1)$ times.

2. We attempt to obtain $O(1)$*-block sharing* in our algorithms.

   Definition. A task τ of size r is L*-block sharing*, if there are $O(L(r))$ blocks which τ can share with all other tasks that could be scheduled in parallel with τ and could access a location in the block.

   A computation is L-block sharing if every task in it is L-block sharing.

REDUCING BLOCK MISS COSTS: ALGORITHMIC TECHNIQUES

1. We enforce *limited access writes*: An algorithm is limited access if each of its writable variables is accessed $O(1)$ times.

2. We attempt to obtain $O(1)$*-block sharing* in our algorithms.

   Definition. A task τ of size r is L*-block sharing*, if there are $O(L(r))$ blocks which τ can share with all other tasks that could be scheduled in parallel with τ and could access a location in the block.

   A computation is L-block sharing if every task in it is L-block sharing.

3 When $O(1)$ block sharing is not achieved in an algorithm, we use *gapping* to reduce the cost of block misses.

REDUCING BLOCK MISS COSTS: ALGORITHMIC TECHNIQUES

1. We enforce *limited access writes*: An algorithm is limited access if each of its writable variables is accessed $O(1)$ times.

2. We attempt to obtain $O(1)$*-block sharing* in our algorithms.

   Definition. A task τ of size r is L*-block sharing*, if there are $O(L(r))$ blocks which τ can share with all other tasks that could be scheduled in parallel with τ and could access a location in the block.

   A computation is L-block sharing if every task in it is L-block sharing.

3 When $O(1)$ block sharing is not achieved in an algorithm, we use *gapping* to reduce the cost of block misses.

4 We take special care to reduce block wait costs at the execution stacks of the tasks.

Summary: Cache-related Parameters

We identify two useful cache-related parameters for algorithm design.

- **Cache-friendly function $f(r)$:**

  $f(r) = O(\sqrt{r})$ suffices for good performance with a standard tall cache.

- **Block-sharing function $L(r)$:**

  $L(r) = O(1)$ is desirable.

ROAD MAP

- Background on multithreaded computations and work stealing.
- Cache and block misses.
- Hierarchical Balanced Parallel (HBP) computations.
- Priority Work Stealing (PWS) Scheduler.
- An example with Strassen's matrix multiplication algorithm.
- Discussion.

BP COMPUTATIONS

Definition. A *BP computation* π is a limited access algorithm that is formed from the down-pass of a binary forking computation tree T followed by its up-pass, and satisfies the following properties.

BP COMPUTATIONS

Definition. A *BP computation* π is a limited access algorithm that is formed from the down-pass of a binary forking computation tree T followed by its up-pass, and satisfies the following properties.

i. Only $O(1)$ computation at every node in the down-pass and the up-pass.

ii. π may use size $O(|T|)$ global arrays for its input and output.

BP COMPUTATIONS

Definition. A *BP computation* π is a limited access algorithm that is formed from the down-pass of a binary forking computation tree T followed by its up-pass, and satisfies the following properties.

i. Only $O(1)$ computation at every node in the down-pass and the up-pass.

ii. π may use size $O(|T|)$ global arrays for its input and output.

