Micro-benchmarking the Tera MTA

E. Jason Riedy
ejr@cs.berkeley.edu

Rich Vuduc
richie@cs.berkeley.edu

Cs258 – Spring 1999

Outline

- Motivation
- Basics: Threads, Streams, and Teams
- Memory System
- Synchronization
- Parallel Challenge: List Ranking
- Tera-able or Terrible?

Prior Work

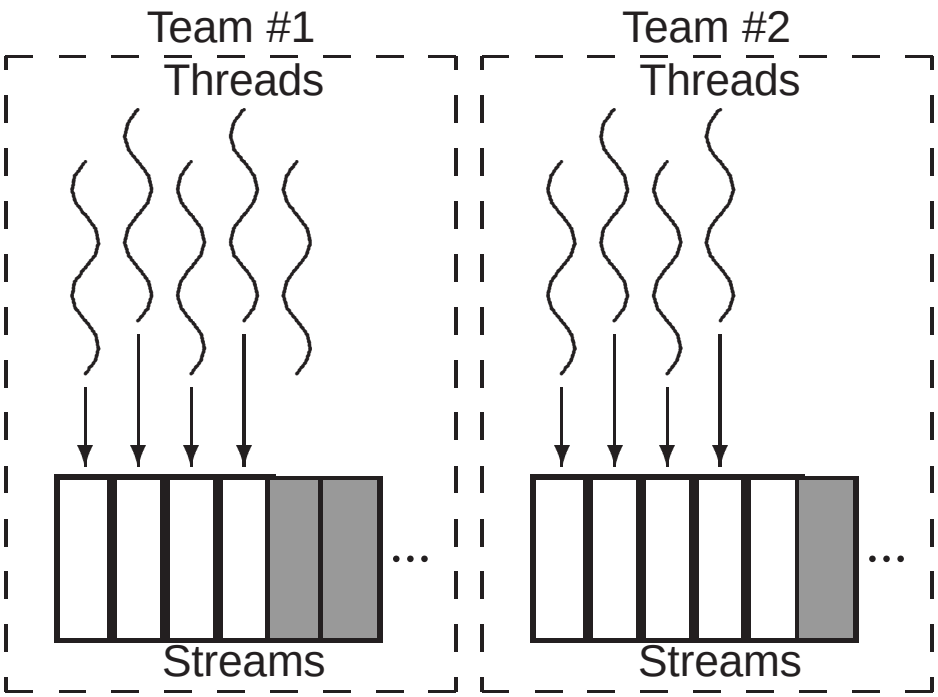
- Recent evaluations have been "macrobenchmarks"
  1. NAS Parallel benchmarks (Snavely, et al., SC'98)
  2. C3I Benchmarks (Brunett, et al., SC'98)
  3. Volume visualization (Snavely, et al., ASTC'99)
- Basic result: Apps need many threads.
- The documentation already says that.

  $\Rightarrow$ So it runs, but what is happening under the hood?

Prior Findings

1. Decent baseline performance on most benchmarks (conjugate gradient, integer sort, multigrid) which vectorize nicely
2. Poor performance on FFT benchmark, which is tuned for memory heirarchy performance
3. Shaky scalability claims?
4. Air Force benchmark; Command, control, communication, and intelligence
5. C3I was selected because it was easy to parallelize — not a very interesting choice, is it?
6. Poor sequential execution

Threads, Streams, and Teams



- A *team* corresponds to a CPU.
- A *thread* is the unit of work specified in the program.
- A *stream* is an instruction-issuing slot within the CPU.

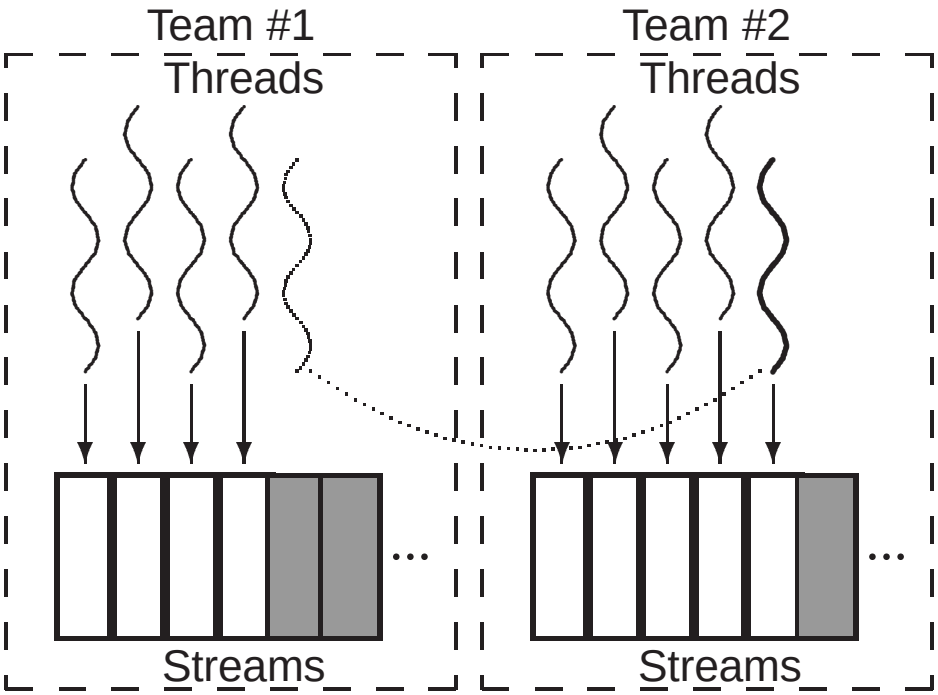
NOTES to slide 5

1. Teams are currently defined to be anonymous CPUs, but that may change.

2. Threads are the units created and managed through the Tera runtime libraries.

3. Streams are created in hardware. Creation involves an instruction to reserve a number of streams, then a sequence of `STREAM_CREATE` instructions. Each stream has its own full register file. The compiler does this directly in some circumstances.

