ILLINOIS INSTITUTE OF TECHNOLOGY

Final Project Report

A Decoupled Execution Paradigm Programming Model

ANTONIOS KOUGKAS

STUDENT ID: A20307881

CS554- SPRING 2015

May 7th, 2015

Abstract

Programming models bridge the gap between the underlying hardware architecture and the supporting layers of software available to applications. Specifically, a programming model is an abstraction of the underlying computer system that allows for the expression of both algorithms and data structures and exists independently of the choice of both the programming language and the supporting APIs. In this project, a Decoupled Execution Paradigm (DEP) new programming model is introduced and evaluated. Our DEPM is focused on achieving increased developer productivity, performance, and portability to other system designs. The complexity of designing an exascale platform provide significant challenges for these goals. The representation and management of increasing levels of parallelism, concurrency and memory hierarchies, combined with the ability to maintain a progressive level of interoperability with today's applications are of significant concern of this programming model. We expect that DEPM can be a building block on future applications using the DEP architecture.

1. Introduction

Many scientific applications running in HPC systems deal with large amount of data. Furthermore, Big Data analytics are not just limited to commodity clusters, many use cases of these data analytics problems started to arise in HPC infrastructure as well.

On the other hand, the computation power of current multicore/manycore architectures has followed a faster trend compared to data access performance improvement (latency and bandwidth). This gap is very unlikely to be overcome in the near future. Thus, accessing large amount of data becomes a bottleneck for many applications [1].

Current HPC architectures lack the support for users to control devices in the I/O stack, hence a data-intensive application that follows the existing architecture is forced to move all input data between the underlying storage solution and the compute nodes, passing through slower networks. This movement of large amount of data constantly in an application can cause severe degradation to the overall performance [2].

There has been work on proposing new architectures to alleviate this bottleneck by decoupling and shipping data intensive operations closer to data. One of the promising architectures is the "Decoupled execution paradigm"(DEP) [3], where the application is decoupled into computive-intensive and data-intensive phases and each are directed to the appropriate capable nodes (more details in the background information section). This new execution paradigm promising as it is, it lacks a programming model to facilitate developers to make the most out of the system with minimal effort. Considering the fact that future generation of supercomputers will have programmable devices in their I/O stack, the idea of having a framework to facilitate this programming for end users sounds appealing.

While the high-performance computing community clearly has developed a set of successful and established programming models and supporting implementations, success for the exascale era which DEP architecture aims, will require substantial efforts to explore the development of this significant new technology. The ability to address the challenges when programming with DEP architectures are covered by this projects proposed solution which includes the design and implementation of a novel programming model called DEPM.

2. Background Information

DEP is the first paradigm enabling users to identify and handle data-intensive operations separately. It can significantly reduce costly data movement and is better than the existing execution paradigms for data-intensive applications. DEP decouples the nodes into data-processing (data) nodes and compute nodes. Data-processing nodes are further decoupled into compute-side data nodes and storage-side data notes. Compute-side data nodes are compute nodes that are dedicated for data processing. Storage-side data nodes are specially designed nodes that are connected to file servers with fast network. Compute-side data nodes reduce the size of computing generated data before sending it to storage nodes. Storage-side data nodes reduce the size of data retrieved from storage before sending it to compute-side data nodes. Data nodes can provide simple data forwarding without any data size reduction, but the idea behind data nodes is to let the data nodes conduct the decoupled data-intensive operations and optimizations to reduce the data size and movement. Figure 1 depicts this architecture.

Active storage [4], [5], [6] leverages the computing capability of storage nodes and performs certain computation to reduce the bandwidth requirement between storage and compute nodes. Active disks [7] and smart disks integrate a processing unit within disk storage devices and offload computations to embedded processing unit. However, these architecture improvements are designed to explore either the idle computing power of storage nodes or an embedded processor, and have limited computation-offloading capability. It is easy to see that DEP provides a much more powerful platform for the same purpose.

