Processing Large-scale XML Files on GPGPU Cluster

Liu, Ping

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
10
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2015-03-24
(URL)
https://doi.org/10.15002/00011664

Processing Large-scale XML Files on GPGPU Cluster

Ping Liu
Graduate School of Computer and Information Sciences, Hosei University
ping.liu.4f@stu.hosei.ac.jp

***Abstract*—XML has been used as a textual data format for transporting and storing information in many areas. However, the cost to process the large-scale XML file will become a serious issue for general processing methods. In this paper, we propose a design and implementation of a large-scale XML processing system on GPU cluster to address the processing performance issue. This system cooperates CPU and GPGPU to the master-slave architecture for processing the XML file. The processing consists of two phases, structure extracting, and tags parsing. The structure extracting uses multiple threads to read the file and recognize the document structure, tags parsing will take advantage of GPGPU to get every tag's name and attributes using the location information got in structure extracting phase.**

Keywords—XML ; CUDA; GPGPU; performance

I. INTRODUCTION

As a semi-structured language, XML has been increasingly used for data transporting and storing, such as online data, logs, configuration file, content-based database and company documents. However, when the XML file gets too large it will become a serious issue to process the large-scale XML file, because analyzing the file from beginning to end will became a nightmare for general processing methods. A number of approaches [1] [2] [4] [5] [6] have been used to address these performance concerns.

Meanwhile, the General-Purpose Computing on Graphics Processing Units (GPGPU) have rapidly evolved to become high performance accelerators for data-parallel computing. Modern GPGPUs contains hundreds of processing units, which makes them well suitable for many data-parallel computing tasks. Moreover fortunately NVIDIA has provided a Compute Unified Device Architecture (CUDA) [11] [12] for programming for GPGPU, which makes it easier to program and possible to optimize the program.

The GPGPU's parallel architecture is suitable for data-parallel computing, which needs to run a lot simple same tasks at the same time. This is the reason why GPGPU is qualified for the work like matrix operation or bit map processing. Nevertheless, GPGPU is not considered suitable for the problem, which contains a complicated logical control work like analyzing the structure of a XML file, because the analyzing work is based on machine abstract while the machine work is hard to be transformed into data-parallel tasks. This is the challenge for using GPU to processing XML files.

In this paper, we propose a design and implement for processing large-scale XML files on GPGPU cluster. This system operates CPU and GPGPUs together into master-slave architecture for processing the XML file. Our processing consists of two phases, structure extracting and tags parsing. In structure extracting phase we use multiple CPU threads to read the file and recognize the document structure, using the location information got in structure extracting phase, we take advantage of GPGPUs to parse every tag's detail, tag name and attributes, in tags parsing. The implementation of our system is based on CUDA and C, and the communication between master and slaves is based on Message Passing Interface (MPI) [10]. Our system gains almost 2 times speedup than single CPU implementation on processing file larger than 3GB.

II. RELATED WORK

XML processing is based on machine abstraction, such as NFA, and the basic idea of using parallel way to process XML is dividing the document into several parts. However, arbitrary division will break the structure information and influence the state of the parsing machine at the beginning of each chunk. The PXP (Parallel XML Parsing) [1] uses a pre-parsing work to determine the XML document structure, which is then used to divide the XML document such the divisions between the chunks occur at well-defined points in the XML grammar. Nevertheless, this is counterproductive because XML parser spends a large percentage of time tokenizing the file in an inherently serial process, typically running a deterministic finite automaton on the input, although PXP have tried to make the pre-parsing light as possible. Besides this, the PXP take advantage of multiple threads to do the further parsing work, the amount of threads is limited by the system resource, although increasing threads can achieve more performance, it seems hard to use enough threads to get peak performance. Therefore, because of the system resource limitation, the PXP is not suit for processing large-scale XML files. The memory in one machine is hard to maintain the process results for large-scale XML file either. Michael R.Head also explored new techniques for parallelizing parsers for very large XML documents [4]. He does not focus on developing parallel XML parsing algorithm, but dividing XML parsing process into several phases, such as XML data loading, XML parsing and result generation, and then scheduling working threads to execute each parsing phase in a pipeline model. Besides, the design of combining CUDA and MPI [7] [8] has occurred in recent years.