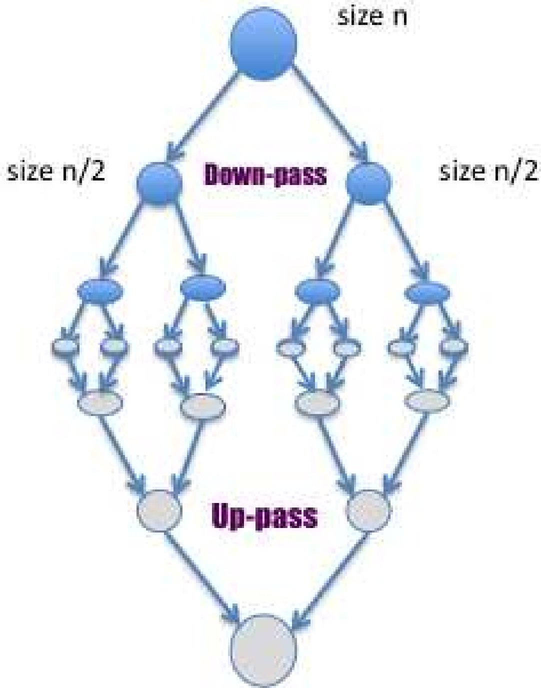
iii. *Balance Condition.*

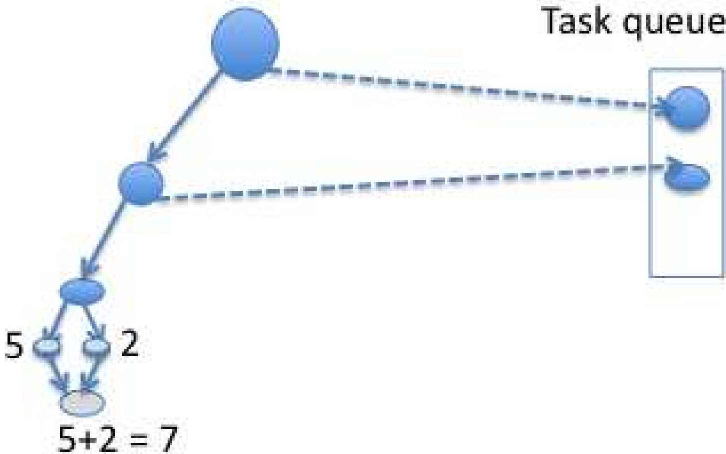
Let the root task have size r; let α be a constant less than 1; and let $c_1,\ c_2$ be constants with $c_1 \leq 1 \leq c_2$.

The size of any task τ at level i in the downpass of T satisfies:

$c_1 \cdot \alpha^i \cdot r \leq |\tau| \leq c_2 \cdot \alpha^i \cdot r$.

BP Computation





Input:

5	2	6	3	1	0	4	8

HBP COMPUTATIONS

A *Hierarchical Balanced Parallel Computations (HBP)* is a limited access algorithm that is one of the following:

- A Type 0 Algorithm, a sequential computation of constant size.
- A Type 1, or BP computation.
- A Type $t+1$ HBP, for $t \geq 1$, which, on an input of size n, calls, in succession, a sequence of $c \geq 1$ collections of parallel recursive subproblems, each of size $s(n) \leq n/b(n)$, with $b(n) > 1$;

  each of these collections can be interspersed with calls to HBP algorithms of type at most t.

HBP Computations (continued)

- A Type $\max\{t_1, t_2\}$ HBP computation results if it is a sequence of two HBP algorithms of types t_1 and t_2.

An HBP computation of type $t > 1$ is *balanced* if the recursive problems at each level of recursion all have sizes within a constant factor of each other.

HBP Results

Algorithm	Type	$f(r)$	$L(r)$	T_∞	$Q(n,M,B)$
Scans (MA, PS)	1	1	1	$O(\log n)$	$O(n/B)$
Matrix Transposition	1	1	1	$O(\log n)$	$O(n/B)$
Strassen	2	1	1	$O(\log^2 n)$	$n^\lambda/(B \cdot M^{\frac{\lambda}{2}-1})$
RM to BI	1	$\sqrt{r}$	1	$O(\log n)$	$O(n^2/B)$
Direct BI to RM	1	$\sqrt{r}$	$\sqrt{r}$	$O(\log n)$	$O(n^2/B)$
BI-RM (gap RM)	1	$\sqrt{r}$	gap	$O(\log n)$	$O(n^2/B)$
FFT	2	$\sqrt{r}$	1	$O(\log n \cdot \log\log n)$	$O(\frac{n}{B}\log_M n)$
LR	3	$\sqrt{r}$	gap	$O(\log^2 n \cdot \log\log n)$	$O(\frac{n}{B}\log_M n)$
CC*	4	$\sqrt{r}$	gap	$O(\log^3 n \cdot \log\log n)$	$O(\frac{n}{B}\log_M n \cdot \log n)$
Depth-n-MM	2	1	1	$O(n)$	$n^3/(B\sqrt{M})$
BI-RM for FFT*	2	$\sqrt{r}$	1	$O(\log n)$	$O(\frac{n^2}{B}\log_M n)$
Sort (SPMS)	2	$\sqrt{r}$	1	$O(\log n \cdot \log\log n)$	$O(\frac{n}{B}\log_M n)$

MA is Matrix Addition and PS is Prefix Sums. RM is Row Major and BI is Bit Interleaved.

