MP-sort: Sorting at Scale on Blue Waters – for a Cosmological Simulation

Yu Feng[1,3], Mark Straka[2], Tiziana Di Matteo[3], and Rupert Croft[3]

[1]Berkeley Center for Cosmological Physics, 341 Campbell Hall, Berkeley, CA 94720
[1]Berkeley Institute for Data Science, 190 Doe Library, Berkeley, CA 94704
[2]National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 1205 W. Clark St., MC-257, Urbana, IL 61801
[3]McWilliams Center for Cosmology, 5000 Forbes Avenue, Pittsburgh, PA 15213

April 6, 2015

Abstract

We implement and investigate a parallel sorting algorithm (MP-sort) on Blue Waters. MP-sort sorts distributed array items with non-unique integer keys into a new distributed array. The sorting algorithm belongs to the family of partition sorting algorithms: the target storage space of a parallel computing rank is represented by a histogram bin whose edges are determined by partitioning the input keys. The algorithm is used in a cosmology simulation (BlueTides) that utilizes 90% of the computing nodes of Blue Waters, the Cray XE6 supercomputer at the National Center for Supercomputing Applications. MP-sort is optimal in communication: any array item is exchanged over the network at most once. We analyze a series of tests on Blue Waters with up to 160,000 MPI ranks. At scale, the single global shuffling of items takes up to 90% of total sorting time, and overhead time added by other steps becomes negligible. MP-sort demonstrates expected performance on Blue Waters and served its purpose in the BlueTides simulation. We make the source code of MP-sort freely available to the public.

1 Introduction

A complex supercomputing application utilizes data algorithms as well as numerical algorithms. Sorting allows efficient random access of the data create by these applications in post-processing and data analysis. The dataset being sorted is typically distributed across many computing ranks of the massively parallel application, and thus parallel algorithms on multiple-instruction, multiple-data (MIMD) computer architectures are of particular interest.

One of such applications is the BlueTides simulation, a smoothed particle hydrodynamics (SPH) simulation to study the formation of the first galaxies in the early universe.

BlueTides is carried out on the Cray XE6 supercomputer, Blue Waters at National Center for Supercomputing Applications (NCSA), and ran on 20,250 Cray XE nodes with 81,000 MPI process ranks (4 MPI tasks per node, each using 8 OMP threads). A total of 697 billion particles are used to model the formation of galaxies and structure in a uniform vol-

[0]This research is supported by the National Science Foundation (award number OCI-0725070, OCI-0749212 and AST-1009781) and by the state of Illinois (the Blue Waters sustained-petascale computing project). Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

ume that is 300 times larger than the largest Hubble Deep Field survey campaign, the BoRG survey[16], and with a resolution that can resolve the physical processes inside individual galaxies. Computation at this scale has only recently become accessible with the increased power of computers like Blue Waters[3].

The simulation software MP-Gadget is based on Gadget [14], but heavily modified to include additional physics and to accommodate such a large computing scale.

Each snapshot of BlueTides consumes 40 TB of disk storage, and poses severe challenges in data analysis. BlueTides utilizes a Friend-of-Friend [4] finder to identify groups (or equivalent classes) of particles that have formed structure. The properties of these particles are stored into "Particle in Group" (PIG) files, where particles of the same group, are stored in a contiguous data chunk. PIG files are much smaller ($\sim$ TB each) than snapshots.

Parallel sorting by the group number is an essential step in building PIGs. We sort in parallel the meta-data of particles twice calculate the shuffling scheme of the particle data cross 81,000 MPI ranks, before calling the particle exchange module to order the particles into the correct ordering in PIG files.

As we scaled up the production runs for BlueTides, we discovered that the original parallel merge-sorting module in MP-Gadget had serious scaling limitations. At redshift[1] $z = 14$, most of the MPI ranks are coupled into the sorting (because gravitational collapse happens at all places in the universe). The original merge-sort based parallel sorting algorithm degrades significantly, even though the total number of items sorted is only a few billion (out of 700 billion particles in BlueTides). Ultimately it was this sorting that brought the simulation to a halt. This inspired the construction of MP-sort, a partition based parallel sorting algorithm.

