Radenski, A. Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs. Proc. PDPTA'11, the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press (H. Arabnia, Ed.), 2011, pp. 367 - 373.

Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs

Atanas Radenski
School of Computational Sciences, Chapman University, Orange, California, USA
radenski@chapman.edu http://www.chapman.edu/~radenski

Abstract – *While merge sort is well-understood in parallel algorithms theory, relatively little is known of how to implement parallel merge sort with mainstream parallel programming platforms, such as OpenMP and MPI, and run it on mainstream SMP-based systems, such as multi-core computers and multi-core clusters. This is misfortunate because merge sort is not only a fast and stable sort algorithm, but it is also an easy to understand and popular representative of the rich class of divide-and-conquer methods; hence better understanding of merge sort parallelization can contribute to better understanding of divide-and-conquer parallelization in general. In this paper, we investigate three parallel merge-sorts: shared memory merge sort that runs on SMP systems with OpenMP; message-passing merge sort that runs on computer clusters with MPI; and combined hybrid merge sort, with both OpenMP and MPI, that runs on clustered SMPs. We have experimented with our parallel merge sorts on a dedicated Rocks SMP cluster and on a virtual SMP luster in the Amazon Elastic Compute Cloud. In our experiments, shared memory merge sort with OpenMP has achieved best speedup. We believe that we are the first ones to concurrently experiment with - and compare – shared memory, message passing, and hybrid merge sort. Our results can help in the parallelization of specific practical merge sort routines and, even more important, in the practical parallelization of other divide-and-conquer algorithms for mainstream SMP-based systems.*

Keywords: Parallel merge sort, OpenMP, MPI, SMP, Cluster computing, cloud computing

1 Introduction

Merge sort is an efficient divide-and-conquer sorting algorithm. Because merge-sort is easier to understand than other useful divide-and-conquer methods, it is often considered to be a typical representative of such methods, and frequently used to introduce the divide-and-conquer approach itself [3, Ch 2].

Intuitively, merge sort operates on an array of n objects as follows: (1) if n > 1, divide the array into two sub-arrays of about half the size each; (2) apply merge sort on each sub-array; (3) merge the two sorted sub-arrays from step 2 into one sorted array. For small arrays, some implementations switch from recursive merge sort to non-recursive methods, such as insertion sort – an approach that is known to improve execution time. (Fig. 1 in Section 2.1 outlines a serial merge sort implementation in C.)

The average complexity of merge sort is O(n log n) [7], the same as quick sort and heap sort. In addition, best-case complexity of merge sort is only O(n), because if the array is already sorted, the merge operation perform only O(n) comparisons; this is better than best case complexity of both quick sort and heap sort. The worst case complexity of merge sort is O(n log n) [7], which is the same as heap sort and better than quick sort. However, classical merge sort uses an additional memory of n elements for its merge operation (the same as quick sort), while heap sort is an in-place method with no additional memory requirements.

The average/best/worst asymptotic complexity of merge sort is at least as good as the corresponding average/best/worst asymptotic complexity of heap sort and quick sort; despite of this, merge sort is often considered to be slower than the other two in practical implementations. On the positive side, merge sort is a stable sort method, in contrast to quick sort and heap sort, which fail to maintain the relative order of equal objects. The practical performance of merge sort is known to improve with recursion removal and cache memory utilization [8].

The focus of this paper is not on efficiency improvements that are specific to merge sort. Instead, we regard recursive merge sort as a typical and well-understood representative of the divide-and-conquer approach. We use merge sort as a test bed to explore parallelization schemes that may possibly apply without significant changes to other divide-and conquer methods.

Merge sort parallelization is well-studied in theory. For example, Cole [2] describes a O(log n) parallel merge sort algorithm for a CRW PRAM (an abstract machine which neglects synchronization and communication), while Cormen et al outline another O(log n) parallel merge sort for abstract comparison networks [3, Ch. 27].

In contrast to theory, little is known of how to implement parallel merge sort on mainstream architectures (such as standalone and clustered Symmetric Multiprocessing Systems, SMPs), by means of mainstream shared memory and

message passing platforms (such as OpenMP [13] and MPI [12]). Our goal in this paper is to provide a better understanding in this direction.