Thread Migration

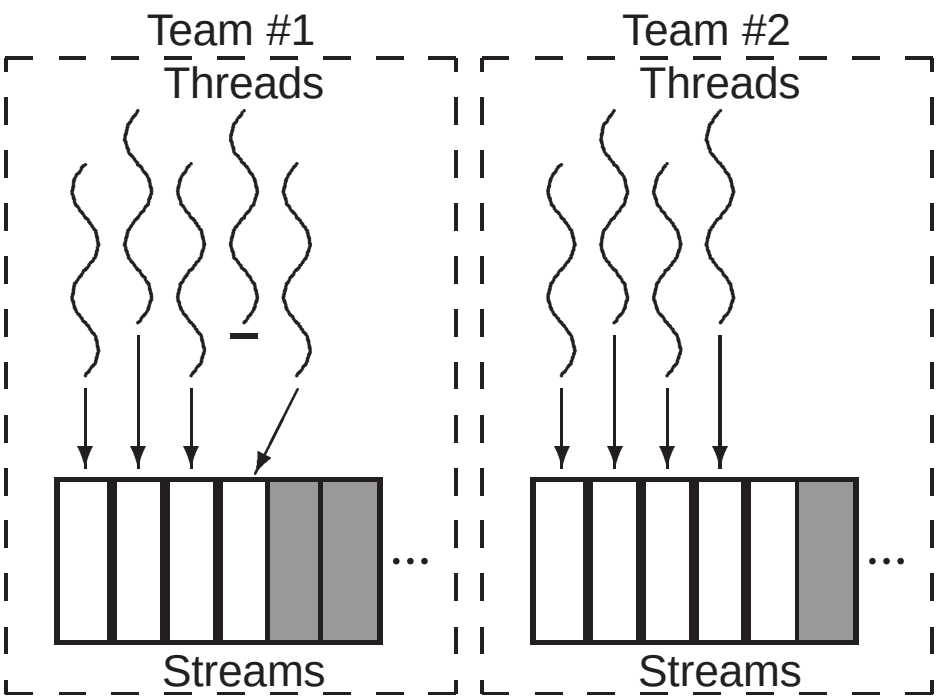


When there is excess parallelism, the runtime migrates threads.

NOTES to slide 6

1. Because teams are anonymous, threads can be transparently migrated.
2. The runtime also tries to manage the cost of team creation...
3. There is no way to keep track of migration, and no way to measure the overhead. It's likely a trap, a save of about 70 words, and some notification of the other team. The run-time streams on the other team probably watch for new threads, but there could be an inter-processor interrupt.
4. If the program is not already executing in the other team, there will be some cold-start misses.

Thread Re-mapping



Blocked streams are re-mapped by the runtime.

- Streams block due to traps and politeness (system calls).
- Streams are not re-mapped due to latency, unless the latency is so large that it traps out.

NOTES to slide 7

1. Streams can also be locked to a stream.

Measuring the Re-mapping Cost

Outer thread:

```
terart_create_thread_on_team (...);
purge (&end_time);
writeef (&start_time, TERA_CLOCK (0));
readff (&end_time);
```

Inner thread:

```
writeef (&end_time, TERA_CLOCK (0));
```

From 100 samples:

	Clocks	μseconds
Mean	1.72×10^5	6.60×10^{-1}
Std	1.55×10^4	5.95×10^{-2}

NOTES to slide 8

1. A memory access is on the order of 10^3 cycles.
2. Most run-time features (especially auto-growth) must be disabled. We disable them in all micro-benchmarks.

Measuring the Stream Creation Cost

Outer thread:

```
purge (&end_time);
writeef (&start_time, TERA_CLOCK (0));
terart_create_stream (...);
readff (&end_time);
```

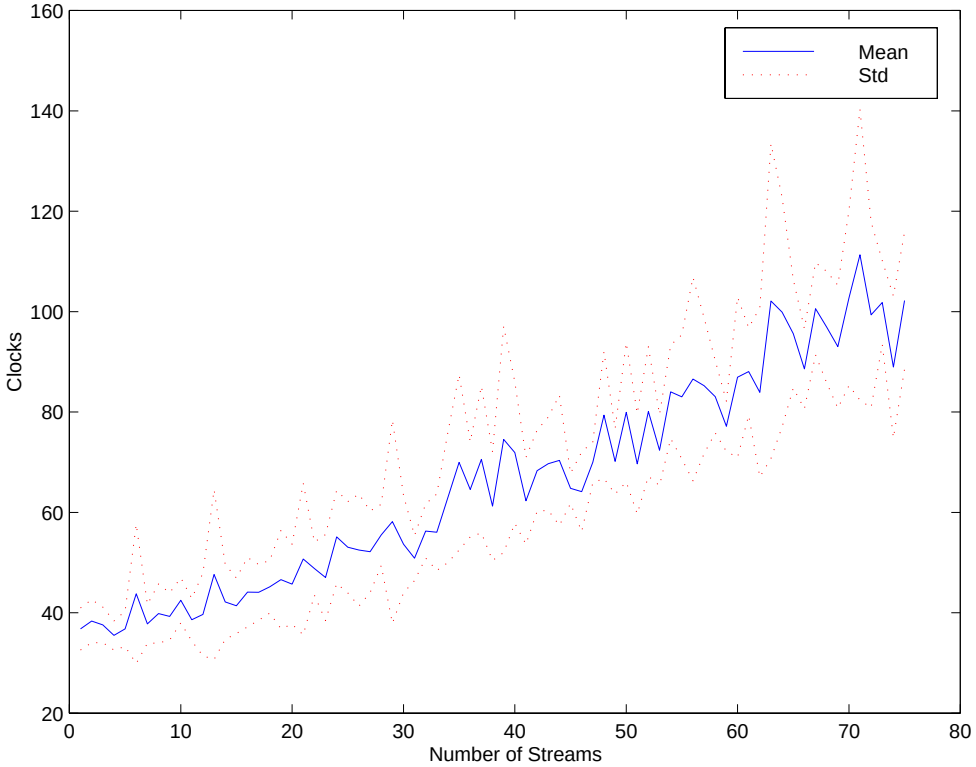
Inner thread:

```
writeef (&end_time, TERA_CLOCK (0));
```

From 100 samples, through runtime:

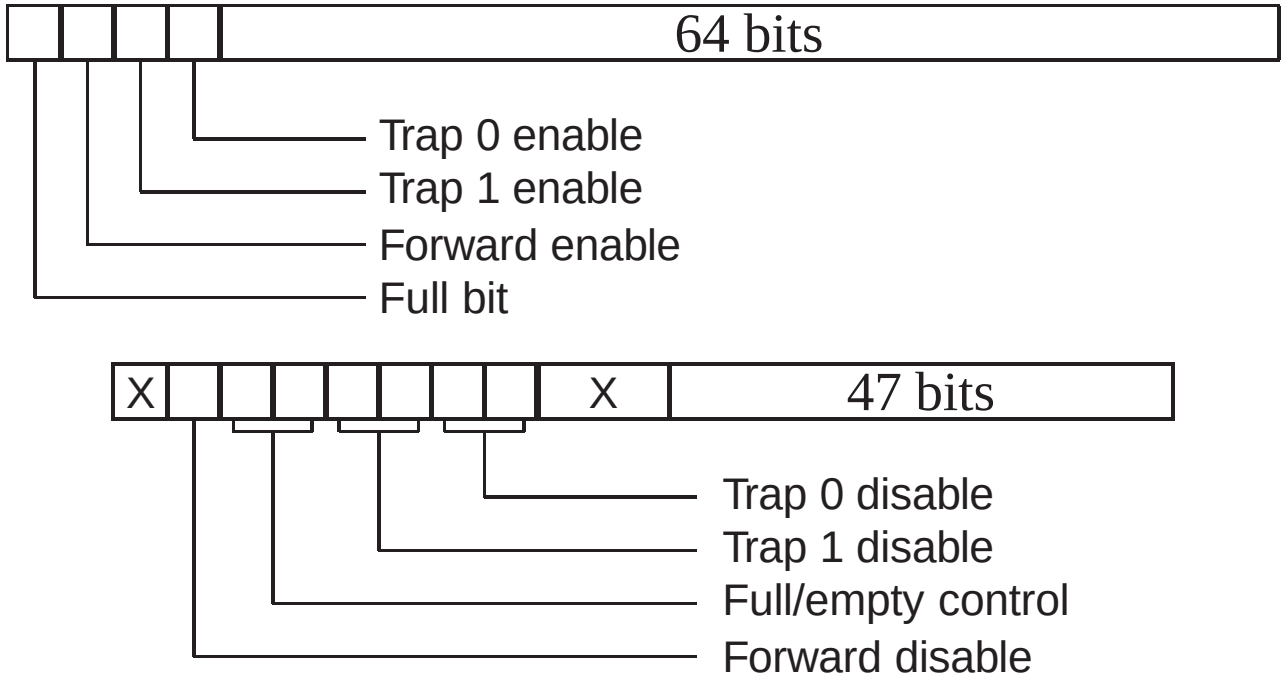
	Clocks	μseconds
Mean	2.72×10^4	1.00×10^{-4}
Std	9.72×10^2	3.74×10^{-6}