Sorting is one of the fundamental algorithms in computing, and parallel sorting algorithms have been extensively studied, on both shared memory and distributed memory systems[see, e.g. 1, 10]. Of various proposed algorithms in the literature, partition based algorithms have been shown to out-perform other algorithms by a large margin on a massively parallel computing environment[see, e.g. 5, 8, 9].

Our new parallel sorting module for the BlueTides simulation, MP-sort, falls into the partition based algorithm family: every item of the input array is exchanged at most *once* across all computing nodes. The source code of MP-sort is available[2]. In this paper, we studied the scaling of the algorithm on Blue Waters at extremely large scale (160,000 MPI ranks), and show that at scale, the global shuffling of items uses more than 90% of total wall-time, and overhead time in other steps is negligible. We also emphasize that the algorithm has been used in the BlueTides Simulation that runs continuously for several days on 20,250 nodes with 81,000 MPI ranks and 8 OpenMP threads per rank. This is 90% of the full capability of Blue Waters.

The sorting algorithm that is implemented in MP-sort is described in section 2, where we also show that the sorting algorithm can be made stable. In section 3, we report and analyze the results of the scaling tests on Blue Waters, as well as the improvement due to switching to MP-sort in BlueTides simulation. Section 4 is our conclusion. We describe the architecture of Blue Waters in the appendix.

2 MP-sort: the Algorithm

In this section we describe the algorithm that is implemented in MP-sort. The algorithm, as well as many other variants of partition based sorting algorithms, is based on the idea of partitioning the local array into buckets that are directly sent to the target rank. Because the amount of communication is minimal, (without global merging) these algorithms out-perform others at an extreme scale [8]. MP-sort in BlueTides is a particular version of partition sort, where the ordering of items is defined via a key function

$$K : a \to k, k \in \text{Integer},$$

where a is an item being sorted. We note that it is crucial NOT to require the keys to be unique, as

[1] redshift marks the time of the simulation. The number decreases as the simulation evolves from the past toward now.

[2] `http://github.com/rainwoodman/MP-sort`

in BlueTides, the group numbers of particles in the same group are identical. Also, BlueTides does not need a stable sorting algorithm.

2.1 Overview of the Algorithm

In general, a partition based parallel sorting algorithm contains 5 steps (Algorithm 1). In order to illustrate the algorithm in a clear way, we will define a few variables and discuss their relevant properties in this section.

Definition 1 (Distributed Array)**.** *A distributed array $A[i, p; N]$ is distributed to ranks $p \in [1, P]$. $i \in [1, n_p]$ is the index of the item on rank p.*

We also define the distribution $N = (n_1, n_2, ...n_P)$ as the number of items per rank, and cumulative distribution $N^c[p] = \sum_{q<p} n_p$, the cumulative sum of N.

Definition 2 (Splitters)**.** *The splitters are integer values $E[p]$, such that for any item in sorted distributed array $B[i, p]$,*

$$E[p-1] \leq B[i,p] \leq E[p].$$

It is implied that $E[0] = 0$.

Definition 3 (Local Bounds)**.** *The lower bound $C_1[q, p]$ is the total number of items with keys less than $E[q]$, on rank p. The upper bound $C_2[q, p]$ is the total number of items with keys less than or equal to $E[q]$, on rank p. For convenience, we also define cumulative sums of C_1 and C_2*

$$R_{1,2}[q] = \sum_{q \in P} C_{1,2}[q, p].$$

$R_{1,2}[p]$ is the total number of items less than (or less than or equal to) $E[p]$. The computation of $C_{1,2}$ is completely local, while computing each of $R_{1,2}$ requires a single global communication of size P.

Definition 4 (Shuffling matrix)**.** *A rank p sends items (of the locally sorted array) $L[q-1, p] : L[q, p]$ to another computing rank q, and $L[q, p]$ is called the Shuffling Matrix.*

We see that $L[p, q]$ is constrained by the target distribution N,

$$\sum_{q \in P} L[q, p] = N^c[p],$$

and also by the lower and upper bounds $C_{1,2}[q, p]$

$$C_1[q, p] \leq L[q, p] \leq C_2[q, p].$$

Algorithm 1 MP-sort: top-level algorithm

1 Locally sort $A[:, p]$;
2 Find splitters $E[p]$ and local bounds $C_1[i]$, $C_2[i]$;
3 Solve for a shuffling matrix $L[i, p]$ based on $E[p]$, $C_{1,2}$;
4 Shuffle locally sorted $A[:, p]$ into $B[:, p]$;
5 Locally sort $B[:, p]$;

2.2 Details of the Algorithm

2.2.1 Step 1: Local Sorting

The initial local sort serves two purposes. First, the shuffling can be implemented via a MPI_Alltoall operation when all items that are targeted to the same bucket are collected into a contiguous array. Second, when the local array is sorted, counting the number of local items within a bucket can be accelerated with a binary search algorithm.

