Problem Based Benchmarks: and their role in parallel algorithms

Guy Blelloch

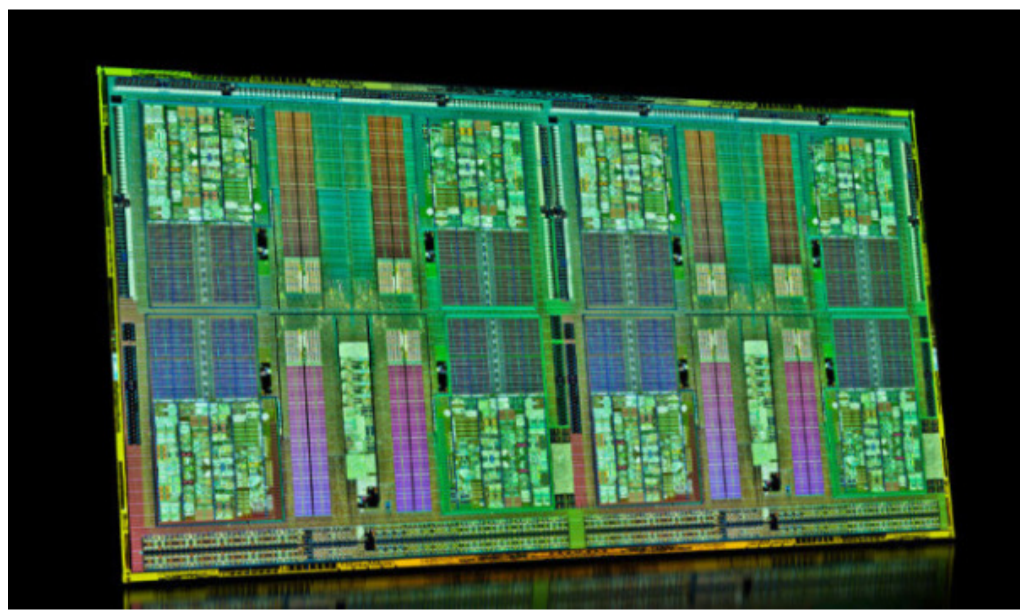
Carnegie Mellon University

Also: Jeremy Fineman, Phil Gibbons (Intel), Julian Shun, Harsha Vardham Simhadri, …

Outline

- The challenge with parallel algorithms
- The problem based benchmark suite
- How they do on modern multiprocessors

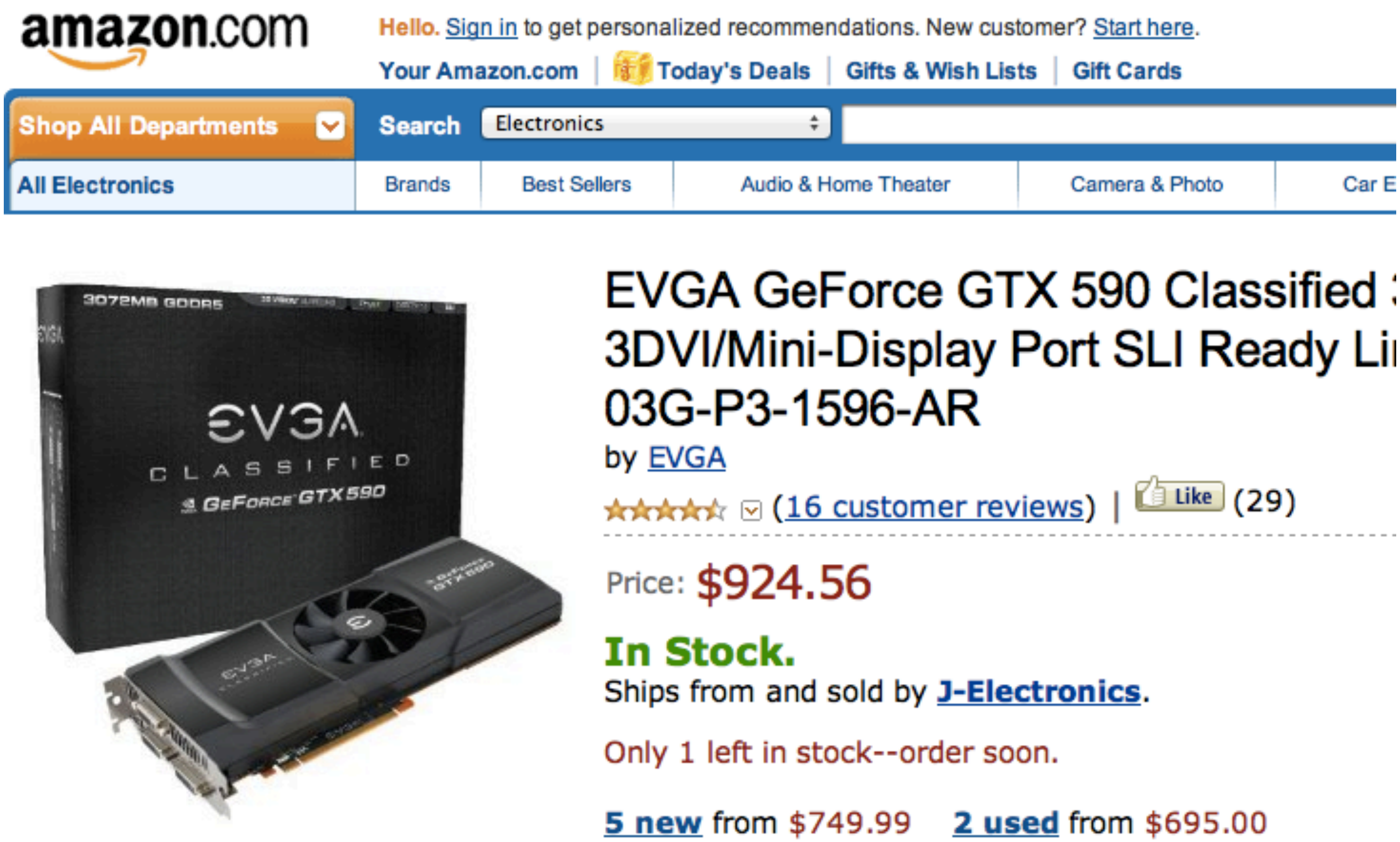
16 core processor

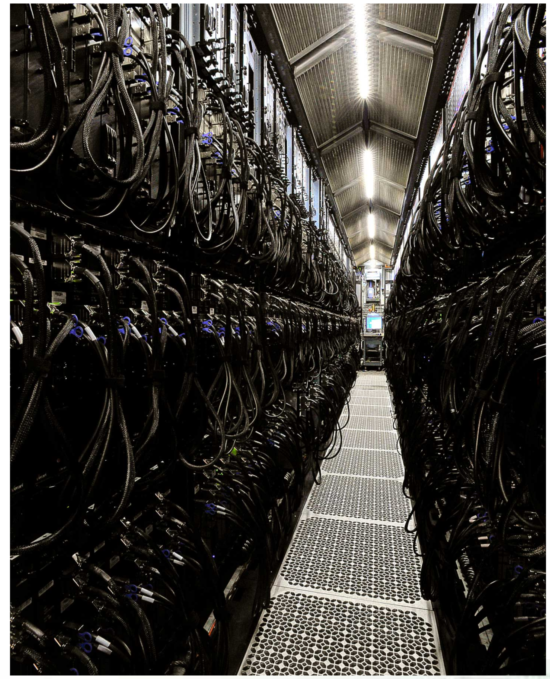


64 core blade servers ($6K) (shared memory)

x 4 =

1024 “cuda” cores





Up to 300K servers

PCWorld » Phones

Recommend: Like 113 171 +1 16 1K Email 44 Comments Print

Quad-Core Phones: What to Expect in 2012

Revolutionary a year ago, dual-core mobile processors are now standard; next, chipmakers say, quad-core processors will support mobile multitasking comparable to the performance of a desktop computer.

By Ginny Mies, PCWorld Dec 11, 2011 8:30 pm

Different Architectures

- Multicore (shared memory)
- GPUs
- Distributed memory
- FPGAs

Different Programming Approaches

- transactions
- futures
- nested parallelism
- map-reduce
- CUDA/GPU programming
- data parallelism
- PRAM
- bulk synchronization

Different Programming Approaches

- threads
- message passing
- parallel I/O models
- partitioned global address space
- coordination languages
- concurrent data structures
- events
- …

But….