We choose OpenMP to parallelize merge sort on SMPs and MPI to parallelize it on clustered systems. We choose OpenMP to implement shared-memory merge sort on SMPs because (i) OpenMP is standardized and comes ready-to-use with contemporary C/C++ compilers, including compilers that are freely available; (ii) OpenMP is easier to use than various thread libraries because it supports a higher level parallel programming model; (iii) OpenMP is can work on a wider number of shared-memory computers as opposed to other interesting yet less available higher-level frameworks, such as UPC [4] and Orio [9]. We choose MPI to implement message-passing merge sort on computer clusters because (i) MPI is implemented for a broad variety of architectures, including implementations that are freely available; (ii) MPI is well documented; (iii) MPI has grown much more popular than alternative platforms, such as PVM [5]. Finally, our preference for an implementation language is ANSI C because (i) C is fast and available on virtually any platform; (ii) C can be used to implement merge sort versions with both OpenMP and MPI, including a hybrid implementation of parallel merge sort, based on both OpenMP and MPI (see Section 2.3).

In the rest of this paper, we describe parallel merge-sort algorithms with OpenMP and MPI, and evaluate their performance (Section 2); then we offer conclusions (Section 3). Section 2.1 is devoted to a shared memory OpenMP implementation of merge sort, while Section 2.2 delivers a message-passing merge sort with MPI. Section 2.3 is focused on a hybrid parallel sort that combines both OpenMP and MPI. Section 2.4 evaluates and compares the performance of the three parallel merge sorts as measured on a dedicated SMP cluster. In addition, Section 2.5 describes experience with the same parallel merge sorts on AWS, the Amazon cloud computing platform [1] and provides performance evaluation accordingly.

2 Recursive Merge Sort Parallelization and Evaluation

Recursive merge sort is a typical and well-understood divide-and-conquer algorithm (Fig. 1).

```
void mergesort_serial(int a[], int size, int temp[]) {
   if (size < SMALL) { insertion_sort(a, size); return; }
   mergesort_serial(a, size/2, temp);
   mergesort_serial(a + size/2, size - size/2, temp);
   merge(a, size, temp);
}
```

Fig. 1. Serial recursive merge sort in C. It sorts an array *a* using additional array *temp* of the same size as *a*

We design parallel versions of this algorithm not as much for the sake of merge sort parallelization alone, but to also hopefully provide insights into parallelization of divide-and-conquer algorithms in general. This is why we do not to employ parallelization techniques that are (i) too specific for merge sort or (ii) founded on specific functionality of particular parallel computers.

2.1 Shared Memory Merge Sort with OpenMP

The OpenMP API [13] supports, on a variety of platforms, programming of shared memory multiprocessing. With OpenMP, C/C++ and Fortran programmers use a set of compiler directives (pragmas), library routines, and environment variables to specify multi-threaded execution that is implicitly managed by the OpenMP implementation.

OpenMP supports a straightforward conversion of serial recursive merge sort (Fig. 1) into a multi-threaded recursive merge sort (Fig. 2). A *parallel sections* directive calls for enclosed independent sections of code – as defined by nested instances of the *section* directive - to be divided between automatically generated threads (Fig. 2).

By default, the additional array *temp* is shared by all threads. Therefore, the second recursive call from the serial version (Fig. 1) must be modified to provide to each thread a unique part of the shared additional *temp* array (Fig. 2).

```
void mergesort_parallel_omp
      (int a[], int size, int temp[], int threads) {
   if ( threads == 1) { mergesort_serial(a, size, temp); }
  else if (threads > 1) {
     #pragma omp parallel sections
     {
        #pragma omp section
        mergesort_parallel_omp(a, size/2, temp, threads/2);
        #pragma omp section
        mergesort_parallel_omp(a + size/2, size - size/2,
                          temp + size/2, threads - threads/2);
     }
     merge(a, size, temp);
   } // threads > 1
}
```

Fig. 2. Shared memory parallel merge sort with OpenMP (it uses parallel sections to assign recursive calls to threads)

It is possible to further parallelize the OpenMP merge sort by parallelizing the *merge* operation as well. This can be done by a conversion into OpenMP of a platform-specific technique originally developed for the .Net Task Parallel Library [6]. Reportedly, this technique can make parallel merge sort 25% faster than parallel quick sort, probably because the merge operation is easier to parallelize than quick sort's partition operation.