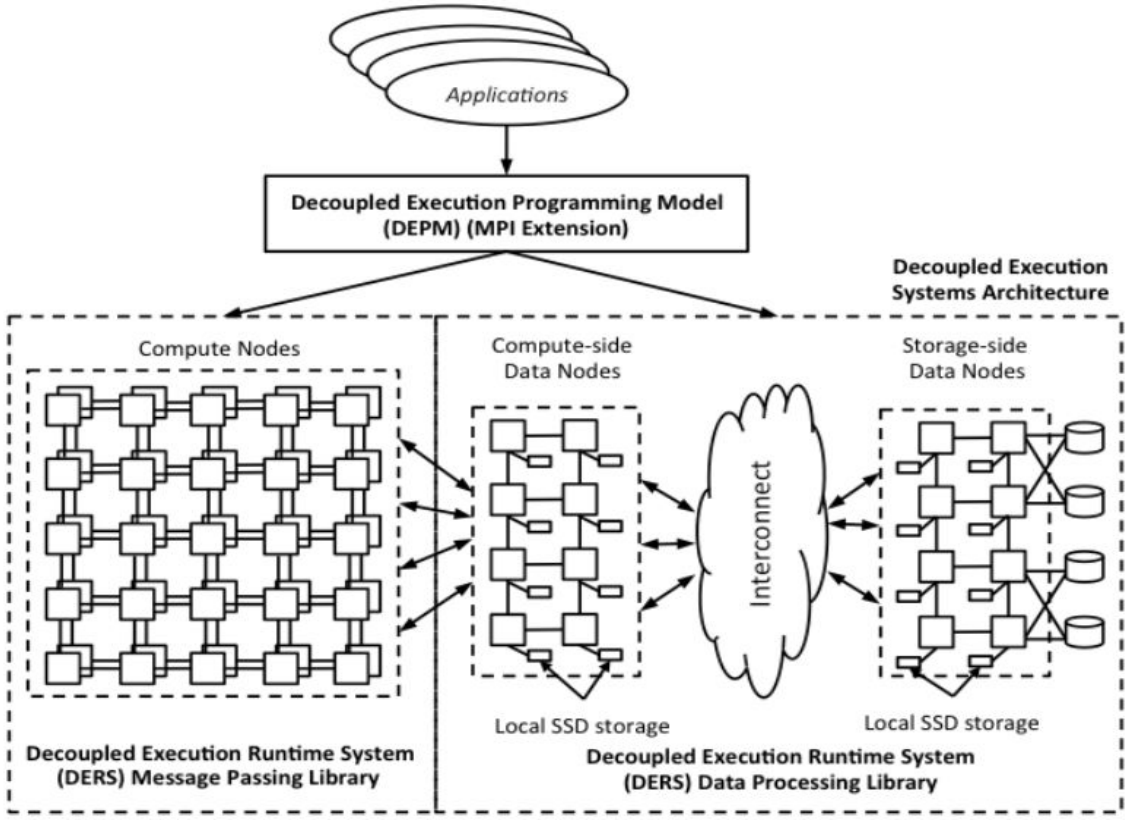

Figure 1, Decoupled Execution Paradigm for Data-Intensive Applications

3. Problem Statement

Even though new system architectures for data-intensive computing are proposed, scientists still use the old-fashion compute-centric parallel programming models to program these system. These programming models primarily focus on the memory abstractions and communication mechanism among processes. I/O is treated as a peripheral activity and often a separate phase in these programming models, which is often achieved through a subset of interfaces such as MPI-IO. This mismatch of programming data-intensive applications with compute-centric approaches makes the programming difficult and leads to error prone and complex codes.

4. Related Work

There have been some development of advanced I/O libraries, such as Hierarchical Data Format (HDF), Parallel netCDF (pnetCDF) and Adaptable IO System (ADIOS), that all provide high level abstractions, map the abstractions to the I/O and try to complement parallel programming models, such as Message Passing Interface (MPI) [8] and others, by managing I/O activities.

MapRecude [9] as mentioned before, is another example of a programming model aiming to ease data-intensive applications, but is typically layered on top of distributed file systems and was not designed for high performance computing semantics and infrastructure. It also requires specific Map and Reduce abstractions.

Lastly, OpenMP [10] is an API that supports multi-platform shared memory multiprocessing programming on most processor architectures and operating systems. It consists of a set of compiler directives, library routines and environment variables that influence runtime behavior. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for HPC. Much like this, DEPM will provide the simple and flexible interface for developing DEP applications running on this new decoupled execution framework.

