Utilizing Underlying Synchronization Mechanisms for Efficient Support of Different Programming Models

Nikos Anastopoulos

Computing Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
anastop@cslab.ece.ntua.gr
http://www.cslab.ece.ntua.gr

July 26, 2009

Talk Outline

1. Lab Profile
2. Part I: Supporting Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors
3. Part II: Combining TM with Helper Threads for Exploiting Optimistic Parallelism

Talk Outline

1. **Lab Profile**
2. Part I: Supporting Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors
3. Part II: Combining TM with Helper Threads for Exploiting Optimistic Parallelism

General Info

People:

- Nectarios Koziris (Associate Professor, NTUA)
- 4 post-doc researchers
- more than 15 graduate students

Research Areas

- High performance computing
  - Optimizations for challenging applications
    - Lack of inherent parallelism
    - Memory bandwidth saturation (e.g. SpMxV, Floyd-Warshall)
    - Memory latency (graph algorithms)
  - Studies of architectures' impact on applications
  - Different architectures (PC clusters, CMPs, GPGPUs, Cell B/E)

Research Areas

- Computer architecture
  - Caches for CMPs (e.g. cache-partitioning)
  - SMTs (e.g. thread synchronization, speculative precomputation)
- High performance systems and interconnects
  - Study the effects of shared resources on SMP clusters
  - Focus on I/O and scheduling techniques
- Grid computing & P2P networks and distributed systems
- More info at lab's wiki:
  `http://www.cslab.ece.ntua.gr/cgi-bin/twiki/view/CSLab/`

Recent Work (HPC)

- Transformations to increase data locality
  - "Loop Transformations for Cache-Friendlier Floyd-Warshall", (submitted to ACM Journal of Experimental Algorithms)
- Data compression to decrease memory traffic
  - "Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression", (Comp. Frontiers 2008)
  - "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression", (ICPP 2008)
- Optimizing communication for message-passing applications
  - "Overlapping Computation and Communication in SMT Clusters with Commodity Interconnects", (CLUSTER 2009)

Recent Work (CA)

- CMP caches
  - "An Adaptive Bloom Filter Cache Partitioning Scheme for Multicore Architectures", (SAMOS 2008)
- SMT processors
  - "Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors", (MTAAP 2008)
- Transactional memory
  - "Early Experiences on Accelerating Dijkstra's Algorithm Using Transactional Memory", (MTAAP 2009)
  - "Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm", (ICPP 2009)

Talk Outline

1. Lab Profile
2. **Part I: Supporting Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors**
3. Part II: Combining TM with Helper Threads for Exploiting Optimistic Parallelism

Application Model - Motivation

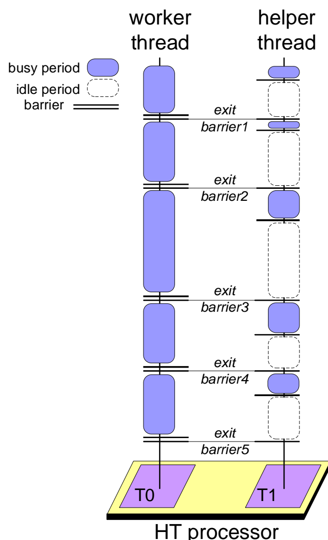
Threads with asymmetric workloads executing on a single HT processor, synchronizing on a frequent basis

In real applications, usually a *helper* thread that facilitates a *worker*

- speculative precomputation
- network I/O & message processing
- disk request completions

How should synchronization be implemented for this model?

