PARALLEL AND DISTRIBUTED PROGRAMMING

ASSIGNMENT 2

Group 50

Jiahao LU, You WU, Zonghao LU

14 May 2019

## Problem Formulation

Given a sequence of n integers, the task is to implement the Parallel Quicksort algorithm using C and MPI and evaluate the performance.

The input file contains n + 1 numbers. The first number is n (that is, the number of elements to sort) and the n subsequent numbers are the actual number sequence.

## Solution Method

1. **Algorithm implementation**

The Parallel Quick-sort algorithm is implemented as follows:

1. Divide the data into p equal parts, one per process.
2. Sort the data locally for each process.
3. Perform global sort.
   1. Select pivot element within each process set.
   2. Locally in each process, divide the data into two sets according to the pivot (smaller or larger).
   3. Split the processes into two groups and exchange data pairwise between them so that all processes in one group get data less than the pivot and the others get data larger than the pivot.
   4. Merge the two sets of numbers in each process into one sorted list.
4. Repeat 3.1 - 3.4 recursively for each half until each group consists of one single process.

The algorithm converges in steps.

For stability reasons, the built-in function qsort() was used. In the implementation detail, the recursion is converted to iteration by keeping tracking of the variable group\_size. The iteration is ended when group\_size reaches 1.

1. **Pivot strategies**

Three pivot strategies are implemented and can be chosen by specifying the third input argument as 1, 2, or 3:

1. Select the median in one processor in each group of processors.
2. Select the median of all medians in each processor group.
3. Select the mean value of all medians in each processor group.

## Experiments

1. **Testing environment**

Host: rackham.uppmax.uu.se

1. **Testing method**

The strong scalability is tested for each input file, running from 1 to 16 processors with the number of processors being 2k. The output time is measured in seconds by MPI\_Wtime() function on the master processor. Only the parallelised computation time is measured.

The weak scalability is tested when number of processors are increased from 1 to 8 with the number of processors being 2k, while number of intervals are increased simultaneously in a way that the interval load per processors is kept to 125000000.

## Results and Discussion

The tested strong and weak stabilities are shown as follows (each element in the tables represents “Time(s)/Speedup”).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input125000000 | 1 | 2 | 4 | 8 | 16 |
| Pivot Strategy-1 | 26.00/1.000 | 12.91/2.014 | 6.71/3.875 | 3.77/6.897 | 2.20/11.818 |
| Pivot Strategy-2 | 26.01/1.000 | 12.89/2.018 | 7.45/3.491 | 4.05/6.422 | 2.30/11.309 |
| Pivot Strategy-3 | 25.43/1.000 | 13.21/1.925 | 7.02/3.622 | 3.55/7.163 | 2.17/11.719 |

图片包含 文字, 地图

描述已自动生成

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input250000000 | 1 | 2 | 4 | 8 | 16 |
| Pivot Strategy-1 | 52.65/1.000 | 27.32/1.924 | 13.40/3.922 | 8.23/6.386 | 4.56/11.526 |
| Pivot Strategy-2 | 53.21/1.000 | 27.95/1.904 | 14.54/3.660 | 8.06/6.602 | 4.62/11.517 |
| Pivot Strategy-3 | 52.58/1.000 | 27.72/1.897 | 14.52/3.621 | 7.83/6.715 | 4.61/11.406 |

图片包含 文字, 地图

描述已自动生成

Since only the parallelised computation time is measured, the theoretical speedup for fixed problem size should equal to the number of processors. As the figure shows above, the measured speedup for fixed problem size curve is approximately linear. Therefore, it is reasonable to assess that the strong scalability is well achieved.

Later on, the results of weak scalability evaluation are presented below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 125000000/PE | 1 | 2 | 4 | 8 |
| Pivot Strategy-1 | 25.56/1.000 | 27.03/0.946 | 30.18/0.847 | 33.62/0.760 |
| Pivot Strategy-2 | 24.73/1.000 | 29.43/0/840 | 31.68/0.781 | 33.41/0.740 |
| Pivot Strategy-3 | 27.59/1.000 | 28.20/0.978 | 29.82/0.925 | 33.59/0.821 |

图片包含 文字, 地图

描述已自动生成

The ideal speedup for scaled problem size should be 1 since the load per individual processor is kept large enough and approximately constant. That the curve moves further from the ideal line as the number of cores increases is because of the overhead and the time spent on message passing among processors. Besides, it is not absolute that each processor is under the same workload since the load balance will be broken once the amount of received is not equal to the amount of sent. These are also the reasons why ideal speedup for fixed problem size cannot be achieved.

Theoretically, it is the most troublesome for quick sort when the sequence is in descending order, especially when using a bad pivot choosing strategy. The ideal speedup is supposed to be harder to achieve. Below, a backwards125\* file is tested to verify the hypothesis.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Backwards125000000 | 1 | 2 | 4 | 8 | 16 |
| Pivot Strategy-1 | 8.61/1.000 | 5.35/1.609 | 3.10/2.777 | 1.86/4.629 | 1.63/5.282 |
| Pivot Strategy-2 | 8.03/1.000 | 5.43/1.479 | 3.22/2.494 | 1.94/4.139 | 1.19/6.748 |
| Pivot Strategy-3 | 9.26/1.000 | 4.93/1.878 | 2.91/3.182 | 1.67/5.545 | 1.19/7.781 |

图片包含 文字, 地图

描述已自动生成

It is a surprise that input file with descending order is a lot faster than with random order. It can be attributed to the -O3 optimisation flag which probably much accelerated the serial quicksort. But the speedup is worse than with the random order as expected, and we guess the descending order of numbers aggravates the disbalance of the workload in different processor.

Additionally, it can be seen from the figures that, the pivot strategy 3 achieved the best result in almost all of the testing. The reason is that the pivot strategy 3 selects the mean value of all medians in each processor group thus is more random and achieves better load balance.