5. Proposed Solution

In this study, we are proposing a new programming model that will facilitate the decoupled execution paradigm and help developers code easily and efficiently. Much like MapReduce programming model, which was an instant hit for many data-intensive applications in the distributed computing, we thrive to offer HPC users of the DEP architecture increased programming capabilities. Lastly, this new programming model can also allow more complex applications to run on DEP architectures and be evaluated easily. The DEPM (DEP programming model) uses a source-to-source compiler that can translate DEPM code to the traditional HPC code. DEPM is inherently tied not only to the underlying hardware infrastructure but also to the requirements of applications and libraries. Our model exposes features of the runtime software layers and also effectively presents to the software developer traditional and new software development tools. Finally, we believe that any new programming model should aim for desirable features such as abstraction and separation of concerns which DEPM will provide.

The proposed solution comprises by:

- Implementing the DEP programming model (C++ code)
- Evaluating the efficiency of the DEPM

As part of the new programming model proposed solution, we provide here an example code to demonstrate how someone could use DEPM (original code calculates the sum from a file):

Original MPI code:

```
sum = 0.0
MPI_File_open(..., &fh);
double *tmp = (double*) malloc(n*sizeof(double));
int offset = rank*n*sizeof(double);
MPI_File_read_at(fh, offset, tmp, n, MPI_DOUBLE, &status);
for(i = 0; i < n; ++i)
     sum += tmp[i];
```

DEP code:

```
sum = 0.0;
MPI_File_open(..., &fh);
#pragma dep file_handler(fh) input(n,rank) output() input(sum)
{
double *tmp = (double*) malloc(n*sizeof(double));
int offset = rank*n*sizeof(double);
MPI_File_read_at(fh, offset, tmp, n, MPI_DOUBLE, &status);
for(i = 0; i < n; ++i)
     sum += tmp[i];
}
```

In this project so far, we primarily focus on disk-to-disk sorting algorithm as a use case scenario while trying to design our programming model. The goal of this application is to sort a large amount of data (integers, key-value pairs, etc) using the limited memory available to the nodes in an HPC environment. An efficient implementation of this application should overlap computation needed to merge and sort the data in memory with I/O requests. Furthermore, on a uniformly distributed input data-set, sorting imposes a lot of communication messages among compute nodes. Therefore, This application touches all parts of DEP architecture, and will be a good candidate to start in order to characterize basic programming constructs for this architecture. We have defined and achieved the following goals:

- Understanding the performance limits of disk-to-disk sorting,
- Coming up with a performance model for sort in current HPC architecture,
- Characterizing bottlenecks and contended resources for this application.
- Design a new algorithm to address performance bottlenecks of sorting in DEP architecture,
- Performance modeling of the new proposed algorithm in the context of DEP,
- Implementing and evaluating the new algorithm on a modest size DEP cluster

Disk-to-disk sorting is known as BigSort among CORAL benchmark suite. The most recent work on disk-to-disk sort is DataSort project, which uses non-blocking point-to-point communication in MPI to overlap communication with computation and I/O requests. There is a PGAS implementation of DataSort that uses hybrid MPI+SHMEM model to use RDMA to optimize the communication in this application. Fortunately, the source code of all mentioned implementation is available. In this project, we started from BigSort source code, however, the final source code for DEP architecture will probably be a complete re-write to benefit all parts of the architecture. Following we present BigSort algorithm and our DEP algorithm. Finally, our expectations are listed at the end of this midterm report.

BigSort algorithm brief description:

The BigSort algorithm has two main phases.

1. Hash all elements to N bins, where N = # of nodes. Note that BigSort uses 1 MPI rank per node. By the end of this phase, each bin contains all elements falling within a sub range. For example, the first bin contains numbers ranging from 0 to T/N. Each bin is then stored as a "bin file". In this phase, each rank performs three steps:

a. Read a part (proportional to the DRAM size) of the input file (each rank reads a different part)

b. Hash this part locally according to the bins defined above.

c. Exchange elements among MPI ranks (AllToAll communication) to send them to the MPI rank that owns the corresponding bin.

d. Each rank writes the received elements to its "bin file".

2. Sort each "bin file" individually in each node. (Note: BigSort assumes the numbers are uniformly distributed so there is little concern for load balancing). No MPI communications. Each MPI rank treats its bin file as a collection of chunks, each chunk is of page_size (page_size is recommended to be no smaller than the page size of the file system).

a. Sort chunks in parallel (with OpenMP)

b. Merge every M chunks, repeat until there is only 1 chunk left. This chunk is the sorted bin file

3. Concatenate all sorted bin files together into a single file.

We examine this sorting algorithm to find that obviously the whole algorithm is I/O bound and most of the execution time is spent in reading and/or writing to the disk. It involves two clear phases: in the first phase it reads the data, does the binning and writes back the "bin files" to the disk. In the second phase, it again reads the "bin files", sorts individually those files, and then write them back to the disk. Next we point out how DataSort [11] has improved this algorithm by overlapping the I/O with the computation.