III. Achitecture and Design

In order to process file effectively, large-scale XML processing is executed as pipeline. The whole processing is divided into two stages, structure extracting and tags parsing in the system. These two stages can be executed as pipeline based on parsing tasks, which means structure extracting inputs the XML file in parallel way and outputs the parsing tasks managed by task manager, the stage Tags parsing inputs the parsing tasks and outputs parsed results, which will be stored in different nodes. After all structure extracting is done, we will merge the whole document structure information in master node and store detailed information in every slave nodes. Both the structure extracting stage and the Tags parsing stage can apply data-parallel mode respectively to exploit more parallelism.

To take advantage of multiple CPUs and GPUs, we try to find out the parallelizable part in full processing work. We use a number of file reader to read the file in parallel way and each reader contains a parser to recognize tags in the file stream s show in Figure 1. The readers and parsers run on the CPU for extracting document structure. The GPUs will parse tags' names and attributes in tags parsing phase.

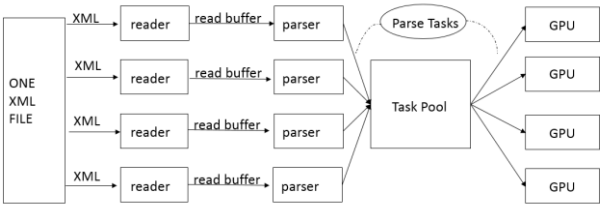

Fig. 1. System Architeture

In the architeture showed in Figure 1, readers and parsers run on the CPU, they are responsible for extracting document structure.

A. *Structure extracting*

Structure extracting is designed for extracting tags' location information and packing the information into parsing tasks, detecting tags' location is a specific step for extracting structure. In structure extracting stage, the performance to read the XML file through I/O is a big bottleneck for our system. Therefore, we design the parallel read to scan a large-scaled XML document in parallel way. The XML document is divided into chunks so that each reader works on a chunk independently. One of keys to the parallel reading is how to divide XML document without seeing the whole document. A chunk may start in the middle of some string whose context and grammatical role is unknown. For example, a chunk may start as a part of element name or attribute or text value.

We propose a method called, Broken XML Seaming, to address this issue. Broken XML Seaming can recognize the broken elements, caused by dividing XML file, at the end of the chunk and heal the broken element automatically.

After dividing the XML file, we should parse the partial XML Information set in every chunk. The parsing work includes searching out all tags and building structure. Actually in the readers mentioned above, there is a corresponding parser embedded for handling the XML stream, consequently the parsing work is parallel as the reading work.

1) *Broken XML Seaming*

When we divide the file into equal size chunk, we will meet an obvious problem, *broken tags*, which involves partial tag in the end of chunk. For example in Figure 2, a start tag is split into two broken parts, in this case the prior part of the start tag belongs to thread 1 but the rest part belongs to thread 2. We design an auto-complement method to *heal* the broken tag.

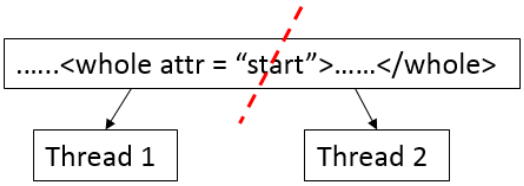

Fig. 2. Broken tag caused by we simplly split the file into equal parts. The left part is read by thread 1 and the right part is read by thread 2.

To heal the broken tag, we need to recognize the broken tag first. Because we prepare a NFA to recognize tags, we can figure out whether the character belongs to a tag, when thread finishes parsing all the characters in current chunk and the state in Figure 3 is *Content*, it means that there is no broken tags; otherwise, the last few bytes compose a broken tag. When a broken tag appears, the thread continues to read characters until the NFA state changes to *Content*, and then broken tag is healed.

2) *Searching tags*

The output of parallel parser in this phase is the tags' location information and partial tree structure responding to the chunk read by reader, we have to mention that the structure extracting will not get tags' names or attributes to make the structure extracting work as light as possible. As a result, the node in the tree structure is unnamed and we identify them with IDs. For recognizing tags, we took advantage of a NFA designed in [1] to recognize tags. We quote the automaton in Figure 3, here we just need to restore the offset and length of every tag but ignoring the text parts of nodes. To identify the tag, we give every start tag or whole tag an ID.

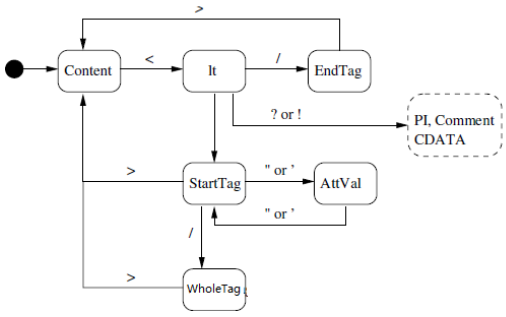

Fig. 3. NFA for recognizing tags.

