Modeling Parallel Sorts on the CM-5 with LogP

Andrea C. Dusseau

http://http.cs.berkeley.edu/~dusseau/Papers/papers.html

Computer Science 267

University of California, Berkeley

April 1995

Goals

- Learn about three parallel sorting algorithms
  - Bitonic, Radix, and Sample Sort (Column in paper)
  - Emphasis on real implementation
- Know the best algorithm for sorting on the CM-5
- See uses of the LogP model
  - Guides algorithmic development

    Choose data layouts which minimize communication
  - Predicts execution time

    Extrapolate to larger machine sizes
  - Shows implementation defects

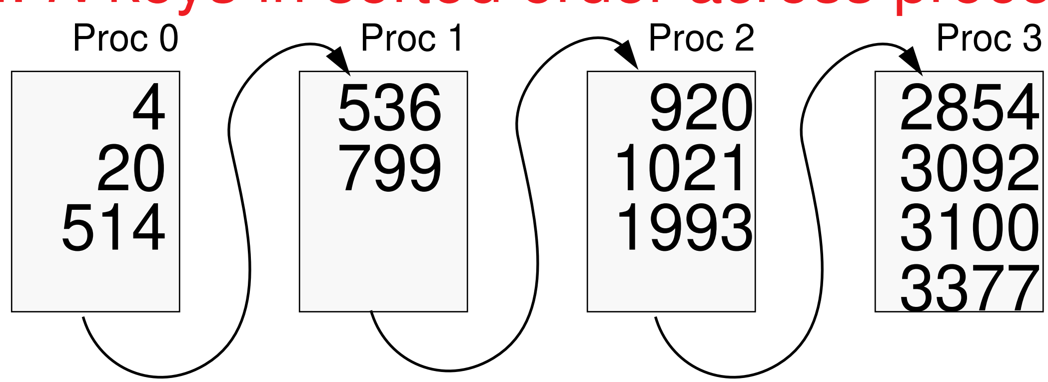
    When model does not match measured performance

Overview

- Problem Statement
- Local Sort (Radix Sort)
- LogP Review
- Bitonic Sort
  - Transpose
- Radix Sort
  - Pipelined Multi-Scan
- Sample Sort
- Comparison

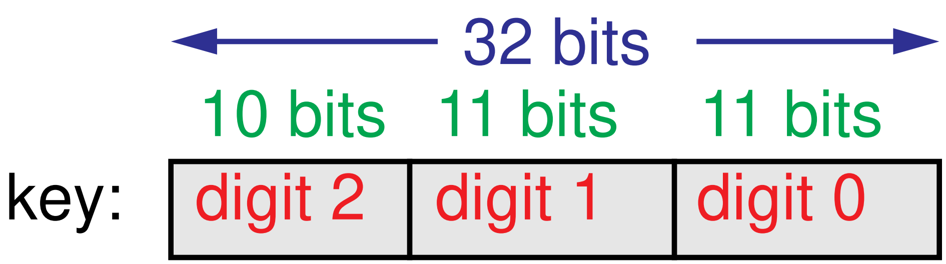
Problem Statement

- Start: *N* 32-bit unsorted keys
  - Blocked layout: *n=N/P* keys per processor (*N* >> *P*)
- Parameters:
  - Number of processors (*P* = 32 through 512)
  - Keys per processor (*n* = 16K through 1M)
  - *(Distribution of keys)*
- Finish: *N* keys in sorted order across processors



Local Sort: Radix Sort

- Many parallel sorting algorithms leverage a local sort (up to 80% of the execution time)
- Radix Sort (11-bit radix) fastest for 16K to 1M keys
  - Counting sort --> O(n)
  - Treat each key as $\left\lceil \frac{b}{r} \right\rceil$ digits



b = 32
r =11
digits = 3

Local Radix Sort: Algorithm

$\left\lceil \frac{b}{r} \right\rceil$ Repetitions:

1. Construct histogram: 2^r entries
   - Count number of occurrences of each digit
   - O(N)
2. Calculate rank of each digit
   - Scan across histogram
   - $O(2^r)$
3. Permute keys according to rank
   - Move key to intermediate destination; update ranks
   - O(N)

Repeat

Local Radix Sort: Example

Start: N = 4 keys, b = 4 bits, radix = 2

Keys: | 10 | 01 | | 01 | 10 | | 11 | 00 | | 01 | 00 |

1. Construct Histogram

00	01	10	11

2. Calculate Ranks

00	01	10	11

3. Permute Keys

Local Radix Sort: Example

2nd Pass:

Keys: 11 00 | 01 00 | 10 01 | 01 10

1. Construct Histogram

00	01	10	11

2. Calculate Ranks

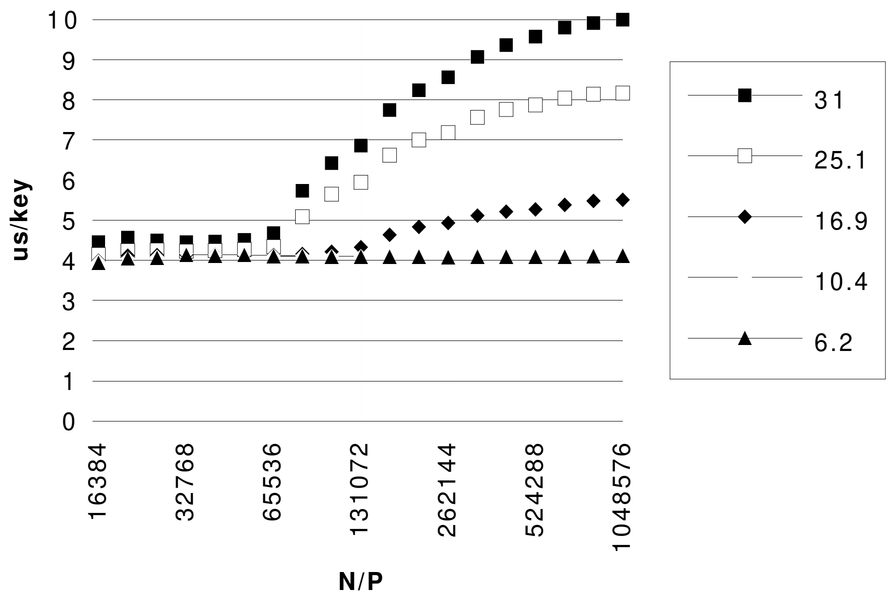
00	01	10	11

3. Permute Keys

Local Radix Sort: Performance

- Use measured performance for local model
- Time per key increases with *n*
  - TLB misses when storing keys in destination array
  - Lower entropy --> better locality --> faster execution

