Multicore Processors & Microparallelism

Lawrence Spracklen
Sun Microsystems

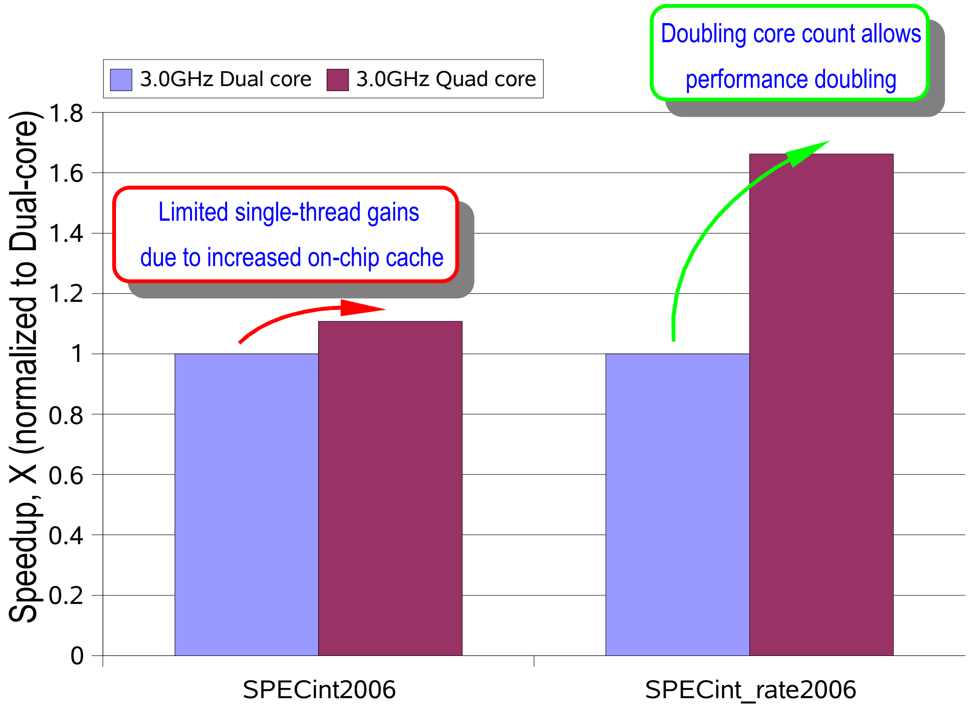
lawrence.spracklen@sun.com

Overview

- Next generation processors
- Exploiting the advantages of multicore
- The challenges of multicore architectures
- “Microparallelism”
- Final tweaks
- Conclusions

Next generation processors

- Single-thread performance has stagnated
  - Gains coming from compiler optimizations & immense on-chip caches
- Processor core count doubling with each new generation
- Multithreaded software is becoming essential
  - Only way to benefit from new processors
- Next-generation multiprocesors present some interesting challenges
- To-date rudimentary multi-threading has generally sufficed
- Going forward complete & efficient multithreading will be necessary