How Many Instructions Is That?



The minimum possible instruction latency is 21, the depth of the pipeline. The maximum *should* be 128, the number of hardware streams, if scheduling is strictly round-robin. The CPU prefers previously unready streams, however.

The Tera Memory System



- The segment size can vary from 8 KB to 256 MB.
- The TLB (512 entry, 4-way assoc.) holds privilege information and can toggle stalling and distribution.
- The maximum memory size is currently $2^{42} = 4$ TB, not 2^{47}.

NOTES to slide 11

1. There can be up to 8 memory transactions per stream outstanding at once; branches can optionally wait for these to complete.

2. The stall field allows the OS to perform quick memory management without having to stop all work.

3. Memory accesses can be re-distributed so consecutive addresses are spread over the entire memory space.

4. The TLB coherence mechanism is not specified.

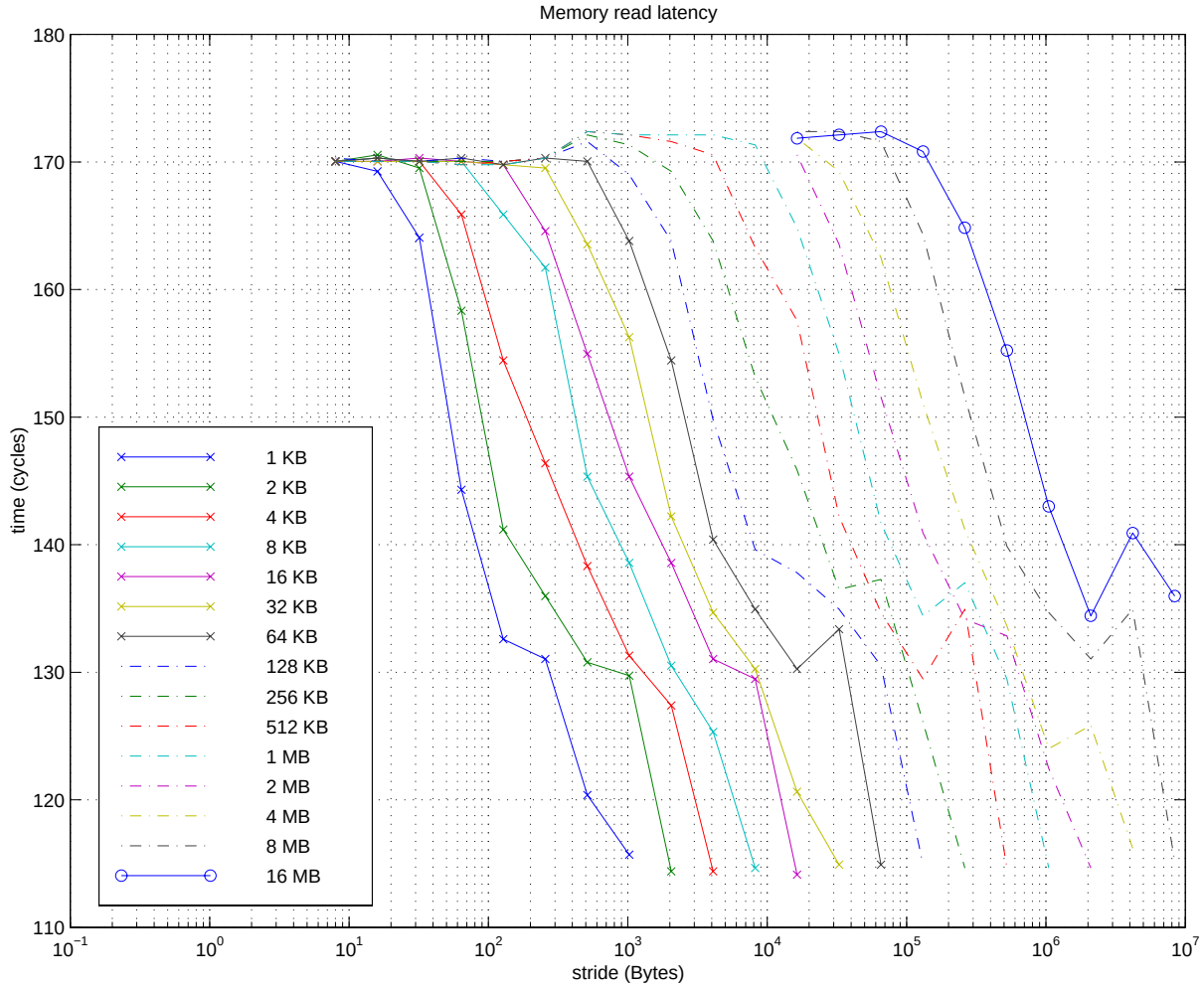
5. Privileges are handled as a stack.

Memory Bandwidth

The standard STREAM benchmark compiled with Tera's parallelizing compiler shows fair speedup for almost no effort.

Teams	Bandwidth
1	1.7 GB/s
4	5.8 GB/s
Speedup	3.4

Memory read latency



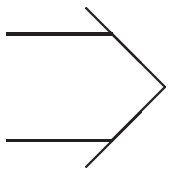
LMBENCH results, not parallelized

NOTES to slide 13

1. Measured the time to perform a load by streaming through an array of memory addresses
2. Drop-off: hot-spot caches
3. Highest point: maximum time to perform a load
4. Lowest point: round-trip network latency
5. Difference: corresponds with claimed memory access time
6. Number of points between any drop-off and the end: 8

Effects of Software Pipelining

```
for i = 1 to n,
    read A[i]
    loop
        operate on A[i]
```



```
for i = 1 to n step 2,
    read A[i]
    read A[i+1]
    loop
        operate on A[i]
        operate on A[i+1]
```