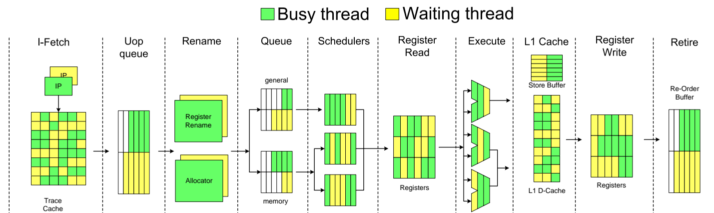
- resource-conservant
- worker: fast notification
- helper: fast resumption



Option 1: spin-wait loops

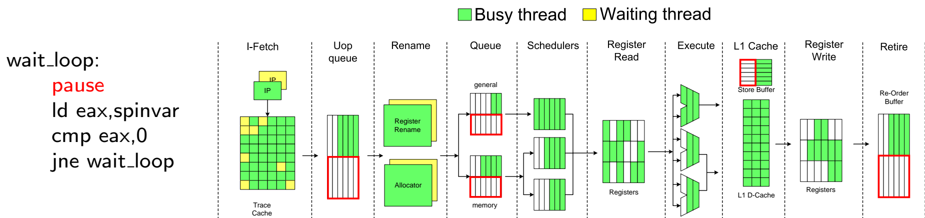
- commonplace as building blocks of synchronization in MP systems
- pros: simple implementation, high responsiveness
- cons: spinning in resource hungry!
  - loop unrolled multiple times
  - costly pipeline flush penalty
  - spins a lot faster than actually needed

```
wait_loop:
    ld eax,spinvar
    cmp eax,0
    jne wait_loop
```



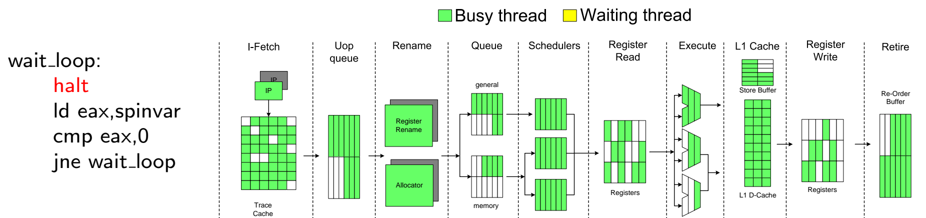
Option 2: spin-wait, but loosen the spinning...

- slight delay in the loop (∼pipeline depth)
- spinning thread effectively de-pipelined → *dynamically shared* resources to peer thread
  - execution units, caches, fetch-decode-retirement logic
- *statically partitioned* resources are not released (but still unused)
  - uop queues, load-store queues, ROB
  - each thread can use at most half of total entries
- up to 15-20% deceleration of busy thread



Option 3: spin-wait, but "HALT"...

- partitioned resources recombined for full use by busy thread (*ST-mode*)
- IPIs to wake up sleeping thread, resources then re-partitioned (*MT-mode*)
- system call needed for waiting <u>and notification</u> ☹
- multiple transitions between ST/MT incur extra overhead



Option 4: MONITOR/MWAIT loops

```
while (spinvar!=NOTIFIED) {
   MONITOR(spinvar,0,0)
   MWAIT
}
```

- condition-wait close to the hardware level
- all resources (shared & partitioned) relinquished
- require kernel privileges
- obviate the need for (expensive) IPI delivery for notification ☺
- sleeping state more responsive than this of HALT ☺

Option 4: MONITOR/MWAIT loops

```
while (spinvar!=NOTIFIED) {
   MONITOR(spinvar,0,0)
   MWAIT
}
```

- condition-wait close to the hardware level
- all resources (shared & partitioned) relinquished
- require kernel privileges
- obviate the need for (expensive) IPI delivery for notification ☺
- sleeping state more responsive than this of HALT ☺

Contribution:

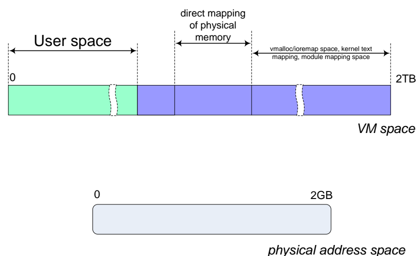
- framework that enables use of MONITOR/MWAIT at user-level, with least possible kernel involvement
  - so far, in OS code mostly (scheduler idle loop)
- explore the potential of multithreaded programs to benefit from MONITOR/MWAIT functionality

