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

Amd Opteron (sixteen-core) Model 6274
by AMD

Be the first to review this item | 0 like

Price: $792.99

In Stock.
Ships from and sold by 3-Electronics.

Only 1 left in stock--order soon.

4 new from $714.03
64 core blade servers ($6K) (shared memory)

\[ x \times 4 = 64 \]
1024 “cuda” cores

EVGA GeForce GTX 590 Classified 3DVI/Mini-Display Port SLI Ready Li 03G-P3-1596-AR
by EVGA

⭐⭐⭐⭐⭐ (16 customer reviews) | Like (29)

Price: $924.56

In Stock.
Ships from and sold by J-Electronics.

Only 1 left in stock--order soon.

5 new from $749.99  2 used from $695.00
Up to 300K servers

Alenex 2012
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
- ...

Alenex 2012
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.
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, ALP Bench, 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequences</td>
<td>* Comparison Sorting</td>
</tr>
<tr>
<td></td>
<td>* Removing Duplicates</td>
</tr>
<tr>
<td></td>
<td>* Dictionary</td>
</tr>
<tr>
<td>Graphs</td>
<td>* Breadth First Search</td>
</tr>
<tr>
<td></td>
<td>Graph Separators</td>
</tr>
<tr>
<td></td>
<td>* Minimum Spanning Tree</td>
</tr>
<tr>
<td></td>
<td>* Maximal Independent Set</td>
</tr>
<tr>
<td>Geometry/</td>
<td>* Delaunay Triangulation and Refinement</td>
</tr>
<tr>
<td>Graphics</td>
<td>* Convex Hulls</td>
</tr>
<tr>
<td></td>
<td>* Ray Triangle Intersection (Ray Casting)</td>
</tr>
<tr>
<td></td>
<td>Micropolygon Rendering</td>
</tr>
</tbody>
</table>
# Preliminary Benchmarks II

<table>
<thead>
<tr>
<th>Category</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Learning</td>
<td>* All Nearest Neighbors</td>
</tr>
<tr>
<td></td>
<td>Support Vector Machines</td>
</tr>
<tr>
<td></td>
<td>K-Means</td>
</tr>
<tr>
<td>Text Processing</td>
<td>* Suffix Arrays</td>
</tr>
<tr>
<td></td>
<td>Edit Distance</td>
</tr>
<tr>
<td></td>
<td>String Search</td>
</tr>
<tr>
<td>Science</td>
<td>* Nbody force calculations</td>
</tr>
<tr>
<td></td>
<td>Phylogenetic tree</td>
</tr>
<tr>
<td>Numerical</td>
<td>* Sparse Matrix Vector Multiply</td>
</tr>
<tr>
<td></td>
<td>Sparse Linear Solve</td>
</tr>
</tbody>
</table>
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

- Sort
- Duplicate Removal
- Min Spanning Tree
- Max Independent Set
- Spanning Forest
- Breadth First Search
- Delaunay Triangulation
- Triangle Ray Intersect
- Nearest Neighbors
- Sparse Matrix Multiply
- Nbody
- Suffix Array

Bar chart showing performance comparisons between T1/T32 and Tseq/T32 for various algorithms.
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

\[
\begin{align*}
\text{Sample} & \quad \text{sort} & \quad P \\
\text{Divide using } P & \quad \text{S} & \quad \text{MERGE}
\end{align*}
\]
Finally, sort buckets.

- Depth(n) = $O(\log^2(n))$
- Work(n) = $O(n \log n)$
- $Q_1(n; M,B) = O((n/B)(\log_{(M/B)}(n/B)))$
Sort Performance, More Detail

<table>
<thead>
<tr>
<th></th>
<th>weight</th>
<th>STL Sort</th>
<th>Sanders Sort</th>
<th>Quicksort</th>
<th>SampleSort</th>
<th>SampleSort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform</td>
<td>.1</td>
<td>15.8</td>
<td>1.06</td>
<td>4.22</td>
<td>.82</td>
<td>20.2</td>
</tr>
<tr>
<td>Exponential</td>
<td>.1</td>
<td>10.8</td>
<td>.79</td>
<td>2.49</td>
<td>.53</td>
<td>13.8</td>
</tr>
<tr>
<td>Almost Sorted</td>
<td>.1</td>
<td>3.28</td>
<td>1.11</td>
<td>1.76</td>
<td>.27</td>
<td>5.67</td>
</tr>
<tr>
<td>Trigram Strings</td>
<td>.2</td>
<td>58.2</td>
<td>4.63</td>
<td>8.6</td>
<td>1.05</td>
<td>30.8</td>
</tr>
<tr>
<td>Strings Permut</td>
<td>.2</td>
<td>82.5</td>
<td>7.08</td>
<td>28.4</td>
<td>1.76</td>
<td>49.3</td>
</tr>
<tr>
<td>Structure</td>
<td>.3</td>
<td>17.6</td>
<td>2.03</td>
<td>6.73</td>
<td>1.18</td>
<td>26.7</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>36.4</td>
<td>3.24</td>
<td>10.3</td>
<td>.97</td>
<td></td>
<td>28.0</td>
</tr>
</tbody>
</table>

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 \( V \) : \( S[u] = \text{Remain} \)

for each \( u \) in \( V \)

    if for all \( v \) in \( N(u) \), \( v < u \), \( S[v] = \text{Out} \)
    then \( S[u] = \text{In} \)
    else \( S[u] = \text{Out} \)
Maximal Independent Set

Sequential algorithm:
for each $u$ in $V$ : $S[u] = \text{Remain}$
for each $u$ in $V$
    if for all $v$ in $N(u)$, $v < u$, $S[v] = \text{Out}$
    then $S[u] = \text{In}$
    else $S[u] = \text{Out}$

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 \( V \) : \( S[u] = \text{Remain} \)
for each \( u \) in \( V \)
    if for all \( v \) in \( N(u) \), \( v < u \), \( S[v] = \text{Out} \)
    then \( S[u] = \text{In} \)
    else \( S[u] = \text{Out} \)
Maximal Independent Set

same algorithm: with speculation on prefix

for each $u$ in $V$ : $S[u] = \text{Remain}$
for each $u$ in $V$
  if for all $v$ in $N(u)$, $v < u$, $S[v] = \text{Out}$
    then $S[u] = \text{In}$
  else $S[u] = \text{Out}$
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

different algorithm: with speculation on prefix

for each \( u \) in \( V \) : \( S[u] = \text{Remain} \)

for each \( u \) in \( V \)

if for all \( v \) in \( N(u) \), \( v < u \), \( S[v] = \text{Out} \)

then \( S[u] = \text{In} \)

else \( S[u] = \text{Out} \)
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
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 = $O(n/d_{\text{max}})$

Deterministic:

– result only depends on initial permutation of vertices
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

- **Sort**
- **Duplicate Removal**
- **Min Spanning Tree**
- **Max Independent Set**
- **Spanning Forest**
- **Breadth First Search**
- **Delaunay Triang.**
- **Triangle Ray Inter.**
- **Nearest Neighbors**
- **Sparse MxV**
- **Nbody**
- **Suffix Array**

The graph compares the performance of different workloads using T1/T32 and Tseq/T32 on a 32 core Intel Nehalem setup.
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.