We measured the apparent bandwidth as a function of

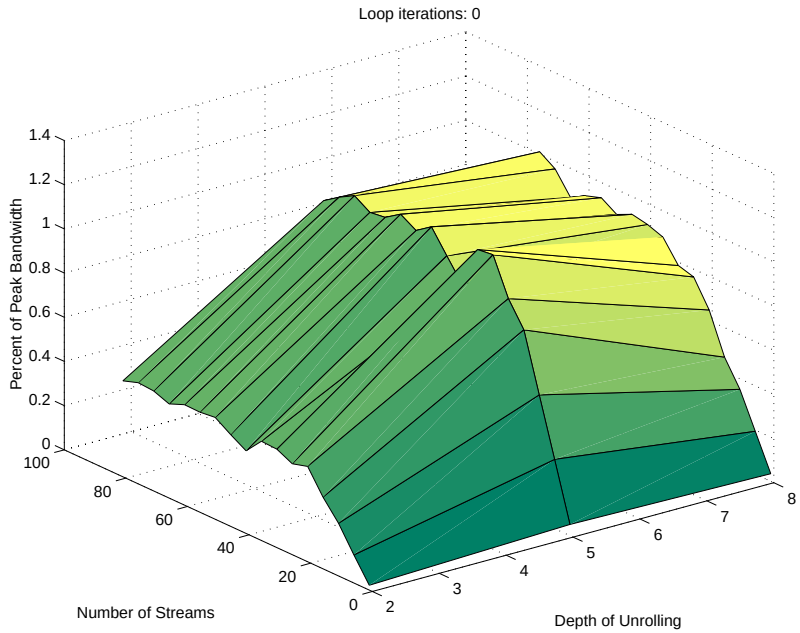
- the depth of unrolling,
- the number of streams,
- the number of teams, and
- the number of loop iterations.

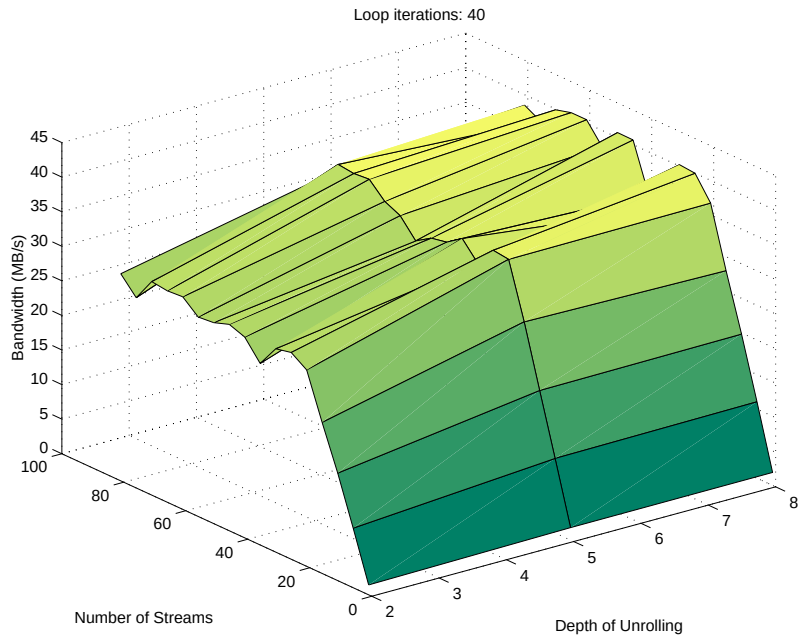
NOTES to slide 14

1. The loop does trivial floating-point work, but the compiler is instructed not to optimize floating-point operations.

2. All loops are due by the time the loop begins because of this.

Bandwidth by Depth and Streams





Depth	N. Streams	% of Peak
2	51	46%
5	26	111%
8	76	101%

Depth	N. Streams	% of Peak
2	71	60%
5	76	76%
8	51	80%

NOTES to slide 15

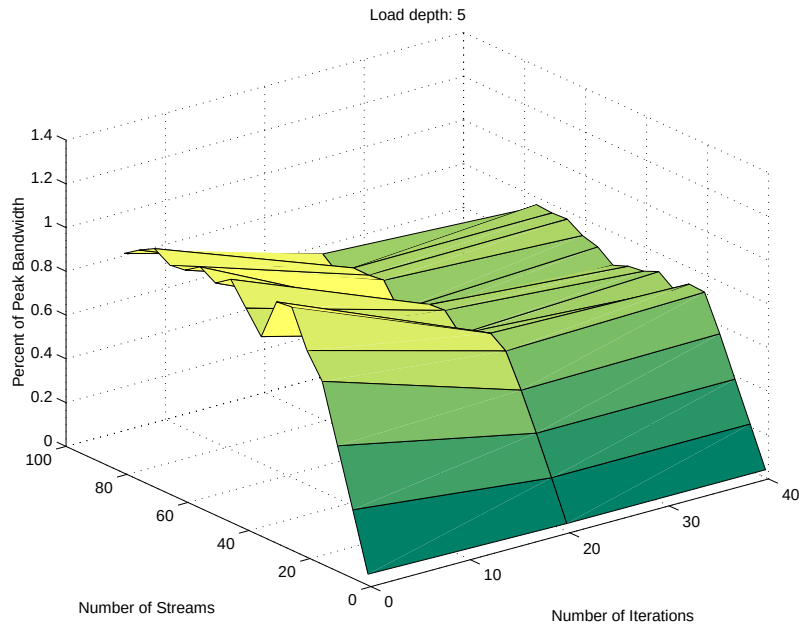
1. We assume that the Tera can load one word (8 bytes) per clock cycle, giving a peak of 2.08 GB/s.

2. The $> 100\%$ results must be due to multiple loads completing at once.

3. The loop overhead is very small; the scheduler packs the instructions into other parts of the instruction word.

4. The number of instructions words in the loop is actually a bit over half the number of instructions. Two floating-point additions can be packed into one instruction.

Bandwidth by Iterations and Streams



N. Iterations	N. Streams	% of Peak
0	26	111%
20	61	76%
40	76	77%

NOTES to slide 16

1. Bandwidth used scaled linearly with teams, once adjusted by number of streams per processor.

The Full / Empty Bit

- Cases when retrieving a value from memory:

  Full Return with the value, possibly toggling the f/e state.

  Forwarding Retry with the new address.

  Empty Retry with the same address

- Too many retries triggers a trap.

  1. Allocate a handler structure.
  2. Copy the current value into that structure.
  3. Set forwarding to that value.
  4. Set the appropriate trap bit in memory.
  5. Add current stream's thread to waiting list and re-map the stream.

The Full / Empty Bit

- The only interesting case when storing a value is one that changes the f/e state to full.
- When this happens, the access will trap.
  1. Ignoring forwarding, fetch the pointer in memory.
  2. Subtract off the offset, giving the handler structure.
  3. Store the value and turn off forwarding and trapping in the original location.
  4. Wake threads as appropriate, allowing them to be mapped to streams again.
  5. De-allocate the handler structure.

NOTES to slide 18

1. There must be some synchronization again in this. This is carried out in by the program's linked runtime environment, so it is only local to the program. Most likely: The thread doing this is locked to a stream, preventing traps while waiting for another stream to set this up.

