Parallel Sorting

Chris Kauffman

CS 499: Spring 2016 GMU
Logistics

Today

▶ Trailing questions on HW2?
▶ Review Parallel Performance Theory
▶ Parallel Sorting

Normal Office Hours

▶ Tue 3/1 3:30-5:30

Reading: Grama Ch 9

▶ Sorting
▶ Focus on 9.4: Quicksort

Schedule

Tue 2/23   PageRank & MPI
Thu 2/25   Performance Analysis
Mon 2/29   HW 2 Due 11:59pm
Tue 3/1    Performance, Parallel Sorting
Thu 3/3    Guest Lecture, Mini-Exam 2
3/2-3/4    HW 2 Interviews
Quick Review

- What is Amdahl’s law? What does it say about the speedup achievable by parallel programs?
- How does one calculate the following for a parallel algorithm:
  - S: Speedup
  - E: Efficiency
  - C: Cost
- How does the Efficiency of a parallel usually change if the problem size increases but the number of processors P stays the same?
- How does the Efficiency of a parallel usually change if the number of processors P increases but the problem size stays the same?
- What is Parallel Overhead?
- What is Isoefficiency?
Sorting

- Much loved computation problem
- What is the best complexity of general purpose (comparison-based) sorting algorithms?
- What are some algorithms which have this complexity?
- What are some other sorting algorithms which aren’t so hot?
- What issues need to be addressed to parallelize any sorting algorithm?
Partition and Quicksort

- Quicksort has $O(N \log N)$ average complexity
- In-place, low overhead sorting, recursive

Partition

- Partition: select pivot value
- On completion
  - Left array is $\leq$ pivot
  - Right array is $>$ pivot
  - pivot is in "middle"

```
algorithm partition(A, lo, hi) is
  pivot := A[hi]
  boundary := lo
  for j := lo to hi - 1 do
    if A[j] <= pivot then
      boundary++
  swap A[i] with A[hi]
  return boundary
```

Quicksort

- Partition into two parts
- Recurse on both halves
- Bail out when boundaries lo/hi cross

```
algorithm quicksort(A, lo, hi) is
  if lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)
```
Practical Parallel Sorting Setup

- Input array A of size N is already spread across P processors (no need to scatter)
  P0: A[] = { 84 31 21 28 }
  P1: A[] = { 17 20 24 84 }
  P2: A[] = { 24 11 31 99 }
  P3: A[] = { 13 32 26 75 }

- Goal: Numbers sorted across processors. Smallest on P0, next smallest on P1, etc.
  P0: A[] = { 11 13 17 20 }
  P1: A[] = { 21 24 24 26 }
  P2: A[] = { 28 31 32 33 }
  P3: A[] = { 75 84 84 99 }

- Want to use P processors as effectively as possible
- Bulk communication preferred over many small messages
Exercise: Parallel Quicksort

- Find a way to parallelize quicksort
- Hint: The last step is each processor sorting its own data using a serial algorithm. Try to arrange data so this is possible.

START:

P0: A[] = { 84 32 21 28 }
P1: A[] = { 17 20 25 85 }
P2: A[] = { 24 11 31 99 }
P3: A[] = { 13 32 26 75 }

GOAL

P0: A[] = { 11 13 17 20 }
P1: A[] = { 21 24 25 26 }
P2: A[] = { 28 31 32 33 }
P3: A[] = { 75 84 85 99 }

SERIAL ALGORITHM

algorithm quicksort(A, lo, hi) is
if lo < hi then
  p := partition(A, lo, hi)
  quicksort(A, lo, p - 1)
  quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
  pivot := A[hi]
  boundary := lo
  for j := lo to hi - 1 do
    if A[j] <= pivot then
      boundary++
    swap A[i] with A[hi]
  return boundary
Parallel Quicksort Ideas 1

\[
A[] = \{ 84 \ 32 \ 21 \ 11 \ | \ 17 \ 20 \ 25 \ 85 \ | \ 24 \ 28 \ 31 \ 99 \ | \ 13 \ 32 \ 26 \ 75 \ \}
\]

\[\begin{array}{cccc}
P0 & P1 & P2 & P3 \\
\end{array}\]

Partition(pivot=26) on each processor
\[
A[] = \{ 21 \ 11 \ 84 \ 32 \ | \ 17 \ 20 \ 25 \ 85 \ | \ 24 \ 28 \ 31 \ 99 \ | \ 13 \ 26 \ 32 \ 75 \ \}
\]