Top line is uniformly distributed data

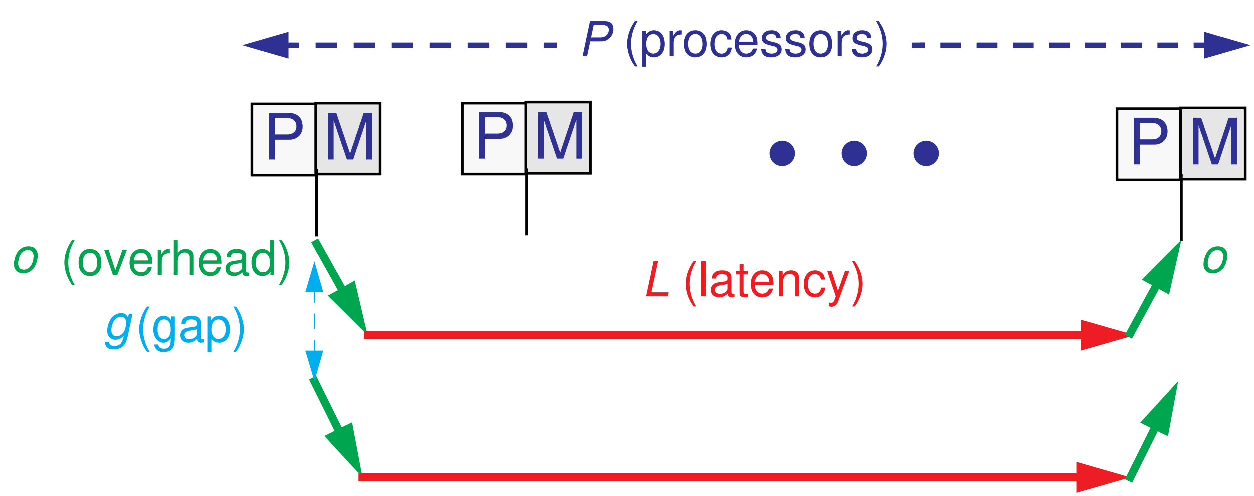


entropy (different distributions)

LogP Review

Model of distributed-memory multiprocessor

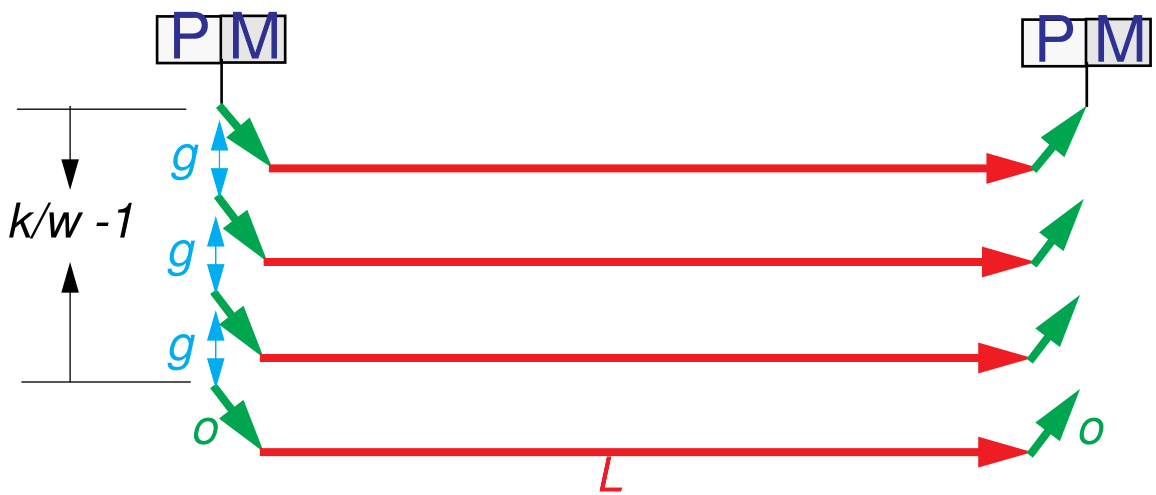
- Communicate with point-to-point messages



- Small Messages: *w* words per message
- Capacity Limit: Sender stalls if more than *L/g* messages in transit to a destination

LogP: Example

- Processor sends *k* words to another
  - $\left\lceil \frac{k}{w} \right\rceil$ messages
  - Assume $o < g$
  - Pipeline messages at interval g



$$Tsend(k) = \left(\left\lceil \frac{k}{w} \right\rceil - 1\right)g + o + L + o$$

LogP: Example

- Pair-wise exchange of *k* words: *k* large:

$$T_{exch}(k) = \left\lceil \frac{k}{w} \right\rceil max(g, 2o)$$

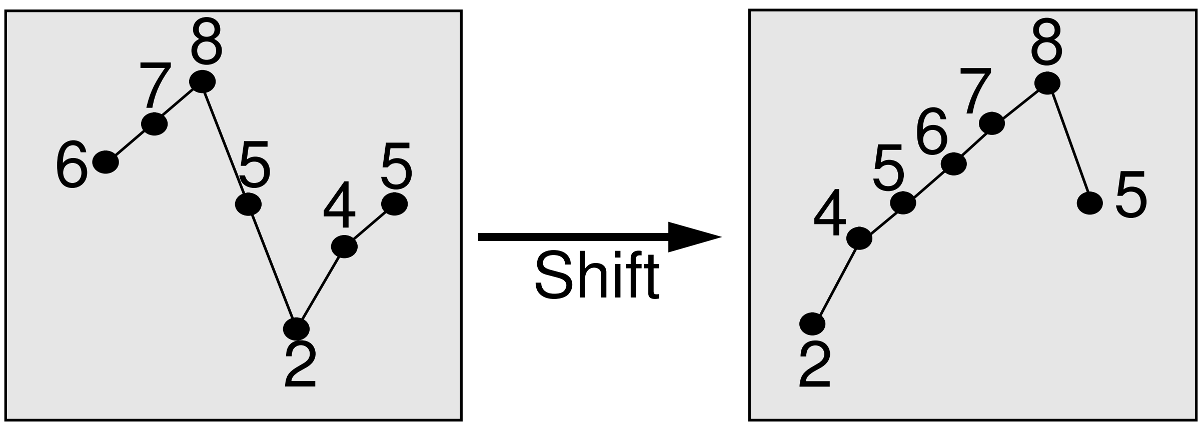
- Boundary conditions:

$$T_{exch}(k) = L + \left(\left\lceil \frac{k}{w} \right\rceil - 1 - \frac{L}{g}\right) max(g, 2o) + o + L + o$$