DataSort algorithm brief description:

The whole algorithm can be split into two main stages: the *read* and the *write* phases. See figure 2 for details. During the read stage, the read group processes sequentially read the data from the global file system and transfer this data to the sort group processes. The sort group processes, on receiving M records, bins these into q buckets much like BigSort. During the write stage, the flow of information is mostly reversed, with the sort group processes reading the q local files, one at a time, synchronized across all processes, sorting them globally, and then writing the final sorted data back to the global file system. We modeled the DataSort algorithm just to have a better understanding of what we should expect of our own DEP sorting algorithm.

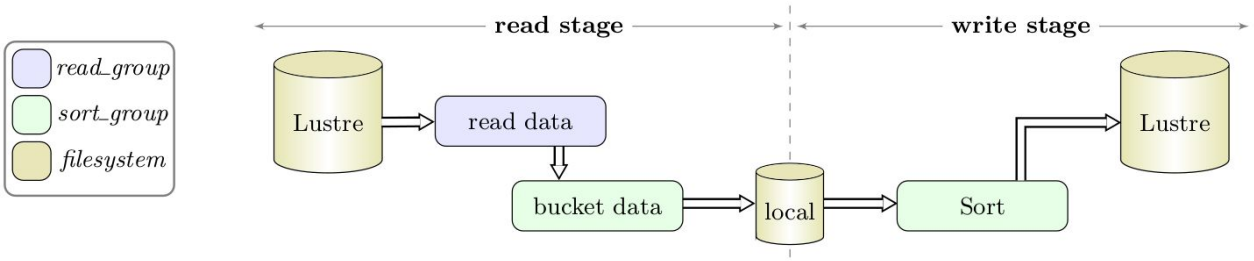

Figure 2, Overview of DataSort end-to-end sort process.

So, roughly speaking the DataSort algorithm comprises the following timings:

$$R+W$$

$$(r+b)+(r+m+w)$$

Where, R is the reading phase further analyzed as reading from the disk and do the binning and W is the write phase further broke down into read again the intermediate data, mergesort and write back the final result. Of course for out-of-core sorting problem, this routine happens in rounds or waves or runs, all with the same meaning.

DEPSort algorithm brief description:

In our design, we have two clear phases: the read and the write phase much like DataSort. But we have more than few changes coming from two different directions. First, DEP architecture suggests the existence of programmable nodes "in the middle" i.e. Compute-Side Data Nodes (CSDN) and Storage-Side Data Nodes (SSDN) which can be used to do reductions in the data movement from the shared file system and the compute nodes. We introduced *compression* and thus we are expecting a significant reduction in the volume of the data moved through the slower interconnect network. Second, the existence of those nodes allows us to change the flow of the sorting algorithm and use them to store any intermediate results, avoiding to touch the shared file system. Let us describe first the algorithm and at the end we will provide the benefits of our design and our expectations in terms of performance through a simple performance model we built. Note that we currently restrict this to integer sorting for the programming model building but it is easy to extend it to more general sorting. We also provide a figure of the components of the DEP architecture to easily follow the steps of the algorithm.