The performance of the above shared memory (with

OpenMP) implementation has been measured (i) on a stand-alone multi-core computer and (ii) on an Amazon AWS's large multi-core instance; performance results are reported in Sections 2.4 and 2.5 correspondingly.

2.2 Message-Passing Merge Sort with MPI

The MPI API [12] supports, on a variety of platforms, programming of message-based communication between processes and is typically used in distributed-memory systems, such as computer clusters. With MPI, programmers in a wide variety of languages use a set of library routines to implement communication and synchronization between processes.

Recall that OpenMP threads are dynamically assigned to parallel sections when the execution reaches such sections. This means that with OpenMP, the tree of recursive merge sort calls is automatically mapped onto threads. In contrast to OpenMP, all MPI processes start at once at the very beginning of program execution, and all processes concurrently execute the same code – the entire program. Consequently, the MPI program must permit each process to recognize its own place and role in the recursion tree. With MPI, processes need to be explicitly programmed to map themselves to nodes in the recursion tree, while with OpenMP, it is OpenMP itself that straightforwardly maps nodes from the recursion tree to threads. This difference makes the task of the MPI programmer more complicated in comparison to the task of the OpenMP programmer.

As MPI processes map themselves to nodes from the recursion tree, they form a virtual *process tree*. Process 0 is at the root of the tree, with the remaining processes appearing as nodes of the tree (Fig. 3). The *root process* splits the data and sends half of it to a *helper process* which sorts the data and returns it to the root process (send operations are visualized as arrows in Fig. 3). The other half of data is retained by the root process for further sorting by using this same procedure (data retention within processes are visualized by dotted lines in Fig. 3). Once sorted, the two halves of data are merged by the root process.

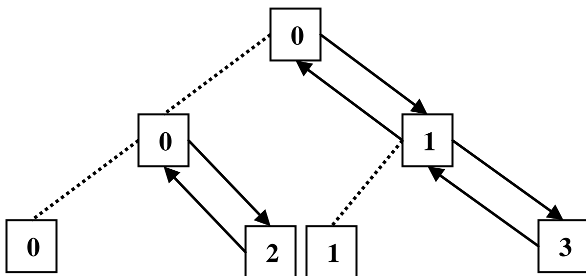

Fig. 3. MPI process tree for recursive merge-sort. Arrows visualize communications with helper processes; dotted lines represent data retained by process for further sorting

Note that the root process can further split its retained data and send half of it to yet another helper process. Helper processes themselves can follow the same procedure as the root process. Splitting and sending data continues until each MPI process becomes a node in the virtual process tree, i.e. until all processes are sent some amount of data to sort.

All MPI processes run the same main function (Fig. 4) which differentiates between the root process and helper processes. The root process prepares the array to sort and then invokes parallel merge sort while each helper process: (i) receives data from its parent process; (ii) invokes parallel merge sort; and (iii) sends sorted data back to parent (Fig. 3). Note that each helper process calculates the level of its top-most appearance in the process tree and passes it to the parallel merge sort function (see Fig. 4).

```
int main(…) {
   // ask MPI for my_rank;
   if (my_rank == 0) {
      // allocate array to sort then run root to sort it:
      run_root_mpi(a, size, temp, …);
   } else {
      run_helper_mpi(my_rank, …);
   }
   // array is sorted;
}
void run_root_mpi (int a[], int size, int temp[], …) {
   int level = 0;
   mergesort_parallel_mpi(a, size, temp, level,…);
}
void run_helper_mpi(int my_rank, …) {
   // probe MPI for a message from parent process
    // and identify message size and parent_rank;
    // allocate int a[size], temp[size];
    MPI_Recv(a, size, …, parent_rank, …);
   int level=my_topmost_level(my_rank);
   mergesort_parallel_mpi(a, size, temp, level, …);
    // send sorted array to parent process:
    MPI_Send(a, size,… , parent_rank, …);
}
int my_topmost_level_mpi(int my_rank) {
   int level = 0;
   while (pow(2, level) <= my_rank) level++;
   return level;
}
```

