On the Scalability of Parallel Quicksort: A Case Study on Distributed vs. Shared-Memory Models

David Paulius  
Department of Computer Science and Engineering  
University of South Florida  
Tampa, Florida, USA  
davidpaulius@mail.usf.edu

Marc Boumedine  
Computational and Computer Sciences Department  
College of Science and Mathematics  
University of the Virgin Islands  
St. Thomas, U.S.V.I  
mboumed@uvi.edu

ABSTRACT/POSTER PAPER — The increasing availability of parallel systems affords undergraduates opportunities to experience firsthand the potential benefit and pitfalls of parallel programming. However, a clear understanding of the underlying architecture is critical to achieve the anticipated speed up of serial programs. To this end, analyzing performance metrics is essential for tuning parallel implementations and determining what improvement can be expected by scaling out or increasing the number of processes [1-4]. Without considering important aspects such locality, load balancing and communication overheads associated with each available parallel system, students are having a hard time obtaining the intended performance and determining which approach is suitable to solving the problem at hand. Finding the best mapping between the parallel implementation and the parallel platform available is hard to achieve without clear guidance or practical experiences. Given some legacy code, a first approach consists of looking at parallel programming patterns such as divide-and-conquer or decomposition techniques. This pattern has been used widely to design a number of efficient algorithms for a variety of applications [5]. Students are already familiar with the approach of recursively partitioning a problem into smaller sub-problems. Identification of sub-problems which can be solved independently is an intuitive choice for achieving speed up. Even if this is not always the case, this approach allows students to quickly implement embarrassingly parallel programs using message passing and shared-address space programming paradigms. Ultimately, performance metrics are collected and contrasted with the serial and parallel runtime implementations. Sources of overhead affecting the performances of the algorithms are examined and discussed. This practical approach can be repeated on various classical problems (matrix multiplication, sorting, numerical integration and the gravitational N-body) and extended to new problems by applying various parallel programming patterns.

This case study walks through the steps of solving the problem of speeding up sorting serial problems from legacy code for both shared memory and distributed memory models on a small system multi-core cluster called Bucc. Bucc consists of a head node with 2 Intel Westmere, 2.40GHz, six-core, processors and five twin compute nodes, each with two Intel 2.40GHz six-core processors. It is configured with 2 GB of RAM per core. The interconnects are InfiniBand 40Gbps.

This study focuses specifically on the sorting task for both the distributed and shared-memory models on the Bucc system. Sorting is an recurrent critical problem for organizing and processing efficiently information in many fields such as combinatorial searching, optimization, simulation, information retrieval, scientific computing etc. Sorting is also used as a core utility in many libraries. Because of its essential role computer scientists have devoted many studies on sorting techniques and have been focusing on finding the fastest serial and parallel implementations. The quicksort algorithm is one of the most popular algorithms known for its time complexity of $O(N \log N)$. As a divide-and-conquer algorithm, it uses the notion of pivoting to divide the data set into two halves or subsets that are recursively solved; there are several ways of determining the pivot such as finding the median, selecting the middle value, or simply selecting a random value in the data set. Selecting the pivot is important as it ultimately affects the time taken to sort and merge all the results together. The quicksort algorithm has the worst case of being $O(N^2)$, which can render it as slow as the selection and bubble sorting algorithms. Due to its divide-and-conquer nature, intuitive parallel implementations are usually implemented using message passing interface libraries (MPI) and shared-address space libraries (OpenMP). Other hybrid programming models are also considered but not examined in this study.

The naive parallel formulation of the serial quicksort consists of sorting a global array $A$ of $n$ element on $p$ processes. Each process is given a sub-array which is a sequence of $n/p$ items. The sub-array is partitioned recursively around a pivot using a compare and split routine. The algorithm terminates when the sub-array can not be partitioned anymore. Fig. 1 shows the serial quicksort runtime of Bucc. The performance and scalability of parallel programs (MPI and OpenMP) are discussed. OpenMP quicksort spawns $t$ threads each of them handle a partition of the array. Each thread uses the serial $qsort()$ from the C libraries [6]. Once all partitions have been sorted, the sub-arrays are then merged together. The
runtimes were recorded during various experiments by varying the size of the data size and the number of processes (see Figs. 2-5). The array sizes were increased incrementally (1000 x $2^N$ items, where $N=0,...,12$). The MPI experiments shown on Figs. 2-3 were derived with $p$ processes ($p=1,...,128$). Figs. 4-5 show OpenMP implementation results by increasing the number of threads from 1 to 512. The data sets were randomly generated with integers using the C’s `drand()` function for better generalization purposes. Runtimes were recorded and plotted on Figs. 1-5. Common performance metrics such as speedup and efficiency were computed on Bucc. Speedup is the performance gain of parallel processing versus sequential processing. It is defined as [7-8]:

$$S(p) = \frac{T(N,1)}{T(N,p)}$$

where $T(n,1)$ is the runtime on one processor executed on size $N$ input, $T(n,p)$ is the runtime on $p$ processors executed on size $N$ input.

Efficiency of a parallel program is a measure of how much of available processing power is being used.

$$E(p) = \frac{S(p)}{p}$$

where $S(p)$ is the parallel runtime on $p$ processors.

This metric provides an accurate measure of the true efficiency of a parallel program compared to CPU usage (redundant calculations and idle times are included) [1-3].

Fig. 6 shows speedup results for the MPI and OpenMP implementations over the serial quicksort. Overall, the results for the OpenMP implementation show significant performance improvement on over the MPI implementation on relatively large problem size ($N=4,096,000$), where 16 threads yield a ~700% time performance increase (see Fig. 7). Both parallel implementations improve performance when item sizes are greater than 32,000. However, the naive parallel implementations have no gain for small problem sizes ($N < 32,000$) on the Bucc architecture. Figs 6-7 show that the current quicksort implementations are not scalable on Bucc. For some experiments, the parallel versions solve the problem with a superlinear speed up for (OpenMP with 8, 16 and 32 threads). On the other hand, the execution with 2 and 4 threads results in a sublinear speedup (see Fig. 7). Based on these results, it is clear that the experiments conducted with the current parallel implementations must be re-examined in order to identify sources of parallel overhead such as load imbalancing, intranode and internode communication overhead, the partitioning of the array and the size of the cache available at each compute node. The pivot selection is also critical in avoiding idle processes/threads and must be considered in revised versions.

This case study highlighted the challenges undergraduate students are facing during the design and development of parallel applications. Students were tasked to parallelize the serial quicksort algorithm on Bucc cluster. Naive parallel algorithms were implemented using OpenMP and MPI. These implementations were compared in terms of speed up, efficiency and scalability. Results have shown that the parallel naive implementations can be improved significantly by reviewing the designs and implementations as well as taking into account the underlying architecture. In the future we want to combine the benefits of MPI and OpenMP code with a hybrid programming model and examine speedup and scalability metrics on Bucc.
REFERENCES


