

# Parallel Sorting Algorithms

- 1 Sorting in C and C++
  - using `qsort` in C
  - using STL `sort` in C++
- 2 Bucket Sort for Distributed Memory
  - bucket sort in parallel
  - communication versus computation
- 3 Quicksort for Shared Memory
  - partitioning numbers
  - quicksort with OpenMP

MCS 572 Lecture 12  
Introduction to Supercomputing  
Jan Verschelde, 6 February 2012

# Parallel Sorting Algorithms

- 1 **Sorting in C and C++**
  - using `qsort` in C
  - using STL `sort` in C++
- 2 **Bucket Sort for Distributed Memory**
  - bucket sort in parallel
  - communication versus computation
- 3 **Quicksort for Shared Memory**
  - partitioning numbers
  - quicksort with OpenMP

## using `qsort`

C provides an implementation of quicksort. The prototype is

```
void qsort ( void *base, size_t count, size_t size,  
            int (*compar)(const void *element1,  
                          const void *element2) );
```

`qsort` sorts an array whose first element is pointed to by `base` and contains `count` elements, of the given `size`.

The function `compar` returns

- $-1$  if `element1 < element2`,
- $0$  if `element1 = element2`,
- $+1$  if `element1 > element2`.

We will apply `qsort` to sort a random sequence of doubles.

## generating and writing numbers

```
void random_numbers ( int n, double a[n] )
{
    int i;
    for(i=0; i<n; i++)
        a[i] = ((double) rand())/RAND_MAX;
}
```

```
void write_numbers ( int n, double a[n] )
{
    int i;
    for(i=0; i<n; i++) printf("%.15e\n", a[i]);
}
```

## using qsort

```
int compare ( const void *e1, const void *e2 )
{
    double *i1 = (double*)e1;
    double *i2 = (double*)e2;
    return ((*i1 < *i2) ? -1 : (*i1 > *i2) ? +1 : 0);
}
```

in the function main():

```
double *a;
a = (double*)calloc(n,sizeof(double));
random_numbers(n,a);

qsort((void*)a,(size_t)n,sizeof(double),compare);
```

## code to time qsort

We use the command line to enter the dimension and to toggle off the output.

To measure the CPU time for sorting:

```
clock_t tstart, tstop;
tstart = clock();
qsort((void*)a, (size_t)n, sizeof(double), compare);
tstop = clock();
printf("time elapsed : %.4lf seconds\n",
      (tstop - tstart)/((double) CLOCKS_PER_SEC));
```

## timing `qsort` on 3.47Ghz Intel Xeon

```
$ time /tmp/time_qsort 1000000 0
time elapsed : 0.2100 seconds
real      0m0.231s
user      0m0.225s
sys       0m0.006s
$ time /tmp/time_qsort 10000000 0
time elapsed : 2.5700 seconds
real      0m2.683s
user      0m2.650s
sys       0m0.033s
$ time /tmp/time_qsort 100000000 0
time elapsed : 29.5600 seconds
real      0m30.641s
user      0m30.409s
sys       0m0.226s
```

Observe:  $O(n \log_2(n))$  is almost linear in  $n$ .

# Parallel Sorting Algorithms

- 1 **Sorting in C and C++**
  - using `qsort` in C
  - using **STL `sort`** in C++
- 2 **Bucket Sort for Distributed Memory**
  - bucket sort in parallel
  - communication versus computation
- 3 **Quicksort for Shared Memory**
  - partitioning numbers
  - quicksort with OpenMP

## using the STL container vector

```
#include <vector>
using namespace std;

vector<double> random_vector ( int n );
// returns a vector of n random doubles

vector<double> random_vector ( int n )
{
    vector<double> v;
    for(int i=0; i<n; i++)
    {
        double r = (double) rand();
        r = r/RAND_MAX;
        v.push_back(r);
    }
    return v;
}
```

## writing STL vectors

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

void write_vector ( vector<double> v );
// writes the vector v

void write_vector ( vector<double> v )
{
    for(int i=0; i<v.size(); i++)
        cout << scientific
            << setprecision(15)
            << v[i] << endl;
}
```

## using the STL sort

```
#include <vector>
#include <algorithm>
using namespace std;

struct less_than // defines "<"
{
    bool operator()(const double& a,
                    const double& b)
    {
        return (a < b);
    }
};
```

in the main program:

```
sort(v.begin(), v.end(), less_than());
```

## timing STL sort on 3.47Ghz Intel Xeon