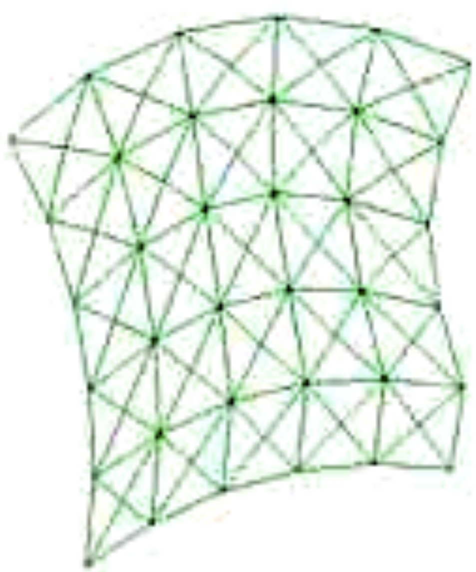
- How well do these work on standard problems?
- How do they compare?
- What kind of algorithms work best?
- How easy are they to program?

Outline

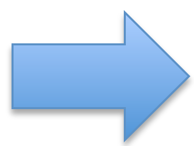
- The challenge with parallel algorithms
- The problem based benchmark suite
- How they do on modern multiprocessors

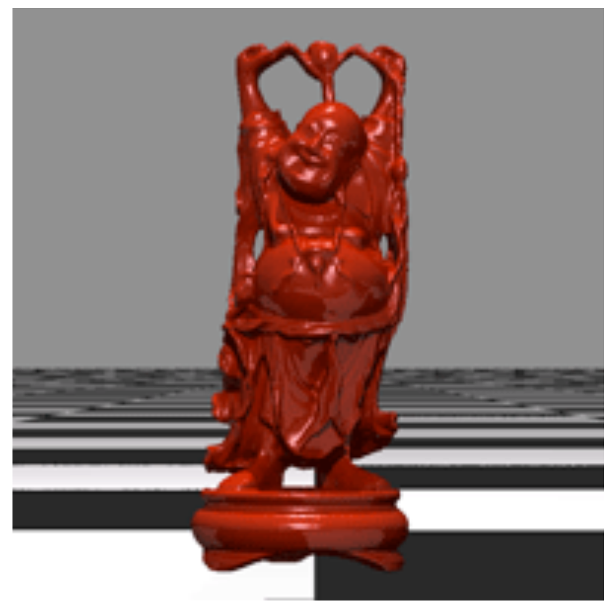
Problem Based Benchmarks

- Define a set of benchmarks in terms of **Input/Output** behavior on specific inputs, and use them to compare solutions.



Input





Output

Problem Based Benchmarks

- Judge based on:
  - Performance and scalability
  - Ability to reason about performance
  - Quality of code
  - Generality over inputs
  - Platform independence

Some aspects can be judged qualitatively, others aspects will be at the eye of the beholder.

Therefore making code public is very important.

The PBBS effort

Benchmarks with following characteristics

- Well known and understood
- Concisely described
- Implementable in under 1000 lines of code
- Broad representation of domains
- Correctness or quality of output easily measured
- Independent of machine type

Many Existing Benchmarks

But none we know of match the spec

- **Code Based** : SPEC, Da Capo, PassMark, Splash-2, PARSEC, fluidMark
- **Application Specific**: Linpack, BioBench, BioParallel, MediaBench, SATLIB, CineBench, MineBench, TCP, ALPBench, Graph 500, DIMACS challenges
- **Method Based**: Lonestar
- **Machine analysis**: HPC challenge, Java Grande, NAS, Green 500, Graph 500, P-Ray, fluidMark

Status

- About 15 benchmarks defined with supporting code
- Sequential implementations
- Multicore implementations
- Will make public in February

Preliminary Benchmarks I

Sequences	* Comparison Sorting
	* Removing Duplicates
	* Dictionary
Graphs	* Breadth First Search
	Graph Separators
	* Minimum Spanning Tree
	* Maximal Independent Set
Geometry/ Graphics	* Delaunay Triangulation and Refinement
	* Convex Hulls
	* Ray Triangle Intersection (Ray Casting)
	Micropolygon Rendering

Preliminary Benchmarks II

Machine Learning	* All Nearest Neighbors
	Support Vector Machines
	K-Means
Text Processing	* Suffix Arrays
	Edit Distance
	String Search
Science	* Nbody force calculations
	Phylogenetic tree
Numerical	* Sparse Matrix Vector Multiply
	Sparse Linear Solve

Each Benchmark Consists of:

- A precise specification of the problem
- Specification of Input/Output file formats
- A set of input generators.
- A weighting on the inputs
- Code for testing the results
- Baseline sequential code
- Baseline parallel code(s)

Example Input

Sorting:

- Random floats (uniform)
- Random floats (exponential bias)
- Almost sorted
- Strings generated from trigram probability and randomly permuted
- Structures with float key and 3 additional fields

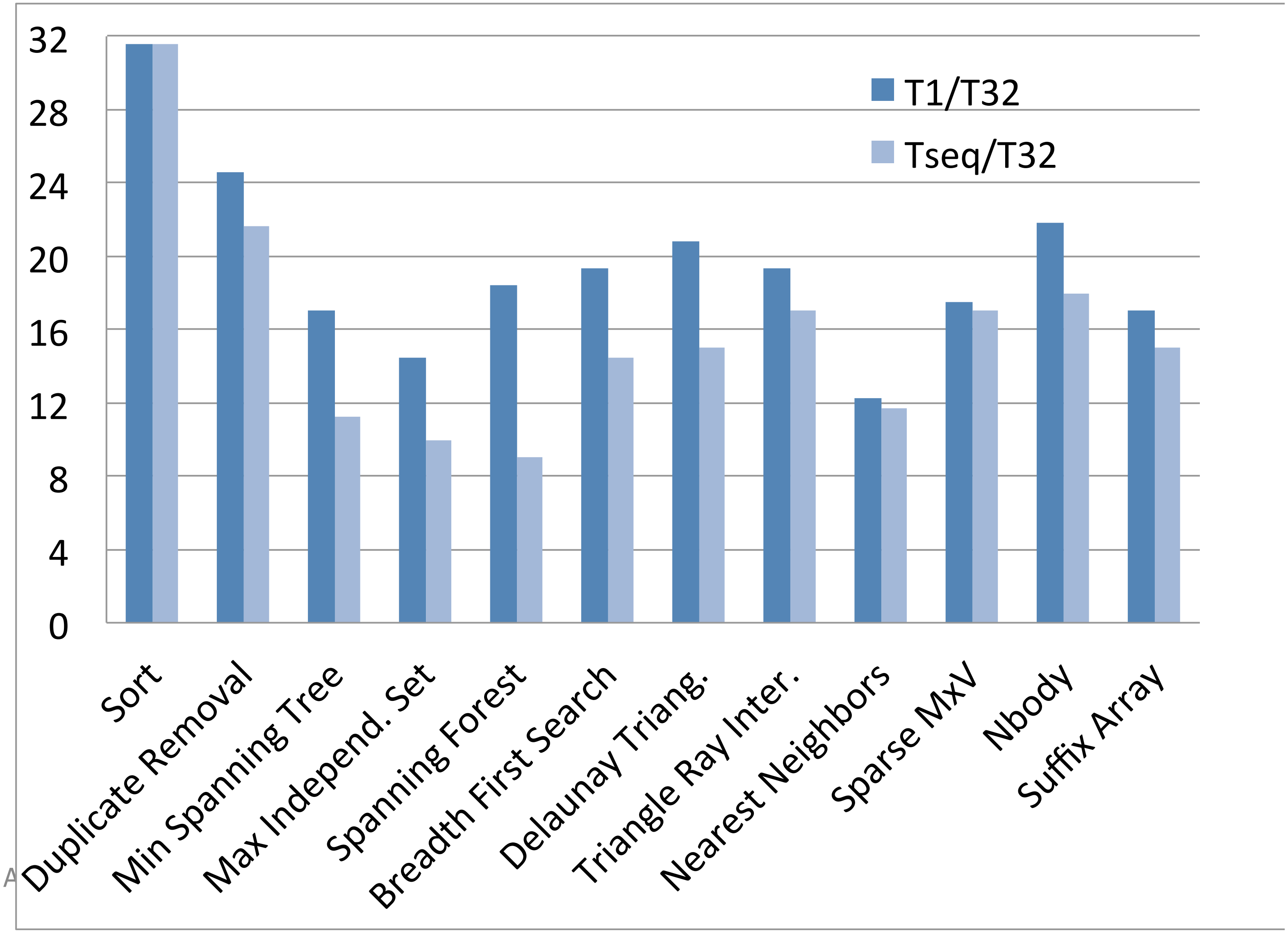
Outline

- The challenge with parallel algorithms
- The problem based benchmark suite
- How they do on modern multiprocessors
  - Using 32-core Intel Nehalem
  - What parallel algorithms work