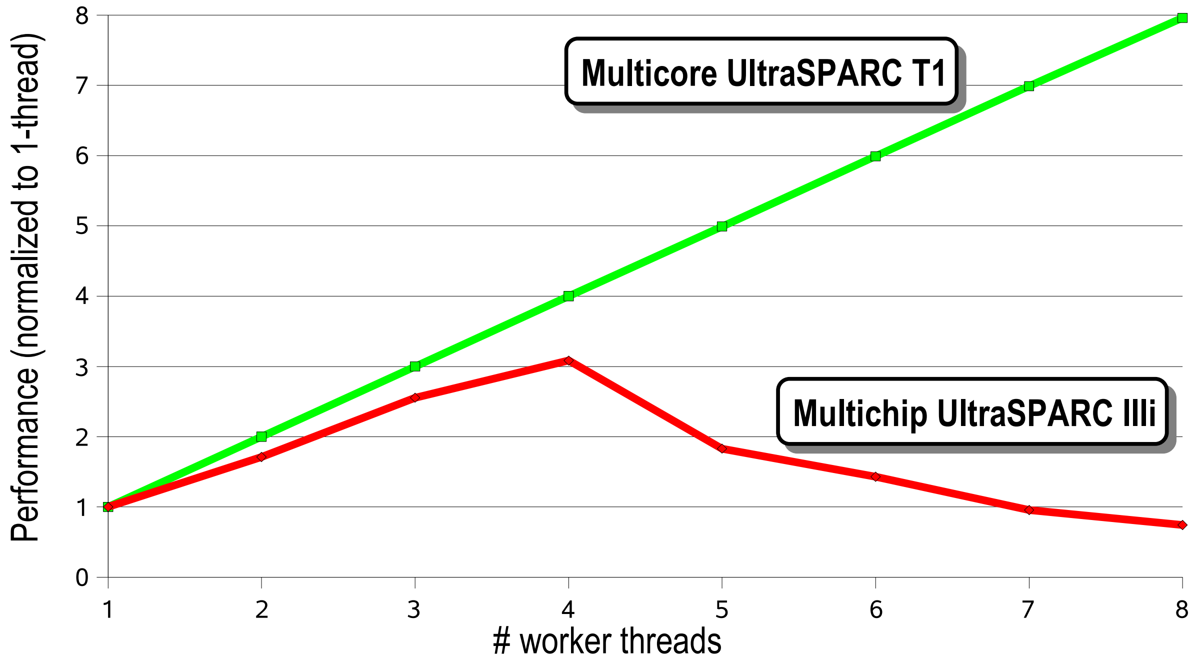
Multicore & MT code

- 2 competing factors affect the ease of parallelism
  1) More threads sharing cache resources
  2) More threads in total
- (1) accelerates inter-thread communication making threading easier
  > HW designs already mitigating many of the negative impacts of resource sharing
- (2) requires improved scaling efficiency making threading complex
  > Most multiprocessor configurations already present tens of threads; trend will accelerate
- Multithreading is **required** to achieve significantly improved performance moving from one processor generation to the next
- We may soon need to start augmenting traditional threading techniques to achieve desired performance
- Much can potentially be automated by next-generation compilers

Benefits of cache sharing #1

- Significantly reduced performance impact from hot locks
  - Reduced lock ping-ponging compared to traditional SMP systems
- Can greatly simplify the process of introducing critical sections
  - Reduces burden of iterative lock tweaking

```
void *
worker_thread(void *arg)
{
   int i, tmp = 0;
   int *data;

   data = (int *)(arg)

   for (i= 0; i < SIZE; i++)
   {
        tmp += data[i];
   }

   mutex_lock(&accum_mutex);
   global_accum += tmp;
   mutex_unlock(&accum_mutex);

   return 0;
}
```