2.2.2 Step 2: Splitters and Local Bounds

In this step, we want to find the splitters and local bounds such that partitions the output array according to the distribution $N = (n_1, \ldots, n_P)$, where n_p is the number of items on rank p.

Because we can locally sort $A[:, p]$ in previous step, for any give spliters $E[p]$, $C_{1,2}$ (and hence $R_{1,2}$) matrices can be quickly calculated via a binary search, with wall-time complexity $O[P \log N/P]$.

We then want to find $E[p]$ and $C_{1,2}$ such that

$$R_1[p] \leq N^c[p] \leq R_2[p].$$

$R_{1,2}[p]$ is monotonic function of $E[p]$, and $E[p]$ is bound. Therefore a solution can be found via binary

search in the "key" domain. The total number of iterations is only proportional to the number of bits of the key, which is typically in logarithm of the total number of items.

2.2.3 Step 3: Shuffling Matrix

As the keys are non-unique, the splitters $E[p]$ insufficiently determines the splits. We begin with a serial algorithm to solve for L (Algorithm 2). We see that the algorithm ensures that the total number of items rank p receives $s = N[p]$.

Algorithm 2 Serial Algorithm to solve for $L[i,p]$

```
for i = 1 to P do
  L[i,p] ← C1[i,p]
  s ← Σ_p L[i,p] − L[i−1,p]
  if s < N[i] then
    for p = 1 to P and s ≠ N[i] do
      d ← min(C2[i,p] − L[i,p], N[i] − s)
      L[i,p] ← L[i,p] + d
      s ← s + d
    end for
  end if
end for
```

For a very large dataset decomposed over a large number of ranks (P $>$ a few thousand), the matrices become too large to be practical. We decompose the matrices along the row direction, and use a parallel algorithm. The rows are solved in parallel. The algorithm is listed in Algorithm 3. The parallel algorithm is similar to the serial algorithm, but we use the cumulative distribution N^c to avoid data dependency between different rows.

2.2.4 Step 4: Shuffle

In this step, we call an MPI_Alltoallv function to exchange the items between computing ranks according to the constructed communication layout $L[:,:]$. We note that this step is the only place in MP-sort where a large amount of communication is involved. Every item is sent directly to the destination rank at this step. The total number of items exchanged within the communication network is N.

Algorithm 3 Parallel Algorithm to solve for $L[i,p]$

```
Require: Arrays L[i,p], C1[i,p], C2[i,p] are distributed to computing rank i by rows.
  L[i,p] ← C1[i,p]
  s ← Σ_p L[i,p]
  if s < N^c[i] then
    for p = 1 to P and s ≠ N^c[i] do
      d ← min(C2[i,p] − L[i,p], N^c[i] − s)
      L[i,p] ← L[i,p] + d
      s ← s + d
    end for
  end if
```

2.2.5 Step 5: Final local sort

After the final local sort, the distributed array will be sorted. We note that because the distributed array is already partially sorted, the local algorithm may perform slightly better than in the initial local sort. Some authors further exploit this and replace the step with merging [e.g. in 8]; however we note that the gain is negligible at a massive scale, for the time spent in local sort is less than 5% of the total wall-time in our largest test runs.

2.3 Stability

To guarantee a stable algorithm, two conditions must be met: (1) the algorithm that is used for local sort is stable; (2) the layout solver preferentially consumes items from lower ranks. Condition 1 preserves the relative ranking of two items with identical keys within a computing rank; Condition 2 preserves the relative ranking of two such items after the All-to-All exchange.

For local sorting, we use the unstable qsort_r implementation in glibc version 2.18[11]. Therefore, our implementation violates Condition 1.

