高性能 PA1 : Odd-Even Sort

计 83 李天勤 2018080106

实验目的

Become familiar with MPI through the implementation of odd-even sort

实验实现

The description of the code is directly commented.

代码

```
void Worker::sort() {
  /** Your code ... */
  // you can use variables in class Worker: n, nprocs, rank, block_len, data,
last_rank, out_of_range

  // directly skip if out of range
  if (out_of_range) return;

  // directly sort if only one process
  if (nprocs == 1) {
    std::sort(data, data + block_len);
    return;
  }

  // algorithm to qsort the data in current process
  std::sort(data, data + block_len);

  // merge once if only two processes
  if (nprocs == 2) {
    int rank_one_block_len = 1;
    if (rank == 1) MPI_Send(&block_len, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    else if (rank == 0) MPI_Recv(&rank_one_block_len, 1, MPI_INT, 1, 0,
MPI_COMM_WORLD, NULL);

    float* rank_one_proc = new float[block_len + rank_one_block_len];

    if (rank == 1) {
      MPI_Send(&data[0], block_len, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&data[0], block_len, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, NULL);
    }
    else if (rank == 0) MPI_Recv(&rank_one_proc[block_len], rank_one_block_len,
MPI_FLOAT, 1, 0, MPI_COMM_WORLD, NULL);

    if (rank == 0) {
        // printf("merging ...\n");
        std::merge(data, data + block_len, rank_one_proc + block_len,
rank_one_proc + block_len + rank_one_block_len, rank_one_proc);
        std::copy(rank_one_proc, rank_one_proc + block_len, data);
        // for(size_t i = 0; i < block_len + rank_one_block_len; i++) printf("%f
", rank_one_proc[i]);
```

```
        MPI_Send(&rank_one_proc[block_len], rank_one_block_len, MPI_FLOAT, 1, 0,
MPI_COMM_WORLD);
    }
    return;
  }

  // check_proc inter-process on n > 2 proceess
  int block_len_next = 1;
  if (rank != 0) MPI_Send(&block_len, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
  if (!last_rank) MPI_Recv(&block_len_next, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD, NULL);

  // used to store merged array (current process data and adjacent process data)
  float* merged_data = new float[block_len + block_len_next];

  bool is_finished = 0;
  bool parity = 0;

  // track whether a process has exchanged
  int check_proc = 0, check_proc_before = 0, check_proc_after = 0;
  int check_count = 0;

  while (!is_finished) {
    check_proc = 0;

    // parity = 0, even-odd-sort, parity = 1, odd-even-sort
    if (rank % 2 == parity) {
      if (!last_rank) {
        MPI_Recv(&merged_data[block_len], block_len_next, MPI_FLOAT, rank + 1,
0, MPI_COMM_WORLD, NULL);

        // if last element of left process data array is <= first element of
right process data array, then sorted
        if (data[block_len - 1] <= merged_data[block_len]) check_proc = 0;
        // if not, merge sort and
        else {
          check_proc = 1;
          // merge sort + split data
          std::merge(data, data + block_len, merged_data + block_len,
merged_data + block_len + block_len_next, merged_data);
          std::copy(merged_data, merged_data + block_len, data);
        }

        MPI_Send(&merged_data[block_len], block_len_next, MPI_FLOAT, rank + 1,
0, MPI_COMM_WORLD);
      }
    }
    else {
      if (rank != 0) {
        MPI_Send(&data[0], block_len, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&data[0], block_len, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD,
NULL);
      }
    }

    // the two ends of the processes pass "information" to the middle,
    // and then the middle process judges whether the sorting has finished,
ending the sort
```

```
    // information being passed is whether or not adjacent process has exchaged
any data
    if (rank < nprocs / 2) {
      check_proc_before = 0;

      // check_proc if current process exchanged or previous process exchanged,
pass result to next process
      if (rank == 0) check_proc_before = 0;
      else MPI_Recv(&check_proc_before, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
NULL);
      check_proc = check_proc || check_proc_before;
      MPI_Send(&check_proc, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);

      // track and pass check_count to know when finished
      MPI_Recv(&check_count, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, NULL);
      if (rank > 0) MPI_Send(&check_count, 1, MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);
    }
    // same as above, just in reverse
    else if (rank > nprocs / 2) {
      check_proc_after = 0;
      if (last_rank) check_proc_after = 0;
      else MPI_Recv(&check_proc_after, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD,
NULL);
      check_proc = check_proc || check_proc_after;
      MPI_Send(&check_proc, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);


      MPI_Recv(&check_count, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, NULL);
      if (!last_rank) MPI_Send(&check_count, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
    }
    // pass information to center process
    // if no exchanges has occured on either side, sorting can stop
    else if (rank == nprocs / 2) {
      MPI_Recv(&check_proc_before, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
NULL);
      MPI_Recv(&check_proc_after, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD,
NULL);

      if (check_proc == 0 && check_proc_before == 0 && check_proc_after == 0) {
        check_count++;
        // printf("rank = %d, check_count = %d\n", rank, check_count);
      }
      else check_count = 0;

      // send check_count value throughout all process starting from process
neighboring center process
      MPI_Send(&check_count, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
      MPI_Send(&check_count, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
    }

    // if process recieve check_count 2 (no more exchanging between process )
    // from middle process, sorting is finished
    if (check_count == 2) break;

    // switch from odd even to even odd
    parity = !parity;
```

```
  }
  delete[] merged_data;
}
```