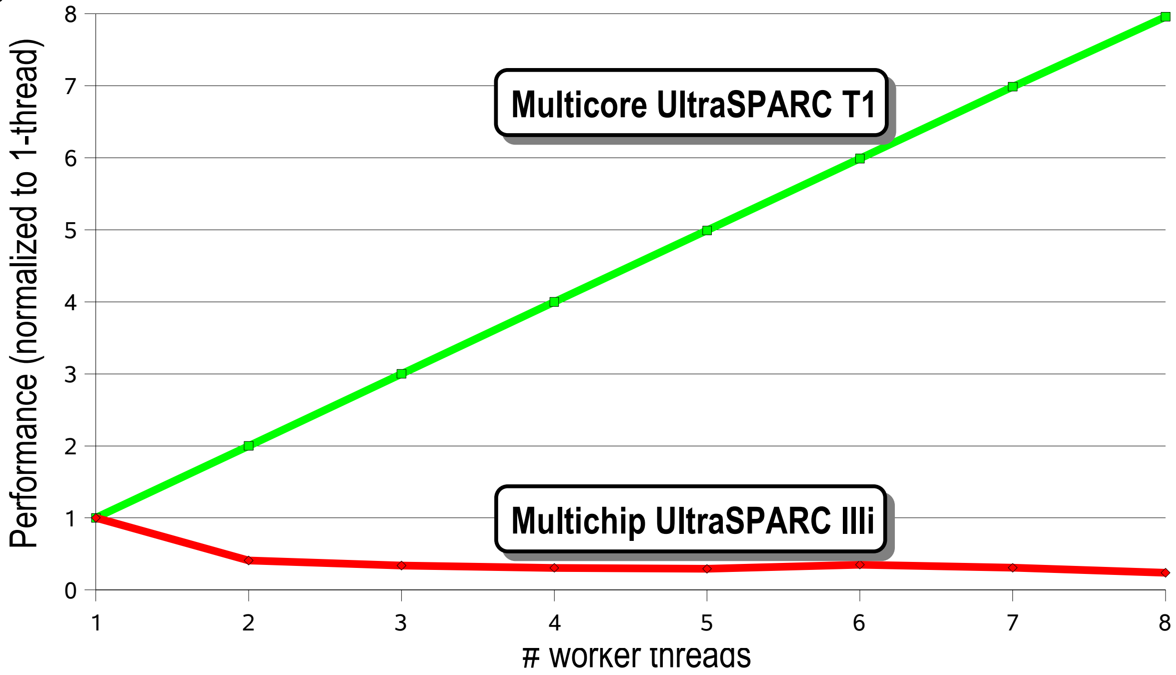


- Very heavily contended locks are still problematic though....

Benefits of cache sharing #2

- Data layout was critical to ensure no false sharing
- Frequently necessitated data layouts be modified
  - Significantly increases cost of threading single-threaded code
  - Potentially error prone process

```
void *
worker_thread(void *arg)
{
  int i, tmp = 0;
  int id = thr_self();
  int *data;

  data = (int *) arg;

  for (i= 0; i< SIZE; i++)
  {
       thr_accum[id] += data[i];
  }

  return 0;
}
```

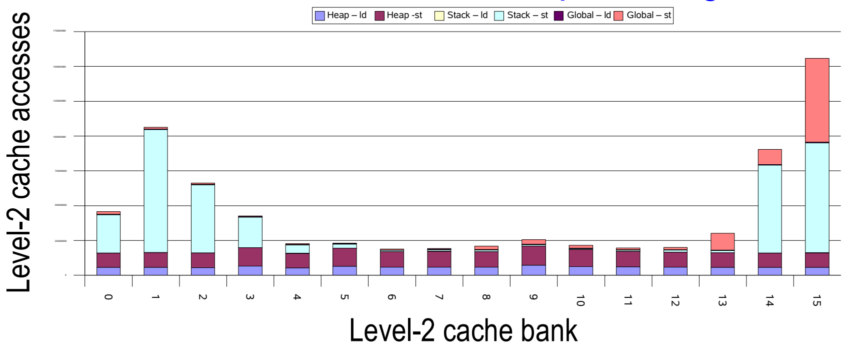


- Performance benefits still associated with eliminating false sharing
  - Magnitude dependent on closeness of shared cache

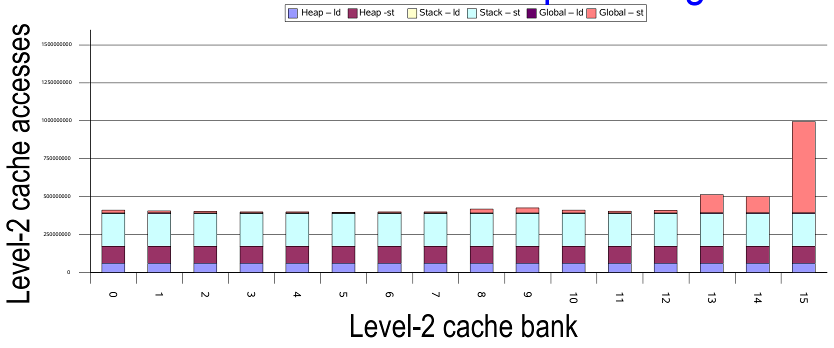
Hardware offers a helping hand

- Simple hardware & OS enhancements can help prevent pathological problems associated with highly shared caches
  - Hot sets
  - Hot banks