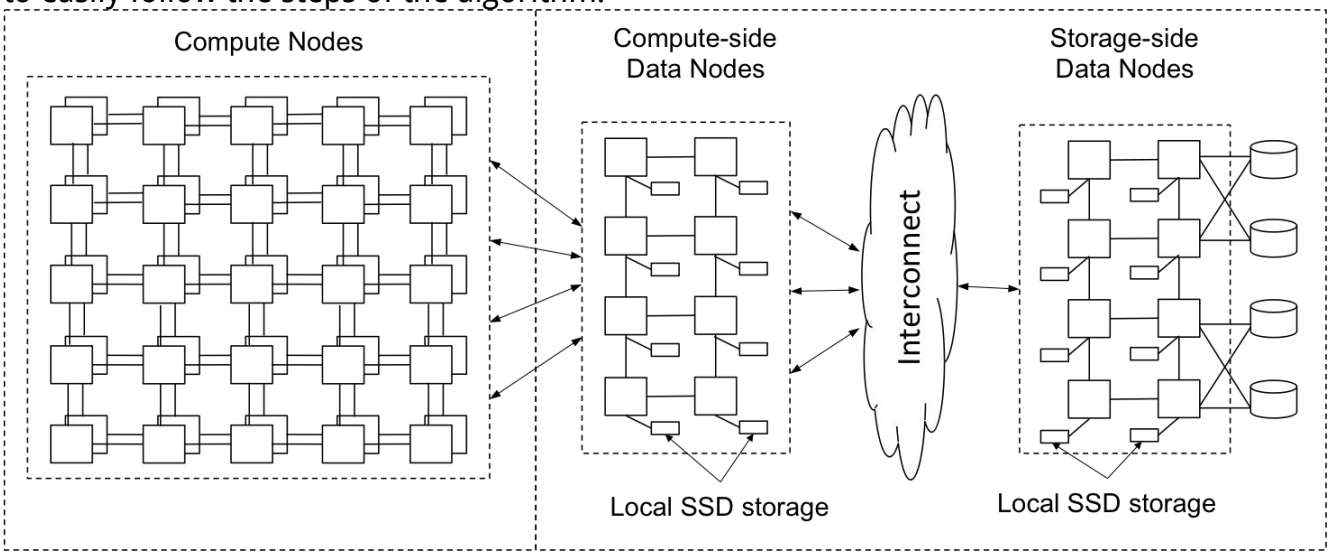

Figure 3, DEP architecture components

Read phase:

1. SSDNs start reading data from the shared file system in chunks (proportional to RAM size). Each MPI rank is issuing the read requests and using OpenMP threads it starts sorting them. Once it reaches a threshold (e.g. 2GB) it compresses the already sorted sequence and it sends it to the appropriate compute node. It keeps doing this, overlapping the reading, sorting and compressing until it finishes the allocated portion of the raw unsorted data.
2. Compute nodes upon receiving a compressed, sorted sequence it decompresses it and merge sorts it with existing sequences in memory. Once the memory reaches a threshold (e.g. almost full) it compress it and sends it to the CSDNs to store it as intermediate results.

Write phase:

1. Each compute nodes starts reading compressed, sorted, intermediate results previously stored in CSDNs. It executes a multi-way merge sort that produces the first piece of the final sorted data and ships it to the SSDNs after it compresses it once last time.
2. SSDNs upon receiving final compressed, sorted pieces it decompresses and stores them in the shared file system. Once every final piece arrives, they just concatenate the final output as a sorted file.

Note that we use a compression algorithm that has great speed and compression ratio. Specifically, after testing a lot of compression algorithms we chose the SIMD integer compression algorithm [12] that achieves 6307 MB/s per core compression rate and 4725 MB/s decompression rate. The ratio achieved is approximately ~9 bits per integer giving us almost 1/4 reduction in the size. Also note that the *memcopy* bandwidth in the same machine tested by STREAM [13] is 6842 MB/s.

Back to the performance model, according to DataSort, sorting 100TB, they achieved with 1792 machines on Stampede machine, a 1,24 TB/min improving the, at the time, Daytona record holder by 65%. Thus, it took roughly 4800 sec to sort this dataset. With reported read bandwidth of about 40GB/s and write bandwidth 120GB/s, the 4800 sec are split as follows:

R + W = Sort time --> 2300 + 2500 = 4800

Now, the reading time cannot be optimized because we still need to read the data from the disks. But the second timing of reading intermediate results, mergesorting them and writing them back as final results can be reduced by our DEPSort algorithm by roughly 1/4 due to the compressed data moving through the slow network plus the DEP architecture in where we store intermediate results to the SSDs of the CSDNs giving a total of approximately 3300 sec to sort the same dataset. This is just an expectation though and we test, evaluate, and compare the algorithms in the next section. Note, that since DEPSort touches all DEP components it can successfully be used to design the new DEP programming model and runtime system.

6. Evaluation

The evaluation of this project is comprised by the following:

- Comparison of programming effort by use case (number of lines of codes) in DEP programming model vs. naive decoupling code vs. code in existing architecture

- Performance comparison of DEP architecture code with existing architecture's implementation
- For the sake of these comparisons we use Sorting as our use case (scenario):

Measurements will be done for total execution time (sec) and programming effort (LOC).