  - *L* units: Send *L/g* messages at rate *g*
  - Send and receive remaining msgs at rate *max(g, 2o)*
  - *o* units: Send last message
  - Final *L+o* units: Receive last *L/max(g,2o)* messages

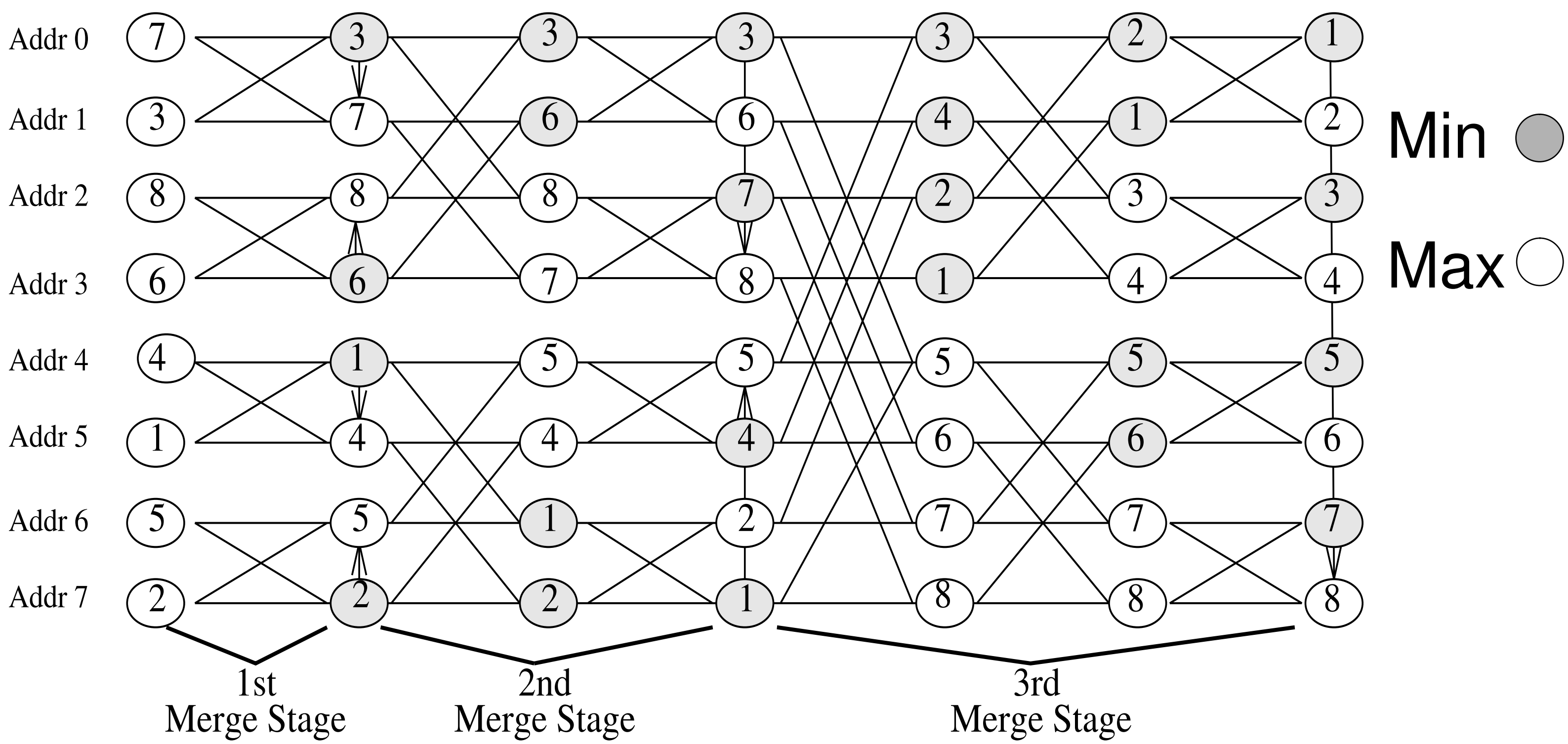
Bitonic Sort

- Classic parallel sorting algorithm
- Repeatedly merge two bitonic sequences to form a larger bitonic sequence
  - Bitonic sequence:
    A sequence that can be circularly shifted to first increase and then decrease monotonically
  - Example:



Bitonic Sort Algorithm

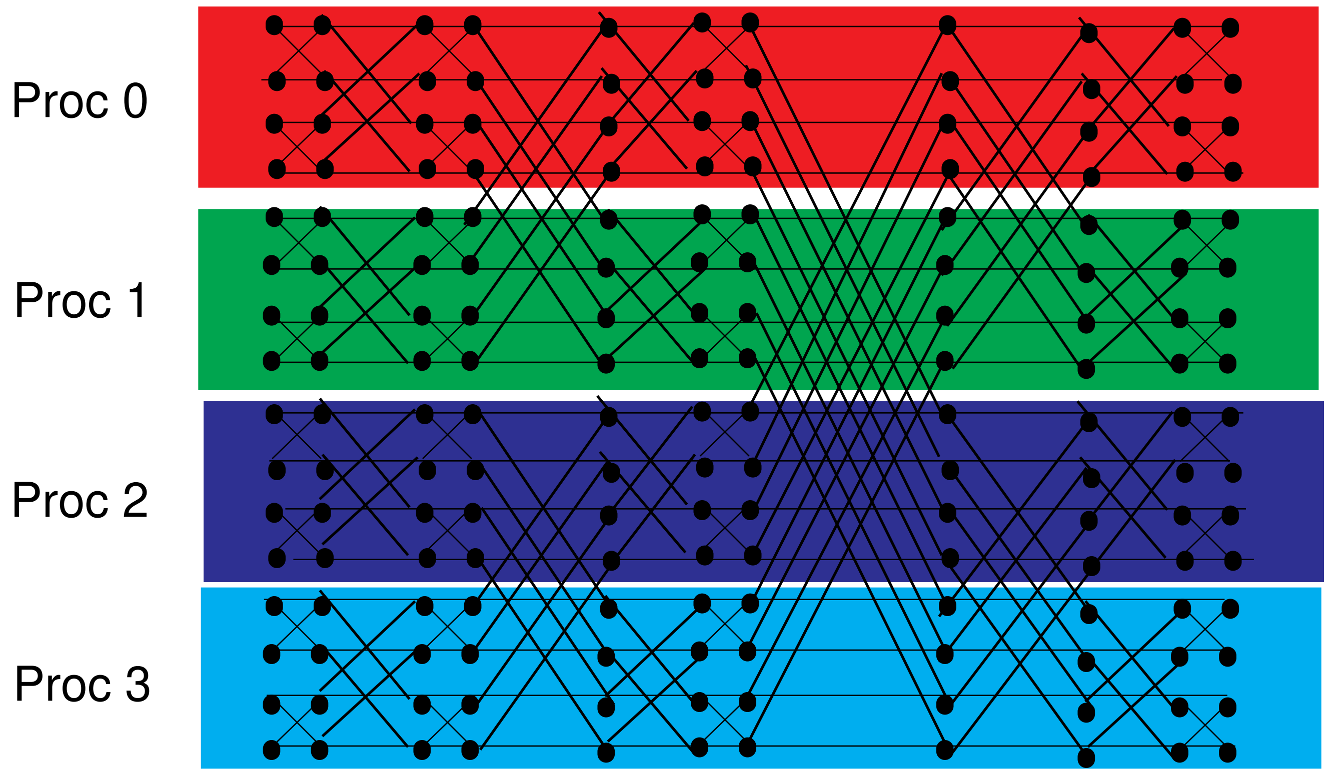
- *Lg N* merge stages to sort *N* keys
  - Bitonic sequences at end of each merge stage
  - Each node compares 2 keys and selects min or max



Bitonic Sort: Data Placement

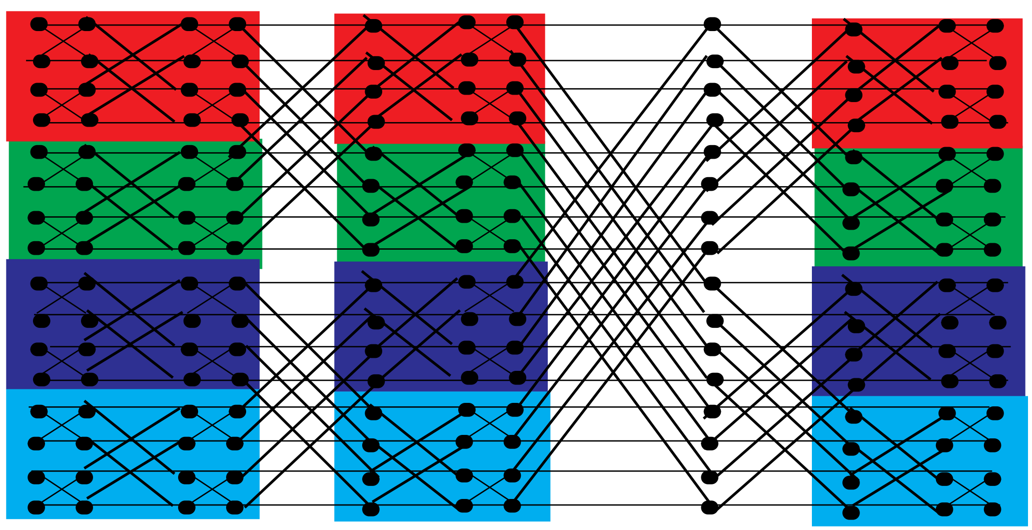
- Standard approach: Simulate nodes of butterfly
  - Blocked layout of keys to processors

N=16
P=4
n=4



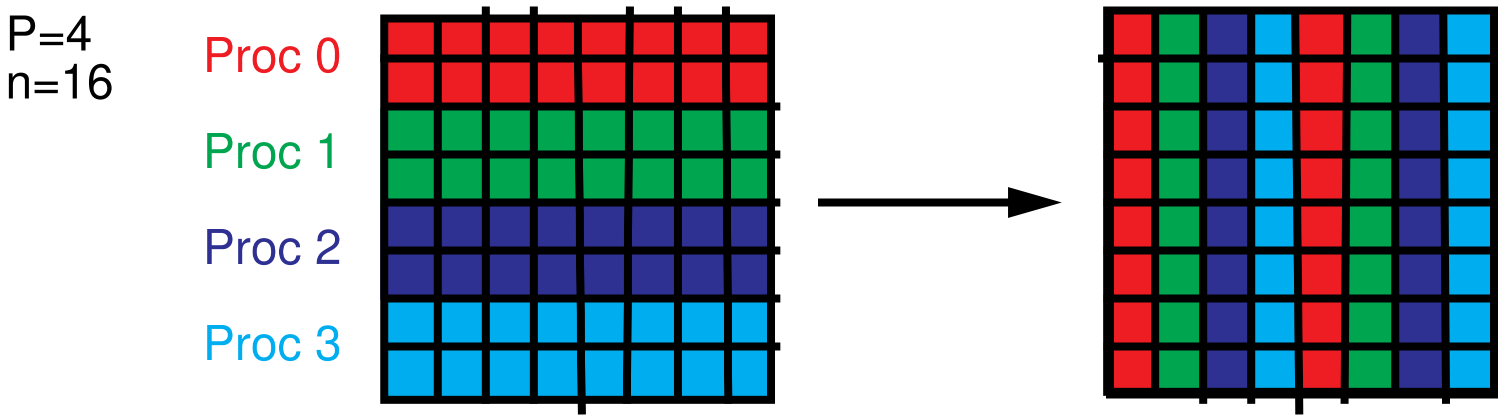
Bitonic Sort: Data Placement

- Reduce communication: Transpose blocked to cyclic
  - Begin with blocked layout
    First *lg n* merge stages entirely local: use radix sort
  - Transpose to cyclic
    i - lg n steps of merge stage *i* local: cmp & swap
  - Tranpose to blocked: replace last *lg n* steps with sort



Bitonic Sort: Transpose

- Blocked to Cyclic Layout



- Alternatives:
  - Send one key at a time
  - Gather keys and perform *P-1* bulk stores each of size *n/P* keys

