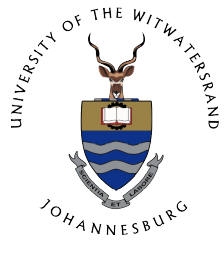

University of the Witwatersrand

COMS4022

High Performance Computing and Scientific Data Management

Parallel Quicksort

Amrit Purshotam

384256

Lecturer:
Prof Ekow Otoo

10 April 2012

Abstract

Quicksort is a sorting algorithm that makes O($n$log$n$) comparisons to sort n items and is generally considered to be the fastest. Due to its 'divide and conquer' nature, quicksort can be parallelised and two slightly different algorithmic approaches were used to implement it: MPI and a hybrid of MPI and OpenMP. Performance analyses were performed on each of the implementations as well as on the sequential quicksort. The MPI only implementation was alot faster than the hybrid of MPI and OpenMP while it was about equal to the sequential implementation, but this was attributed to the fact that large enough lists were not used.

Contents

1 Introduction

Quicksort is a sorting algorithm that makes O($n$log$n$) comparisons to sort n items and is generally considered to be faster than other O($n$log$n$) sorting algorithms. Due to its 'divide and conquer' nature, quicksort can be parallelised. Two slightly different algorithmic approaches i.e. MPI and a hybrid of MPI and OpenMP will be used to implement the parallel quicksort. The purpose of this paper is to compare the two approaches by conducting performance analyses and illustrating them with speed-up and scalability plots.

2 Material

2.1 Hardware and Software

An Intel(R) Core(TM) i5 CPU with 4GB of RAM running Ubuntu 10.10 (*Maverick Meerkat*) was used to conduct the performance analyses. The programming language of choice was C with OpenMP and MPI libraries for parallelism (discussed below).

2.2 OpenMP and MPI

OpenMP (or Open specifications for Multi Processing) is an API for explicit multi-threaded and shared memory parallelism (Otoo [2012b]). MPI (or the Message Passing Interface) is a specification for the developers and users of message passing libraries. Each process within this platform has its own exclusive address space and all data has to be explicitly sent and received piece-meal (Otoo [2012a]). As stated above, the parallel quicksort

will be implemented in two ways: only MPI and also in a hybrid of MPI and OpenMP.

3 Theory

3.1 Sequential Quicksort

Sequential quicksort first chooses a pivot, usually the left-most or right-most element in the list, and partitions the list into two sublists with one sublist less than or equal to the pivot and the other greater than the pivot. The algorithm is then run recursively on each sublist until eventually the the entire list is sorted.

3.2 Hypercubes

Parallel algorithms have their own natural communication structure when executed on a distributed-memory computer. If the logical or virtual topology matches the physical topology of the distributed-memory computer, then performance is enhanced. Thus the virtual topology being used to implement the parallel quicksort is a 2-dimensional hypercube. A d-dimensional hypercube consists of $n = 2^d$ processors where two processors are connected if their Hamming distance is 1 (each processor has a number whose binary representation has d digits) [USC].

3.3 Hypercube Quicksort

Let n be the number of elements to be sorted and $p = 2^d$ be the number of processors in a d-dimensional hypercube. Each processor is assigned a block of n/p elements.

The algorithm starts by selecting a common pivot value, which is broadcast to all processors. Each processor partitions its local elements into two blocks, one with elements smaller than the pivot, and the other with elements larger than the pivot. Then the processors connected along the d-th communication link exchange blocks: Each processor with a 0 in the d-th bit retains the smaller elements, and each processor with a 1 in the d-th bit retains the larger elements. After this step, each processor in the (d-1)-dimensional hypercube whose d-th bit is 0 has elements smaller than the pivot, and each processor in the other (d-1)-dimensional hypercube has elements larger than the pivot. This procedure is performed recursively. After d such splits, the sequence is sorted with respect to the global ordering imposed on the processors. Then each processor sorts its local elements by using a sequential quicksort [USC]. See Appendix A for the algorithm in pseudocode.

3.4 Pivot Selection

Pivot selection in quicksort is critical and if not done properly can cause worst case behaviour on already sorted arrays. Two pivot selection methods were explored, calculating the mean as the pivot, or by using a median of three i.e. the median of the first, middle and last element in the list is chosen as the pivot. Each method has its own advantages and disadvantages, the mean

is more likely to always split the list into two equal halves but at a cost of being more computationally expensive. The median of three, however, is alot less expensive computationally expensive but may not always give a pivot that is likely split the list into two equal halves. The analysis was performed using the mean as the pivot as trying to keep the list sizes roughly the same size was deemed more important as performance will degrade poorly if a bad pivot was chosen.