Algorithmic Models

- PRAM
- BSP
- Nested Parallelism with Work and Span
  - Compose work by summing
  - Compose span by taking the max
- Parallel Cache Oblivious Model
  - Count Sequential Cache misses
  - Can be used to bound parallel cache misses

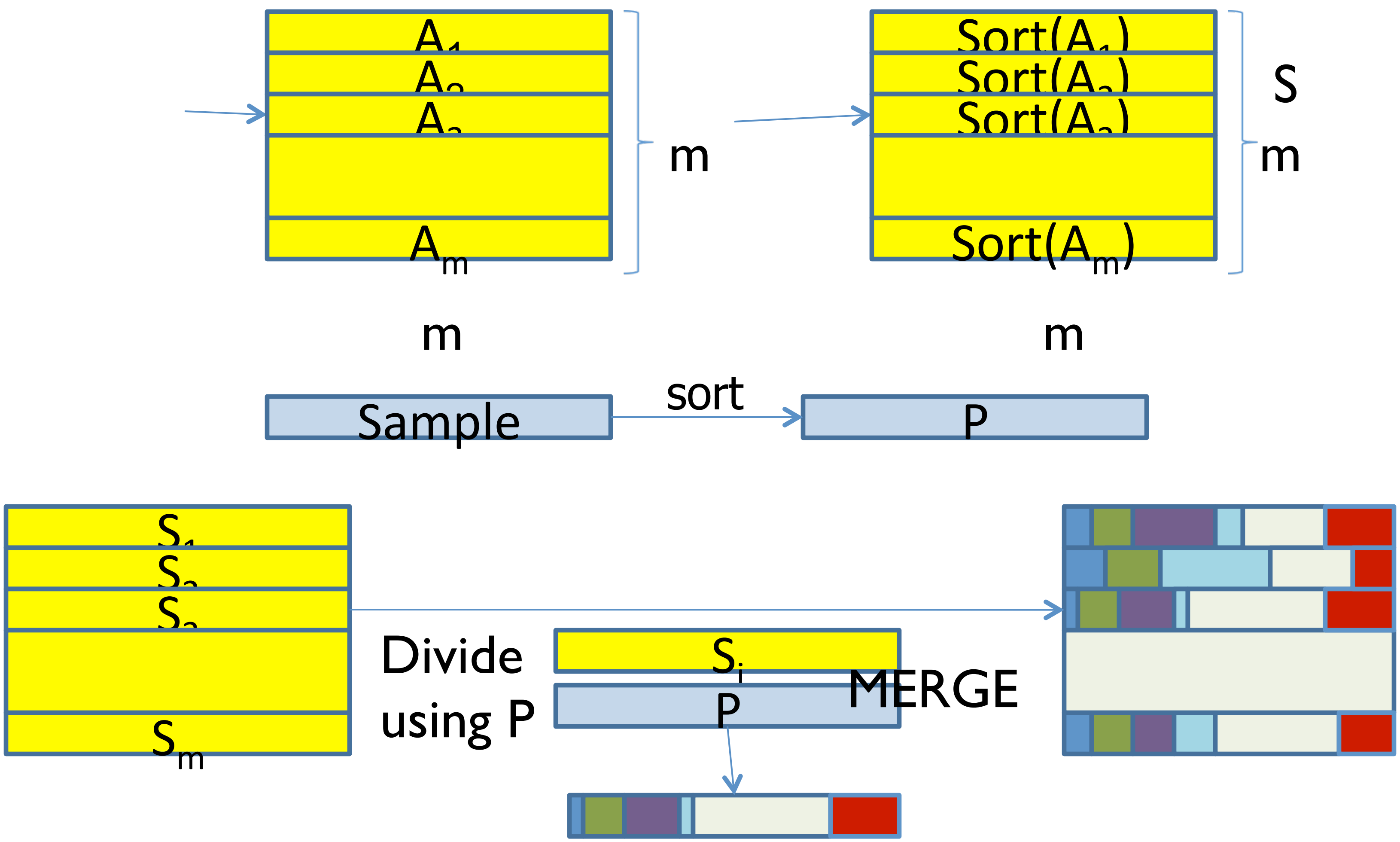
How do the problems do on a modern multicore



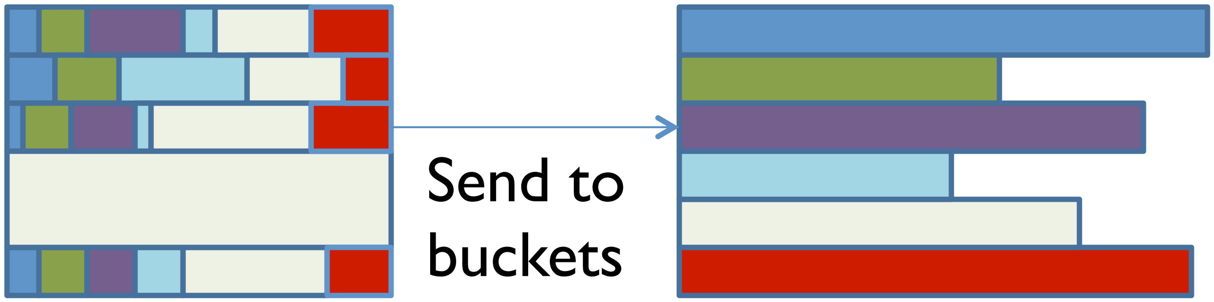
Divide and Conquer

- Sorting : Sample sort
- Nearest neighbors : building quad-oct trees
- Triangle-ray intersect : k-d trees
- N-body simulation: Callahan-Kosaraju