3) *Building structure*

Supervisor: Prof. Toshio Hirotsu

Because we have divided the file in to chunk, we will build the partial tree (P-Tree) in each chunk. P-TREE contains the structure information of the chunk and it will be merged with others later. Therefore, while we create the P-TREE we also need to do some prepare work for merging.

According to the automaton in figure 2, we can recognize three types of tag, start tag, end tag and whole tag. The example of each kind of tag is showed in TABLE I:

TABLE I. TAG TYPES

Tag types	***illustrate***
Start Tag	<tag attr='a'>
End Tag	</tag>
Whole Tag	<tag/>

Three tag types we can recognize from Fig.3

When we meet a start tag, we push it into a stack, and when we meet an end tag, we pop a start tag from the stack, which should be paired with the end tag. Moreover, the start tag's parent should be the top element of the stack now. We show the process in Figure 4. When we meet a whole tag, we treat it as a start tag and an end tag like above.

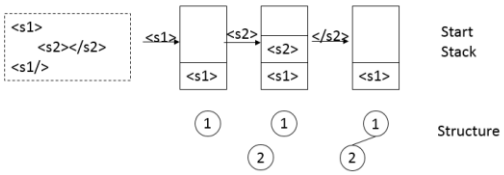

Fig. 4. Process to build the P-TREE.

The algorithm for creating the structure of file in each thread is showed below:

1) *Create a stack S and an List L for start tags*
2) *If meeting a start tag P, push it into S*
3) *If meeting an end tag, pop a start tag Q from S, set the parent of Q as the top element of S if S is not empty, add Q to L.*
4) *If meeting a whole tag W, set the parent of W as the top element of S if S is not empty.*

After searching tags, there will be a tree structure and a list of start tags in each thread. Up to now, the start tags in every thread only include the location information and an ID, and the tree structure is the structure of partial XML document, next we need to parse the details of every start tag and merge the tree structures in all threads into a whole structure.

While searching out all the tags location in structure extracting, we have built the P-TREE structure about the partial document in every thread, to merge the structures into a whole structure, we will figure out the relationship between current structure and the structure in prior thread. We take advantage of the ill-formed exceptions [2] caused by splitting file to merge the structures. The ill-formed document is showed in Figure 5, the whole node's start tag and end tag are divided into different chunks.

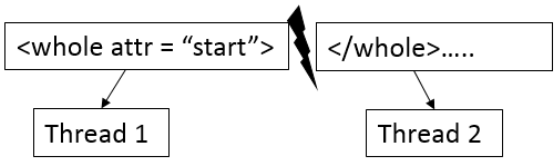

Fig. 5. Ill-formed partial XML document caused by splitting. The start tag of *whole* node belongs to thread 1, but its corresponding end tag belongs to thread 2.

There are two types of ill-formed structure in Figure 5.

Unresolved start tag: which there is no paired end tag in current chunk, like the whole start tag, <whole attr="start">, in thread 1 in Figure 5.

Unresolved end tag: which has no paired start tag in current chunk, like the whole end tag, </whole>, in thread 2 in Figure 5.

Then we will use the number of unresolved start tags and unresolved end tags before a node to find the parent of this node.

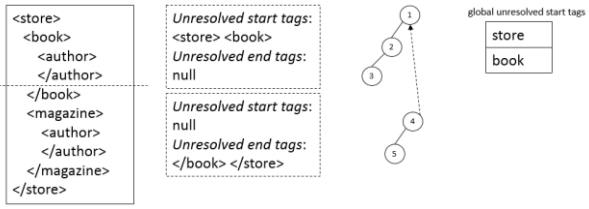

Fig. 6. Process to merge two partial structure.The parent of magazine node is unknown. In fact we still don't know the name of tags, in the structure tree, we use an ID to identify a node. 1 is store, 2 is book, 3 is author, 4 is magazine, 5 is author in second chunk.

In Figure 6, we try to find the parent of node magazine in second node. First, we add the unresolved start tags to the global unresolved tags. Then the number of global unresolved start tag in first chunk is 2, the number of unresolved end before magazine is 1, which means there is 1 unresolved start tag between magazine node and its parent. We use the number of global unresolved start tags, which is 2, to minus the number of unresolved end tags before magazine, we will get the order number of its parent in global unresolved start tags, which is 1, this means the first unresolved start tag is the parent.