Figure 4. Root and helper processes in MPI merge sort

Parallel merge sort is executed by various processes at various levels of the process tree, with the root being at level 0, its children at level 1, and so on (Fig. 3). In that, the process's level and the MPI process rank are used to calculate a corresponding helper process's rank (Fig. 5). Then, merge sort communicates for further sorting half of the array with that helper process. Serial merge sort is invoked when no more MPI helper processes are available. The helper's rank calculation method is adopted from Perera's MPI quick sort

algorithm [10].

```
void mergesort_parallel_mpi
   (int a[], int size, int temp[], int level, …) {
   // my_rank is used to calculate helper rank:
   int helper_rank = my_rank + pow(2, level);
   if (helper_rank > max_rank) {
      mergesort_serial(a, size, temp);
   } else {
      // send second half of array, asynchronous:
      MPI_Isend(a+size/2, size-size/2, …, helper_rank, …);
      // sort first half:
      mergesort_parallel_mpi(a, size/2, temp, level+1, …);
      // receive second half sorted:
      MPI_Recv(a+size/2, size-size/2, …, helper_rank, …);
      // merge the two sorted sub-arrays:
      merge(a, size, temp);
   }
}
```

Fig. 5. Message-passing parallel merge sort with MPI. It uses explicit mapping of recursive calls to helper processes

The performance of the above message-passing (with MPI) implementation is evaluated in Section 2.4.

2.3 Hybrid Merge Sort with MPI and OpenMP

A hybrid parallel architecture combines distributed and shared memory in the same computing system. Some authors prefer the term "multi-level" parallel architecture but we choose to use "hybrid" for its brevity. An SMP cluster of multi-processor multi-core nodes is a typical example of a hybrid parallel system. Besides computer clusters, NUMA computers, such as Compaq's Alpha EV6 and SGI Origin can also be viewed as hybrid parallel systems.

```
void mergesort_parallel_mpi_and_omp
   (int a[], int size, int temp[], int level, int threads, …) {
   int helper_rank = my_rank + pow(2, level);
   if (helper_rank > max_rank) {
      mergesort_parallel_omp(a, size, temp, threads);
   } else {
      MPI_Isend(a+size/2, size-size/2, …, helper_rank, …);
      mergesort_parallel_mpi_and_omp
         (a, size/2, temp, level+1, threads, …);
      MPI_Recv(a+size/2, size-size/2, …, helper_rank, …);
      merge(a, size, temp);
   }
}
```

Fig. 6. Hybrid parallel merge sort with MPI and OpenMP

Recursive merge sort can be mapped rather straightforwardly onto a hybrid parallel architecture by means of MPI and OpenMP. On a hybrid system, MPI can provide coarse-grain parallelism by mapping merge sort recursive invocations onto a process tree (Fig. 3), as already discussed in Section 2.2. In addition, OpenMP can provide finer-grain parallelism by introducing multiple threads within individual MPI processes, namely those MPI processes that are visualized as leaf nodes in the process tree (Fig. 3). A more formal outline of this approach is shown in Fig. 6.

Note that hybrid merge sort (Fig. 6) switches to shared memory merge sort (rather than to serial merges sort) when no more MPI helper processes are available, thus utilizing all available processors and cores on each cluster node.

The performance of the above MPI + OpenMP hybrid implementation is evaluated in Section 2.4.

2.4 Performance Evaluation

We measured the performance of our shared memory, message-passing, and hybrid parallel merge sorts on a five-node Rocks 5.2 cluster running OpenMPI 1.3 and OpenMP 3 under GNU/Linux. Each cluster node contained two Intel Xeon quad-core processors running under a 2.80 MHz clock. We executed our merge sorts with randomly generated arrays of 10^7 integer elements. Note that cluster node capacity permitted experiments with arrays consisting of up to $3*10^7$ integer elements. No other applications were active on the cluster during our performance measurements.

Table 1. Performance results on a standalone Rocks cluster (all times are in seconds)

Program	OpenMP Threads	MPI Processes	Nodes Used	Cores Used	Average Rocks Time	Rocks Speedup
Serial			1	1	4.1	1.0
OpenMP	2		1	2	2.4	1.7
	4		1	4	1.6	2.6
	8		1	8	1.3	3.2
MPI		8	1	8	2.9	1.4
		16	2	16	2.2	1.9
		24	3	24	2.1	2.0
		32	4	32	1.9	2.2
		40	5	40	2.0	2.1
Hybrid	8	1	1	8	1.3	3.2
	8	2	2	16	1.5	2.7
	8	3	3	24	1.5	2.7
	8	4	4	32	1.9	2.2
	8	5	5	40	1.8	2.3