TYPE refers to the HBP type.

Input size is n^2 for matrix computations, and n otherwise.

All algorithms, except those marked with $*$, match their standard sequential work bound.

$\lambda = \log_2 7$ in Strassen's algorithm.

ROAD MAP

- Background on multithreaded computations and work stealing.
- Cache and block misses.
- Hierarchical Balanced Parallel (HBP) computations.
- <u>Priority Work Stealing (PWS) Scheduler.</u>
- An example with Strassen's matrix multiplication algorithm.
- Discussion.

Priority Work Stealing Scheduler (PWS)

Consider BP computation π.

- PWS proceeds in *rounds*, one for each depth in π.
- *Priority* of a task is its depth in the computation.
- In a round for depth d:
  - (a) task at head of every non-empty task queue has priority at least d.
  - (b) only tasks of priority d are stolen in this round
  - (c) next round starts when task at head of task-queue at every non-idle core has priority greater than d.

A steal request is *unsuccessful* if no task priority d task available.

This triggers another steal attempt at priority $d + 1$.

PWS: Some Observations

Observation 1. There are at most $p - 1$ tasks that are stolen at any given depth of the computation.

Observation 2. The total number of steal attempts (including both successful and unsuccessful steals) across all cores is $< 2 \cdot p \cdot D$, where D is the depth of the computation.

Expected number of steals in randomized work-stealing steals is $O(pD)$ [BL99].

Cache Misses in a BP Computations Under PWS

Lemma. Consider the down-pass of a BP computation Π of size n scheduled under PWS. Let sequential cache complexity of Π be Q, and let $f(r) = O(\sqrt{r})$.

Then, with a tall cache $M = \Omega(B^2)$, the number of cache misses is bounded by $O(Q + pM/B\ )$.

If $n = \Omega(Mp)$ then the number of cache misses is $O(Q)$.

Cache Misses: HBP Computations

Lemma. Let Π be a balanced Type 2 HBP computation of size $n \geq Mp$, and let $c,\ s(n)$, and $f(r)$ be as defined earlier. Then, the cache miss excess for Π when scheduled under PWS has the following bounds with a tall cache $M \geq B^2$.

(i) If $c = 1,\ f(r) = O(\sqrt{r})$: $O(p\frac{M}{B}s^*(n, M))$.

(ii) If $c = 2,\ f(r) = O(\sqrt{r})$, and $s(n) = \sqrt{n}$: $O(p\frac{M}{B}\frac{\log n}{\log M})$.

(iii) If $c = 2,\ f(r) = O(\sqrt{r})$, and $s(n) = n/4$:
$O(p[\frac{\sqrt{nM}}{B} + \frac{\sqrt{n}}{\sqrt{M}}\sum_{i \geq 0} 2^i f(M/4^i)])$.

where $s^*(n, M)$ is the number of iterations of s needed to reduce n to M.

Block Misses Under PWS

Block Misses in a BP Computation

Lemma. Let π be the down-pass of a BP computation of size n, and let Q be its sequential cache complexity. If $L(r) = O(1)$, then, when scheduled by PWS, the block wait cost is $O(Q + pB \log B)$ if $n \geq B$.

Proof. By limited access, the block wait cost of any stolen task is $O(B)$.

For stolen tasks of size $\Omega(B^2)$ this cost is dominated by the $\Omega(B^2/B) = \Omega(B)$ cache miss cost.

There are at most $p - 1$ steals at each level, hence $O(p \log B)$ stolen tasks of size $O(B^2)$, and their total block miss cost is $O(B \cdot p \log B)$.

BLOCK MISSES AT THE EXECUTION STACKS

Block wait cost can be incurred at the *execution stack.*

- An execution stack S_τ is created for a task τ when a core C starts executing it.

  S_τ keeps track of the procedure calls and variables in the work performed on τ.

  The variables on S_τ may be accessed by stolen subtasks also.