```
$ time /tmp/time_stl_sort 1000000 0
time elapsed : 0.36 seconds
real      0m0.376s
user      0m0.371s
sys       0m0.004s
$ time /tmp/time_stl_sort 10000000 0
time elapsed : 4.09 seconds
real      0m4.309s
user      0m4.275s
sys       0m0.033s
$ time /tmp/time_stl_sort 100000000 0
time elapsed : 46.5 seconds
real      0m48.610s
user      0m48.336s
sys       0m0.267s
```

Different distributions may cause timings to fluctuate.

# Parallel Sorting Algorithms

- 1 Sorting in C and C++
  - using `qsort` in C
  - using STL `sort` in C++
- 2 **Bucket Sort for Distributed Memory**
  - **bucket sort in parallel**
  - communication versus computation
- 3 Quicksort for Shared Memory
  - partitioning numbers
  - quicksort with OpenMP

# bucket sort

Given are  $n$  numbers, suppose all are in  $[0, 1]$ .

The algorithm using  $p$  buckets proceeds in two steps:

- Partition numbers  $x$  into  $p$  buckets:  
 $x \in [i/p, (i+1)/p[ \Rightarrow x \in (i+1)\text{th bucket.}$
- Sort all  $p$  buckets.

The cost to partition the numbers into  $p$  buckets is  $O(n \log_2(p))$ .

Note: radix sort uses most significant bits to partition.

In the best case: every bucket contains  $n/p$  numbers.

The cost of Quicksort is  $O(n/p \log_2(n/p))$  per bucket.

Sorting  $p$  buckets takes  $O(n \log_2(n/p))$ .

Total cost is  $O(n(\log_2(p) + \log_2(n/p)))$ .

## parallel bucket sort

On  $p$  processors, all nodes sort:

- 1 Root node distributes numbers: processor  $i$  gets  $i$ th bucket.
- 2 Processor  $i$  sorts  $i$ th bucket.
- 3 Root node collects sorted buckets from processors.

Is it worth it? Recall: serial cost is  $n(\log_2(p) + \log_2(n/p))$ .

Cost of parallel algorithm:

- $n \log_2(p)$  to place numbers into buckets,
- $n/p \log_2(n/p)$  to sort buckets.

$$\begin{aligned} \text{speed up} &= \frac{n(\log_2(p) + \log_2(n/p))}{n(\log_2(p) + \log_2(n/p)/p)} \\ &= \frac{1 + L}{1 + L/p} = \frac{1 + L}{(p + L)/p} = \frac{p}{p + L}(1 + L), \quad L = \frac{\log_2(n/p)}{\log_2(p)}. \end{aligned}$$

## comparing to quicksort

$$\begin{aligned}\text{speed up} &= \frac{n \log_2(n)}{n(\log_2(p) + n/p \log_2(n/p))} \\ &= \frac{\log_2(n)}{\log_2(p) + 1/p(\log_2(n) - \log_2(p))} \\ &= \frac{\log_2(n)}{1/p(\log_2(n) + (1 - 1/p) \log_2(p))}\end{aligned}$$

Example:  $n = 2^{20}$ ,  $\log_2(n) = 20$ ,  $p = 2^2$ ,  $\log_2(p) = 2$ ,

$$\begin{aligned}\text{speed up} &= \frac{20}{1/4(20) + (1 - 1/4)2} \\ &= \frac{20}{5 + 3/2} = \frac{40}{13} \approx 3.08.\end{aligned}$$

# Parallel Sorting Algorithms

- 1 Sorting in C and C++
  - using `qsort` in C
  - using STL `sort` in C++
- 2 **Bucket Sort for Distributed Memory**
  - bucket sort in parallel
  - **communication versus computation**
- 3 Quicksort for Shared Memory
  - partitioning numbers
  - quicksort with OpenMP

# communication and computation

The scatter of  $n$  data elements costs  $t_{\text{start up}} + nt_{\text{data}}$ ,  
where  $t_{\text{data}}$  is the cost of sending 1 data element.

For distributing and collecting of all buckets,  
the total communication time is  $2p \left( t_{\text{start up}} + \frac{n}{p} t_{\text{data}} \right)$ .

The computation/communication ratio is

$$\frac{(n \log_2(p) + n/p \log_2(n/p)) t_{\text{compare}}}{2p \left( t_{\text{start up}} + \frac{n}{p} t_{\text{data}} \right)}$$

where  $t_{\text{compare}}$  is the cost for one comparison.

## the computation/communication ratio

The computation/communication ratio is

$$\frac{(n \log_2(p) + n/p \log_2(n/p)) t_{\text{compare}}}{2p \left( t_{\text{start up}} + \frac{n}{p} t_{\text{data}} \right)}$$

where  $t_{\text{compare}}$  is the cost for one comparison.

We view this ratio for  $n \gg p$ , for fixed  $p$ , so:

$$\frac{n}{p} \log_2 \left( \frac{n}{p} \right) = \frac{n}{p} (\log_2(n) - \log_2(p)) \approx \frac{n}{p} \log_2(n).$$

The ratio then becomes  $\frac{n}{p} \log_2(n) t_{\text{compare}} \gg 2n t_{\text{data}}$ .

Thus  $\log_2(n)$  must be sufficiently high...

# Parallel Sorting Algorithms

- 1 Sorting in C and C++
  - using `qsort` in C
  - using STL `sort` in C++
- 2 Bucket Sort for Distributed Memory
  - bucket sort in parallel
  - communication versus computation
- 3 Quicksort for Shared Memory
  - **partitioning numbers**
  - quicksort with OpenMP

## a recursive algorithm