Our testbed:

Linux Cluster with 65 nodes (IIT machine named HEC in SCS lab)

OS: Ubuntu 9.04

CPU: 8-cores on each chip

RAM: 8GB each node

HDD: 120GB HDDs

Network: Ethernet 10Gbit/s

PFS: OrangeFS 2.8.8 (previously known as PVFS2)

Dataset:

24GB of random integers (shuffled in 64 phases)

32GB of random integers (shuffled in 64 phases)

Configuration:

4 compute nodes - 8 pvfs2-servers

8 compute nodes - 8 pvfs2-servers

16 compute nodes - 8 pvfs2-servers

Programming effort

For this comparison, we simply examine the programming effort by counting the number of lines of code for BigSort vanilla, BigSort on DEP and DEPsort. Our goal here is to demonstrate how difficult is for a legacy code to be ported to DEP architecture. DEPsort is our implementation from scratch, taking into consideration the entire DEP architecture and its components. The following table summarizes the LOC for each version of disk-to-disk sorting. Note that this numbers do not include any makefiles or other supplementary files and it concerns only code used for the sorting algorithm.

	BigSort Vanilla	BigSort DEP	DEPsort
LOC	**2578**	**2764**	**1413**

Note that DEPsort is using compression and for this extra functionality we built a new version of compression library based on [12] which is a library for compressing 32bit sorted sequences of integers. We extended this large library (~45000 LOC) to 64bits integers.

It is clear from the table above that, changing a legacy code to be able to run on top of the

DEP architecture does not involve a tremendous programming effort with only some minimal changes. We demonstrate here the steps we changed for the BigSort algorithm. The sorting algorithm itself is untouched and we only modified it to use the DEP components.
BigSort Vanilla:

1) Binning Phase
   a. Read a part of the input file (each rank reads a different part)
   b. Hash this part locally according to the bin splitters.
   c. Exchange elements among MPI ranks (AllToAll) to send them to the MPI rank that owns the corresponding bin.
   d. Each rank writes the received elements to its “bin file”.
2) External Sort Phase: Sort each "bin file" individually in each node (in multiple waves)
3) Write back to the global sorted file

BigSort on DEP (minimal changes):

1) Binning Phase
   a. (SSDN) Read a part of the input file (each rank reads a different part)
   b. (SSDN) Hash this part locally according to the bin splitters.
   c. (SSDN → CN) Exchange elements among MPI ranks (DEP_AllToAll) to send them to the MPI rank that owns the corresponding bin.
   d. (CN → CSDN) Each rank writes the received elements to its “bin file”.

2) (CSDN x CN x CSDN) External Sort Phase: Sort each "bin file" individually in each node (in multiple waves)
3) (CSDN → SSDN) Write back to the global sorted file

The blue parts describe on which nodes of the DEP architecture was executed. Remember SSDN means Storage-Side Data Node etc. Please see DEPsort description for more details. These changes resulted in about ~150 more lines of code that are considered minimal.

DEPsort demonstrated the less LOC and our programming model seems to successfully abstract a lot of DEP specific functionalities and lets the user focus more on the sorting algorithm which was our goal in the first place.

Performance comparison

For the performance comparison we created a simple performance model trying to understand better where the performance gains were coming from. We start with BigSort Vanilla and we compare it with our own DEPsort.

BigSort algorithm:

Read Time + Binning + Writing Bin-Files+ Read Bin-Files + Sorting and Merging + Write back

DEPsort algorithm:

Read Time + Sorting + Compressing + Decompressing + MergeSorting + Writing Intermediate files + MultiWay MergeSort + Write back Final result

We break down the algorithms in these basic steps and we simplify the execution timings for each. We conducted our first test using 16 nodes in total while sorting 24GB of raw unsorted data.

BigSort algorithm:

(Read Time + Binning + Writing Bin-Files)=446.5 + (Read Bin-Files + Sorting and Merging + Write back)=920.4

Total time= 1366.9 sec

Read+Computation+Write= R+(7.5+2.9+8.0)+W= I/O time + Computation= 1349.0 +18.4

Clearly BigSort is I/O bound and its performance is relatively poor achieving an aggregate bandwidth of 19 MB/s!

We did the same test with DEPsort using 8 CNs, 4 CSDNs and 4 SSDNs again sorting 24GB of raw unsorted data.

DEPsort algorithm:

(Read Time + Sorting + Compressing)=248.9 + (Decompressing + MergeSorting + Writing Intermediate files)=220.4 + (MultiWay MergeSort + Write Final Result)=307.8

Total time= 556.7 sec

Due to ~¼ compression ratio on the data and the overlapping of the computation we get 2.45x performance improvement and DEPsort achieved an aggregate I/O bandwidth of about 45 MB/s.