  As S_τ grows and shrinks it may use and then stop using a block β repeatedly in an HBP computation.

BLOCK MISSES AT THE EXECUTION STACKS

Block wait cost can be incurred at the *execution stack.*

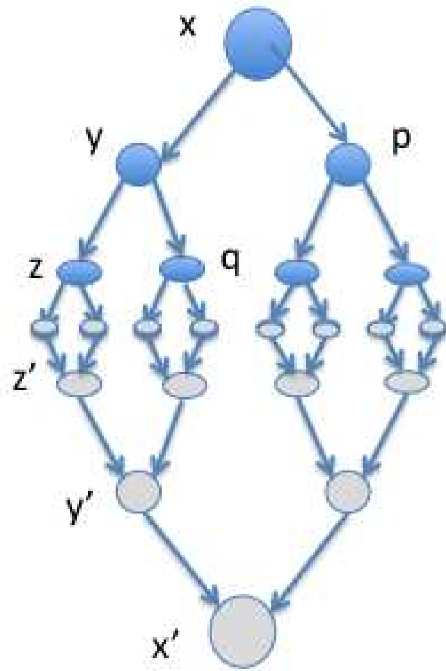
- An execution stack S_τ is created for a task τ when a core C starts executing it.

  S_τ keeps track of the procedure calls and variables in the work performed on τ.

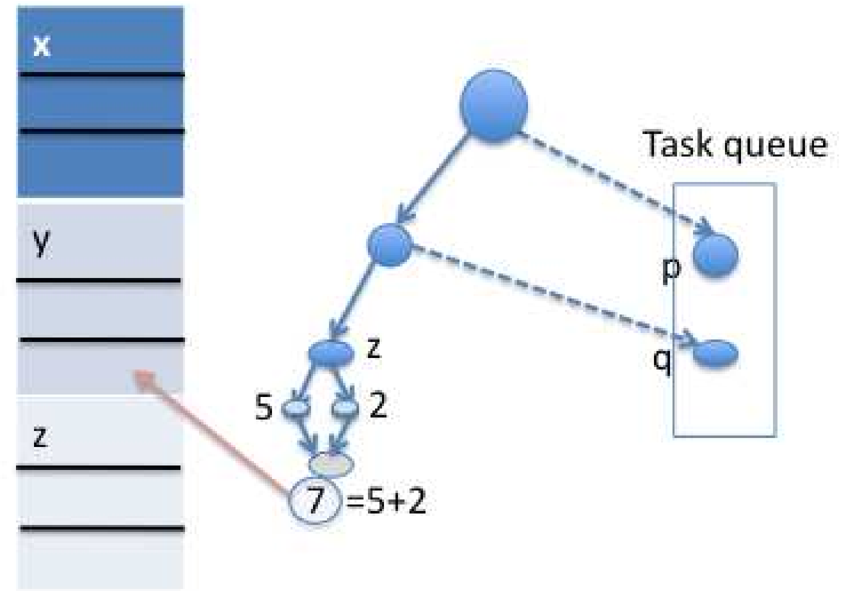
  The variables on S_τ may be accessed by stolen subtasks also.

  As S_τ grows and shrinks it may use and then stop using a block β repeatedly in an HBP computation.

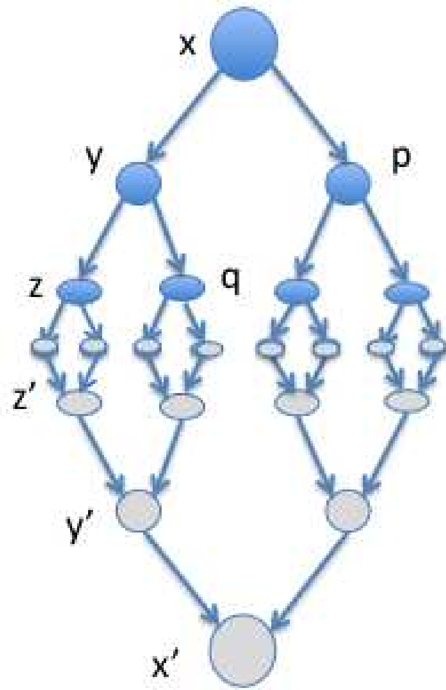
- Thus, even with limited-access and $O(1)$-block sharing, a large block wait cost could be incurred due to accesses to the execution stacks.