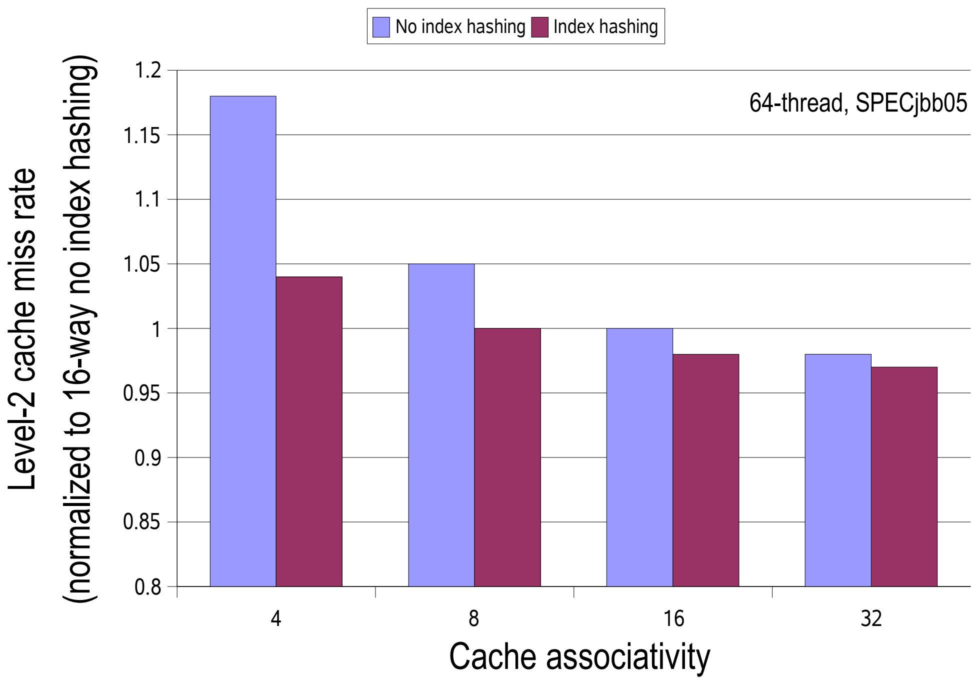
No SW stack or heap slewing



SW stack and heap slewing

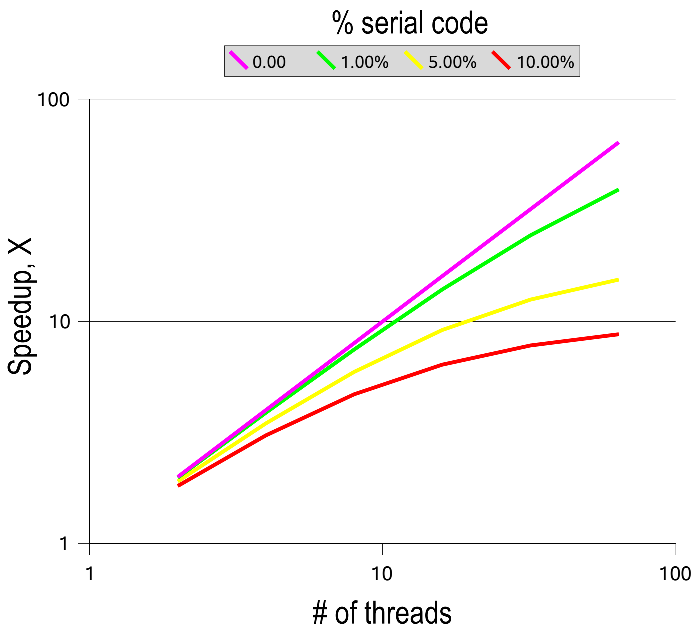


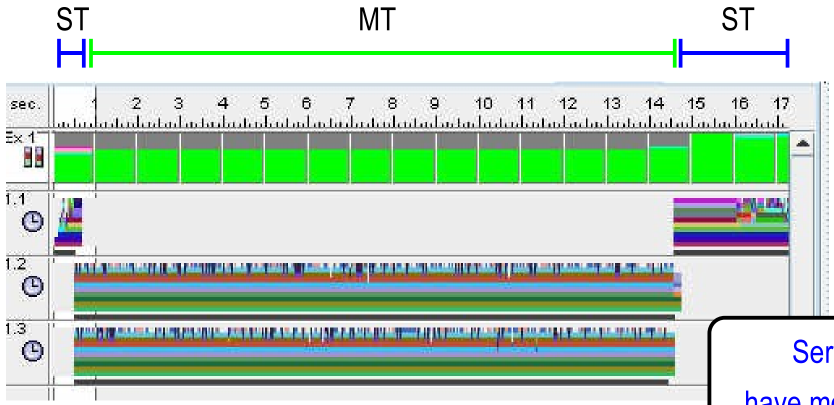
HW index hashing



Implications of Amdahl's law

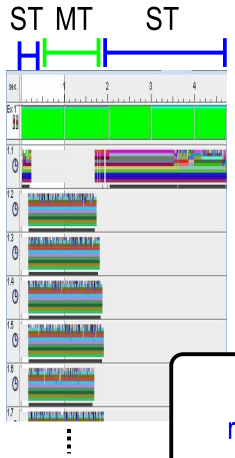
- More complete MT coverage is required as the # of threads is increased
  - > Even modest single-thread components rapidly dominate execution time & curtail scaling