Shared memory merge sort (with OpenMP, Section 2.1) was executed on 1, 2, 4, and 8 cores on the master node of the Rocks cluster. Message-passing merge sort (with MPI,

Section 2.2) was executed on 1 to 5 nodes by using all available cores on all nodes for MPI processes. Hybrid memory merge sort (with MPI and OpenMP, Section 2.3) was executed on 1 to 5 nodes by using one core on each node for distributed MPI processes and all 8 cores for shared memory OpenMP processes. Table 1 presents average wall-clock times and speedup for serial, shared memory, message-passing, and hybrid merge sorts.

In our experiments, shared memory merge sort runs faster than message-passing merge sorts. Hybrid merge sort, while still slower than shared memory merge sort, is faster than message-passing merge sort.

Different OpenMP sections (see Fig. 2) may – or may not - be executed by different threads. It is up to the runtime environment to assign threads to sections, and the OpenMP programmer has no control over thread-to-section assignment. Our experiments show that the runtime environment may overuse some threads and underuse others, as illustrated by Table2. Despite of this inadequate load balancing, shared memory merge sort with OpenMP still performs faster that message-passing merge sort with MPI.

Table 2. Merge sort calls per a thread in a test execution

Thread #	0	1	2	3	4	5	6	7
Assigned calls	4	3	3	0	1	0	1	3

Our merge sorts process a single array that can be entirely held in RAM on a single node. This setup is advantageous for single node implementations with OpenMP and disadvantageous for multiple-node implementations with MPI. Indeed, such a centralized setup involves multiple MPI data transmissions that begin and end with the root node; at the same time, OpenMP is exempt from such transmissions. Should the setup change to permit handling of “big data” that do not fit in a single node RAM, all implementations would require multiple I/O operations. In a “big data” setup, MPI’s parallel I/O functionality may possibly provide considerable advantages in comparison to pure OpenMP implementations.

Eventually, using OpenMP 3 *tasks* may offer some performance benefits in comparison to parallel sections. We chose to use parallel sections because they are simpler and more intuitive; better documented; and more widely implemented at this time.

2.5 Parallel Merge Sort on the Amazon Elastic Compute Cloud

Amazon Web Services (AWS) is the first – and currently the largest – public cloud computing platform that provides virtual computing resources on a metered, pay-per-use basis [1]. A goal of the AWS development was to offer as a public utility part of the extensive Amazon data centers by means of service-oriented virtualization. AWS incorporates a number of services, most notably the Elastic Compute Cloud (EC2), and also services built on top of EC2, such as the Elastic MapReduce.

Using AWS’s EC2, we (i) launched a single server instance; (ii) uploaded and compiled our OpenMP-based shared memory merge sort; (iii) ran merge sort experiments and collected performance data; (iv) terminated the server instance. In the process, we were charged only for the actual time during which our server instance was running, and the charges were covered by a grant provided by Amazon.

To launch our AWS server, we used an abstract machine image (AMI) provided by Amazon itself, a CentOS system with an OpenMP-enabled C-compiler readily available. We launched this AMI as a single 64-bit cluster compute instance with 8 physical cores from two quad-core Intel Xeon processors, running under a 2.93 GHz clock. With hyper threading, the server provided 16 virtual cores. We executed our shared memory merge sort with randomly generated arrays of 10^7 integer elements, much like we did on the standalone Rocks cluster (Section 2.4). Note that AWS server capacity permitted experiments with arrays consisting of up to 10^9 integer elements, much larger than the 30^7 limit of our standalone Rocks cluster nodes. On AWS, we chose to systematically experiment with arrays of 10^7 integer elements for the sake of performance comparisons with the standalone Rocks installation.

Shared memory merge sort (with OpenMP) was executed on 1, 2, 4, 8, and 16 cores on our AWS server. Table 3 presents average wall-clock times and speedup - for serial and shared memory merge sorts on the AWS virtual server. For the sake of more convenient comparisons, Table 3 includes standalone Rocks cluster performance data from Table 1 (Section 2.4).