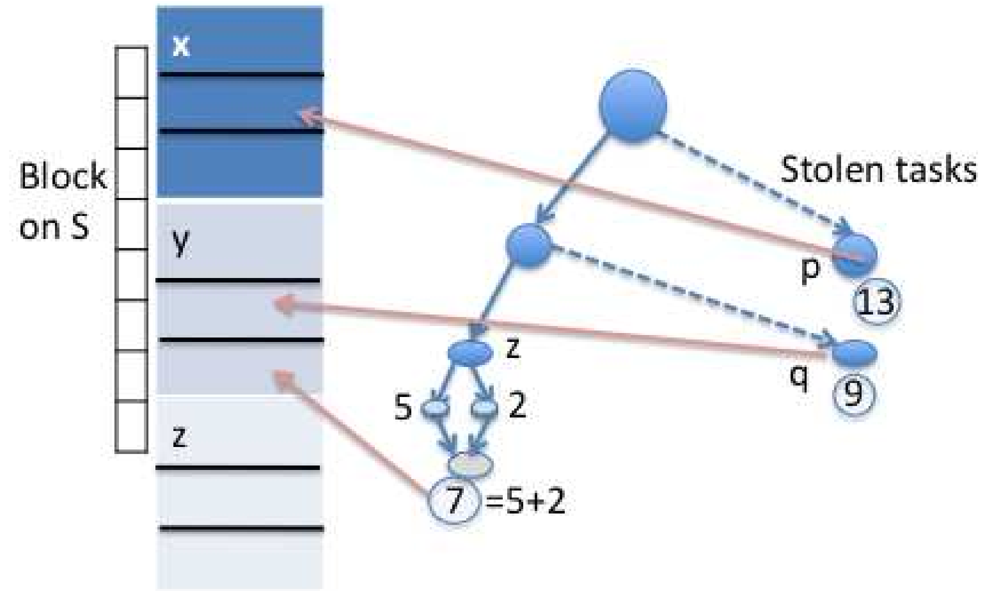


Execution stack S



Input: 5 2 6 3 1 0 4 8





Input:

5	2	6	3	1	0	4	8

EXECUTION STACK: BP AND HBP COMPUTATIONS

We establish the following:

- In a BP computation, block wait cost on any block on the stack is $O(B)$.

- In HBP computations, the block wait cost at a block could be in excess of B, due to repeated use of the block for different recursive calls.

  — To bound this cost we require an HBP computation τ of Type ≥ 2 to use $\Omega(|\tau|)$ space on the execution stack.

  — With this requirement we can bound the block wait cost on any block on the stack as $O(B)$.

Block Misses in Type 2 HBP Computations

Lemma. Let Π be a balanced Type 2 HBP computation of size $n \geq Mp$ with $\alpha = 1/2$ and $L(r) = O(1)$, which is exactly linear space bounded, and let $c,\ s(n)$, and $L(r)$ be as defined earlier. Then, the block miss excess for Π when scheduled under PWS has the following bounds.

(i) $c = 1$: a cost of $O(pB \log B \cdot s^*(n))$ cache misses.

(ii) $c = 2$ and $s(n) = \sqrt{n}$: a cost of $O(pB \log n \log\log B)$ cache misses.

(iii) $c = 2$ and $s(n) = n/4$: a cost of $O(pB\sqrt{n})$ cache misses.

Wrap-up: Overhead of Steals

- Other costs, including usurpations, cost of up-pass, and idle time are dominated by the cache and block miss excesses incurred by steals under PWS.

- For any given HBP algorithm, we can apply the results we have obtained for cache and block miss excess for PWS to determine the PWS scheduling overhead.

ROAD MAP

- Background on multithreaded computations and work stealing.
- Cache and block misses.
- Hierarchical Balanced Parallel (HBP) computations.
- Priority Work Stealing (PWS) Scheduler.
- An example with Strassen's matrix multiplication algorithm.
- Discussion.