2T run
(1.2GHz N1)

Serial components have modest performance impact with limited thread count



32T run
(1.2GHz N1)

Serial components rapidly become performance limiters as thread count increases

Avoiding Amdahl's implications

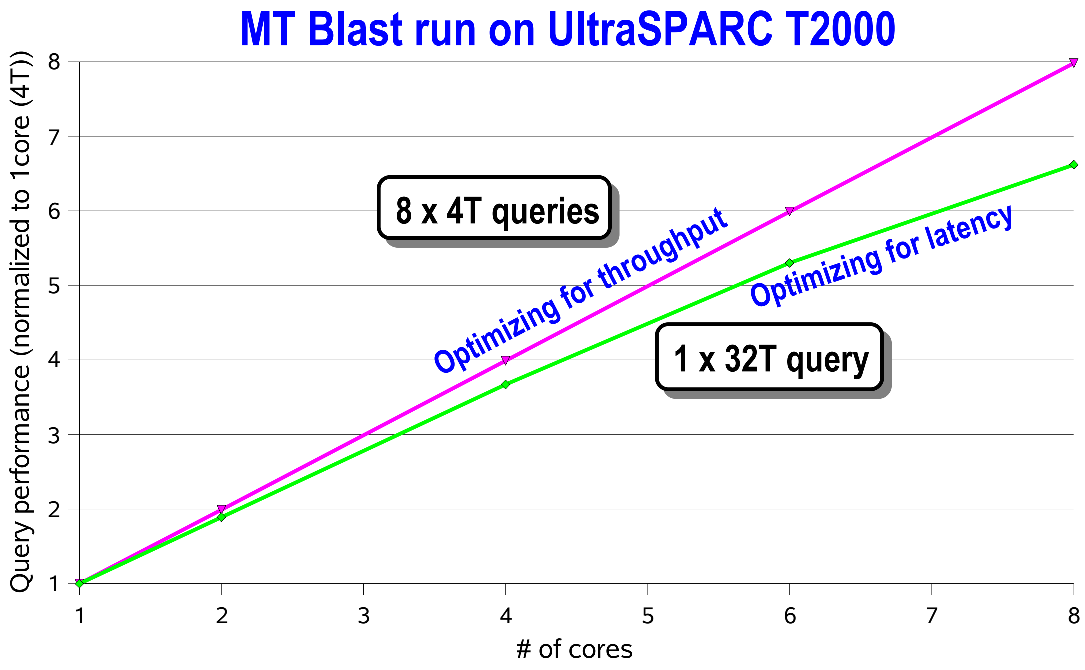
- Many tasks lend themselves to division between multiple instances of an application