4 Results

MPI Times

Size	Sequential	2D	3D	4D
1000	0.00027	0.000714	0.007751	0.03041
1500	0.000415	0.000841	0.006343	0.032755
2000	0.000563	0.000927	0.008035	0.0374
2500	0.00072	0.001015	0.007322	0.038647
3000	0.000882	0.00117	0.008502	0.040479
3500	0.001031	0.000979	0.008571	0.037317
4000	0.001209	0.001382	0.00885	0.038157
4500	0.00137	0.001452	0.008932	0.041003
5000	0.001549	0.001493	0.009136	0.039796
5500	0.001693	0.001478	0.010095	0.042775
6000	0.001869	0.001508	0.010237	0.044914
6500	0.002055	0.001948	0.010194	0.04107
7000	0.002226	0.002041	0.010172	0.042433

Figure 4.1

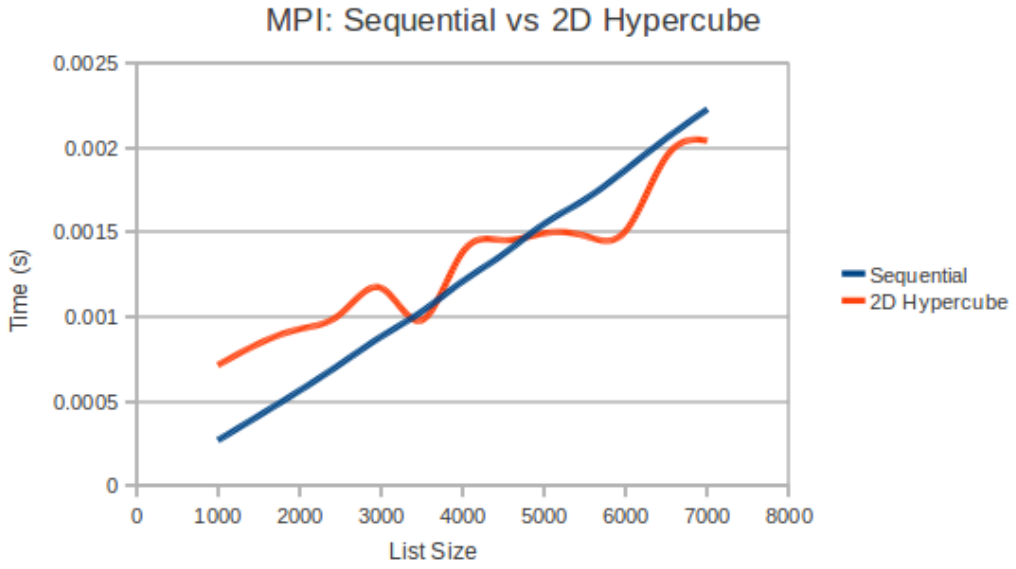

Figure 4.2

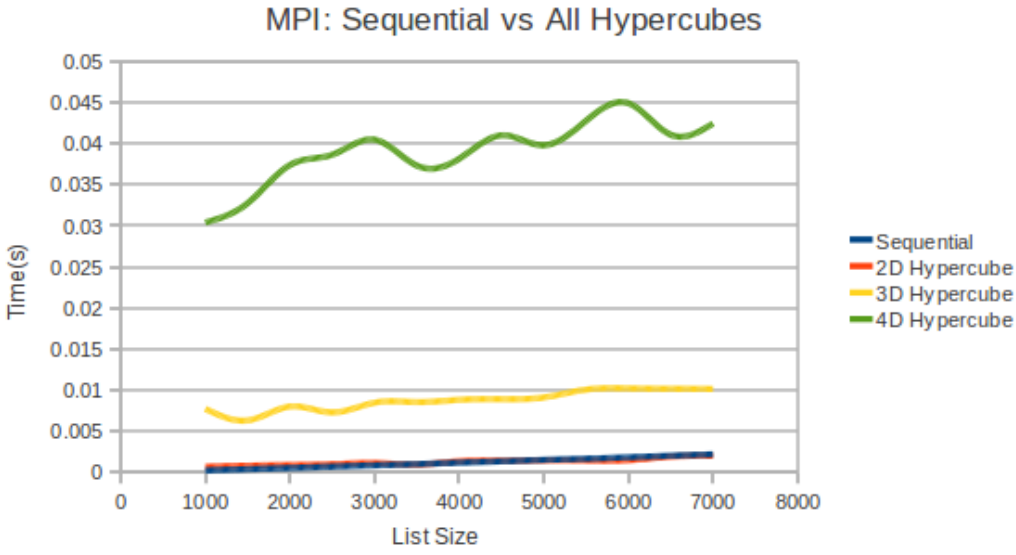

Figure 4.3

MPI and OpenMP Times

Size	Sequential	2D Hypercube	3D Hypercube	4D Hypercube
1000	0.00027	0.054941	0.109559	0.112452
1500	0.000415	0.048694	0.10593	0.125439
2000	0.000563	0.054302	0.100981	0.133252
2500	0.00072	0.057606	0.074321	0.143643