To count the number of unresolved start tags and the number of unresolved end tags before a node in each chunk, we need to modify the algorithm of building partial tree. The modified algorithm is showed in Figure 7.

```
Data: start tag stack: S , start tag list: L, n_unresolved_start, n_unresolved_end
WHILE reading the file
    current tag T ← parser
    IF T is a start tag
     THEN
        push T to S
    ELSE IF T is an end tag
      THEN
        IF S is empty THEN n_unresolved_end+1
        ELSE
          tag P ← S.pop
          add P to L
          tag O ← S.top
          IF O is empty
            the number of unresolved end tag before P ← n_unresolved_end
          ELSE
            P.parent ← O
n_unresolved_start ← the number of start tags left in S
```

Fig. 7. Algorithm for creating partial tree.

The next is the algorithm to merge the partial tree in each chunk.

```
Data: global unresolved start tags stack: G, global_n_unresolved_start
G ← the start tag stack in first chunk
global_n_unresolved_start ← n_unresolved_start in first chunk
FOR each chunk from 2 to n
    FOR each tag T in current chunk
        IF T .parent == null
          THEN
            n ← global_n_unresolved_start – the number of unresolved end tag before T
            T.parent ← the nth element of G from bottom
    m ← n_unresolved_end in current chunk
    pop m elements from G
    add the unresolved start tags in current chunk to G
    global_n_unresolved_start← global_n_unresolved_start – n_unresolved_end in current chunk
```

Fig. 8. Algorithm for merging partial trees in all threads.

B. *Tags Parsing*

In tags parsing phase, we use GPUs to parse the tags' details. GPU's parallel multi-core architecture allows it run hundreds or thousands of threads simultaneously. The threads are grouped into blocks and the blocks are grouped into grids. The shared memory in each bock can be accessed by all the threads in current block. Moreover, the CUDA makes it possible for programmer to specify the number of blocks as well as the number of threads per block to find optimal number of GPU thread to be used. After receiving the parsing task. What GPU thread parses will always be a well-formed tag rather than unformed text thanks to the location information of every tag in parsing task. Then every GPU thread uses the NFA in Figure 10 to parse the tag name and attributes.

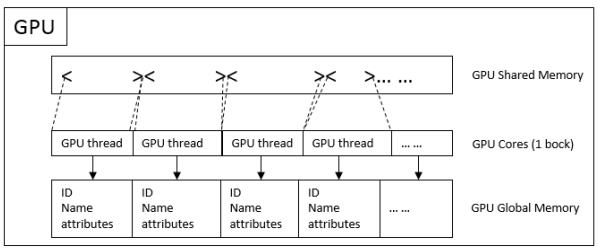

Fig. 9. GPU work in slaves. Every GPU thread only parse one tag for tag name and attributes.

We show the parsing process in GPU in Figure 9. After loading data, one GPU thread just need to parse a little piece of XML stream, and this piece of XML file is a well-formed tag. As a result, we can simplify the NFA for parsing the tags in every core. Figure 10 shows the NFA for parsing tag name and attributes for a well-formed tag.

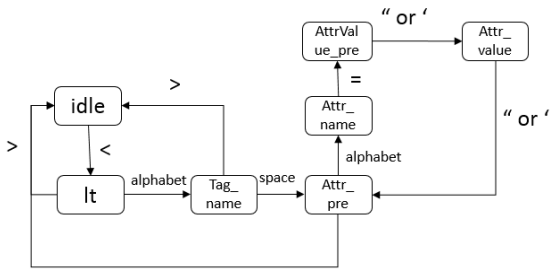

Fig. 10. NFA for recognizing tag name and attributes.

What GPGPU does in this phase is running thousands GPU threads to execute the NFA in Figure 10 to parse tags' name and attributes, one thread parse on tag, then copy the result to memory and store it.

About the NFA in Figure 10, there are no state for text or content, because the data ready for Tags Parsing is prepared as tag format according to the tag location information got in structure extracting.

C. *Parse Task Management*

While we extract document structure, we need to pack the location information for tags parsing. We design a structure, parsing task, containing the location information of thousands of tags, which can be transferred to slaves by MPI quickly because of the light data volume in one parsing task.

On the other hand, we apply the produce-consumer module to parsing task management, the structure extracting will produce tasks and the slaves will consume the task. This will guarantee the master and slaves working in asynchronous way.

In this architecture, the time breakdown of full-parsing work should be like Figure 11.