Table 3. Performance results on an AWS virtual server

Program	OpenMP Threads	Virtual Cores Used	Physical Cores Used	Average AWS Time	Average Rocks Time	AWS Speedup	Rocks Speedup
Serial		1	1	2.5	4.1	1.0	1.0
OpenMP	2	2	2	1.4	2.4	1.8	1.7
	4	4	4	0.8	1.6	3.1	2.6
	8	8	8	0.5	1.3	4.7	3.2
	16	16	8	0.5		4.7	

Time and speedup data from Table 3 clearly indicate that our rented AWS instance, being a physically hosted on a new and more powerful shared memory computer, offered a clear advantage in terms of performance as compared to our dedicated Rocks node.

Looking at Table 3, one may conclude that hyper threading is not particularly beneficial for our shared memory merge sort. In fact, we found out that hyper threading becomes a positive factor for larger arrays. Shared memory merge sort

(with OpenMP) was executed with large data sets on 8 physical and 16 virtual (with hyper threading) cores on our AWS instance. Table 4 outlines performance of serial and shared memory merge sorts on a set of very large arrays. This table indicates speedup gains from hyper threading for larger arrays.

Table 4. Performance with hyper threading on large data

OpenMP Merge Sort Data Size	Serial Time	8 Physical Cores - Time	16 Virtual Cores - Time	8 Physical Cores – Speedup	16 Virtual Cores - Speedup
10^7	2.5	0.5	0.5	4.7	4.7
10^8	29.5	5.4	4.9	5.4	6.0
$5*10^8$	161	28.8	24.4	5.6	6.6
10^9	334	59.5	50.1	5.6	6.7

While launching and using a single high-performance AWS instance is straightforward, configuring a multi-node virtual MPI cluster on AWS is not as easy. As of the time of this writing (March 2011), we are not aware of good quality generic AMIs that can be used to launch MPI clusters by following well documented, sound procedures. At the absence of pre-packaged MPI-enabled cluster nodes, users who would like to run MPI on AWS must act as system administrators and build MPI-enabled, cluster-capable AMIs by themselves. Despite of the technical difficulty of the process, we managed to configure and fire a virtual SMP cluster on the on the Amazon EC2.

To launch our AWS cluster, we created a custom AMI, an Ubuntu Lucid system enhanced with MPI and containing our own merge sort programs. We used this AMI to fire an MPI cluster of five extra-large EC2 instances. In AWS terminology, each instance was a 64-bit platform with 4 virtual cores. Again, we executed our message-passing merge sort with randomly generated arrays of 10^7 integer elements, just like we did on the standalone Rocks cluster (Section 2.4).

Note that although our AWS cluster and the standalone Rocks cluster consisted of the same number of nodes, the two clusters differed in their node architectures, including the number of cores in each node (8 physical cores on the Rock cluster as opposed to 4 virtual cores on the AWS cluster). This is why it is difficult to formally compare performance results obtained on these different clusters with our message-passing merge sort. Yet, it became clear that our message-passing merge sort achieved higher performance on the Rocks cluster than on the AWS cluster. More important, execution times that we measured on the AWS cluster were unstable and varied in a much larger range than execution times on the Rocks cluster.

Data in Table 5 illustrate the performance instability of the AWS virtual cluster. Table 5 includes a representative selection of: average, minimal, and maximal wall-clock times; corresponding standard deviations; corresponding speedup data for message-passing merge sort on the AWS virtual cluster and, for comparison, on the Rocks cluster.

Table 5. Performance deviations, AWS and Rocks clusters

Platform	Nodes	Total Cores	Aver Time	Min Time	Max Time	Standard Deviation	Speedup
AWS	4	16	3.3	2.5	4.9	0.9	1.2
Rocks	*4*	*32*	*1.91*	*1.89*	*1.93*	*0.02*	*2.2*
AWS	5	20	13.6	3.0	40.5	11.9	0.3

Rocks cluster's performance advantages over the AWS cluster can be attributed to the following factors. First, our AWS virtual nodes, being AWS EC2 instances, shared the same hardware with other unknown AWS instances and their applications, and load spikes in those anonymous applications had been probably quite detrimental to the AWS virtual cluster performance; in contrast, we had the Rocks cluster dedicated to our experiments. Second, our Rocks cluster nodes were physically located on the same rack while our AWS virtual nodes were located in the same region, but quite likely on different racks; thus slower and busier network connections negatively affected AWS cluster performance and stability. Last but not least, virtualization in EC2 may induce significant penalties for scientific computing workloads [14].