2. The operations necessary to set this up are privileged. The right traps raise the privilege enough to do this.

Number of Retries

The Tera will keep track of the total number of retries if requested.

1. Purge two sync variables.
2. Create a stream.
   (a) Read (ff) from the first sync variable.
   (b) Write the retry counter to the second.
3. Spin in registers long enough for the other stream to trap.
4. Write the retry counter to the first sync variable.
5. Spin in registers long enough for the other stream to set the counter result.
6. Print second and first variables.

Result: `2045 1022`

Result, locking: `95409 95021`

A Look Back at Re-mapping...

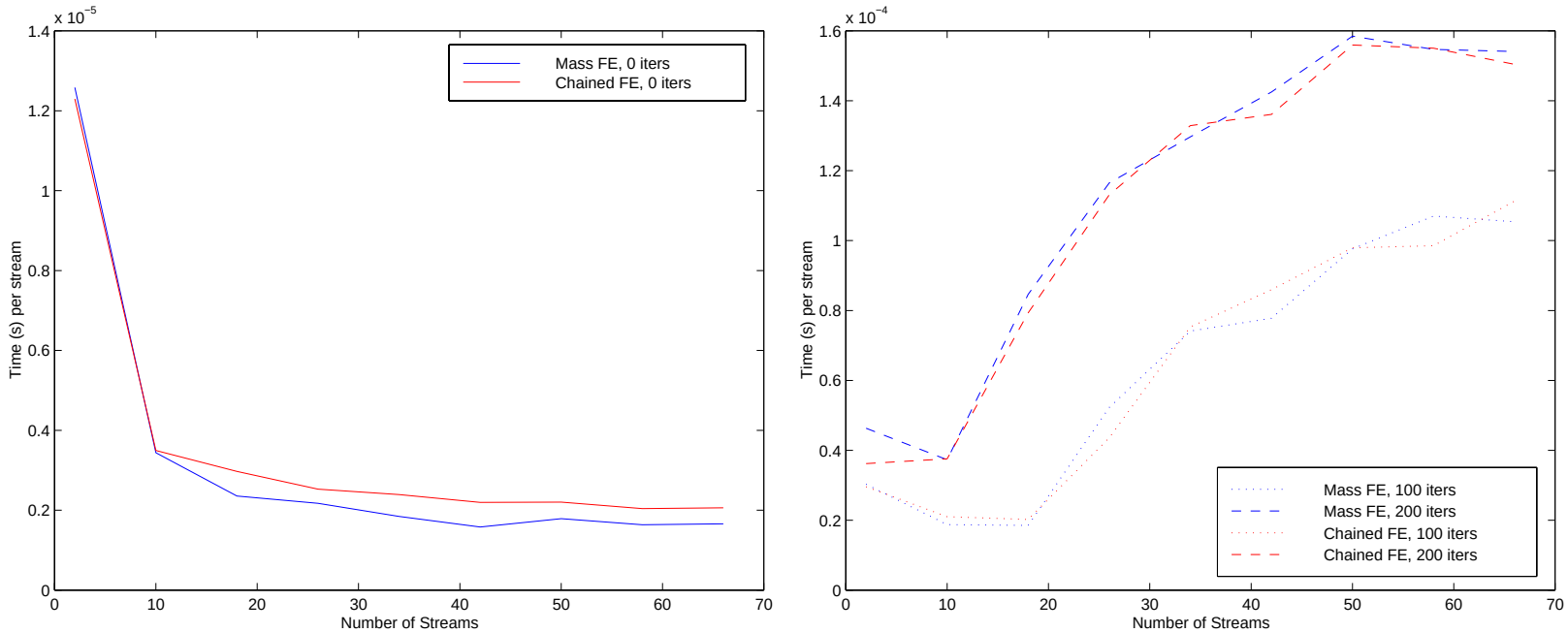
Total		172 000
Retries	1024 accesses $\times$ 120 cycles / access $=$	122 880
Remaining		49 120
State write	70 words $\times$ 170 cycles / word $=$	11 900
State read		11 900
Remaining		25 320
Insn words	30 cycles per instruction $\rightarrow$	$<$ 900

The documentation claims that re-mapping threads takes on the order of a few hundred instructions. Assuming a bit more memory traffic, this result agrees.

NOTES to slide 20

1. Note that it is likely to be $< 1\,800$ instructions, as the compiler seems to average two insns in book-keeping sections.

Synchronization Costs



- The times include a read, at least a test, and a write for each stream.
- One memory transaction should take 6.5×10^{-7}s, two $.13 \times 10^{-5}$s.
- Using only one location might get a boost from the hot-spot cache (or request merging?), but that becomes insignificant with a load.

NOTES to slide 21

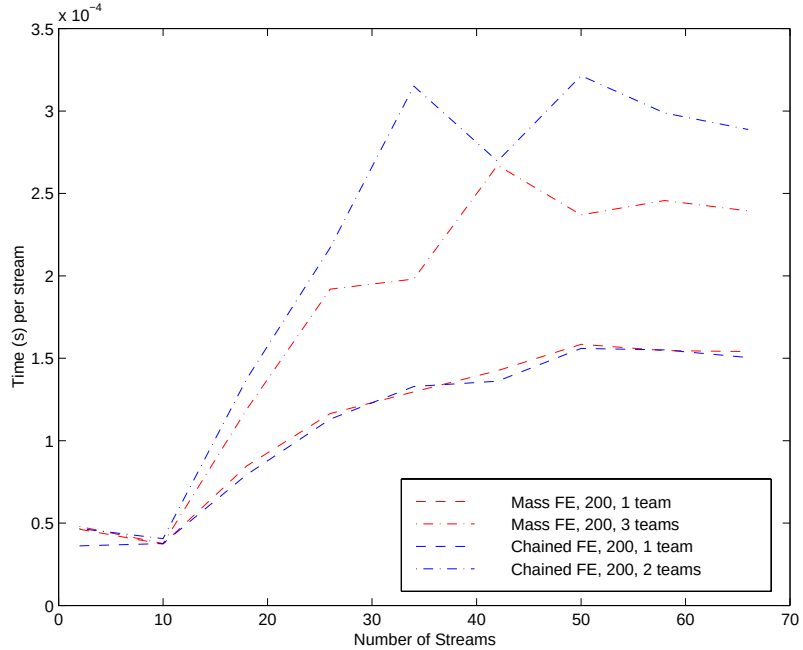
1. The “Chained FE” test measures the time per element to traverse a linked list, emptying a full variable and filling the next.

2. The “Mass FE” test is similar, but all threads are emptying and filling the same location.

3. Only two non-runtime streams are active, so the insn latency should be dominated by the memory latency.

4. Does `mass-fe` cause a thundering herd problem? Or does the trap handler note that the thread to be restarted will leave the memory empty, so it only starts one?

Multi-Team Synchronization Costs



- Until the streams begin to trap, there is no difference between multiple and single-team synchronization.
- Note that each stream in "Chained FE" wakes a stream in another team.