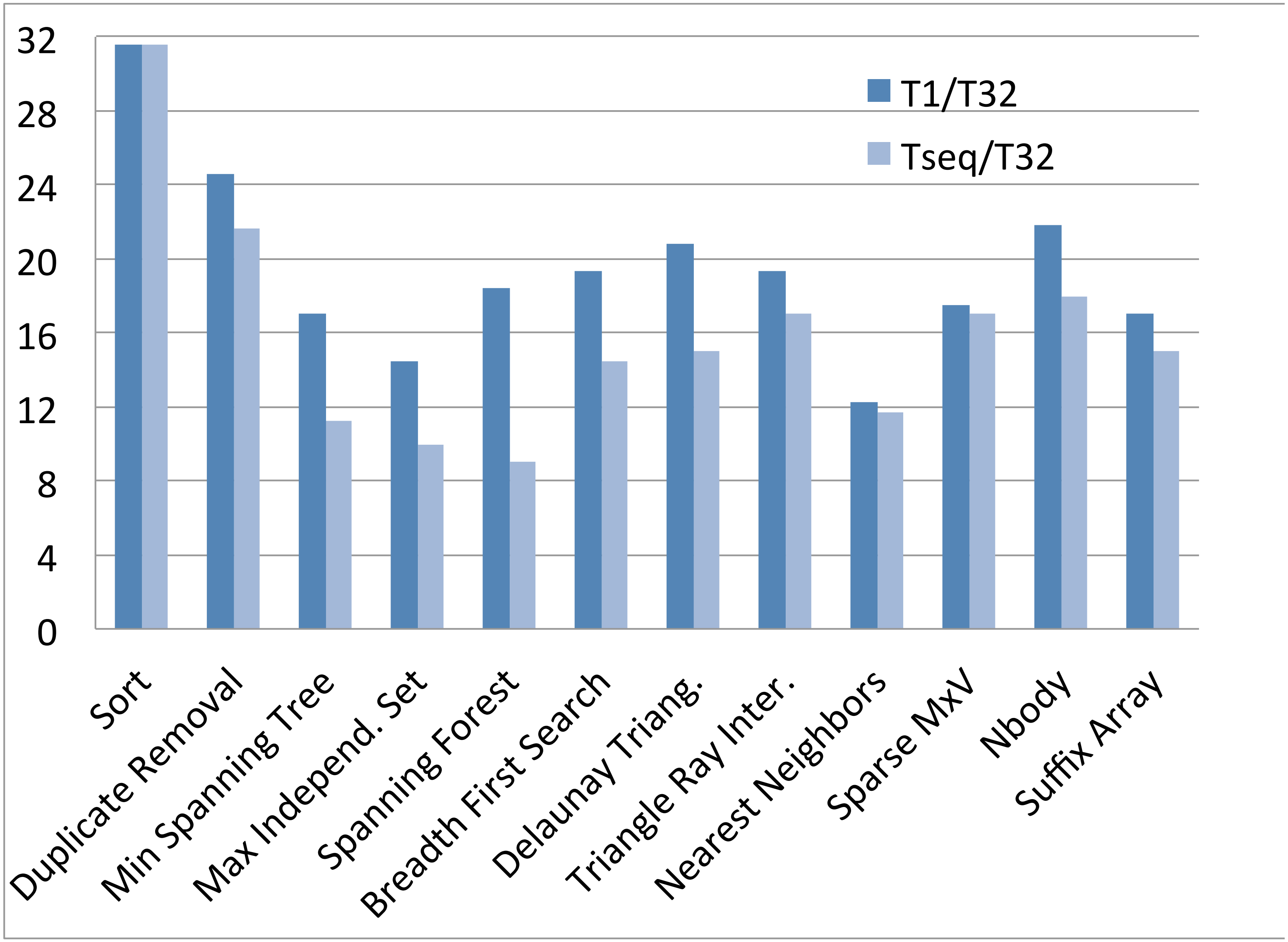
Sorting : Sample Sort



Sorting : Sample Sort



- Finally, sort buckets.
- $Depth(n) = O(\log^2(n))$
- $Work(n) = O(n \log n)$
- $Q_1(n; M,B) = O((n/B)(\log_{(M/B)}(n/B))$


32
28
24
20
16
12
8
4
0
T1/T32
Tseq/T32
Sort
Duplicate Removal
Min Spanning Tree
Max Independ. Set
Spanning Forest
Breadth First Search
Delaunay Triang.
Triangle Ray Inter.
Nearest Neighbors
Sparse MxV
Nbody
Suffix Array

Sort Performance, More Detail

	weight	STL Sort	Sanders Sort	Quicksort	SampleSort	SampleSort
Cores		1	32	32	32	1
Uniform	.1	15.8	1.06	4.22	.82	20.2
Exponential	.1	10.8	.79	2.49	.53	13.8
Almost Sorted	.1	3.28	1.11	1.76	.27	5.67
Trigram Strings	.2	58.2	4.63	8.6	1.05	30.8
Strings Permuted	.2	82.5	7.08	28.4	1.76	49.3
Structure	.3	17.6	2.03	6.73	1.18	26.7
Average		**36.4**	**3.24**	**10.3**	**.97**	**28.0**

All inputs are 100,000,000 long.
All code written run on Cilk++ (also tested in Cilk+)
All experiments on 32 core Nehalem (4 X x7560)

Speculative Execution

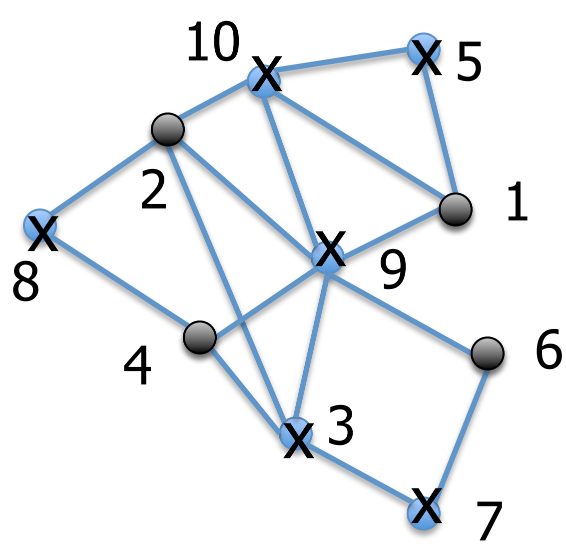
Several efficient sequential algorithms are greedy loops that insert/process items one at a time, but with dependences:

- Maximal independent Set (over vertices)
- Maximal Matching (over edges)
- Spanning Tree (over edges)
- Delaunay Triangulation (over points)

Maximal Independent Set

Sequential algorithm:

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



Maximal Independent Set

Sequential algorithm:

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```

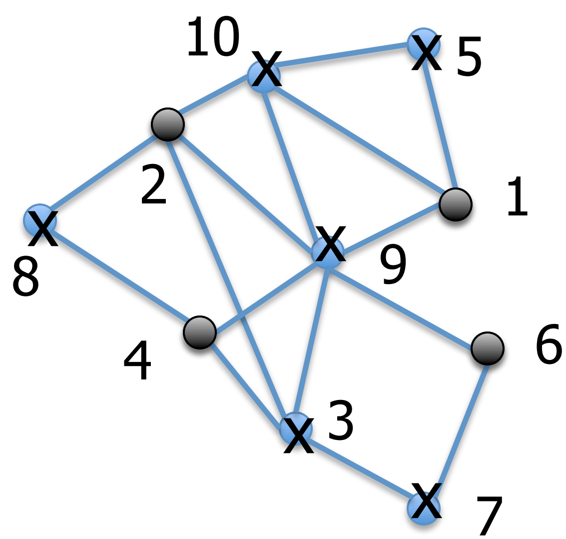
Very efficient: most edges not even visited, simple loops

About 7x faster than sorting m edges

Maximal Independent Set

Same algorithm: with parallel speculation

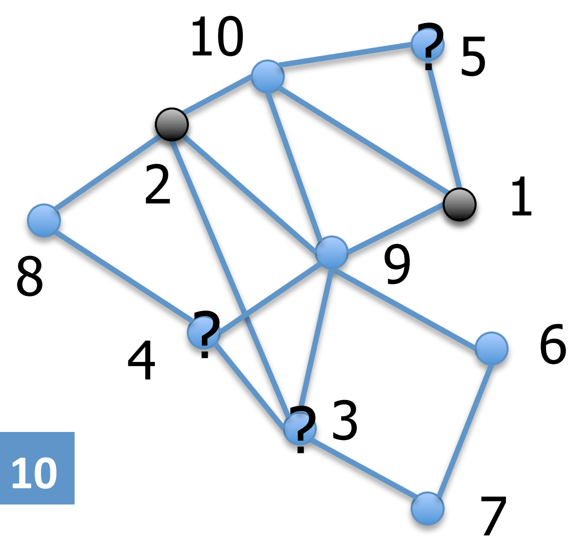
```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



Maximal Independent Set

same algorithm: with speculation on prefix

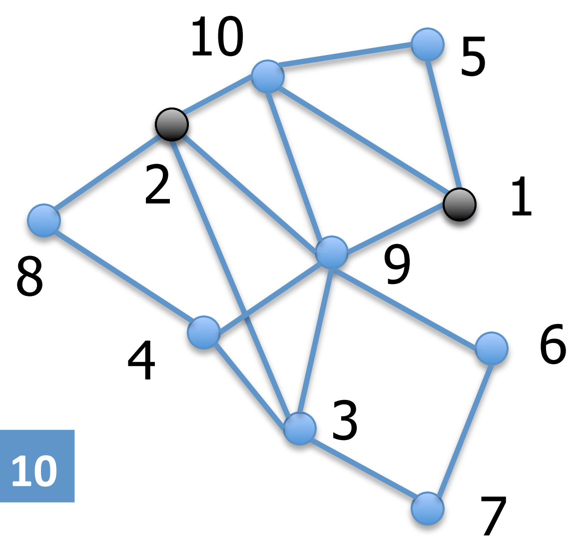
```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