3 Conclusions

This paper introduces three parallel versions of recursive merge sort: shared memory (with OpenMP), message-passing (with MPI) and hybrid (with MPI and OpenMP). While others have developed merge sort algorithms with either multi-threading [6] or message-passing [11], this paper offers comparable multi-threaded, message-passing, and hybrid implementations. The paper reports performance experiments with the three approaches and draws conclusions accordingly (while neither [6], nor [11] report systematic performance results of their individual algorithms). Out performance experiments show that shared memory merge sort (with OpenMP) is faster than message-passing merge sort (with MPI) when applied to arrays that fit entirely in RAM; the performance of hybrid merges sort falls between that of shared merges sort and message passing merge sort. (These relations, however, may not hold for very large arrays that significantly exceed RAM capacity.) Note, however that which programming paradigm is best widely depends on the nature of the problem, the hardware and software in cluster nodes, and the cluster network; for example, a fast network can make a message-passing (with MPI) solution for some problem faster than shared-memory (with OpenMP) and hybrid solutions [15].

Last but not least, this paper describes cloud computing experiments with shared memory merge sort (with OpenMP) and with message-passing merge sort (with MPI) on AWS in general and on the Amazon Elastic Compute Cloud in particular. The use of OpenMP on AWS is straightforward; it has led us to a better speedup, thanks to the readily available high-performance instance on a pay-per-use basis. In contrast, MPI virtual clusters are not readily available on AWS and their configuration for AWS requires technical system administration skills. Our experiments with a virtual AWS cluster exhibited poor and unstable performance. A recent EC2 benchmark performance analysis concludes that the performance and reliability of the EC2 cloud are low [14]. Yet, the EC2 cloud "may still appeal to scientists who need resources immediately and temporarily" [14]; our shared memory merge sort EC2 experiments demonstrate that specific problems and software may actually give performance gains to the high-performance cloud user.

4 References

[1] Amazon Web Services. Retrieved on March 1, 2011 from http://aws.amazon.com/.

[2] Cole , Richard. Parallel merge sort. SIAM Journal on Computing, Volume 17 Issue 4, August 1988, 770-785.

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms (3rd ed.), MIT Press, 2009.

[4] El-Ghazawi, Tarek; Carlson, William; Sterling, Thomas; Yelick, Katherine. UPC: Distributed Shared Memory Programming. Wiley, 2005.

[5] Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, Weicheng; Manchek, Robert; Sunderam , Vaidy. PVM: Parallel Virtual Machine. MIT Press, 1994.

[6] Huba , Dzmitry. Parallel merge sort. Retrieved on March 1, 2011 from http://dzmitryhuba.blogspot.com/2010/10/parallel-merge-sort.html.

[7] Katajainen, Jyrki; Träff, Jesper L. A meticulous analysis of mergesort programs. Lecture Notes in Computer Science, 1997, Volume 1203/1997, 217-228.

[8] LaMarca, Anthony; Ladner, Richard. The influence of caches on the performance of sorting. Proc. 8th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA97), 370–379.

[9] Orio: An Annotation-Based Empirical Performance Tuning Framework. Retrieved on March 1, 2011 from http://trac.mcs.anl.gov/projects/performance/wiki/Orio.

[10] Perera, Prasad. Parallel quicksort using MPI & performance analysis. Retrieved on March 1, 2011 from http://www.codeproject.com/KB/threads/Parallel_Quicksort/Parallel_Quick_sort_without_merge.pdf.

[11] Rolfe ,Timothy J. A Specimen of parallel programming: Parallel merge sort implementation. ACM Inroads, Volume 1, Issue 4, December 2010, 72-79.

[12] The Message Passing Interface (MPI) standard. Retrieved on March 1, 2011 from http://www.mcs.anl.gov/research/projects/mpi/.

[13] The OpenMP specification for parallel programming. Retrieved on March 1, 2011 from http://openmp.org.

[14] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of EC2 cloud computing services for scientific computing. In: Cloud Computing: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer, 2010, Vol. 34, Book Series Editor: O. Akan et al., pp. 115-131.

[15] G. Jost, H. Jin, D. Mey, F. Hatay. Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster. NAS Technical Report NAS-03-019, November 2003.