We note however, our implementation preserves Condition 2, therefore, if desired, it can be made stable by replacing qsort_r with a stable local sorting algorithm.

3 Scaling Tests and Analysis

We performed weak-scaling and strong-scaling tests of MP-sort on Blue Waters at $n_i = 100,000$, $1,000,000$ and $10,000,000$ 64-bit integer items per MPI rank. We vary the number of MPI ranks up to 160,000 (5000 nodes on Blue Waters, 32 ranks per node). Table 1 lists all of the test runs and their total wall-time.

We randomly generate integers with 56 bits: since the key space is much larger than the number of items, key collision is unlikely in the tests.

3.1 Weak-scaling: Component analysis

We break the timing of the algorithm into 5 steps, each corresponding to one major section of the algorithm. (Figure 1)

LocalSort1 and LocalSort2: These are the first and last local sorting steps in the algorithm. The local sorting steps do not utilize any network communication; however there is a necessary barrier synchronization at the end of the step. We observe that the two steps together take a negligible fraction of the total computing time as the size of the problem increases. ($< 1\%$ on 160,000 ranks.)

FindEdges: The step corresponds to Step 3. The algorithm is iterative with a binary search. Fixing the splitters requires 56 iterations (56 corresponds to the number of bits in a key). In an iteration, we perform two MPI_Allreduce operations along the columns of integer arrays $C_1[i,p]$ and $C_2[i,p]$ (reduction of two $P+1$-length 64-bit integer arrays on P ranks). In addition, we perform one MPI_Allreduce of a boolean variable to collectively evaluate the terminal condition. We observe that the time spent in this step scales linearly with the number of MPI ranks.

SolveLayout: The step corresponds to Algorithm 3. The algorithm includes one MPI_Alltoall to transpose the 64-bit integer arrays $C_1[i,p]$ and C_2[i, p]. Overall, we observe that the time spent in this step scales near linearly to the number of MPI ranks, and is dominated by the communication.

Exchange: The step contains (1) an MPI_Alltoall call that builds the 32-bit integer receiving displacement array, and (2) an MPI_Alltoallv call that exchanges all of the items. The total number of items sent out from a rank is N/P, and the average message size is N/P^2, decreasing with the number of MPI ranks. The number of messages posted by a rank is P, and the total number of messages on the communication subsystem is P^2. For example, with the 10M, 80000-rank test, an MPI rank posts 80,000 messages, each of which contains 1250 items or 10 KB, totaling to 80 MB. Overall, we observe that the communication time scales near linearly with the number of MPI ranks.

3.2 Communication and Load Balance Analysis

We observe that the communication performance appears bandwidth-limited for MPI rank counts fewer than about 8k, due to the size of the decomposition at that scale. The weak-scaling plots (Figure 2) suggest that smaller process counts exhibit disproportionately larger exchange times with increasing N, likely due to saturating the bandwidth of the nodes at that scale or perhaps greater load imbalance (discussed below). The configurations with fewer MPI ranks are much more sensitive to very large N, indicating that a performance threshold is crossed between N = 10^6 and 10^7.

The strong scaling charts (Figure 2) show that bandwidth is not yet being saturated at process counts greater than 8k since the times remain flat, or in fact decrease, with super-linear performance, as the average message size decreases with P^2. Although the total communication time naturally increases with the number of ranks, the growth rate above 8k tasks is essentially independent of the message payload. The steady growth in communication time at MPI counts >8k is due to the increasing overhead as the number of messages grows as P^2.

Table 1: Total wall-time of tests

	1024	2048	4096	8192	16384	80000	160000
100K	2.167	4.088	7.094	13.95	20.45	269.5	342.5
1M	3.324	5.404	8.949	13.32	28.48	285.4	462.5
10M	25.35	26.33	32.59	29.9	39.37	309	475.8