优化

The realization of sorting and merging the data of two adjacent processes was quite simple. I simply utilized `std:algorithm` 's sorting algorithms `std::merge` and `std::sort`. This saved the hassle of implementing our own algorithm which may not be as optimized. I first implemented the two base cases which were when `nprocs == 1` and `nprocs = 2`. The implementation for when `nprocs = n`, when $n > 2$, is quite similar to the model of the `nprocs = 2` , but had to consider which sort stage it was (even-odd or odd-even) and whether or not that process had sorted/exchanged with its neigboring process. This a key aspect of this program, as we have to implement a method to know when the processes need to stop, which is when all process no long exchange any data. The most basic implementation is to choose one process to track whether or not all the other process have exhanged data (through a merge sort). And when that process discovers that all other procces no longer are exchanging data, that process can pass a parameter `check_count` that is recieved by all other process, effectively stopping the all the process and thus the program. I chose the middle process `rank = nprocs/2` to process whether or not the program should stop the sort. The two ends of the processes (rank = 0, rank = n) pass information to the middle.

实验结果

1 x 1

```
2018080106@conv0:~/PA1$ srun -n 1 -N 1 ./odd_even_sort 100000000
data/100000000.dat
Process 0 handles [0, 100000000)
Rank 0: pass
Execution time of function sort is 12436.116000 ms.
Process 0: finalize
```

1 x 2

```
2018080106@conv0:~/PA1$ srun -n 2 -N 1 ./odd_even_sort 100000000
data/100000000.dat
Process 0 handles [0, 50000000)
Rank 0: pass
Execution time of function sort is 6763.453000 ms.
Process 0: finalize
Process 1 handles [50000000, 100000000)
Rank 1: pass
Process 1: finaliz
```

1 x 4

```
2018080106@conv0:~/PA1$ srun -n 4 -N 1 ./odd_even_sort 100000000
data/100000000.dat
Process 2 handles [50000000, 75000000)
```

```
Rank 2: pass
Process 2: finalize
Process 3 handles [75000000, 100000000)
Rank 3: pass
Process 3: finalize
Process 0 handles [0, 25000000)
Rank 0: pass
Execution time of function sort is 3931.304000 ms.
Process 0: finalize
Process 1 handles [25000000, 50000000)
Rank 1: pass
Process 1: finalize
```

1 x 8

```
2018080106@conv0:~/PA1$ srun -n 8 -N 1 ./odd_even_sort 100000000
data/100000000.dat
...
Process 0 handles [0, 12500000)
Rank 0: pass
Execution time of function sort is 2451.401000 ms.
Process 0: finalize
...
```

1 x 16

```
2018080106@conv0:~/PA1$ srun -n 16 -N 1 ./odd_even_sort 100000000
data/100000000.dat
...
Process 0 handles [0, 6250000)
Rank 0: pass
Execution time of function sort is 1684.467000 ms.
Process 0: finalize
...
```

2 x 16

```
2018080106@conv0:~/PA1$ srun -n 16 -N 2 ./odd_even_sort 100000000
data/100000000.dat
...
Process 0 handles [0, 6250000)
Rank 0: pass
Execution time of function sort is 1626.972000 ms.
Process 0: finalize
...
```