Those initial results are encouraging and we proceeded to a full scale comparison. Our test case consists of sorting 32GB of raw unsorted data while using only 1GB of DRAM allocation for both algorithms since we wanted to make sure that the disk was touched multiple times and use memory as less as possible to emphasize the data-intensiveness of this problem. We scaled our comparison moderately using 32, 64 and 128 number of processes. For DEPsort the usage of the same resources is coming from different “families” of nodes but we made sure that the total resources used were the same. Please note that through the entire process of testing we encountered some major bugs in the BigSort code provided by Argonne, some of the bugs concerning pvfs2-posix calls and some in the code itself. It took me a long time to debug it and it is still extremely unstable in our environment (namely the HEC machine which is also old and unstable). Nonetheless below are some of the numbers we managed to collect. It is interesting to see that as the scale grows, the performance benefit is more significant. This is because more of those intermediate nodes from DEP architecture are being used making the data reductions and offering a better I/O throughput. The BigSort on DEP using our programming model demonstrated some gains but not as significant as expected. Nonetheless, we can still see the potential when using the model to run legacy code on top of DEP.

This is the point to report that due to significant bug challenges and the shortage of time we didn’t compare our algorithm with DataSort described above. This was our initial target to

be honest but we failed to meet our goal even with the 2 day extension granted. As suggested in the final presentation by the instructor, after we refine and tune our implementation, we consider comparing it not only to BigSort and DataSort (two implementations coming from HPC) but also with Spark or Hadoop from the distributed community (currently holding the sorting benchmark record). We don't know the results of that, but we are optimistic that we can at least offer a decent "fight" against them.

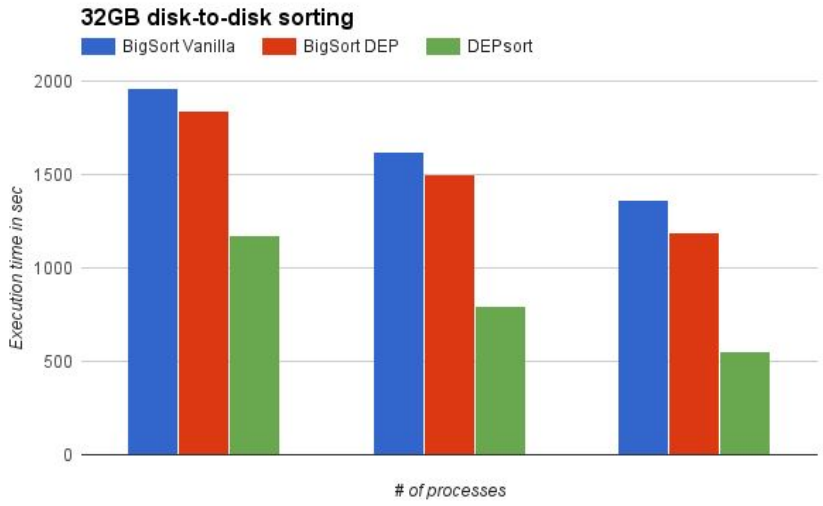

Figure 1, Total execution time

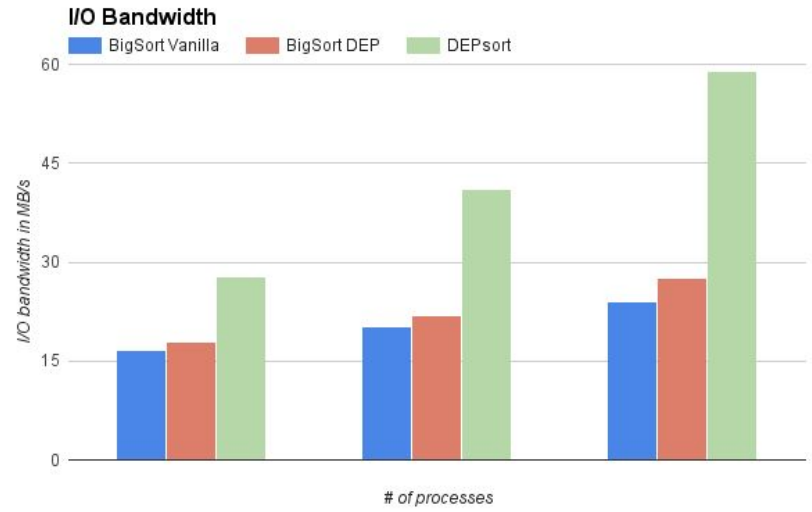

Figure 2, I/O Bandwidth in MB/s

7. Conclusions

Designing a good programming model is just as important as designing a good user interface (UI) and, not so coincidentally, many of the principles used in UI design can be directly applied to programming model design. Good UI design seeks to let the user work at a much higher level of abstraction than the internal implementation. In the same way, a good programming model expresses its functionality in a way that matches how the developer wants to think about it. A good programming model doesn't just expose internal