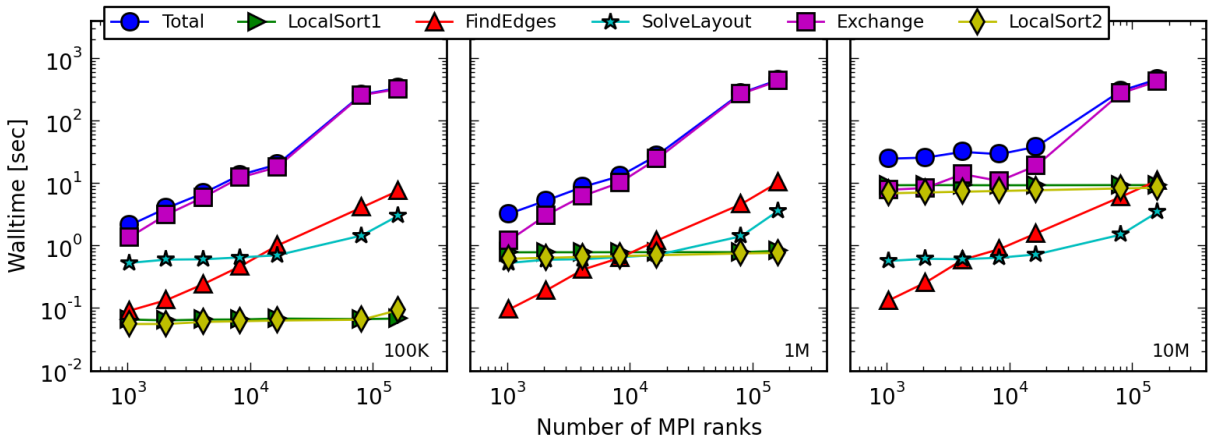

Figure 1: Weak Scaling tests with increasing number of MPI ranks

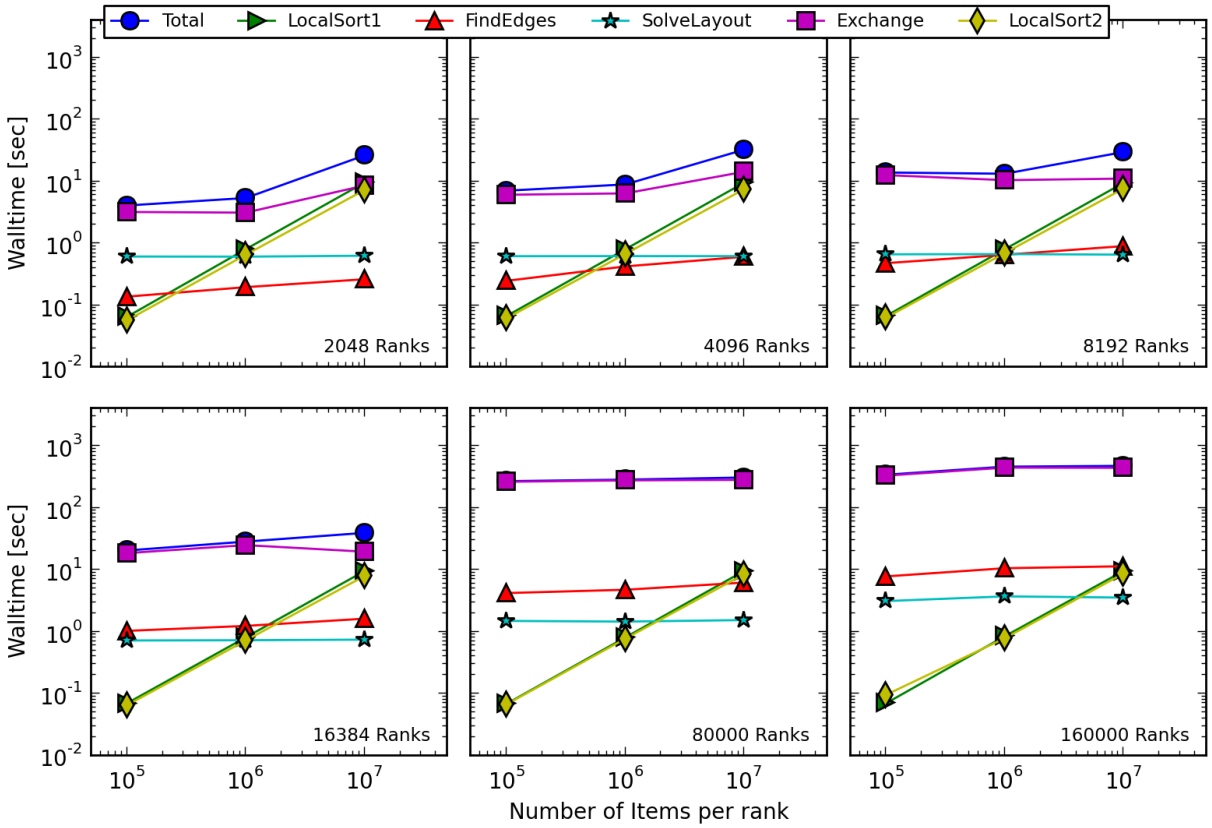

Figure 2: Strong Scaling tests at 100K, 1M, 10M load.

For the global exchange pattern dominating these tests, the global bandwidth is actually twice the bisection bandwidth, since half of the total communication is local, with only half actually crossing the bisection. Depending on a job's size and placement within the 3D torus, the bisection bandwidth of that partition of nodes can determine the performance of some all-to-all communications[6]. The bisection bandwidth available to these larger partitions of Blue Waters would comfortably exceed the requirements of our tests. Therefore we would like to point out that the communication time is likely due to message latency. Other studies [e.g. 13, 15, 17] have addressed bandwidth and latency performance over the Cray Gemini interconnect with both well-defined communication benchmarks and full applications in greater detail.

In these tests (and the full BlueTides simulation as well), the data is random, but the communication pattern is not unpredictable. It is worth commenting on the distribution of message sizes. Since the items are roughly drawn from a uniform distribution, the number of items sent from one rank to another roughly follows a Poisson distribution:

$$N_{ij} \sim P(N/N_P^2)$$

In other words, the 10M load/160K rank run would have on average 100 items per message, with $1-\sigma$ uncertainty of 10 items. This corresponds to a 10 percent message imbalance. With fewer ranks, the imbalance tends to grow; for the 100K load/160K run, with an average of 0.6 items per message, many messages are expected to be zero-length. This explains the slight decrease in communication time of the run. (See the last panel of Figure 2)

3.3 Performance in BlueTides Simulation

Sorting of particles in the BlueTides simulation is implemented as part of the I/O subroutine of the FOF module. Figure 3 shows the timing before and after replacing the old merge-sort module with MP-sort.