3000	0.000882	0.057872	0.054154	0.133563
3500	0.001031	0.057634	0.063245	0.143462
4000	0.001209	0.083566	0.075104	0.151108
4500	0.00137	0.047545	0.082345	0.133201
5000	0.001549	0.090075	0.088085	0.127841
5500	0.001693	0.083654	0.085032	0.142352
6000	0.001869	0.083543	0.081725	0.124809
6500	0.002055	0.076542	0.091234	0.153496
7000	0.002226	0.105423	0.094592	0.162232

Figure 4.4

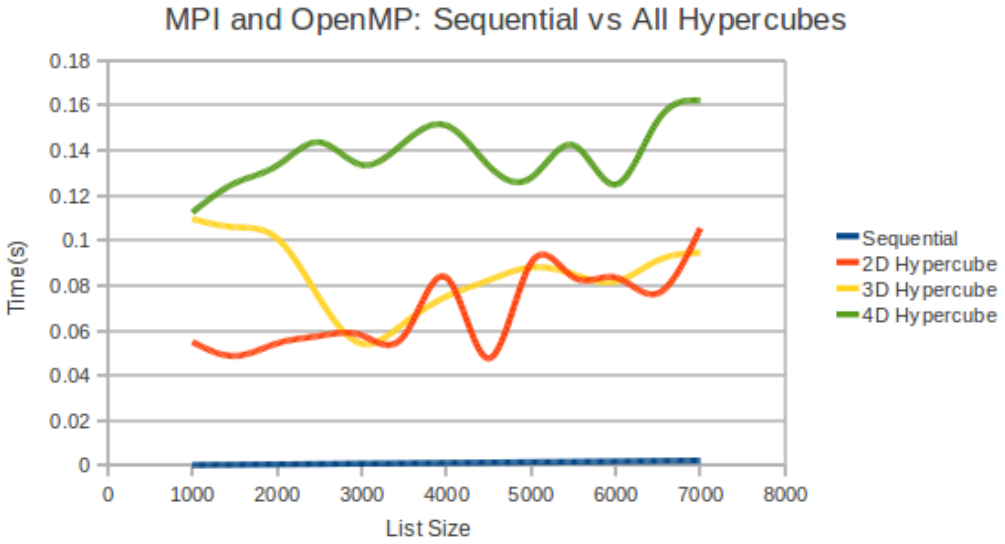

Figure 4.5

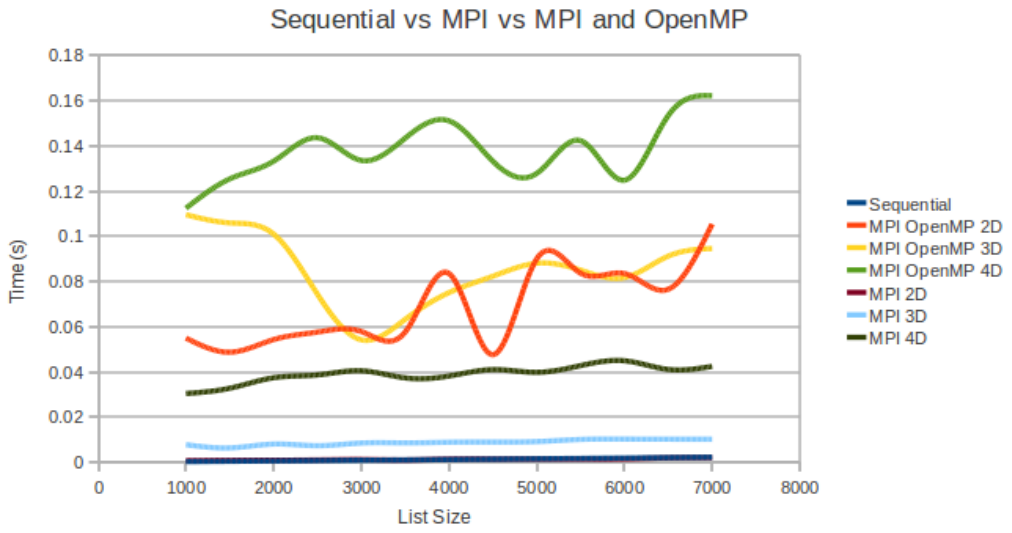

Figure 4.6

5 Discussion

5.1 Speedup and Scalability

Due to only having access to a quad-core CPU, performing speedup and scalability plots were not possible (as more processors would then be needed), thus a 2-dimensional hypercube was used instead of a 6-dimensional one. The performance analyses was then performed by incrementally increasing the list sizes and comparing how long it took to perform the quicksort.