```
void quicksort ( double *v, int start, int end ) {
    if(start < end) {
        int pivot;
        partition(v,start,end,&pivot);
        quicksort(v,start,pivot-1);
        quicksort(v,pivot+1,end);
    }
}
```

where partition has the prototype:

```
void partition
( double *v, int lower, int upper, int *pivot );
/* precondition: upper - lower > 0
 * takes v[lower] as pivot and interchanges elements:
 * v[i] <= v[pivot] for all i < pivot, and
 * v[i] > v[pivot] for all i > pivot,
 * where lower <= pivot <= upper. */
```

## a partition function

```
void partition
( double *v, int lower, int upper, int *pivot )
{
    double x = v[lower];
    int up = lower+1;    /* index will go up */
    int down = upper;    /* index will go down */
    while(up < down)
    {
        while((up < down) && (v[up] <= x)) up++;
        while((up < down) && (v[down] > x)) down--;
        if(up == down) break;
        double tmp = v[up];
        v[up] = v[down]; v[down] = tmp;
    }
    if(v[up] > x) up--;
    v[lower] = v[up]; v[up] = x;
    *pivot = up;
}
```

## partition and qsort in main()

```
int lower = 0;
int upper = n-1;
int pivot = 0;
if(n > 1) partition(v,lower,upper,&pivot);

if(pivot != 0)
    qsort((void*)v,(size_t)pivot,
          sizeof(double),compare);

if(pivot != n)
    qsort((void*)&v[pivot+1],(size_t)(n-pivot-1),
          sizeof(double),compare);
```

# Parallel Sorting Algorithms

- 1 Sorting in C and C++
  - using `qsort` in C
  - using STL `sort` in C++
- 2 Bucket Sort for Distributed Memory
  - bucket sort in parallel
  - communication versus computation
- 3 Quicksort for Shared Memory
  - partitioning numbers
  - quicksort with OpenMP

## a parallel region in `main()`

```
omp_set_num_threads(2);
#pragma omp parallel
{
    if(pivot != 0)
        qsort((void*)v, (size_t)pivot,
              sizeof(double), compare);
    if(pivot != n)
        qsort((void*)&v[pivot+1], (size_t)(n-pivot-1),
              sizeof(double), compare);
}
```

## on dual core Mac OS X at 2.26 Ghz

```
$ time /tmp/time_qsort 10000000 0  
time elapsed : 4.0575 seconds
```

```
real    0m4.299s  
user    0m4.229s  
sys     0m0.068s
```

```
$ time /tmp/part_qsort_omp 10000000 0
```

```
pivot = 4721964
```

```
-> sorting the first half : 4721964 numbers
```

```
-> sorting the second half : 5278035 numbers
```

```
real    0m3.794s  
user    0m7.117s  
sys     0m0.066s
```

Speed up:  $4.299/3.794 = 1.133$ , or 13.3% faster with one extra core.

## recommended reading

- Edgar Solomonik and Laxmikant V. Kale: **Highly Scalable Parallel Sorting**. In the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
- Mirko Rahn, Peter Sanders, and Johannes Singler: **Scalable Distributed-Memory External Sorting**. In the proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), pages 685-688, IEEE, 2010.
- Davide Pasetto and Albert Akhriev: **A Comparative Study of Parallel Sort Algorithms**. In SPLASH'11, the proceedings of the ACM international conference companion on object oriented programming systems languages and applications, pages 203-204, ACM 2011.

# Summary + Exercises

In the book of Wilkinson and Allen, bucket sort is described in §4.2.1 and chapter 10 is entirely devoted to sorting algorithms.

## Exercises:

- 1 Consider the fan out scatter and fan in gather operations and investigate how these operations will reduce the communication cost and improve the computation/communication ratio in bucket sort of  $n$  numbers on  $p$  processors.
- 2 Instead of OpenMP, use Pthreads to run Quicksort on two cores.
- 3 Instead of OpenMP, use the Intel Threading Building Blocks to run Quicksort on two cores.

Homework will be collected on Friday 10 February at noon.

Bring your answers to the problems of lectures 9, 10, and 11 to class.

You may work in pairs. Please hand in only one copy per pair.