Benefits:

- Efficiency
- Simplicity
- Robustness

Cons:

- Introduces load balancer requirements
- Places a significant burden on the caches
- Not always practical



Microparallelism

- Serial components are commonplace in multithreaded applications
  - Most will need to be eliminated in order to achieve acceptable performance on next-generation multicore processors
- Difference between practically serial and fundamentally serial
- Multicore processors enable fine-grain parallelization that was previously unprofitable
- This “Microparallelism” involves dividing small chunks of work between multiple threads
- Microparallelism helper threads are assigned to help master threads rapidly process performance limiting serial components
  - Bottlenecks easy to spot with existing tool chains
- Microparallelism is key to eliminating single-threaded performance limiters

Uses of Microparallelism

Microparallelism attacks a variety of serial code problems:

- **Single-threaded components** – rapidly curtail scaling as thread count is increased
- **Small tasks** – short tasks interposed via synchronization points make threading challenging
- **Critical threads** – scaling may halt once critical threads are 100% busy
- **Critical sections –** scaling is impacted once the threads begin to stall waiting for access

Microparallelism applies to classic single thread situations or when further sub-dividing MT work

- Microparallelism can be simpler and is less intrusive than traditional coarse grain threading
  - > Makes it easy to retrofit existing codes
- Scope of Microparallelism is dictated by inter-thread communication/synchronization overheads