structures—it exposes its functionality at a higher level of abstraction so that the developer can concentrate on what he/she wants to do and not on how one has to accomplish a simple task.
Throughout the entire project:

i. We carefully examined the system architecture of existing HPC machines through case studies
ii. We analyzed the DEP architecture in order to extract knowledge on how we can program on top of it
iii. Designed a prototype programming model for DEP architecture and incorporate a few inherent optimizations
iv. Built a performance model to evaluate our approach.
v. Evaluated our model against a traditional "vanilla" code for a classic data-intensive application like sorting.

As a future work we intend to examine more applications as use cases for building our programming model, enrich the model with some optimizations such as pipelining, memory management and automatic compression and finally to evaluate the model extensively through more micro-benchmarks and real applications.
Our programming model is designed with common scenarios in mind. What will developers typically want to do with our component? Which features are more important for advanced users and which features should be very easy to use? Note that the focus in this project is mostly on the features and not on the internal objects, data structures, or functions. We showed that with little change in legacy data-intensive code, our DEP programming model can achieve significant benefits in the performance. We also conclude that DEP architecture can really help alleviate some of the data-intensive tasks and with our programming model it will become even more easy to implement various algorithms on top of DEP, and we intend to do so.
Finally, some of the lessons learned from this project is that even a simple and very well examined task as sorting can impose serious performance limitations to current systems. Also, designing a programming model is not an easy task and it feels more like predicting what will users want out of the new architecture rather than what do you as a system designer want them to use. Finally, as I was the only student involved in this project, another lesson I've learned is to collaborate with other people into teams whenever you have the chance! It might seem obvious but initially it wasn't to me and so now I know!

8. Additional Resources

Deliverables

- One final report in Word and PDF form.
- One final Powerpoint presentation in PDF format.
- Source code: https://bitbucket.org/akougkas/cs554_dep_sorting .

References

[1] A. Choudhary, W. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham, “Scalable I/O and analytics,” *J. Phys. Conf. Ser.*, vol. 180, p. 012048, Jul. 2009.

[2] P. Roadmap, J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.- Claude, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, K. Hotta, Y. Ishikawa, Z. Jin, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. Mueller, W. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, and M. Valero, “The International Exascale Software.”

[3] Y. Chen, C. Chen, X. H. Sun, W. D. Gropp, and R. Thakur, “A decoupled execution paradigm for data-intensive high-end computing,” *Proc. - 2012 IEEE Int. Conf. Clust. Comput. Clust. 2012*, pp. 200–208, 2012.

[4] C. Chen and Y. Chen, “Dynamic active storage for high performance I/O,” *Proc. Int. Conf. Parallel Process.*, pp. 379–388, 2012.

[5] J. Piernas, J. Nieplocha, and E. J. Felix, “Evaluation of active storage strategies for the lustre parallel file system,” *Proc. 2007 ACM/IEEE Conf. Supercomput. (SC ’07)*, no. 1, 2007.

[6] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W. K. Liao, and A. Choudhary, “Enabling active storage on parallel I/O software stacks,” *2010 IEEE 26th Symp. Mass Storage Syst. Technol. MSST2010*, 2010.

[7] J. Saltz, “Active Disks : Programming Model , Algorithms and Evaluation.”

[8] S. Kumar and M. Blocksome, “Scalable MPI-3 . 0 RMA on the Blue Gene / Q Supercomputer,” pp. 7–12.

[9] J. Dean and S. Ghemawat, “MapReduce : Simplified Data Processing on Large Clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 1–13, 2008.

[10] A. Basumallik, S.-J. Min, and R. Eigenmann, “Programming Distributed Memory Systems Using OpenMP,” *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS 2007)*, pp. 1–8, 2007.

[11] https://github.com/hsundar/datasort

[12] https://github.com/lemire/SIMDCompressionAndIntersection

[13] http://www.cs.virginia.edu/stream/