Implementing Basic Primitives with MONITOR/MWAIT

- condition-wait:
  - must occur in kernel-space → syscall overhead the least that should be paid...
  - must check *continuously* status of monitored memory
- where to allocate the region to be monitored?
  - in user-space...
    - ★ notification requires single value update ☺
    - ★ on each condition check kernel must copy contents of monitored region from process address space (e.g. via `copy_from_user`) ☹
  - in kernel-space...
    - ★ additional system call to enable update of monitored memory from user-space ☹
  - in kernel-space, but map it to user-space for direct access ☺

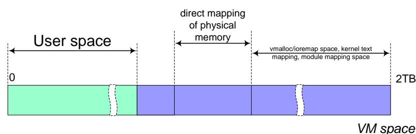
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)



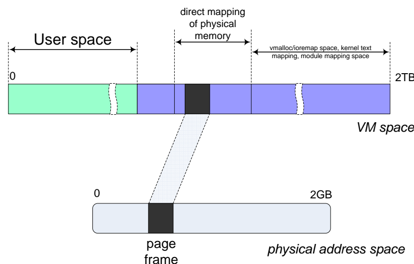
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module



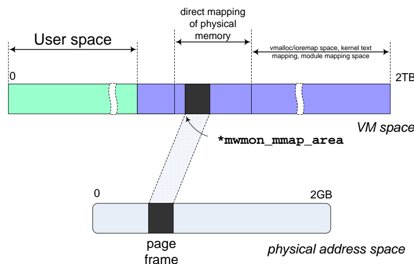
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame



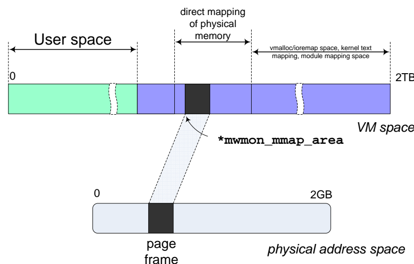
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame



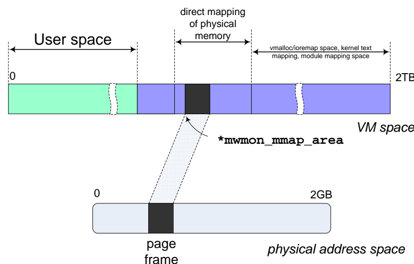
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame
- **open** kmem_mapper
  - initialize monitored region (`MWMON_ORIGINAL_VAL`)



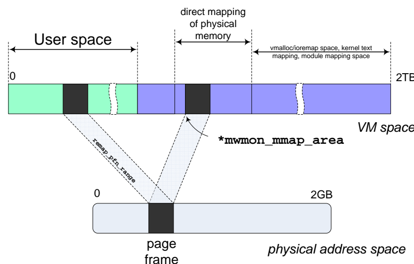
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame
- **open** kmem_mapper
  - initialize monitored region (`MWMON_ORIGINAL_VAL`)
- **mmap** kmem_mapper



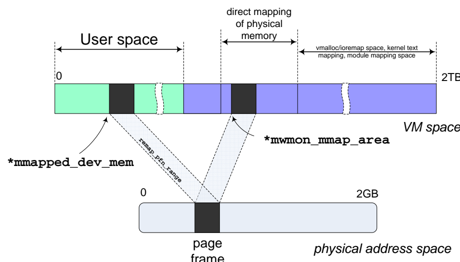
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame
- **open** kmem_mapper
  - initialize monitored region (`MWMON_ORIGINAL_VAL`)
- **mmap** kmem_mapper
  - page-frame remapped to user-space (`remap_pfn_range`)