Maximal Independent Set

same algorithm: with speculation on prefix

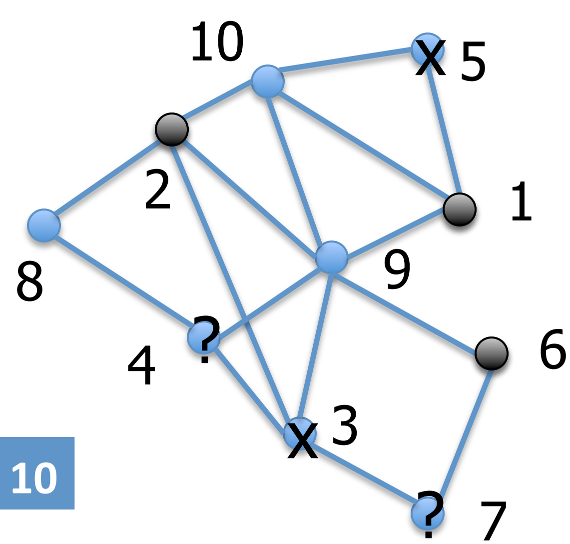
```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



Maximal Independent Set

same algorithm: with speculation on prefix

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```

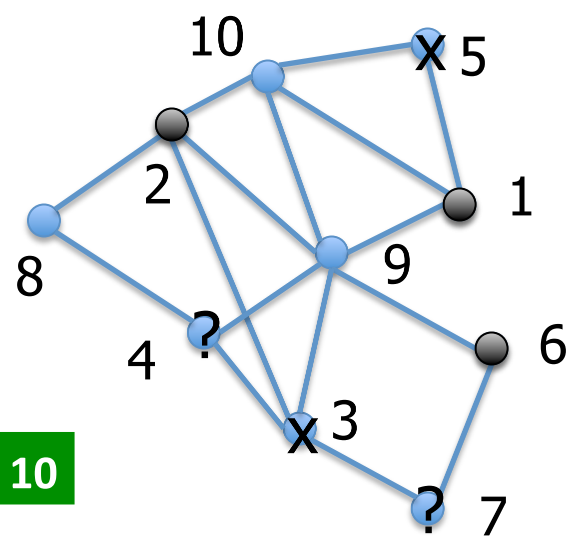


1	2	3	4	5	6	7	8	9	10

Maximal Independent Set

same algorithm: with speculation on prefix

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```

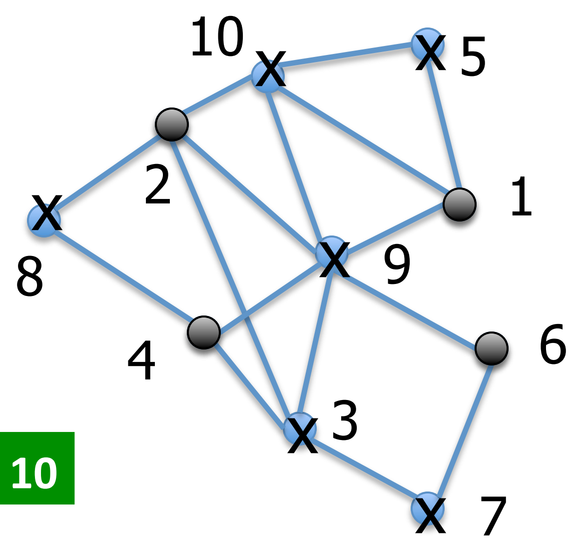


1	2	3	4	5	6	7	8	9	10

Maximal Independent Set

same algorithm: with speculation on prefix

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



1	2	3	4	5	6	7	8	9	10

MIS Parallel Code

```
struct MISStep {
  bool reserve(int i) {
    int d = V[i].degree;
    flag = IN;
    for (int j = 0; j < d; j++) {
      int ngh = V[i].Neighbors[j];
      if (ngh < i) {
        if (Fl[ngh] == IN) { flag = OUT; return 1;}
        else if (Fl[ngh] == LIVE) flag = LIVE; } }
    return 1; }

  bool commit(int i) { return (Fl[i] = flag) != LIVE;}};

void MIS(FlType* Fl, vertex* V, int n, int psize)
  speculative_for(MISStep(Fl, V), 0, n, psize);}
```

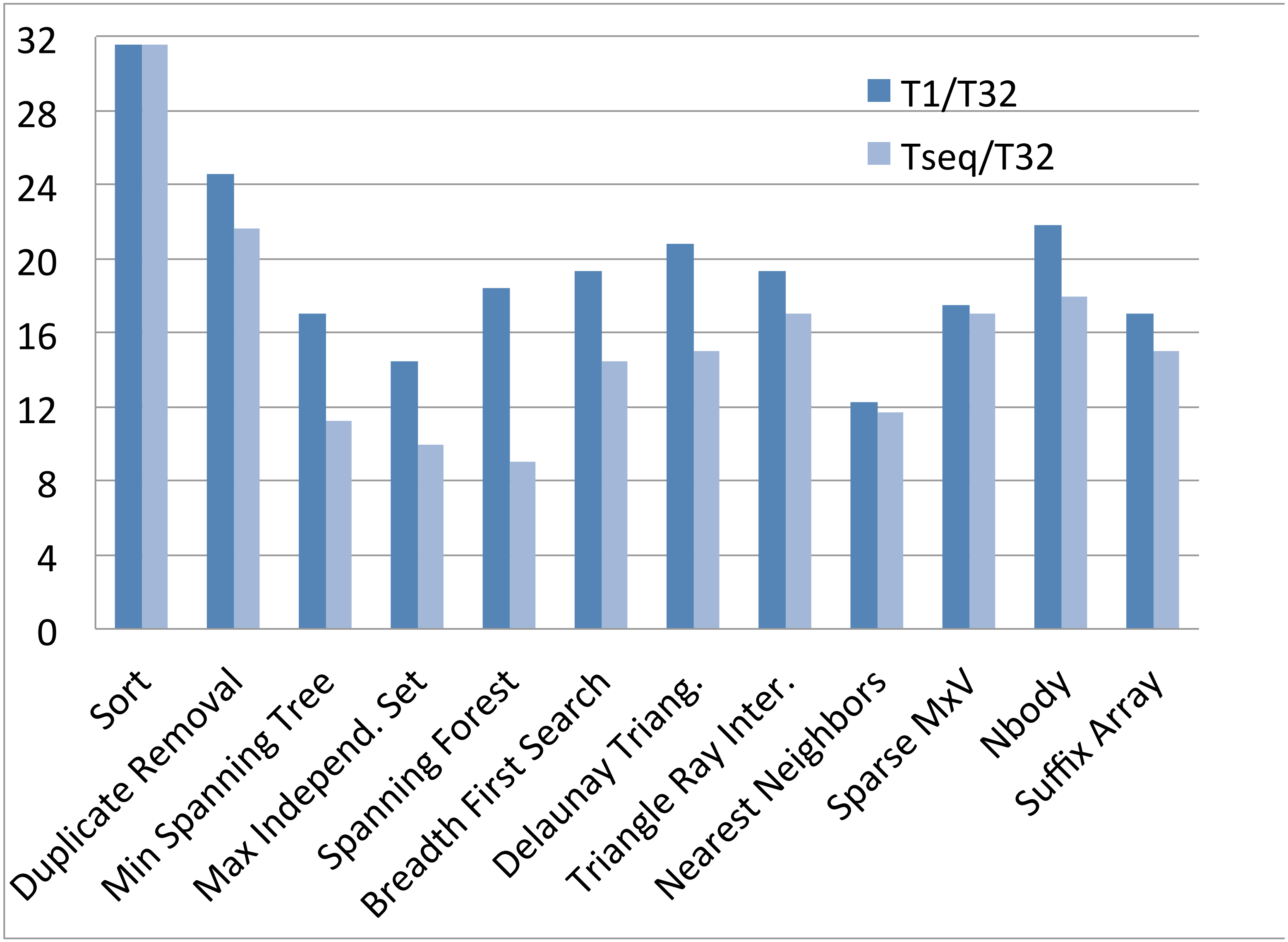
Maximal Independent Set

Costs:

- Span = $O(\log^3 n)$
  Expected case over all initial permutations
- Work = $O(m)$
  if prefix size = $O(n/d_{max})$

Determininistic :

- result only depends on initial permutation of vertices



32
28
24
20
16
12
8
4
0
T1/T32
Tseq/T32
Sort
Duplicate Removal
Min Spanning Tree
Max Independ. Set
Spanning Forest
Breadth First Search
Delaunay Triang.
Triangle Ray Inter.
Nearest Neighbors
Sparse MxV
Nbody
Suffix Array

Spanning Tree

Sequential algorithm:

```
for each (u,v) in E
  u’ = find(u)
  v’ = find(v)
  if (u’ != v’) union(u’,v’)
```

Spanning Tree

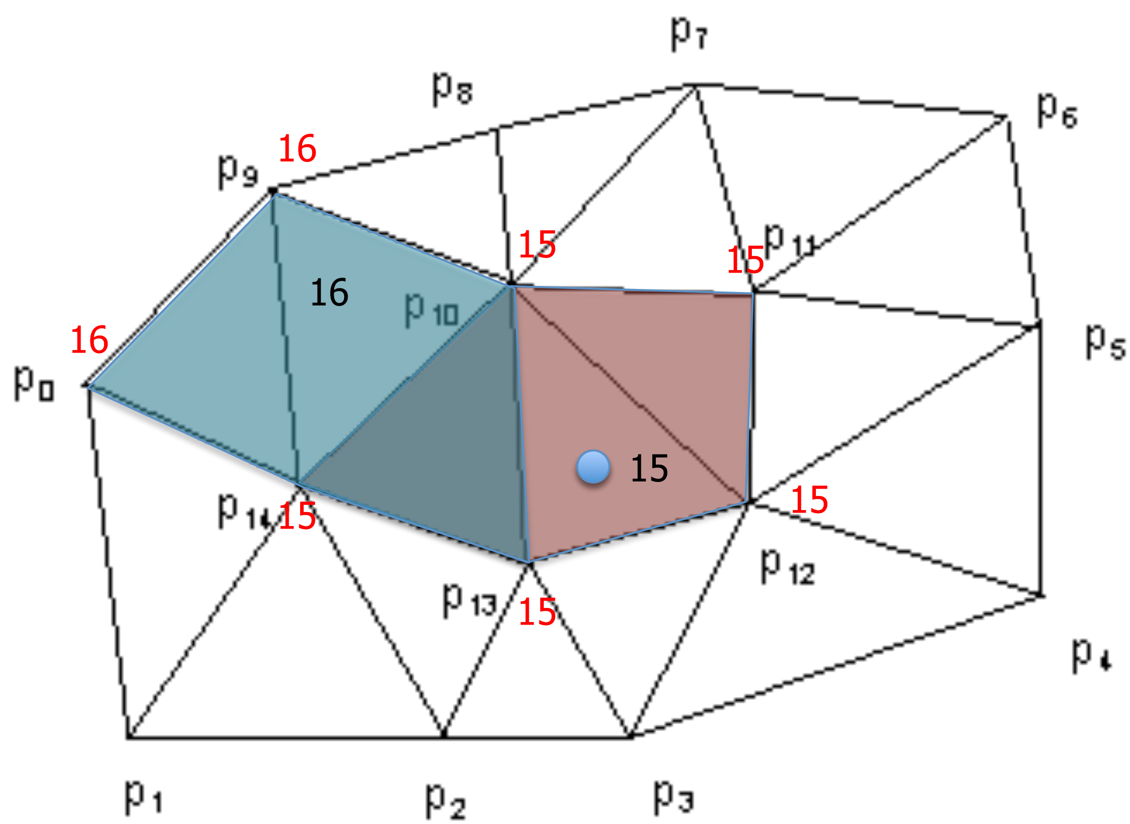
```
struct STStep {
  bool reserve(int i) {
    u = F.find(E[i].u);
    v = F.find(E[i].v);
    if (u == v) return 0;
    if (u > v) swap(u,v);
    R[v].reserve(i);  return 1;}

  bool commit(int i) {
    if (R[v].check(i)) { F.link(v, u); return 1;}
    else return 0;  }};