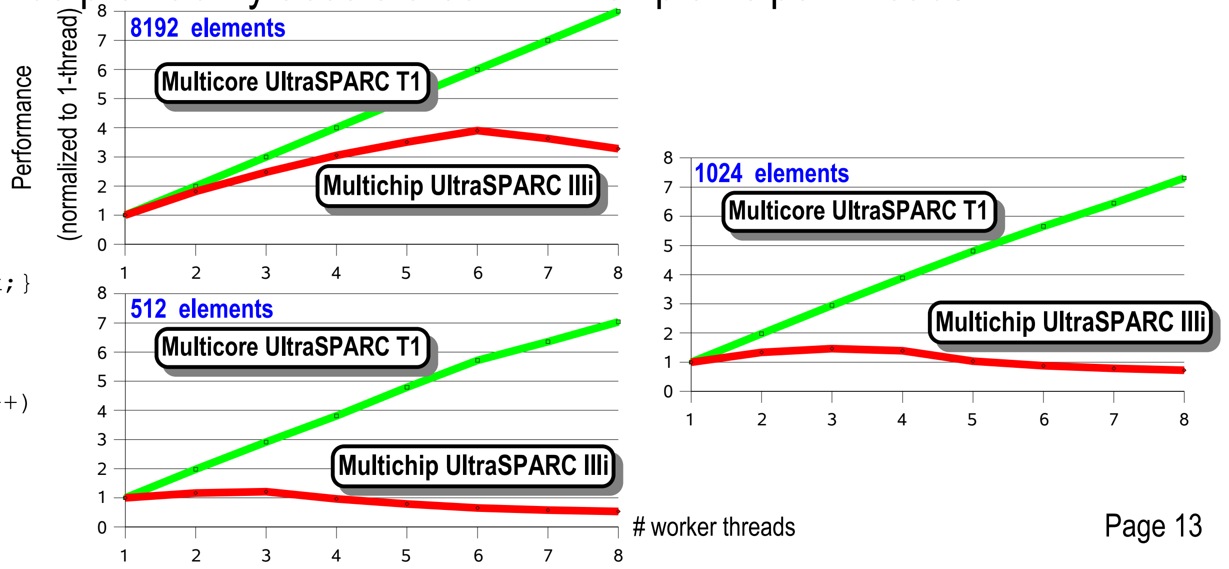
Light-weight synchronization

- Current inter-thread synchronization primitives are typically too heavy-weight for Microparallelism
  - > Up to 700-cycles for a semaphore post on a recent Intel processor running Linux
  - > Impacts the profitability of many Microparallelism opportunities
- Optimal to use own synchronization methods
  - > Frequently easy to employ lock-free synchronization
  - > Interaction between master and helper threads is often simple producer/consumer
  - > Made easier as the interface between helper and master can be tailored to each interaction
- Helper threads spin-wait until they are pointed to new work
- Master thread ensures all helpers complete before proceeding
- Possible to defer the sync point to boost performance even further
  - > Master can offload all processing processing for task A to the helpers and begin processing task B if there are no data dependencies – only check for completion when actually necessary
- A single helper thread can easily provide acceleration for multiple microparallelized tasks across multiple master threads

Microparallelism example #1

- Many serial sections not amenable to traditional threading
- However, these sections are potentially composed of multiple small threadable sections
  - > These operations (e.g. low trip-count loops) traditionally not profitable to thread
  - > Aggregate work performed across all of these sections is significant
- With microthreading consider each section independently and leverage helper threads to accelerate each section separately
  - > Even very short sections can be profitability accelerated with multiple helper threads

```
void *
worker_thread(void *arg)
{
  int i, id = thr_self();
  int *off;

  off = (int *)(arg)

  while (1) {
         //Wait for work
         while (1) {if (start[id]) break;}

         start[id] = 0;

         //Perform copy
         for (i = off[0]; i < off[1]; i++)
               dst[i] = src[i];

         //Signal completion
         finish[id] = 1;
  }
}
```



Profitability

- If work to be undertaken is variable, dynamic profitability analysis is required
- In the previous example it was simple to divide the work between the master and helper threads
  - Makes dynamic profitability analysis simple
- Unfortunately, such an even division of work is not always feasible, making determination of profitability tricky
- However, light-weight synchronization reduces the overheads incurred by the master thread to just a few loads and stores
- Kick-starting the helper threads is a trivial overhead unless the amount of work is very small or ‘failure’ is too frequent
- Possible to employ Microparallelism even if the work to be undertaken by the helper(s) may be occasionally unneeded

Microparallelism example #2

- Consider parsing a string for a specific character sequence
  - Divide the string into multiple regions and hand each region to a separate thread
  - Potentially accelerates processing very significantly -- especially if the sequence is located at the start of the final region
  - However, if desired sequence is located in the master thread's region, threading is purely overhead
- No requirement for master to wait until helpers complete if the master locates the desired sequence
  - Helper needs to complete before next invocation, but master signals early completion

MASTER

Preamble

Update communication structure with latest directions for helper

//Ensure helper is ready

```
While (sync0 == 1);
```

//Kick-start helper

```
sync0 = 1;
```

WORK/2

//Signal early completion iff helper results aren't necessary

```
Sync0 = 2;
```

//Wait for completion iff helper results are required

```
While (sync0 == 1);
```