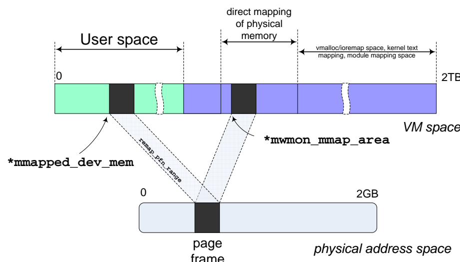
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame
- **open** kmem_mapper
  - initialize monitored region (`MWMON_ORIGINAL_VAL`)
- **mmap** kmem_mapper
  - page-frame remapped to user-space (`remap_pfn_range`)
  - pointer returned points to beginning of monitored region



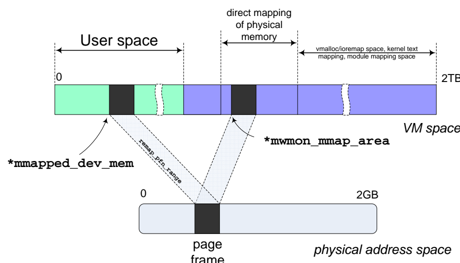
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame
- **open** kmem_mapper
  - initialize monitored region (`MWMON_ORIGINAL_VAL`)
- **mmap** kmem_mapper
  - page-frame remapped to user-space (`remap_pfn_range`)
  - pointer returned points to beginning of monitored region
- **unload** module
  - page `kfree`'d



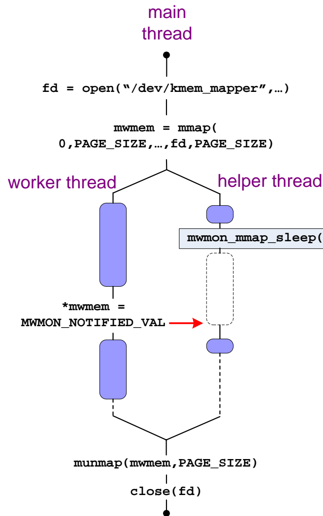
Establishing Fast Data Exchange between Kernel- and User-Space

- monitored memory allocated in the context of a special char device (*kmem_mapper*)
- **load** module
  - `kmalloc` page-frame
  - initialize kernel pointer to show at monitored region within frame
- **open** kmem_mapper
  - initialize monitored region (`MWMON_ORIGINAL_VAL`)
- **mmap** kmem_mapper
  - page-frame remapped to user-space (`remap_pfn_range`)
  - pointer returned points to beginning of monitored region
- **unload** module
  - page `kfree`'d



- `mmapped_dev_mem`: *used by notification primitive at user-space to update monitored memory*
- `mwmon_mmap_area`: *used by condition-wait primitive at kernel-space to check monitored memory*

Use example - System call interface



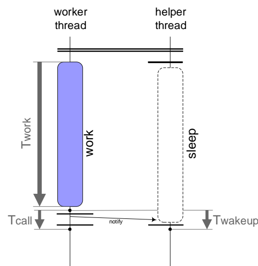

```
asmlinkage long sys_mwmon_mmap_sleep(void)
{
    do {
        local_irq_disable();
        monitor(mwmon_mmap_area,0,0);
        local_irq_enable();
        if(*mwmon_mmap_area == MWMON_NOTIFIED_VAL)
            break;
        mwait(0,0);
    } while (*mwmon_mmap_area != MWMON_NOTIFIED_VAL);
    *mwmon_mmap_area = MWMON_ORIGINAL_VAL;

    return 0;
}
```

System Configuration

- Processor
  - Intel Xeon@2.8GHz (Prescott core), 2 hyper-threads
  - 16KB L1-D, 1MB L2, 64B line size
- Linux 2.6.13, x86_64 ISA
- gcc-4.12 (-O2), glibc-2.5
- NPTL for threading operations, affinity system calls for thread binding on LPs
- `rdtsc` for accurate timing measurements