After switching to MP-sort, we process 10 times more particles in the module within the same amount of wall-time, indicating MP-sort has substantially improved the performance of the simulation.[3]

This can be further seen from the time spent in sorting. The FOF I/O module performs 4 operations:

- Two sorting steps on an array of 128 bit integers with the same length as the number of sorted particles to work out the particle exchange scheme;
- Globally exchange the sorted particles with the particle exchange module in MP-Gadget;
- Write the sorted particles to disk.

Ideally one should have had a separate timer for the sorting module, but unfortunately the simulation did not distinguish these times in this version. As a lower bound, we estimate 1000 seconds are spent in operations other than sorting. Therefore the *upper bound* of the sorting time is $t_{\rm all} - 1000$. We see that the improvement in sorting can be a factor of 4 at 4 billion particles and it increases to a factor of 10 as the number of sorted particles increases to 20 billion. This improvement in sorting is mostly due to the amount of communication dropping from $O[N \log N]$ (old merge-sort) to $O[N]$. As the simulation proceeds, more particles will be included in the sort, and we observe an even larger improvement.

4 Conclusion

We implemented an optimally communication efficient massively parallel sorting algorithm on Blue Waters. Because there is only one global shuffling of items, the algorithm demonstrates sustained parallel scaling as well as vastly improved time-to-solution over the previously implemented method (10x wall-time improvement). For problems at a large scale (large number of MPI ranks and large load), more than 90% of wall-time is spent in the single MPI_Alltoall shuffling operation. Our tests show that for large problems, wall-time spent in other steps

[3] We were inspired to switch to MP-sort at 3.7 billion particles when the run stalled during the old sorting for more than 30 minutes.