Bitonic Sort: Transpose

- *(P-1)* bulk stores each of *n/P* keys
  - Regular and balanced all-to-all communication
  - Construct schedule to prevent destination contention
  - Iteration *i*, processor *p* exchanges with *p* XOR *i*

$$T_{transpose}(n) = (P-1)\,T_{exchange}\left(\frac{n}{P}\right)$$

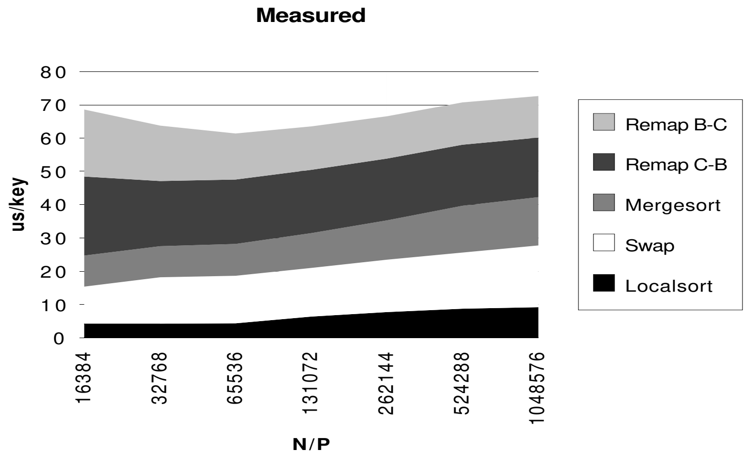
- Model prediction matches measurements within 15%
  - Remove gather/scatter from communication loop (sender stalls if destination ignores network)
  - Do not use *(p+i)%P* schedule (some sensitivity to network topology)

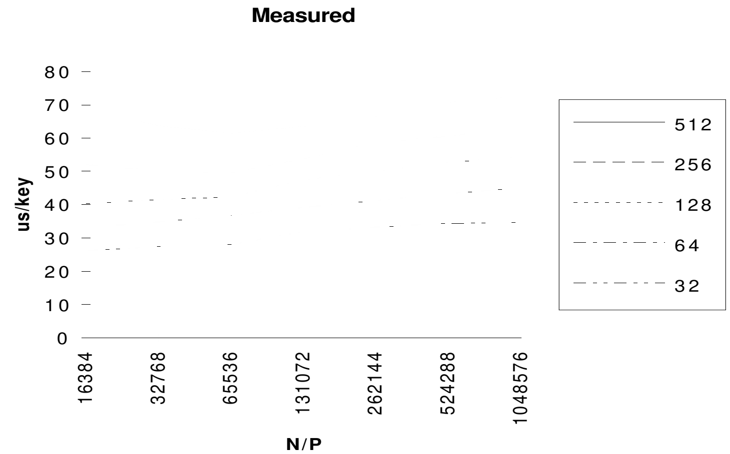
Bitonic Sort: Performance

- LogP Model: Time per local key increases with *P*

$$T_{Bitonic} = n \cdot t_{sort} + lgP\left(T_{xpose} + \frac{1}{2}(lgP)\ (n \cdot t_{swap}) + T_{xpose} + n \cdot t_{mergesort} \right)$$

- Measurements: Within 12% of model prediction





512 processors

Parallel Radix Sort

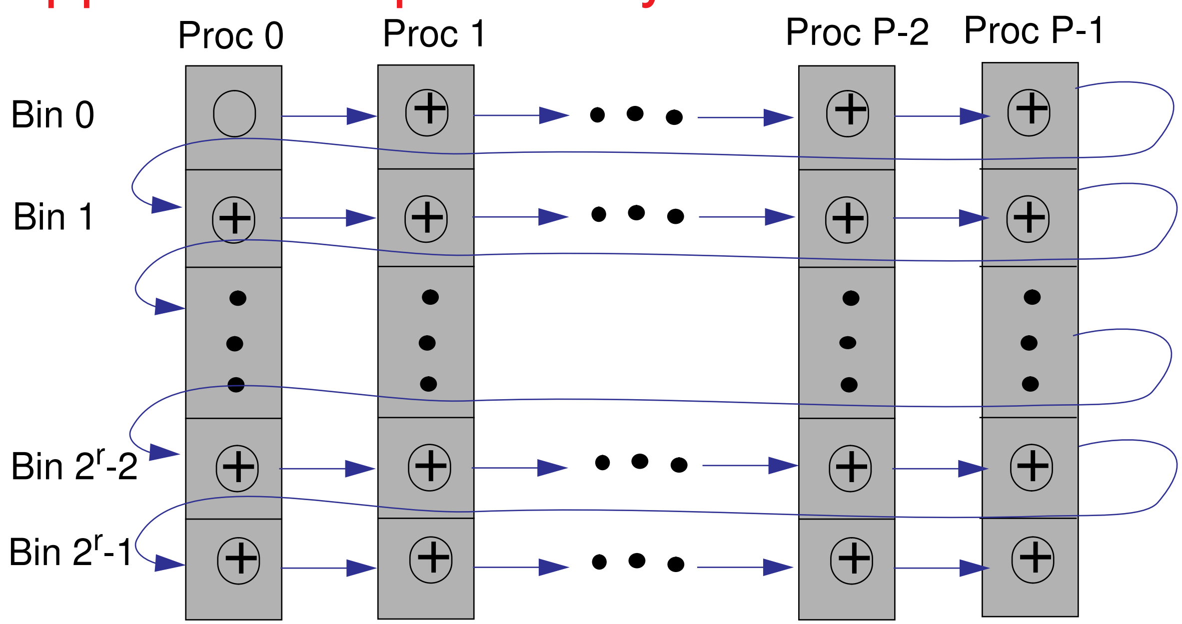
Straight-forward extension to local radix sort

- Blocked layout across processors
- $\left\lceil \frac{b}{r} \right\rceil$ passes; three phases in each pass ($r = 16$)

1. Each processor calculates histogram for local digits
   - Entirely local computation
2. Calculate global rank of digits
   - Scan local histograms to create *global histogram*
3. Permute keys to intermediate destinations
   - Store key at correct offset on destination processor
   - Update local histogram

Radix Sort: Global Histogram (#2)