Case 1: Barriers - Simple Scenario

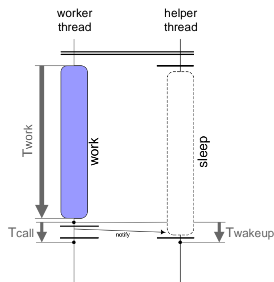
- simple execution scenario:
  - worker: 512×512 matmul (fp)
  - helper waits until worker enters barrier
- direct measurements:
  - $T_{work} \rightarrow$ reflects amount of interference introduced by helper
  - $T_{wakeup} \rightarrow$ responsiveness of wait primitive
  - $T_{call} \rightarrow$ call overhead of notification primitive
- condition-wait/notification primitives as building blocks for actions of intermediate/last thread in barrier



	Intermediate thread *(condition-wait)*		**Last thread** *(notification)*	
		OS?		**OS?**
spin-loops	spin-wait loop + PAUSE in loop body	NO	single value update	NO
spin-loops-halt	spin-wait loop + HALT in loop body	YES	single value update + IPI	YES
pthreads	`futex(FUTEX_WAIT,...)`	YES	`futex(FUTEX_WAKE,...)`	YES
mwmon	`mwmon_mmap_sleep`	YES	single value update	NO

Case 1: Barriers - Simple Scenario

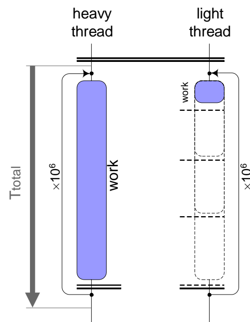
- *mwmon* best balances resource consumption and responsiveness/call overhead
  - 24% less interference compared to *spin-loops*
  - 4× lower wakeup latency, 3.5× lower call overhead, compared to *pthreads*



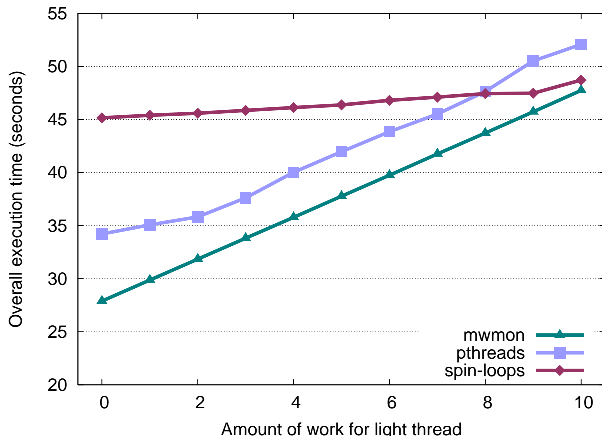
	T_{work} (seconds)	T_{wakeup} (cycles)	T_{call} (cycles)
	lower is better		
spin-loops	4.3897	1236	1173
spin-loops-halt	3.5720	49953	51329
pthreads	3.5917	45035	18968
mwmon	**3.5266**	**11319**	**5470**

Case 2: Barriers - Fine-grain Synchronization

- varying workload asymmetry
  - unit of work = 10×10 matmul (fp)
  - *heavy* thread: always 10 units
  - *light* thread: 0-10 units
- 10^6 synchronized iterations
- overall completion time reflects throughput of each barrier implementation



Case 2: Barriers - Fine-grain Synchronization



Across all levels of asymmetry *mwmon* outperforms *pthreads* by 12% and *spin-loops* by 26%

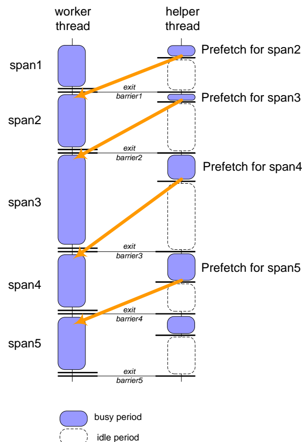
- converges with *spin-loops* as threads become symmetric
- constant performance gap w.r.t. *pthreads*

Case 3: Barriers - Speculative Precomputation (SPR)

Thread based prefetching of top L2 cache-missing loads (*delinquent loads – DLs*)