List ranking (and list scan)

- List ranking: given a linked-list, find the distance of each node from the head (tail) of the list.
- Basic parallel primitive
- Practical parallel implementation is a challenge: irregular data structure, communication intensive
- We examine three practical algorithms
  1. Straight-forward serial implementation
  2. Wyllie's pointer jumping algorithm (Wyllie, 1979)
  3. Reid-Miller/Blelloch (1994)
- Straight-forward serial implementation is tough to beat!

NOTES to slide 23

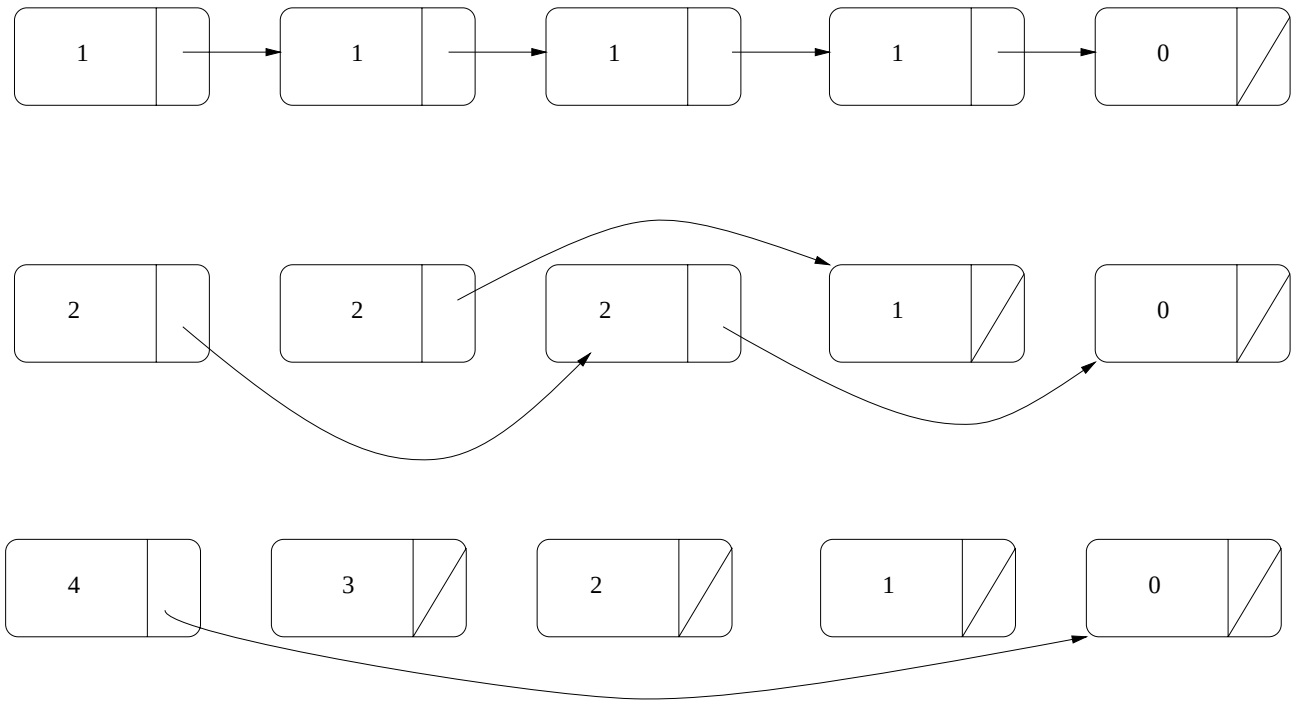
1. List ranking is a special case of list scan; our implementations are actually scan implementations

2. Uses: ordering elements of a list, finding the Euler tour of a tree, load balancing (Gazit, et al. '88, "Optimal tree construction..."), scheduling, parallel tree contraction

3. Other important practical algorithms which we did not try: randomized algorithms by Andersen/Miller, Reif/Miller

Wyllie's Pointer Jumping algorithm

(Wyllie, 1979)



- Idea: Each node adds itself to its neighbor, then links to its neighbor's neighbor
- Simple but not work efficient — O$(n \log n)$ total ops.

NOTES to slide 24

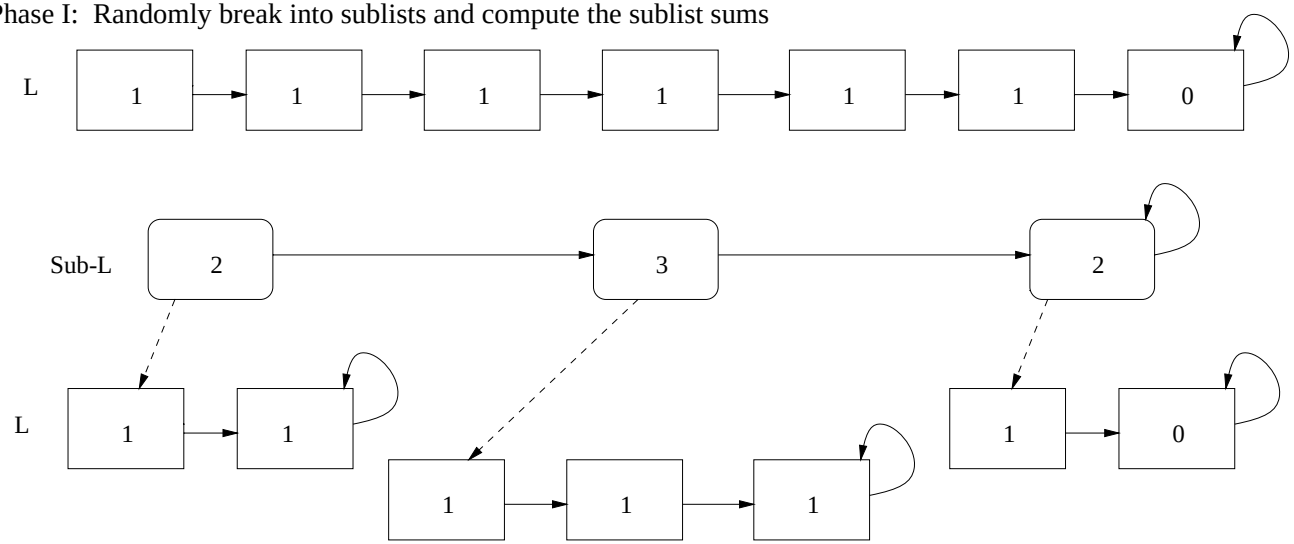
1. At each bulk step, double the number of pointers skipped; must synchronize to get $\log n$ factor

2. Also, must synchronize accesses to value and next fields together.

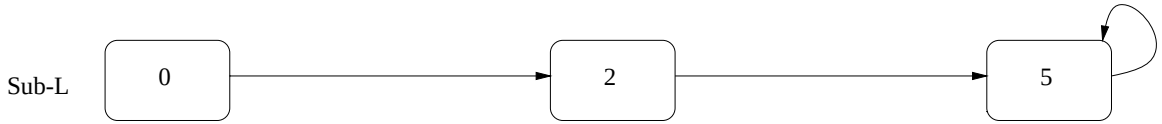
Reid-Miller algorithm

(Reid-Miller and Blelloch, 1994)