Naive approach: Dependency across rows

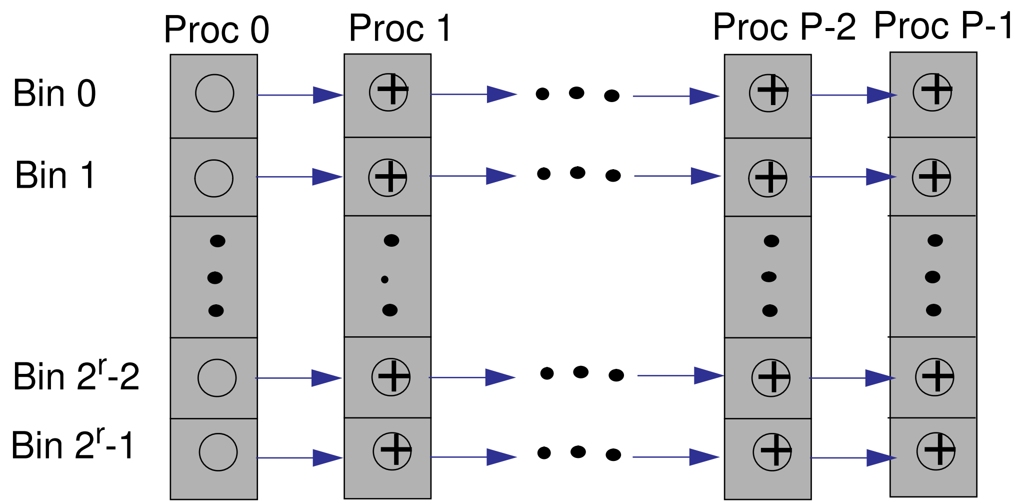


$$GlobalHist[0]\,[p] = \sum_{j=0}^{p-1} LocalHist[0]\,[j]$$

$$GlobalHist[i]\,[p] = GlobalHist[i-1]\,[P-1] + \sum_{j=0}^{p-1} LocalHist[i]\,[j]$$

Radix Sort: Global Histogram

Partial Histogram: Rows independent



$$PartialHist[i][p] = \sum_{j=0}^{p-1} LocalHist[i][j]$$

$$GlobalHist[i][p] = GlobalHist[i-1][P-1] + PartialHist[i][p]$$

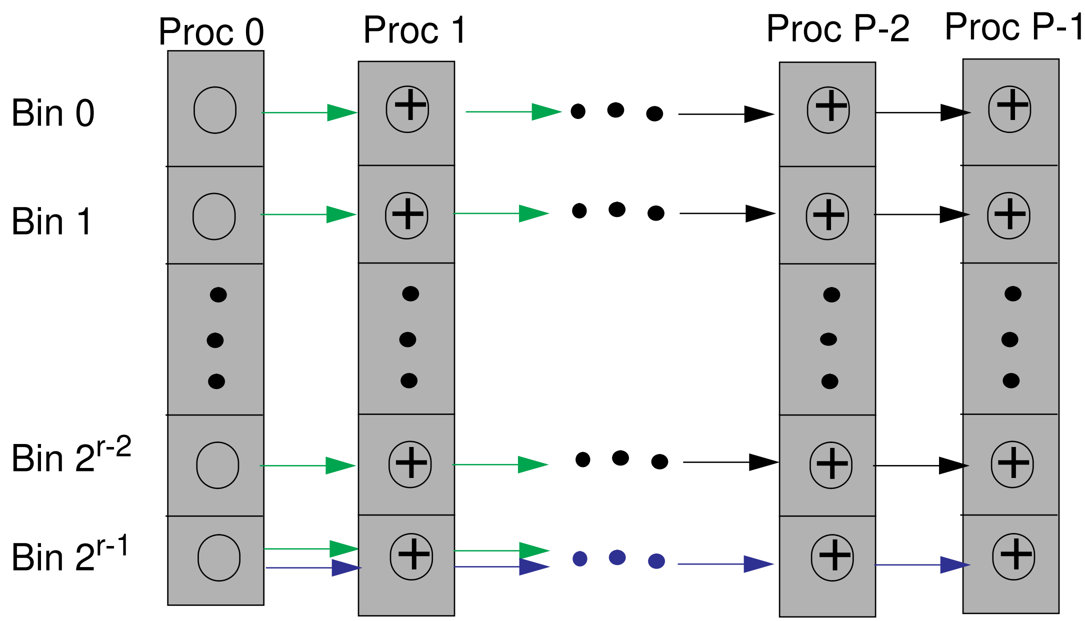
How to implement 2^r-1 independent scans (multi-scan)?

Radix Sort: Multi-Scan

- Single-Scan
  - Tree approaches: O(lg *P*)
  - Pipelined approach: O(*P*)
- Multi-Scan Tree Approach:
  - Sending $n=2^{r-1}$ words requires time *2ng* at root
  - Add time O(lg *P*) to propagate to all processors
- Multi-Scan Pipelined Approach:
  - Requires time *ng* at root
  - Add time O(*P*) to propagate
  - Faster when $n >> P$

Radix Sort: Multi-Scan

LogP analysis of pipelined Multi-Scan



$$Tscan = (2^r)t_{zero} + (2^r)\max(g, 2o + t_{add}) + (P-1)*(L+2o+t_{add})$$

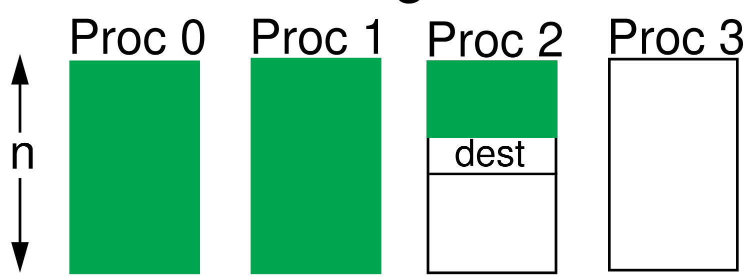
What happens when we implement and measure this?