In phase k helper thread prefetches for phase k+1, then throttled

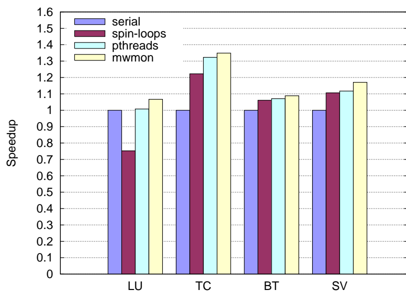
- phases or *prefetching spans*: execution traces where memory footprint of DLs $< \frac{1}{2}$ L2 size



Benchmarks

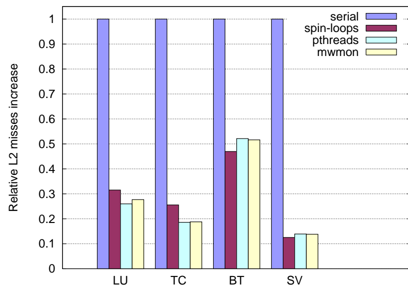
Application	Data Set
LU decomposition	2048×2048, 10×10 blocks
Transitive closure	1.6K vertices, 25K edges, 16×16 blocks
NAS BT	Class A
SpMxV	9648×77137, 260785 non-zeroes

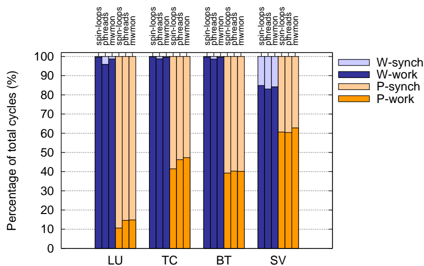
Case 3: SPR Speedups and Miss Coverage



mwmon offers best speedups, between 1.07 (LU) and 1.35 (TC)

- with equal miss-coverage ability succeeds in boosting "interference-sensitive" applications
- notable gains even when worker delayed in barriers and prefetcher has large workload





Conclusions

mwmon primitives make the best compromise between low resource waste and low call & wakeup latency

- efficient use of resources on HT processors
- MONITOR/MWAIT functionality should be made available to the user

Possible directions

- *mwmon*-like hierarchical schemes in multi-SMT systems (e.g. tree barriers)
- other "producer-consumer" models (disk/network I/O applications, MPI programs, work-queuing models, etc.)
- multithreaded applications with irregular parallelism

Talk Outline

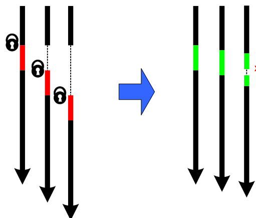
1. Lab Profile
2. Part I: Supporting Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors
3. **Part II: Combining TM with Helper Threads for Exploiting Optimistic Parallelism**

Motivation

- TM community needs real-world applications
- Graph algorithms are described as good candidates for TM, due to irregular accesses of data structures
- Dijkstra's algorithm
  - fundamental SSSP algorithm
  - widely used
  - inherently serial, thus challenging to parallelize
  - previous attempts resulted in major changes in algorithm's semantics (e.g. Δ-stepping, Boost implementation)
- Early results published in MTAAP'09, extended version will appear in ICPP'09