5.2 Problem with MPI Send

Analysis on lists of sizes 4M to 10M could not be performed since the MPI Send function was not behaving as it should for bigger lists. In my code, the MPI Send was used to send the sublists (an array of integers) from one

process to its partner process. This would work perfectly when the list size was small but as soon as the list size increased (i.e. more elements had to be sent), the MPI Send would just block. What stumped me, however, was that a list of size, say 120000, would work perfectly on one computer but would not work on another computer. This erratic behaviour can only lead me to hypothesize that the MPI Send was somehow machine and operating system dependent, and that somehow determined the maximum size of an array it can send. Other factors such as the pivot selection method (i.e. mean or median of three), range of the numbers, and dimension of the hypercube also affected the maximum size of the list that can be sorted. It is for these reasons then that the analysis was performed on lists of sizes 1000 to 7000.

5.3 Performance Analysis

5.3.1 MPI

The data indicates that the sequential quicksort performs at roughly the same speed as the 2D hypercube quicksort (see Figure 4.2) with the 2D hypercube quicksort only slightly edging it out as the list size increased. However the parallel quicksort has alot more (fixed) computational overhead before any actual sorting is done, therefore it can be hypothesized that the parallel quicksort will perform alot better than the sequential quicksort for much larger list sizes. The 3D and 4D hypercube quicksort performed poorly when compared to the sequential and 2D hypercube quicksort. This can be attributed to the fact that more processes than processors were running and so the communication overhead also vastly increased resulting in much higher

run times. It can be hypothesized here that if the 3D and 4D hypercube quicksorts were run on CPUs with 8 cores and 16 cores respectively (thereby matching the virtual topology with the physical), performance will greatly improve.

5.3.2 MPI and OpenMP

OpenMP was used to parallelise the part of the code where all the original numbers are randomly generated and sent to each node. Figures 4.5 and 4.6 seem to indicate that this hybrid of MPI and OpenMP is alot slower than the sequential and MPI only implementation. This goes against what one would normally expect as in theory it should have been a bit faster. As expected, however, the 2D hyrid implementation is faster than the 3D implementation which in turn was faster than the 4D implementation (see Figure 4.5).

6 Compiling and Running Instructions

```
gcc quicksort.c -o quicksort
./quicksort


mpicc hcqs1.c -o hcqs1
mpicc -n x ./hcqs1


mpicc hcqs2.c -o hcqs2 -fopenmp -lgomp
mpicc -n x ./hcqs2
```

Note x is the number of processes needed and is dependent on the dimension used in the program so if d = 2 then x = 4 (i.e. 2^d)

7 Conclusion

The raw data indicates that sequential quicksort was the quickest, however, it also hints that if the lists were alot larger (to make the fixed computational overhead seem small) and if the physical topology matched the virtual topology (to reduce the communication overhead) then the MPI only version would really outperform the other two implementations. In theory, the MPI and OpenMP hybrid should have been quickest but was in fact the slowest which is puzzling.

A Algorithm

```
bitvalue = 2^(dimension-1)
mask = 2^(dimension)-1
for L = dimension down to 1
    if myid AND mask = 0 then
        choose a pivot for the L-dimensional subcube
    broadcast the pivot from the master to the other members of the subcube
    partition list[0:nelement-1] into 2 sublists such that
    list[0:j] <= pivot < list[j+1:nelement-1]
    partner = myid XOR bitvalue
    if myid AND bitvalue = 0 then
        send right sublist list[j+1:nelement-1] to partner
        receive left sublist list[0:j] of partner
        append the received list to my left list
    else
        send left sublist list[0:j] to partner
        receive right sublist list[j+1:nelement-1] of partner
        append the received list to my right list
    nelement = nelement - nsend + nreceive
    mask = mask XOR bitvalue
    bitvalue = bitvalue/2
sequential quicksort to list[0:nelement-1]
```

[USC]

References

[Otoo 2012a] Ekow Otoo. *Parallel Programming Using MPI*, 2012.

[Otoo 2012b] Ekow Otoo. *Programming Shared-Memory Platforms Using Open-MP*, 2012.

[USC] *Divide and Conquer Parallelization Paradigm.* `cacs.usc.edu/education/cs653/02-3DC.pdf`. Accessed 4 April 2012.