Radix Sort: Multi-Scan

- Implementation is slower than model by a factor of two
- What happened?
  - LogP: assumed forward data immediately after receiving
  - Split-C: all arriving messages handled together
  - P0 sends faster than P1 forwards (g vs. $2o + t_{add}$)
  - P1 receives many before forwarding any.
  - Other processors idle
- Solution:
  - Slow down the sending rate of P0
  - Measured time within 3% of model prediction

Radix Sort: Key Permutation (#3)

- Each processor, *p*, loops through local keys
  - The rank of a digit with value *k*: *GlobalHist[p][k]*



Dest Proc: rank div n
Dest Offset: rank mod n

- All-to-all irregular communication
  - Arbitrary permutation leads to contention
  - Ignore contention in model:

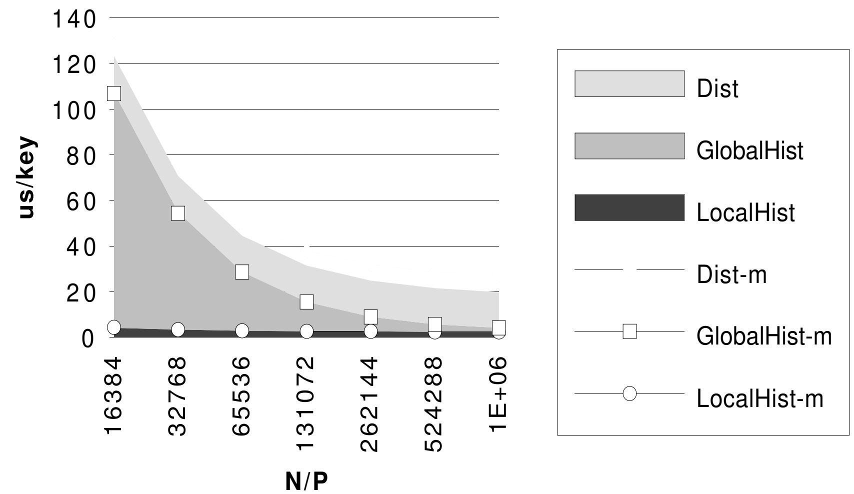
$$T_{Distribute} = n \cdot max\,(g, 2o + t_{AddrLookup})$$

- Alternative: Gather keys for dests to utilize *w* (slower)

Radix Sort: Performance

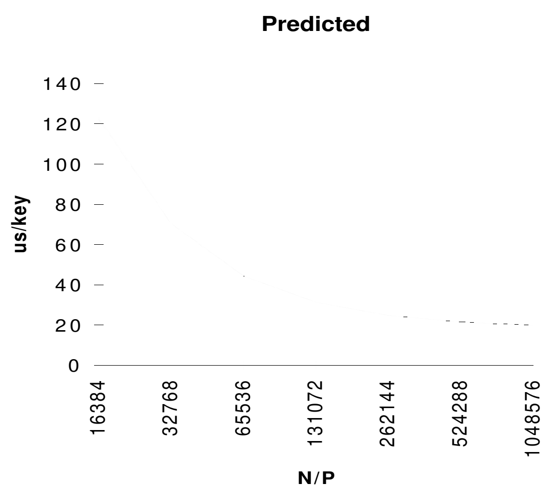
Predicted and Measured Performance

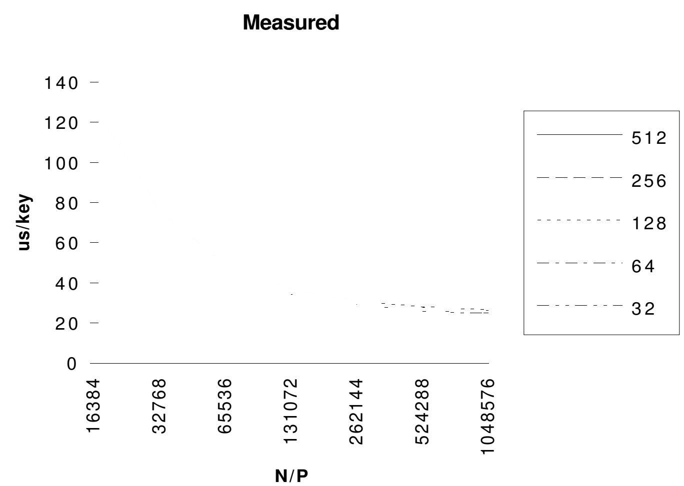
512 Processors



predicted

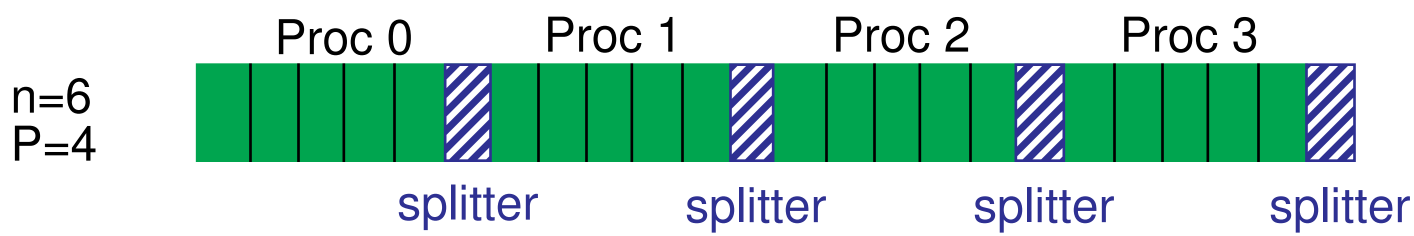
measured





Sample Sort

- Basic idea of Sample Sort:
  - Suppose: Select every *n*-th key in the sorted order
  - These *P* values split the keys into *P* equal pieces



  - Each processor has copy of splitters
  - Loop through local keys and send to dest. processor
  - Determine dest. with binary search on splitter array
  - Sort local keys
- How do we select the splitters?

Sample Sort: Splitter Selection (#1)

- Each processor selects *s* random local keys
  - Larger *s* (sample size) gives better approximation
- Send sampled keys to processor 0
  - Processor 0 sorts the samples
  - Selects every *s*-th key as a splitter
  - Broadcasts the *P* splitters
- Splitters not chosen perfectly
  - After key distribution: uneven # of keys per proc
  - *E* (expansion factor): ratio of max keys to average
  - *E* bounded by small constant w. high probability (measured $1.22 < E < 1.45$)

Sample Sort: Key Distribution (#2)