Strassen's Matrix Multiplication (BI)

- $m = n^2 =$ size of matrix.
- Type 2 HBP with $c = 1$ collection of 7 subproblems, each of size $s(m) = m/4$.
- Uses BP computation MA for the matrix additions.
- Inherently limited access.
- $f(r) = L(r) = O(1)$ if matrix is in BI (bit interleaved) format.
- Sequential cache complexity is $\Theta(\frac{n^\lambda}{BM^\gamma})$, where $\lambda = \log_2 7$ and $\gamma = (\lambda/2) - 1$.

STRASSEN'S MM UNDER PWS

From PWS results:

Cache miss excess when $c = 1$, $f(r) = O(\sqrt{r})$: $O(p\frac{M}{B} \cdot s^*(n, M))$.

Block miss excess when $c = 1$, $L(r) = O(1)$: $O(pB \log B \cdot s^*(n))$ cache misses.

STRASSEN'S MM UNDER PWS

From PWS results:

Cache miss excess when $c = 1$, $f(r) = O(\sqrt{r})$: $O(p\frac{M}{B} \cdot s^*(n, M))$.

Block miss excess when $c = 1$, $L(r) = O(1)$: $O(pB \log B \cdot s^*(n))$ cache misses.

Apply to the HBP for Strassen's algorithm and check for conditions under which sequential cache complexity dominates:

STRASSEN CACHE MISS OVERHEAD. $O(p \cdot \frac{M}{B} \cdot \log \frac{n^2}{M})$

STRASSEN BLOCK MISS OVERHEAD. $Q_B = O(pB \log B \cdot \log n^2)$.

Strassen Block Miss Excess Under PWS

We need $pB \log B \cdot \log n^2 = O\left(\frac{n^\lambda}{BM^{\lambda/2-1}}\right)$.

It suffices to show that $\frac{n^\lambda}{\log n^2} = \Omega\left(pB^2 \log B \cdot M^{\lambda/2-1}\right)$.

When $n^2 \geq Mp$:

$$\frac{n^\lambda}{\log n^2} = \frac{n^\lambda}{(2/\lambda) \cdot \log n^\lambda} = \Omega\left(\frac{(Mp)^{\lambda/2}}{\log (Mp)^{\lambda/2}}\right).$$

So, it suffices to show that

$\frac{(Mp)^{\lambda/2}}{\log (Mp)} = \Omega\left(pB^2 \log B \cdot M^{\lambda/2-1}\right)$, or $Mp^{\lambda/2-1} = \Omega\left(B^2 \log B \log Mp\right)$

By considering the two cases $p = O(M)$ and $p = \omega(M)$, we can see that a tall cache $M = \Omega(B^2 \log^2 B)$ suffices.

Hence, when $M = \Omega(B^2 \log^2 B)$, the block miss (and cache miss) excess under PWS are dominated by sequential cache complexity.

ROAD MAP

- Background on multithreaded computations and work stealing.
- Cache and block misses.
- Hierarchical Balanced Parallel (HBP) computations.
- Priority Work Stealing (PWS) Scheduler.
- An example with Strassen's matrix multiplication algorithm.
- Discussion.

DISCUSSION

- HBP is suitable for Multi-BSP, but is more versatile.

  – Though analyzed under PWS for homogeneous cores, HBP under work stealing adapts gracefully to variations in core speeds, and to cores entering and leaving the computation.

- Block miss costs in parallel computing.

- Other models, e.g., network obliviousness and multicore-obliviousness for multi-level cache hierarchy.

MULTI-LEVEL CACHE HIERARCHY

Multicore oblivious algorithms for multi-level hierarchical caches. [Chowdhury-Silvestri-B-R'10]

- No mention of cache parameters or number of processors within the algorithm.
- Instead, algorithm includes *scheduler hints*:

  – *Coarse-grained contiguous (CGC)* scheduling for scans.

  – *Space-bound (SB)* scheduling for recursive computations such as depth n matrix multiplication. Supplies a *size* bound on task.

  – *CGC on SB scheduling* for more complex recursive computations such as FFT

  The scheduler algorithm, *using its knowledge of cache parameters*, schedules tasks on cores so that caches are effectively used at all levels of cache hierarchy.