void ST(res* R, edge* E, int m, int n, int psize) {
  disjointSet F(n);
  speculative_for(STStep(E, F, R), 0, m, psize);}
```

Delaunay Triangulation/Refinement

- Add points in parallel but detect conflicts



Dictionary

Using hashing:

– Based on generic hash and comparison

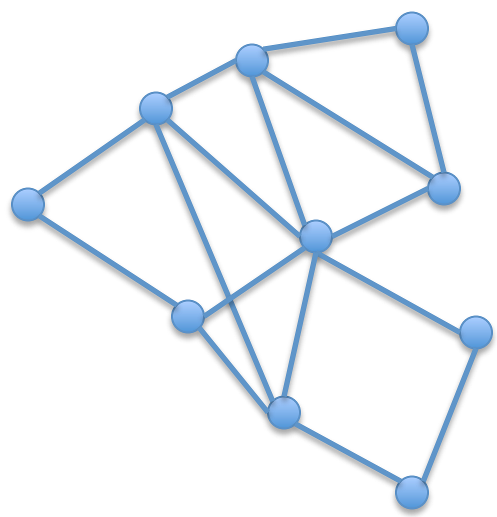
– Problem: representation can depend on ordering. Also on which redundant element is kept.

– Solution: Use history independent hash table based on linear probing…representation is independent of order of insertion

– Use write-min on collision

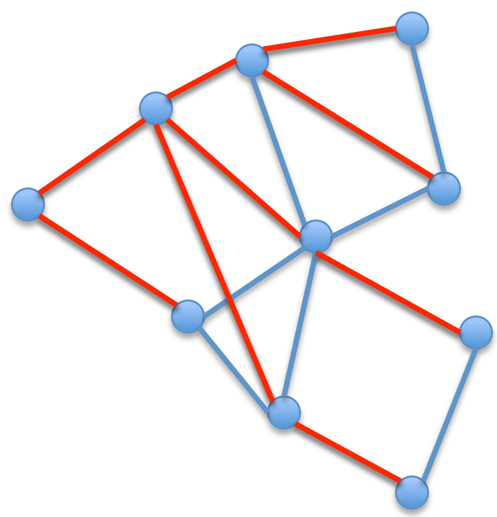
Breadth First Search (BFS)

Goal: generate the same BFS (spanning) tree as the sequential Q based algorithm.



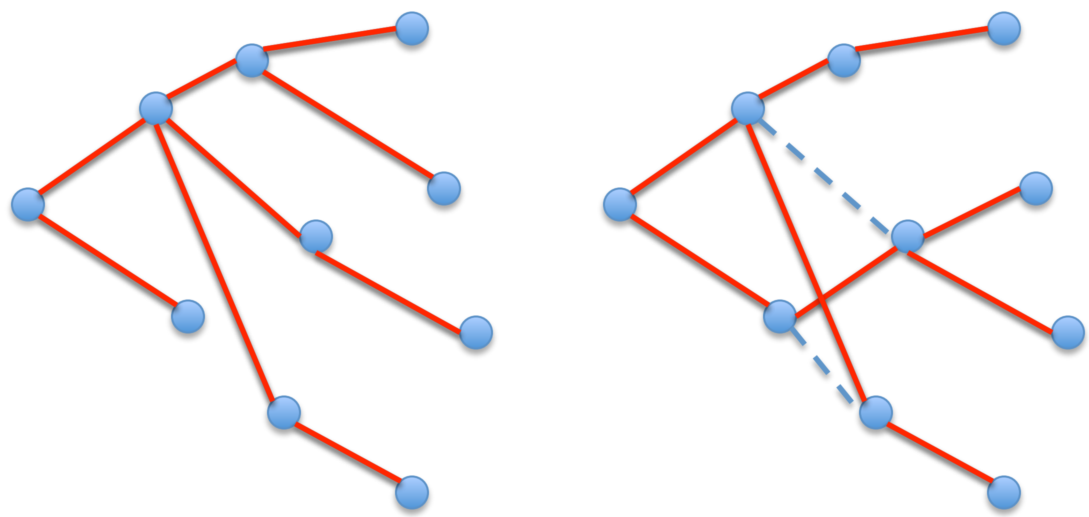
Breadth First Search (BFS)

Sequential algorithm:



Breadth First Search (BFS)

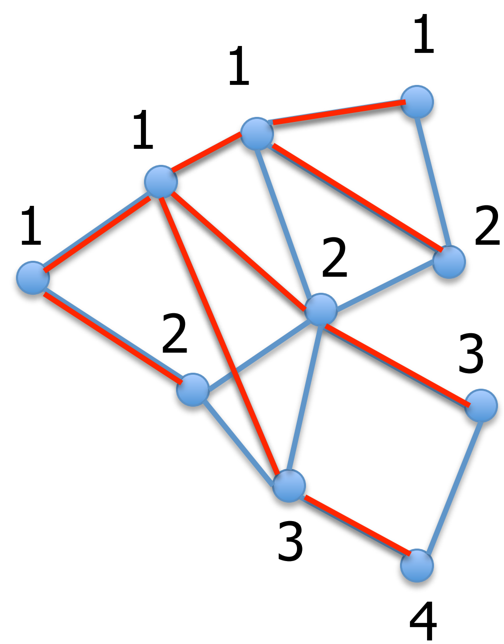
Another possible tree:



Breadth First Search (BFS)

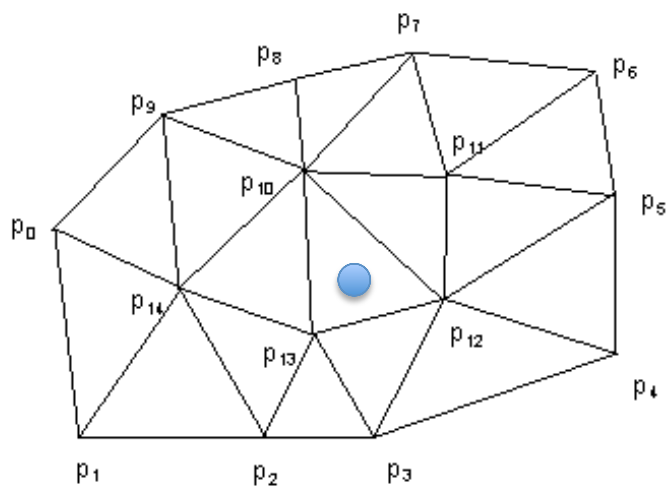
Solution:

- Maintain Frontier and priority order it
- Use writeMin to choose winner.



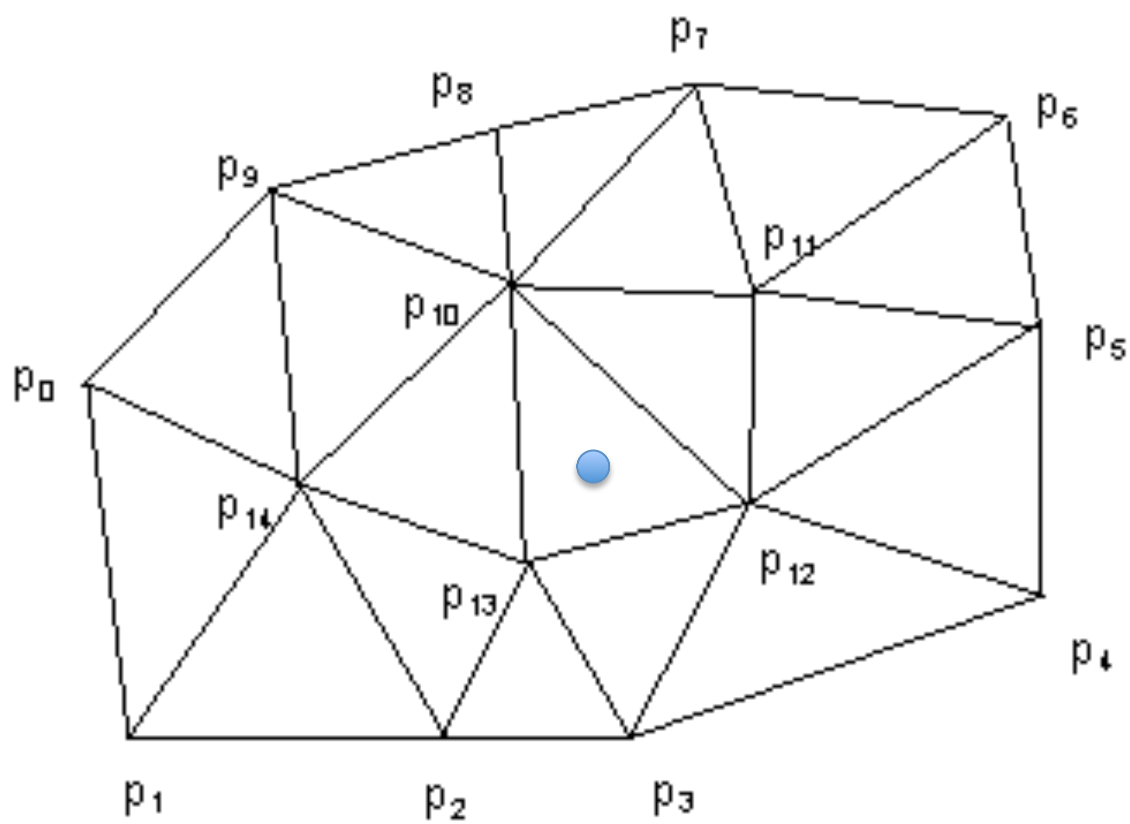
Delaunay Triangulation/Refinement

- Incremental algorithm adds one point at a time, but points can be added in parallel if they don't interact.