HELPER

Possible to handle requests for help with multiple operations

//Wait for request for help

```
While ((sync0 == 0) && (sync1 == 0) ...);
```

Loop

Preamble

Load work info from updated communication structure

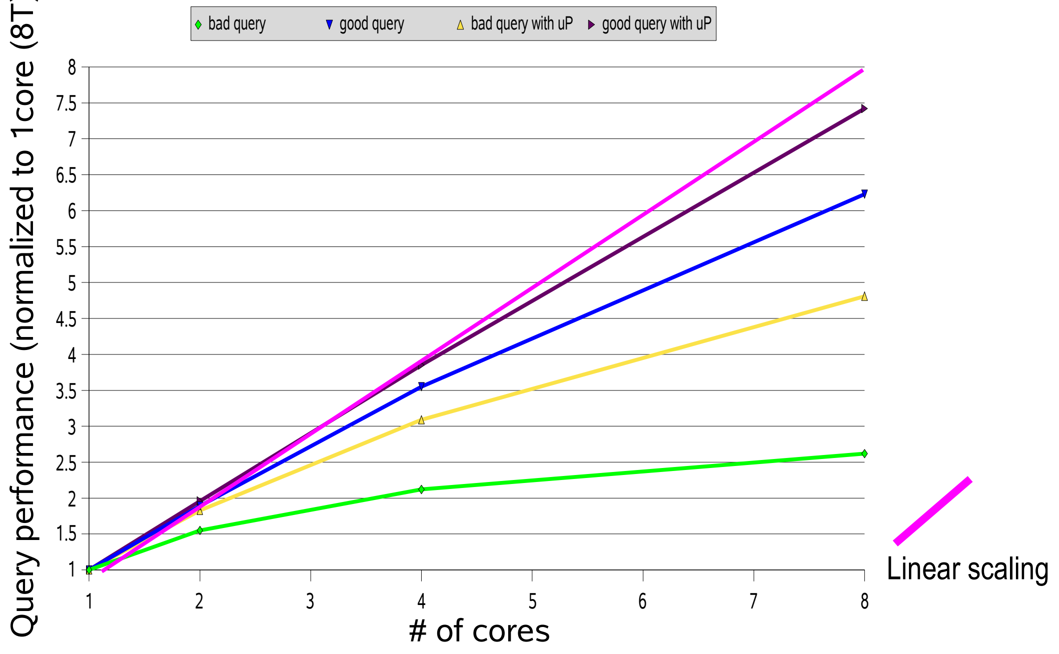
WORK/2 *[periodically checking if early completion requested]*

```
sync0 = 0;
```

Benefits of Microparallelism

- Delivering even 2-4X performance improvement in the single-threaded sections can significantly improve overall scaling
  > Typically just want to deploy 1-7 helper threads to handle Microparallelism
- While the scope of Microparallelism can be impacted by data dependencies, significant opportunity is apparent in many common codes

Blast performance with 3X acceleration of single-thread component



SPECcpu2000; % of loops to which Microparallelism could be safely applied*

Benchmark	% of loops	Benchmark	% of loops
168.wupwise	84.89	164.gzip	12.18
171.swim	45.45	175.vpr	8.72
172.mgrid	30.59	176.gcc	8.37
173.applu	38.92	181.mcf	2.5
177.mesa	29.22	186.crafty	10.52
178.galgel	35.76	197.parser	4.07
179.art	29.13	252.eon	47.59
183.equake	37.89	253.perlbmk	8.7
187.facerec	30.93	254.gap	6.87
188.ammp	5.77	255.vortex	0.44
189.lucas	49.5	256.bzip2	15.84
191.fma3d	49.94	300.twolf	8.68
200.sixtrack	40.96		

*Data from Zoran Radovic [No profitability considerations]

Final tweaks

- Thread placement is important
  - In multiprocessor systems master and helper threads should reside on the same processor
  - Even in uniprocessor systems, thread placement can be important depending on the specifics of the cache hierarchy
- Maximise utilization of each core's resources
  - Mix compute intensive and memory intensive threads
- Heterogeneous cores
  - Disable SMT on cores used by critical single threads
  - Potentially provides a not insignificant boost in performance – gains need to be balanced against losses incurred from reduced thread count
- Significant problems if a processor's cores don't share on-chip cache resources
  - Eliminates Microparallelism opportunities

Conclusions

Stagnation in single-thread performance, coupled with industry-wide focus on increasing core/thread count is radically impacting the way programmers need to tackle multithreading

- Multithreaded applications don't just need to scale to 4-threads
  - 16, 32, 64 and beyond are already commonplace
- Increasing thread count requires applications to be almost fully threaded to ensure decent scalability
- Low inter-thread communication latencies on multicore processors make fine-grain interaction feasible
- This Microparallelism can be employed to thread serial application components that are not amenable to traditional threading techniques
- Even limited acceleration of an application's remaining serial components via Microparallelism can translate into significant improvements in overall application scalability

Questions?