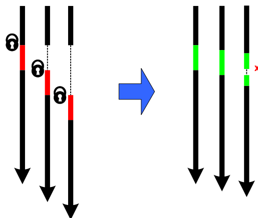
Main Idea

Conventional use of TM: optimistic synchronization

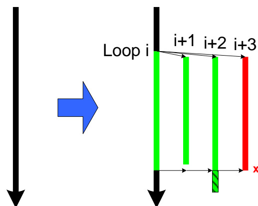


Main Idea

Conventional use of TM: optimistic synchronization



Our view of TM: optimistic parallelization



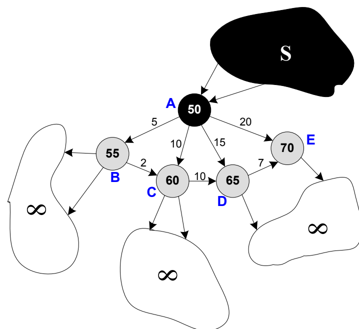
The Basics of Dijkstra's Algorithm

Serial algorithm

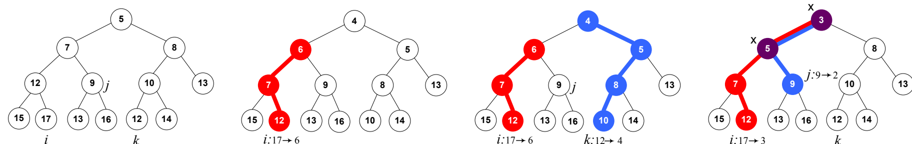
Input : $G = (V, E)$, $w : E \rightarrow \mathbf{R}^{+}$, source vertex s, min Q

Output : shortest distance array d, predecessor array π

foreach $v \in V$ **do**
  $d[v] \leftarrow$ INF;
  $\pi[v] \leftarrow$ NIL;
  Insert(Q, v);
end
$d[s] \leftarrow 0$;
while $Q \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(Q);
  foreach *v adjacent to u* **do**
    $sum \leftarrow d[u] + w(u, v)$;
    if $d[v] > sum$ **then**
      DecreaseKey(Q, v, sum);
      $d[v] \leftarrow sum$;
      $\pi[v] \leftarrow u$;
  end
end



The Basics of Dijkstra's Algorithm



Min-priority queue implemented as binary min-heap

- maintains all but the settled ("optimal") vertices
- min-heap property: $\forall i : d(parent(i)) \leq d(i)$
- amortizes the cost of multiple `ExtractMin`'s and `DecreaseKey`'s
  - $O((|E| + |V|)log|V|)$ time complexity

Straightforward Parallelization

Fine-grain parallelization at the inner loop level

Fine-Grain Multi-Threaded

```
/* Initialization phase same to the serial
   code */
while Q ≠ ∅ do
    Barrier
    if tid = 0 then
        u ← ExtractMin(Q);
    Barrier
    for v adjacent to u in parallel do
        sum ← d[u] + w(u, v);
        if d[v] > sum then
            Begin-Atomic
            DecreaseKey(Q, v, sum);
            End-Atomic
            d[v] ← sum;
            π[v] ← u;
    end
end
```

Issues

- speedup bounded by average out-degree
- concurrent heap updates due to `DecreaseKey`'s
- barrier synchronization overhead

Evaluation

- conventional synch. mechanisms yield major slowdowns
- TM
  - better performance
  - highlights optimistic parallelism
  - suffers from barriers overhead

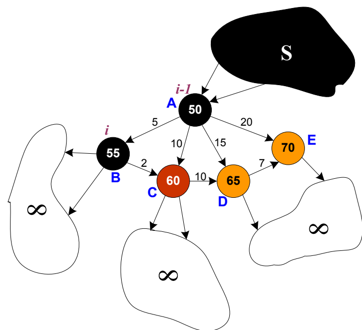
Helper-Threading Scheme

Motivation

- expose more parallelism to each thread
- eliminate costly barrier synchronization

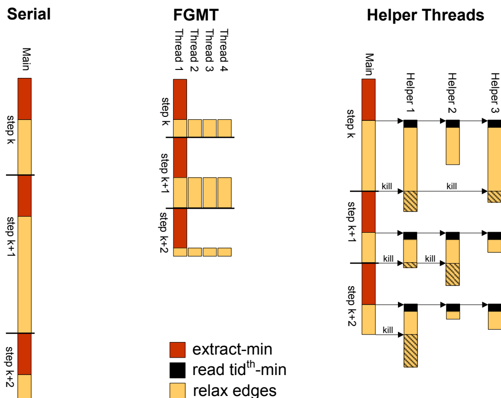
Rationale

- in serial, updates are performed only from *definitely* optimal vertices
- allow updates from *possibly* optimal vertices
  - main thread operates as in the serial case
  - helper threads are assigned the next minimum vertices (x_k) and perform updates from them