Boundary: ~ ~ ~ ~
Counts: P0: 2 P1: 3 P2: 1 P3: 2
Calculate which data goes where
\[
A[] = \{ 21 \ 11 \ 84 \ 32 \ | \ 17 \ 20 \ 25 \ 85 \ | \ 24 \ 28 \ 31 \ 99 \ | \ 13 \ 26 \ 32 \ 75 \ \}
\]

\[\begin{array}{cccccc}
P0 & P0 & P2 & P2 & P0 & P0 & P1 & P2 & P1 & P2 & P3 & P3 & P1 & P1 & P3 & P3 \\
\end{array}\]

Re-arrange so values <= 26 on P0 and P1, > 26 on P2 and P3
\[
A[] = \{ 21 \ 11 \ 17 \ 20 \ | \ 25 \ 24 \ 13 \ 25 \ | \ 84 \ 32 \ 85 \ 28 \ | \ 31 \ 99 \ 23 \ 75 \ \}
\]

\[\begin{array}{cccc}
P0 & P1 & P2 & P3 \\
\end{array}\]

Split the world: 2 groups
\[
A[] = \{ 21 \ 11 \ 17 \ 20 \ | \ 25 \ 24 \ 13 \ 25 \} | \{84 \ 32 \ 85 \ 28 \ | \ 31 \ 99 \ 23 \ 75 \ \}
\]

\[\begin{array}{cccc}
P0 & P1 & P2 & P3 \\
\end{array}\]
Parallel Quicksort Ideas 2

Each half partitions on different value
P0-P1: Partition(pivot=20)
P2-P3: Partition(pivot=32)
A[] = { 11 17 20 21 | 13 25 24 25 } | { 28 32 84 85 | 31 23 99 75 }
Boundary: ^ ^ ^ ^
Counts: P0: 3 P1: 1 P2: 2 P3: 2
Calculate which data goes where
A[] = { 11 17 20 21 | 13 25 24 25 } | { 28 32 84 85 | 31 23 99 75 }
P0 P0 P0 P1 P0 P1 P1 P1 P2 P2 P3 P3 P2 P2 P3 P3
Re-arrange values to proper processors
A[] = { 11 17 20 13 | 21 25 24 25 } | { 28 32 31 23 | 84 85 99 75 }
P0 P0 P1 P1 P2 P2 P3 P3
Split the world: 4 groups
A[] = { 11 17 20 13 } | { 21 25 24 25 } | { 28 32 31 23 } | { 84 85 99 75 }
P0 P0 P1 P1 P2 P2 P3 P3

4 groups == 4 processors, all processors sort locally
A[] = { 11 13 17 20 } | { 21 24 25 25 } | { 23 28 31 32 } | { 75 84 85 99 }
P0 P0 P1 P1 P2 P2 P3 P3
Done
Issues

- Pivots were cherry-picked to get even distribution
- Generally not possible to do: processors might have uneven portions of the array after partitioning
- Will require
- Must figure out how to communicate which elements to each processor
- Must split the world into smaller groups
Prefix Sums / Scan

```c
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- Similar to reduction
- Change: only add on values from procs $\leq \text{proc}_id$
- `op` can be sum/max/min/etc.
- In Quicksort, use All-gather to get an array of counts of small values on each proc, follow with Prefix Sum to calculate how much to send to each processor
All-to-all personalized communication: like every processor scattering to every other processor.

Source: Cornell University Center for Advanced Computing
MPI_Alltoall

- Standard version: every processor gets a slice of sendbuf, same sized data
- Vector version allows different sized slices (appropriate for quicksort)

```c
int MPI_Alltoall(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm);

int MPI_Alltoallv(
    void *sendbuf, int sendcounts[], int sdispls[], MPI_Datatype sendtype,
    void *recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtype,
    MPI_Comm comm);
```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
  MPI_Comm *newcomm);

- comm is the old communicator (start with MPI_COMM_WORLD)
- color is which sub-comm to go into
- key establishes rank in new sub-comm, usually proc_id
- newcomm is filled in with a new communicator
- Examine mpi-code/comm-split.c

Splitting the World
Ultimate Complexity of Parallel Quicksort

Take a moment to calculate $O$ complexity based on

- $N$ elements
- $P$ processors