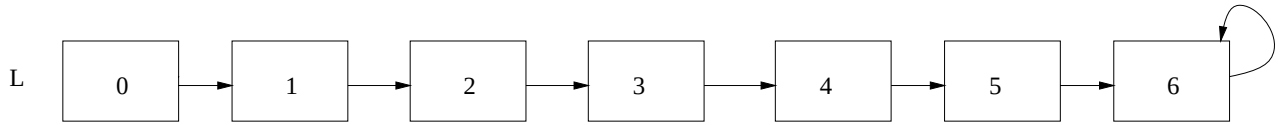
Phase I: Randomly break into sublists and compute the sublist sums



Phase 2: Find the list scan of the sublist list (Sub-L)



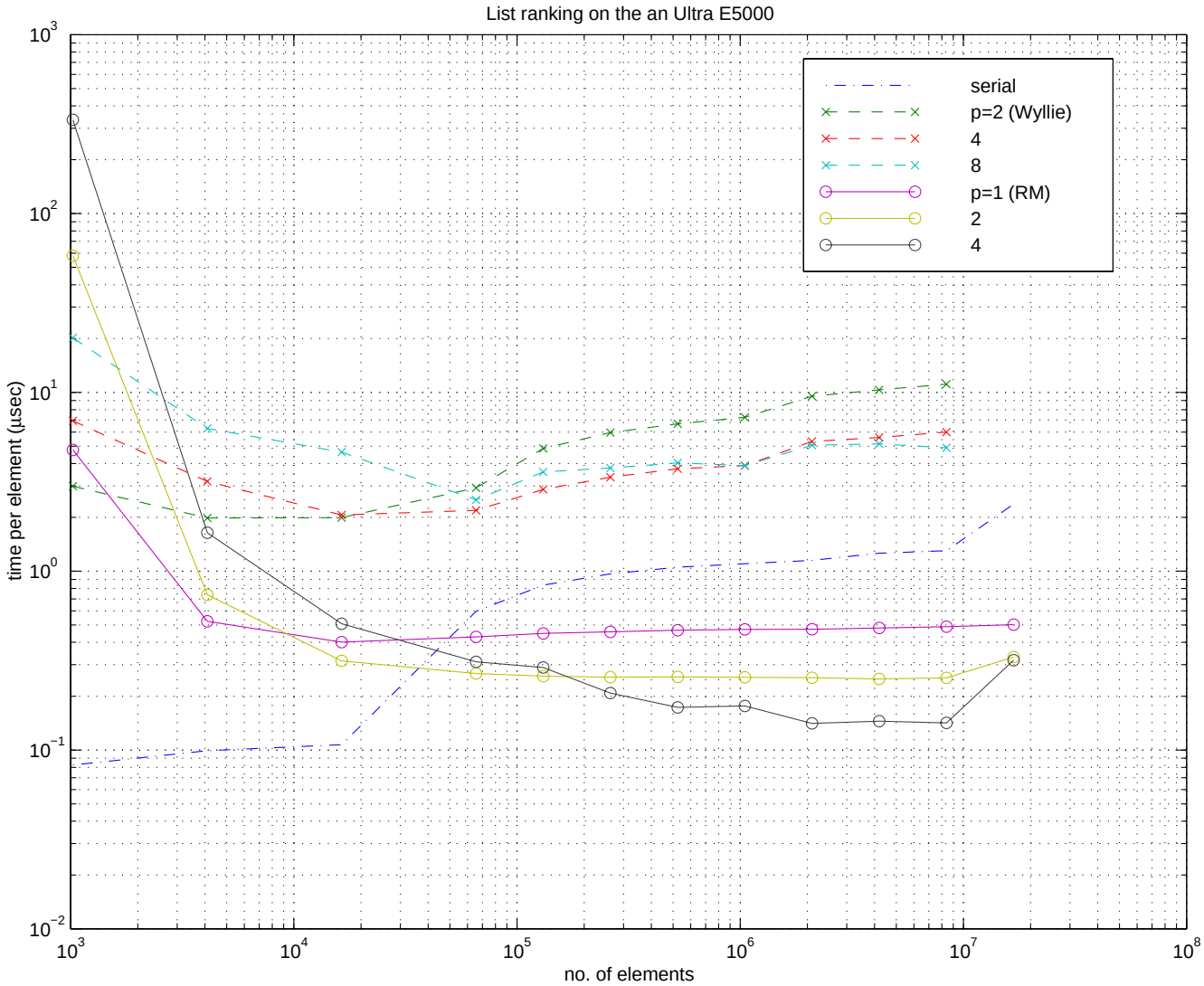
Phase 3: Fill in the scan of the original list using the reduced list results



NOTES to slide 25

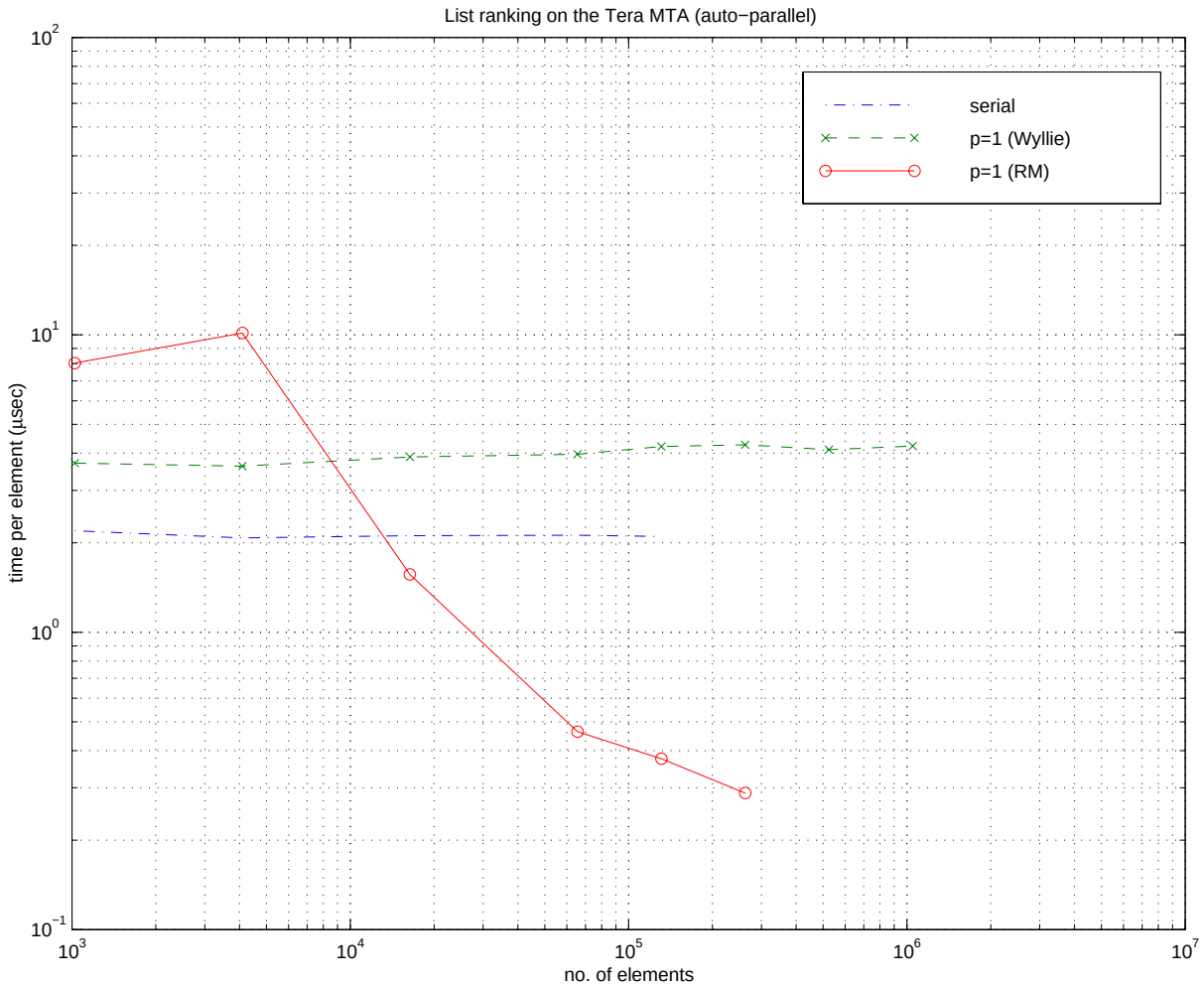
1. Phases 1 and 3 are parallelizable
2. Phase 2 is serial, or a call to Wyllie, or a recursive call
3. Periodic load balancing (packing) during phases 1 and 3
4. Many tuning parameters — size of sublists? when to load balance?
5. Strictly speaking, not work optimal, but has small constants.
6. Expected running time: $O(n/p + (n/m)\log m)$.
7. New algorithm by Helman-JaJa (UMD, 1998) based on the Reid-Miller sparse ruling set idea, tuned for SMPs