- The problem is that the output will depend on the order they are added.



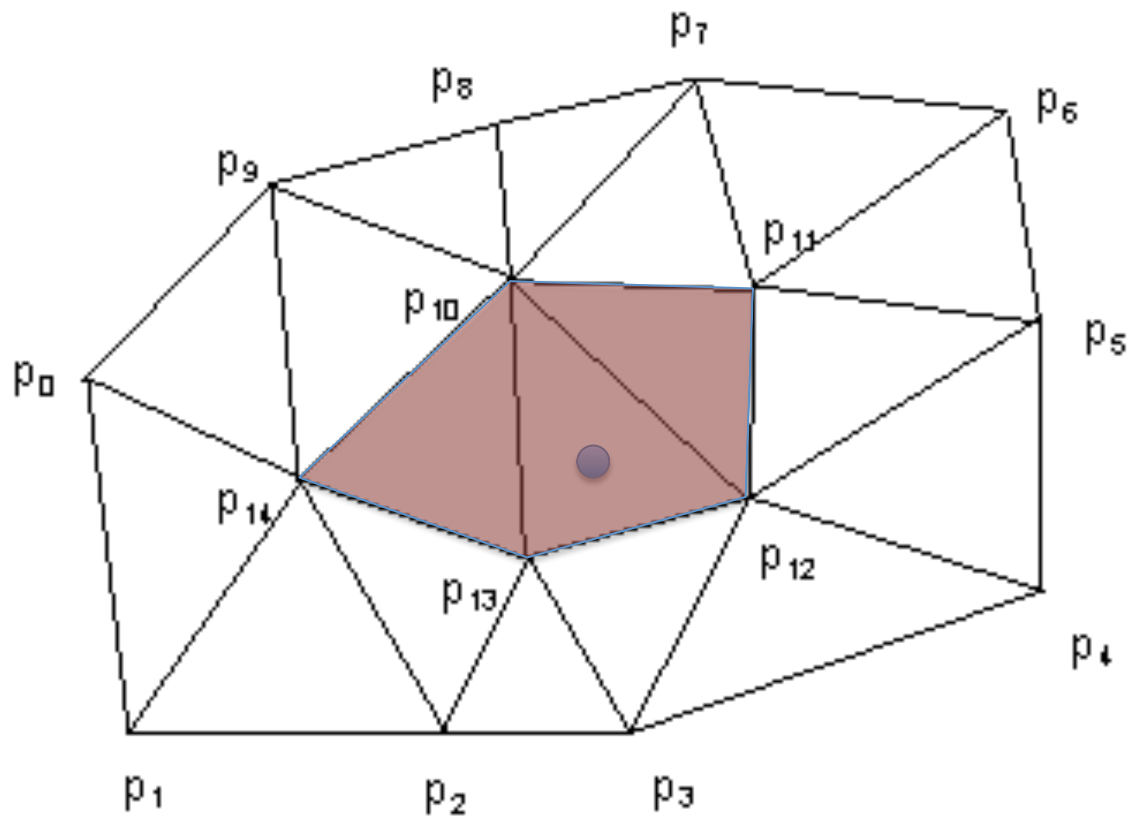
Delaunay Triangulation/Refinement

- Adding points deterministically



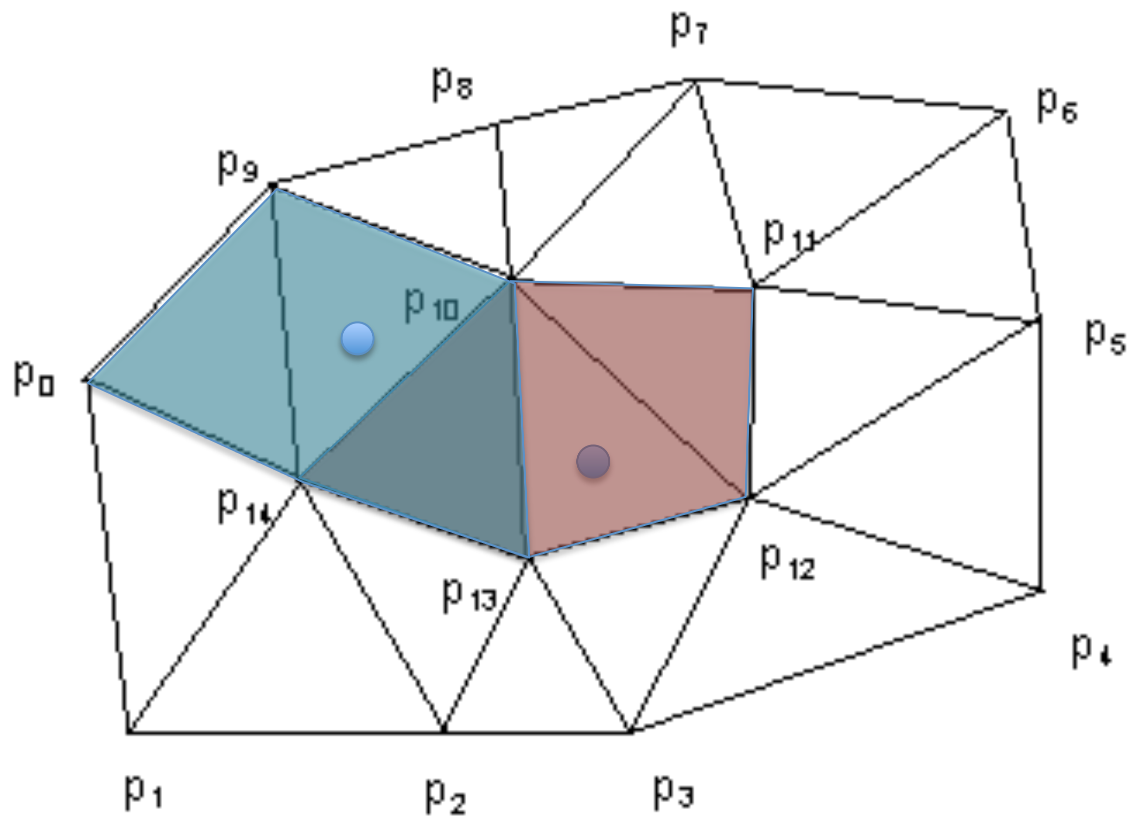
Delaunay Triangulation/Refinement

- Adding points deterministically

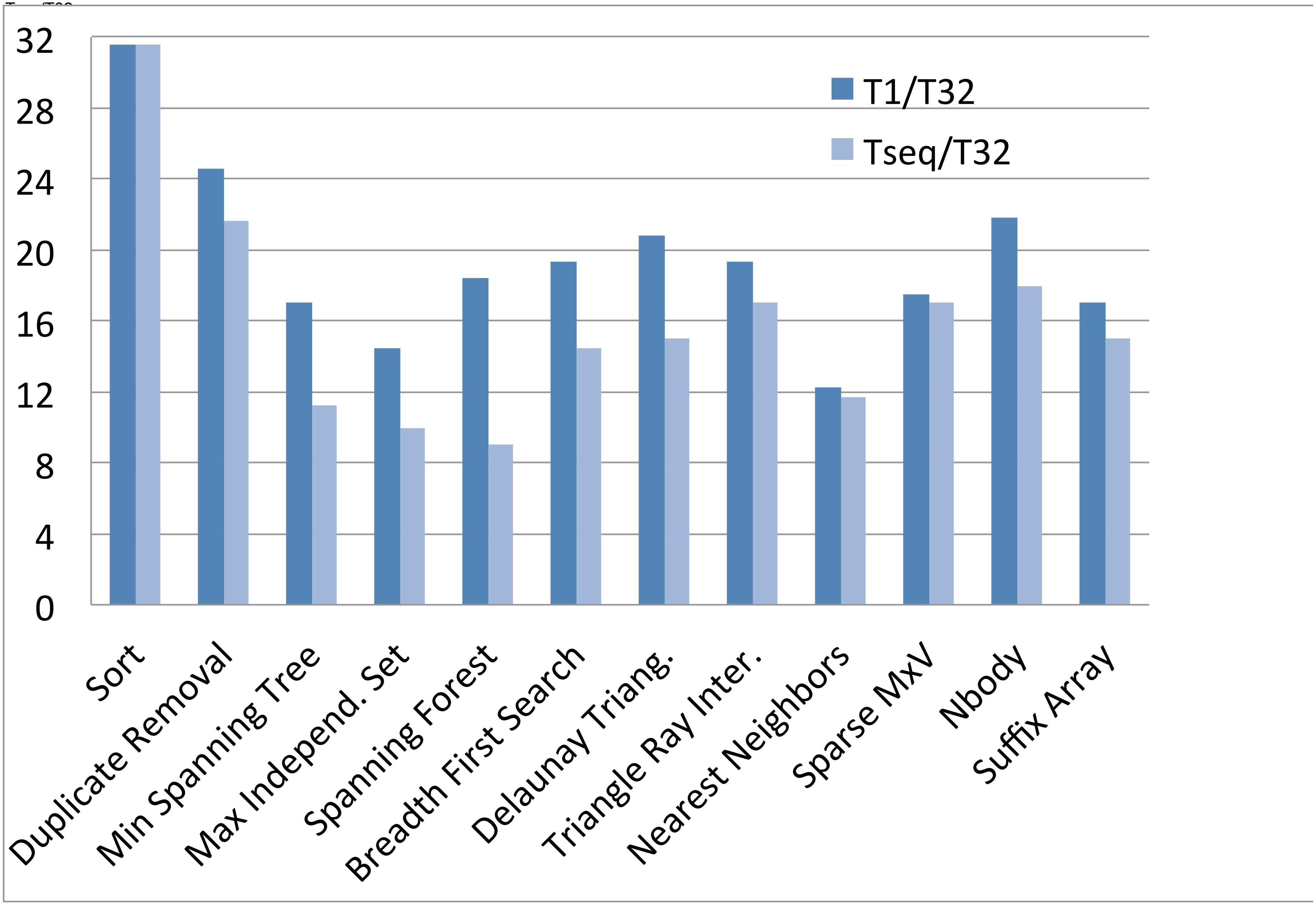


Delaunay Triangulation/Refinement

- Adding points deterministically



Performance on 32 Core Intel Nehalem



Some Conclusions from Experiments

- Multicores work quite well…but there are some issues with memory bandwidth
- Most problems parallelize well.
- Cost models are reasonably accurate
- Parallel code does not need to be complicated
- Need a mix of parallelization techniques

Open Questions

- How do the benchmarks do on other machines….other models?
- Are there better sequential implementations
- Are there better parallel implementations
- More benchmarks – perhaps ones that don't parallelize well (e.g. max flow?).

Back to the benchmarks

- Need for standardized "problem based" benchmarks for comparing approaches.
- Particularly important for parallel algorithms, but also useful for sequential algorithms.
- With adequate framework, should be possible for anyone to submit new benchmarks and solutions.