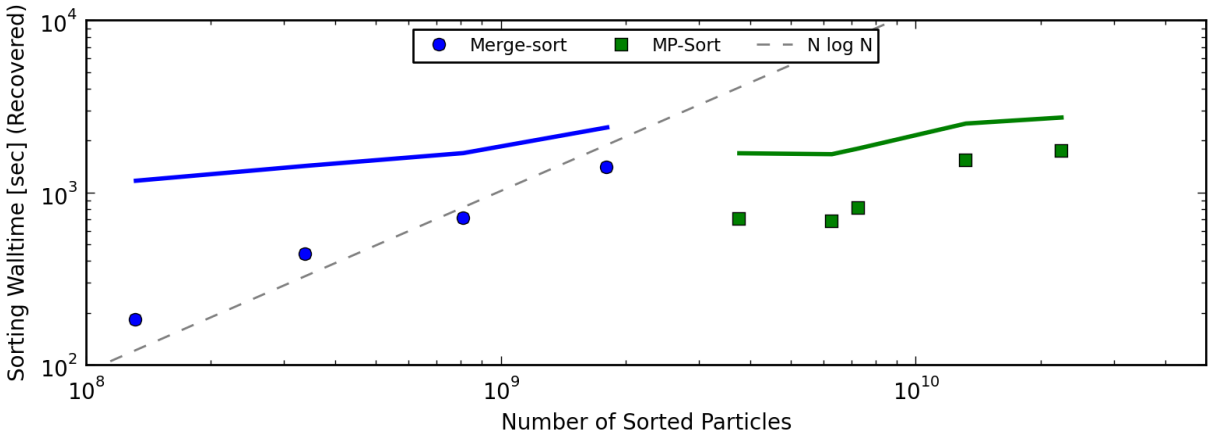

Figure 3: MP-sort in BlueTides Simulation: Before and After. The sorting time is recovered from the timing reported by the full FOF I/O timer. (See text) The solid lines show the total FOF I/O time measured from the simulation. The dash-line gives a projection of the wall-time of the old sorting module.

of a partition based parallel sorting algorithm is essentially negligible, and the sorting time is limited by the message latency of the network. Future improvements in wall-time will benefit the most from increased efficiency and performance of network hardware and software (MPI implementation). This algorithm has been run on NCSA's Blue Waters Cray XE6 at up to 160,000 MPI ranks and is integrated as part of the BlueTides SPH cosmology simulation which has successfully run for days on 90% nodes of Blue Waters.

We release the source code of MP-sort at `http://github.com/rainwoodman/MP-sort`. Contributions to the software package are welcomed. We also provide the IPython notebook and raw data to reproduce the figures and tables in this paper with the source code under the paper/ directory.

A Appendix: Blue Waters System Architecture

The Blue Waters supercomputer provides sustained performance of 1 petaflop/s on a range of real-world science and engineering applications. Blue Waters is composed of 237 Cray XE6 cabinets plus 32 cabinets of Cray XK7 with NVIDIA Kepler(TM) GPU computing capability [12]. The Cray XE6 processor is a 16-core 64-bit AMD Opteron 6276 series (Interlagos). It features 8x64 KB of L1 instruction cache, 16x16 KB of L1 data cache, 8x2 MB of L2 cache per processor core, and 2x8 MB shared L3 cache. Up to 192 processors can populate a cabinet. The memory system is 64 GB, registered ECC DDR3 SDRAM per compute node, with a memory bandwidth of up to 102.4 GB/s per node.

The interconnect is a 3-D torus, with 2 compute nodes connected to a Cray Gemini ASIC router. There are 2 network interface cards, 48 switch ports per Gemini chip providing a 160 GB/s switching capacity per chip[7][2].

Disk storage is comprised of a Sonexion 1600 with the Lustre parallel file system. Total available storage is 26.4 PB and can achieve an aggregate I/O bandwidth of greater than 1 TB/second.

References

[1] Selim G. Akl. 1990. *Parallel Sorting Algorithms*. Academic Press, Inc., Orlando, FL, USA.