On a conventional SMP: Ultra E5000



List scan on the Tera MTA

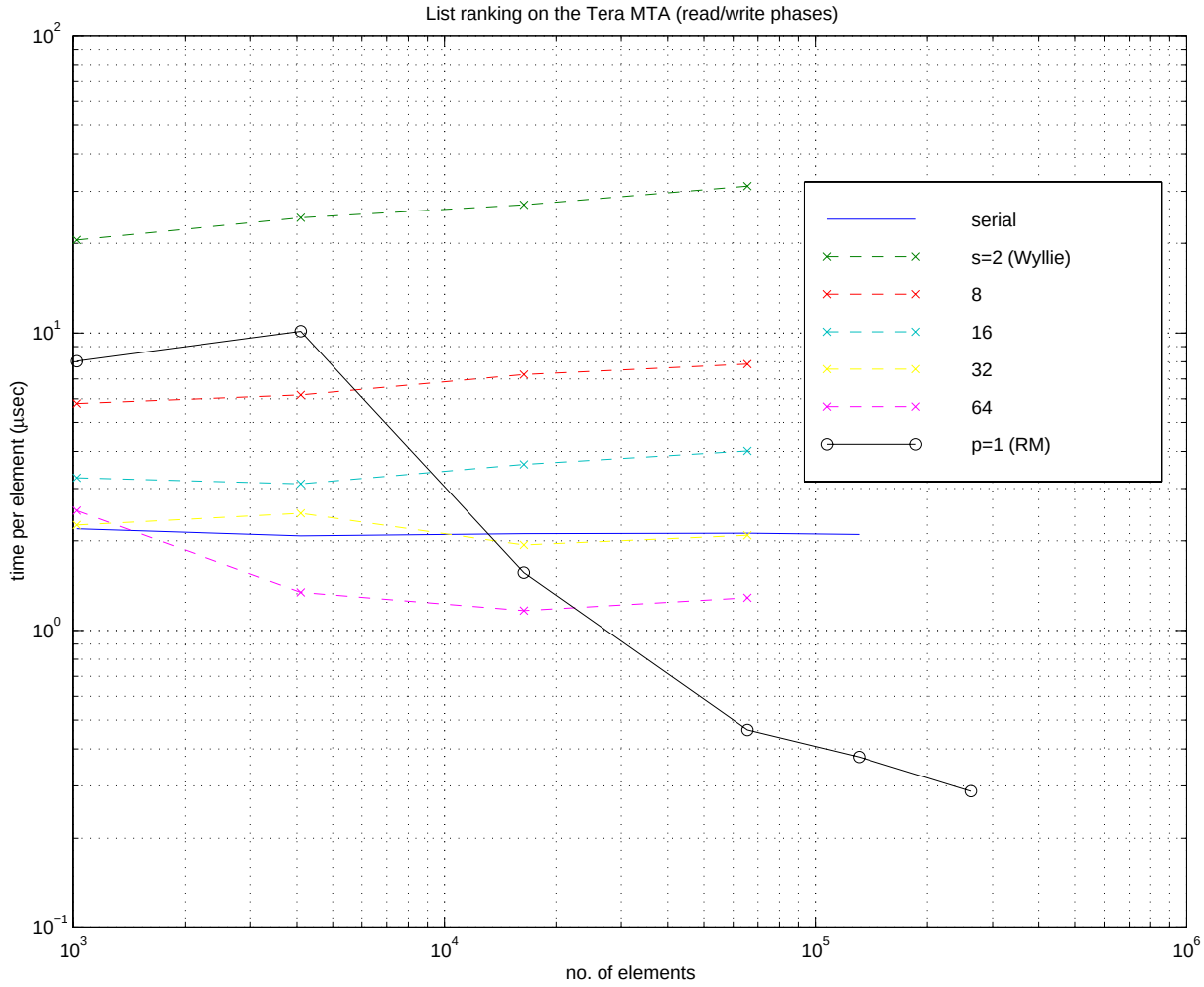
(auto-parallel)



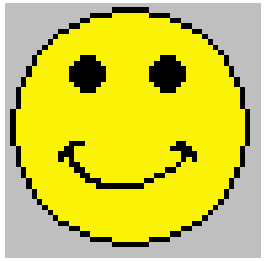
NOTES to slide 27

1. Compiler requested 40 streams for parallel region in Wyllie
2. Compiler requested 120 streams for parallel regions in Reid-Miller

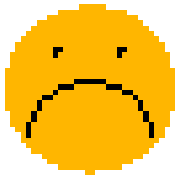
Alternating Wyllie's Algorithm



So, Tera-able or Terrible?



- Given enough streams, a processor can use peak (?) memory bandwidth.
- Stream creation is fast, and the runtime seems to provide flexible performance.
- The fine-grained synchronization is fast, yielding more fine-grained parallelism to invest.



- All the problems of an experimental machine...
- The compiler is tempermental about parallelizing code.
- No magic. Such algorithms as sparse matrix-vector multiplication still need programmer effort, of course.