Parallel Bucket Sort

Sequential Bucket Sort

(a)  

(b)
Sequential Bucket Sort

- Works well only if numbers in list are uniformly distributed across known interval, e.g. $0 - a-1$
- Divide interval into $m$ equal (bucket) regions
- Numbers in buckets then sorted using quicksort or mergesort (not the one we’ve been using)
- Are alternatives, e.g., can recursively divide buckets into smaller buckets $\Rightarrow$ like quicksort without pivot

Figure 4.8 Bucket sort.
Sequential Bucket Sort

- Compare each number with start of bucket => $m-1$ comparisons per number

OR
- Divide each number by $a/m$ to get bucket number.
- If $m$ and $a$ power of 2, just need to look at upper bits of number in binary, e.g., assume $m$ is 8, $a$ is 128 and number in binary is 1100101. This would go into bucket 6 (110) => less expensive than division.
- How do you know which set of bits to look at?

Sequential Bucket Sort

Best case analysis:
- Assume placing a number into bucket requires 1 step and there are $n$ numbers to place. If distribution is uniform $n/m$ numbers/bucket
- Quicksort/mergesort $\Theta(n \log n)$, e.g.
  $$(n/m) \log(n/m)$$ here.
- $T_{seq} = n + m \left( (n/m) \log(n/m) \right) = n + n \log(n/m) = \Theta(n \log(n/m))$
  if no time needed to compile final sorted list
Parallel Bucket Sort (1)

- One bucket per processor
- $T_{par1} = n + n/p \log(n/p)$, where $n$ computations needed to place numbers and then numbers are quicksorted.
Parallel Bucket Sort (2)

Each processor handles $1/p$ of original array and does a bucket sort on it…for mpi

**Phase 1**: Computation and Communication
- $p$ computations needed to partition $n$ numbers into $p$ regions, i.e.,
  $$t_{\text{compl}} = p$$
- $p$ partitions containing $n/p$ numbers are sent to processes using broadcast or scatter routine, (Assumes it takes $n$ pieces of data take $n$ times as long to send as one does, i.e. $t_{\text{data}}$) i.e.,
  $$t_{\text{comm1}} = t_{\text{startup}} + n \, t_{\text{data}}$$
Parallel Bucket Sort (2)

Each processor does a bucket sort into $p$ buckets

**Phase 2: Computation**
- $T_{comp2} = n/p$

Parallel Bucket Sort (2)

If uniform distribution, each small bucket has $n/p^2$ numbers. Each process must send contents of $p-1$ small buckets to other processes

**Phase 3: Communication**
- $p$ processors need to make this communication, if cannot overlap
  $$t_{comp3} = p( p-1 ) ( t_{startup} + ( n/p^2 ) t_{data} )$$
- If communications can overlap, then lower bound is
  $$t_{comm3} = ( p-1 ) ( t_{startup} + ( n/p^2 ) t_{data} )$$
Parallel Bucket Sort (2)

Each processor must sort approximately $n/p$ numbers.

**Phase 4: Computation**

$$t_{comp4} = (n/p) \log(n/p)$$

**Final (lower bound) Cost:**

$$T_{par2} = p + n/p + (n + (p-1)n/p^2) t_{data} + p t_{startup} + (n/p) \log(n/p)$$

---

Questions

- Is/When is $T_{par1}$ less then $T_{par2}$?
- How do you guarantee relatively uniform distribution?
- If you can’t, how do you adjust?
**MPI_AlltoAll and MPI_Alltoallv()**

- First sends equal amount of data from each processor to each other processor, e.g., use to tell each processor how much data to expect from each other processor.
- Second sends unequal amounts

*Figure 4.11* "All-to-all" broadcast.
MPI_Alltoall()

int MPI_Alltoall(  
    void* send_buffer,  /* input */,  
    int send_count,  /* input */,  
    MPI_Datatype send_type,  /* input */,  
    void* recv_buffer,  /* output */,  
    int recv_count,  /* input */,  
    MPI_Datatype recv_type,  /* input */,  
    MPI_Comm communicator  /* input */)  

MPI_Alltoall()

• The effect on process $q$ is to send $send_count$ elements of type $send_type$ to every process (including itself). The first block of $send_count$ elements goes to process 0, the second block to process 1, etc.

• Process $q$ will also receive $recv_count$ elements of type $recv_type$ from every process. The elements from process 0 are received into the beginning of $recv_buffer$. The elements from process 1 are received immediately following those from 0, etc.
MPI_Alltoallv()

int MPI_Alltoallv(
    void* send_buffer /* input */,
    int send_counts[] /* input */,
    int send_displacements[] /* input */,
    MPI_Datatype send_type /* input */,
    void* recv_buffer /* output */,
    int recv_counts[] /* input */,
    int recv_displacements[] /* input */,
    MPI_Datatype recv_type /* input */,
    MPI_Comm communicator /* input */)
Figure 4.12 Effect of "all-to-all" on an array.