[2] Jason Beech-Brandt. 2011. Gemini description, MPI. In *Parallel Programming Workshops and Programming Language Courses 2011*. University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany. https://fs.hlrs.de/projects/par/events/2011/parallel_prog_2011/2011XE6-1/08-Gemini.pdf

[3] Rupert Croft, Tiziana Di Matteo, Nishikanta Khandai, and Yu Feng. 2015. Petascale Cosmology: Simulations of Structure Formation. *Computing in Science & Engineering* 17, 2 (2015), 40–46. DOI:http://dx.doi.org/10.1109/MCSE.2015.5

[4] Marc Davis, George Efstathiou, Carlos S. Frenk, and Si D. M. White. 1985. The evolution of large-scale structure in a universe dominated by cold dark matter. *The Astrophysical Journal* 292 (May 1985), 371–394. DOI:http://dx.doi.org/10.1086/163168

[5] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. Parallel Sorting on a Shared-nothing Architecture Using Probabilistic Splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS '91)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 280–291. http://dl.acm.org/citation.cfm?id=382009.383693

[6] Richard Fiedler. 2013. Improving Performance of All-to-All, Random Pair, and Nearest-Neighbor Communication on Blue Waters. In *Blue Waters User Workshop , February, 2013*. National Center for Supercomputing Applications, Urbana-Champaign, IL, USA. https://bluewaters.ncsa.illinois.edu/documents/10157/12008/AdvancedFeatures_PRAC_WS_2013-02-27.pdf

[7] Forest Godfrey. 2012. *The Cray Gemini Network, Basic Architecture and Failure Analysis*. Technical Report. Urbana-Champaign, IL, USA. http://i2pc.cs.illinois.edu/public_archive/Godfrey_Talk.pdf

[8] Michael Hofmann and Gudula Runger. 2011. A Partitioning Algorithm for Parallel Sorting on Distributed Memory Systems. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC '11)*. IEEE Computer Society, Washington, DC, USA, 402–411. DOI:http://dx.doi.org/10.1109/HPCC.2011.59

[9] Daniel Jiménez-González, Juan J. Navarro, and Josep-L. Larriba-Pey. 2002. The Effect of Local Sort on Parallel Sorting Algorithms. In *Proceedings of the 10th Euromicro Conference on Parallel, Distributed and Network-based Processing (EUROMICRO-PDP'02)*. IEEE Computer Society, Washington, DC, USA, 360–367. http://dl.acm.org/citation.cfm?id=1895489.1895536

[10] Vivek Kale and Edgar Solomonik. 2010. Parallel Sorting Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10)*. ACM, New York, NY, USA, Article 10, 12 pages. DOI:http://dx.doi.org/10.1145/1953611.1953621

[11] David Miller and contributors. 2013. GNU C Library v2.18. (August 2013). https://sourceware.org/ml/libc-alpha/2013-08/msg00160.html

[12] NCSA. 2012. *BlueWaters System Summary*. Technical Report. Urbana-Champaign, IL, USA. https://bluewaters.ncsa.illinois.edu/hardware-summary

[13] Hongzhang Shan, Nicholas J. Wright, John Shalf, Katherine Yelick, Marcus Wagner, and Nathan Wichmann. 2011. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. In *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS '11)*. ACM, New York, NY, USA, 13–14. DOI:http://dx.doi.org/10.1145/2088457.2088467

[14] Volker Springel. 2005. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomy Society* 364 (Dec. 2005), 1105–1134. DOI:http://dx.doi.org/10.1111/j.1365-2966.2005.09655.x

[15] Yanhua Sun, Gengbin Zheng, Laximant V. Kale, Terry R. Jones, and Ryan Olson. 2012. A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 751–762. DOI:http://dx.doi.org/10.1109/IPDPS.2012.127

[16] M. Trenti, L. D. Bradley, M. Stiavelli, P. Oesch, T. Treu, R. J. Bouwens, J. M. Shull, J. W. MacKenty, C. M. Carollo, and G. D. Illingworth. 2011. The Brightest of Reionizing Galaxies Survey: Design and Preliminary Results. ApJ 727 (Feb. 2011), L39. DOI:http://dx.doi.org/10.1088/2041-8205/727/2/L39

[17] Abhinav Vishnu, Monika ten Bruggencate, and Ryan Olson. 2011. Evaluating the Potential of Cray Gemini Interconnect for PGAS Communication Runtime Systems. In *Proceedings of the 2011 IEEE 19th Annual Symposium on High Performance Interconnects (HOTI '11)*. IEEE Computer Society, Washington, DC, USA, 70–77. DOI:http://dx.doi.org/10.1109/HOTI.2011.19