- Irregular, Unbalanced all-to-all communication
  - Destination contention
  - Limited by processor receiving *E*n* keys
- Binary search on splitters before sending each key

$$t_{search} = t_{compare} \cdot lg(P)$$

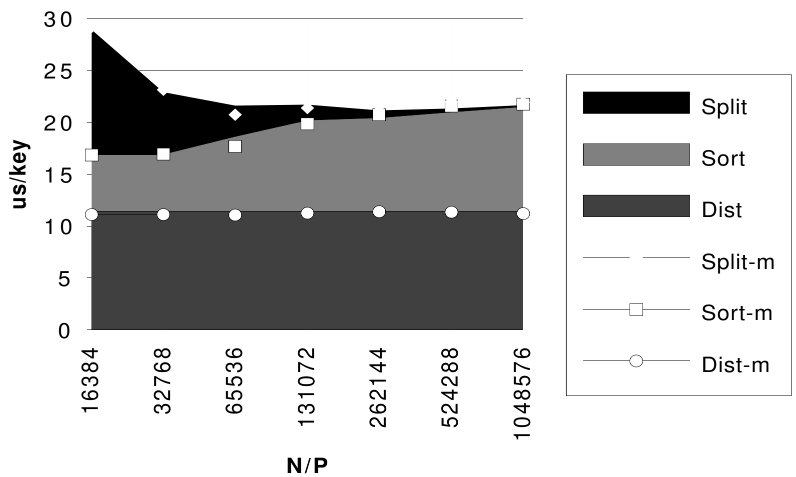
- Total time for key distribution phase

$$T_{Distribute} = n \cdot max(E \cdot g, t_{search} + o + E \cdot o)$$

Sample Sort: Performance

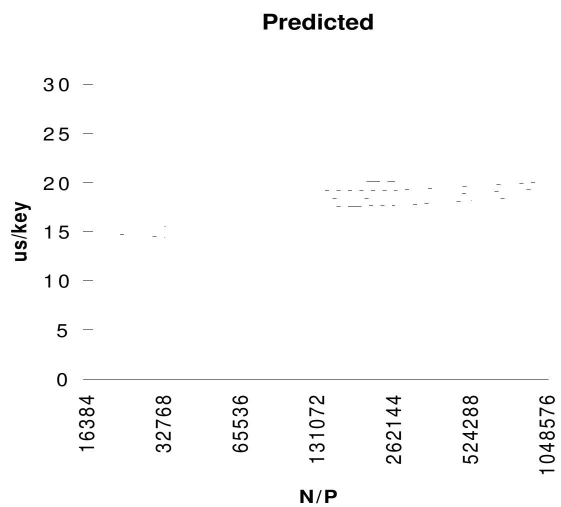
Predicted and Measured Execution Time:

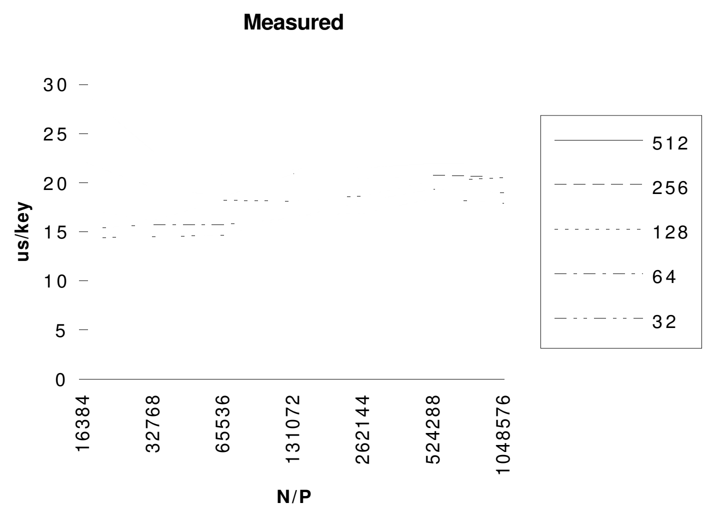
512 Processors



predicted

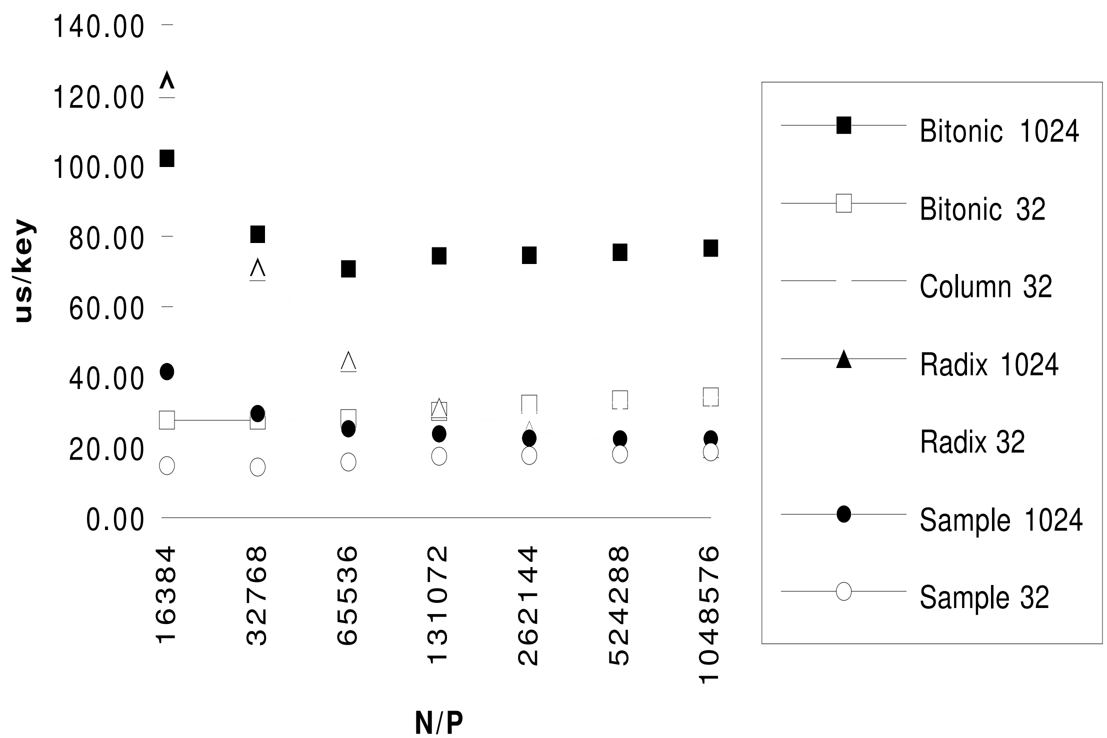
measured





Parallel Sorting: Comparison

Performance of Four Sorting Algorithms



Parallel Sorting: Conclusion

- Bitonic sort does not map well to real processors
- Fastest sorting algorithm depends upon *P* and *n*
  - Radix fastest on 1M+ keys on 1024 processors
  - Sample fastest on less keys and 1024 procs
  - Sample fastest with 32 processors
- Advantages to radix sort:
  - Easier to implement than sample sort
  - Equal number of keys per processor at end