- speculation on the status of x_k
  - if already optimal, main thread will be offloaded
  - if not optimal, any suboptimal relaxations will be corrected eventually by main thread

Execution Pattern



Decoupling of sequential/parallel parts is achieved through TM

- the main thread stops all helpers at the end of each iteration
- unfinished work will be corrected, as with mis-speculated distances

Helper-Threading Scheme

Main thread

while $Q \neq \emptyset$ **do**
  $u \leftarrow \texttt{ExtractMin}(Q)$;
  $done \leftarrow 0$;
  foreach *v adjacent to u* **do**
    $sum \leftarrow d[u] + w(u, v)$;
    Begin-Xact
    if $d[v] > sum$ **then**
      $\texttt{DecreaseKey}(Q, v, sum)$;
      $d[v] \leftarrow sum$;
      $\pi[v] \leftarrow u$;
    End-Xact
  end
  Begin-Xact
  $done \leftarrow 1$;
  End-Xact
end

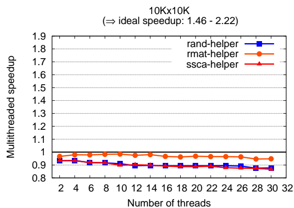
Helper thread

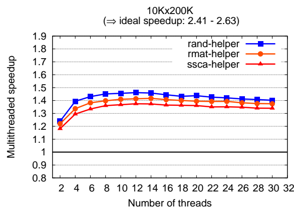
while $Q \neq \emptyset$ **do**
  while $done = 1$ **do** ;
  $x \leftarrow \texttt{ReadMin}(Q, tid)$
  $stop \leftarrow 0$
  foreach *y adjacent to x* **and while** $stop = 0$ **do**
    Begin-Xact
    if $done = 0$ then
      $sum \leftarrow d[x] + w(x, y)$
      if $d[y] > sum$ **then**
        $\texttt{DecreaseKey}(Q, y, sum)$
        $d[y] \leftarrow sum$
        $\pi[y] \leftarrow x$
    else
      $stop \leftarrow 1$
    End-Xact
  end
end

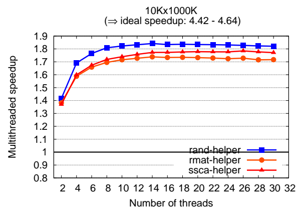
Why with TM?

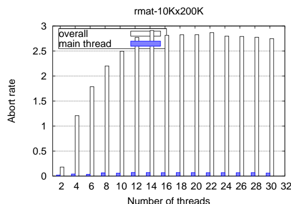
- composable
  - all dependent atomic sub-operations composed into a large atomic operation, without limiting concurrency
- optimistic
- easily programmable

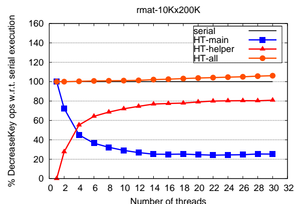
Performance Evaluation

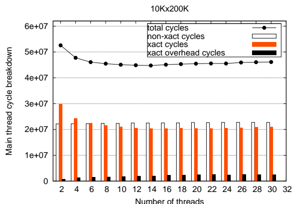












- Simics 3.0.31, GEMS 2.1, LogTM-SE
- speedups in 15 out of 18 graphs, up to 1.84 (max ideal speedup = 4.64)
- main thread not obstructed by helpers (<1% abort rate in all cases)

Conclusions

HT+TM scheme

- exposes more parallelism and eliminates barrier synchronization
- noteworthy speedups with minimal code extensions

Future work

- TM for optimistic *parallelization*
  - HT+TM as a programming model for other graph problems (MSTs, maximum flow, SSSP) and other similar ("greedy") applications
  - adjustments of existing TM systems for explicitly supporting speculative parallelization

Thank you!

Questions?