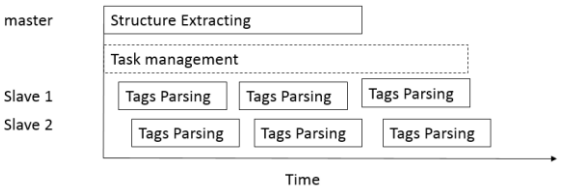

Fig. 11. System time breakdown. The x axis indicates the time.

IV. EXPERIMENT AND DISCUSSION

First, we performed experiment to compare the processing performance with SAX method, and then performed experiments to measure the full parsing performance. The experiments were conducted on a cluster containing CPU and

GPUs, one master and three slaves. Every node is equipped with one Intel Xenon Processor E5-2620(15M Cache, 2.00 GHz, 6 cores) CPU, and one NVIDIA Tesla M2075 GPU. The MPI version complied on the nodes is Open MPI 1.6.3. All the experiments used Wikimedia archive (http://dumps.wikimedia.org/enwiki/latest/) as data input, the file size are 287MB, 446MB, 987MB and 3400MB.

To prove our system is able to achieve a better performance when processing a large-scale file. We performed experiment to compare the execution time with general Simple API for XML (SAX) method.

Next, we measured the time costing for full parsing including structure extracting, task transporting, and Tags parsing work in slaves. Every test was run two times but the measurement of first time was discarded to avoid cached data influencing.

At last, we show the time breakdown for the full parsing.

A. Performance Comparison with SAX

Our original intention to design this system is to achieve speedup of processing big-scale XML file compared with sequence method based on CPU. To prove the speedup of our system, we measured the performance difference between the CPU method and our GPGPUs cluster in Figure 12.

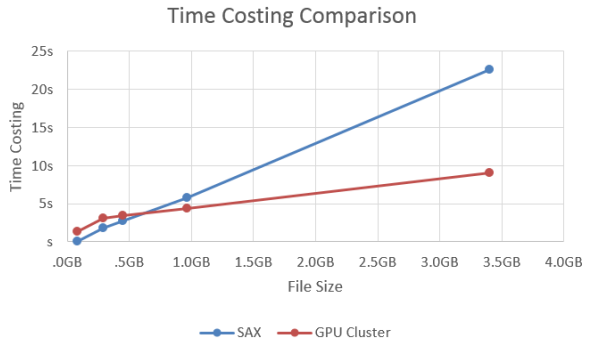

Fig. 12. Experiment for performance comparing between SAX and GPU Cluster in processing XML file. Two slaves nodes were used in this experiment.

We saw that the performance of SAX was better than that of GPU cluster when the file size is less than 500MB. GPU cluster got a better performance in 976MB than SAX and kept performing better. In fact, the SAX parsing is a sequence job, and it will only use CPU to do the parsing work. We got almost 2.5 times speedup than SAX method.

B. Full Parsing Performance

Full parsing work includes the structure extracting, management and tags parsing work. The full parsing is exactly what our system can do, and its performance indicates the system's performance. Because there are two phases, structure extracting and tags parsing, in our full parsing work, we designed two experiments, which make the performance of these two phases permanent respectively in different experiment.

We made the tags parsing performance permanent with using 2 slaves and 1024 GPU threads in each node in experiment showed in Figure 13. The performance is measured as speed of parsing file, the data size parsed per second. The threads applied for structure extracting would influence the structure extracting work, which would influence the whole system's speed thereby. Therefore, we used different number of threads for structure extracting. Full parsing work includes the structure extracting, management and tags parsing work. Moreover, in our system, the manage work, tasks management and MPI invoking, will cause a loss to file parsing performance, to make up this performance lose will measure the speed for parsing larger and larger size of files.

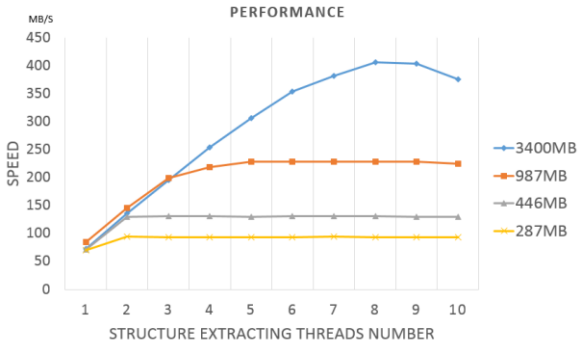

Fig. 13. Performance for full parsing, the different color line indicate different size of file, X ax

According to this experiment in Figure 13, we can figure out that the speed will increase as the structure extracting threads and file size increasing, and then reach a peak.

Next, we made the structure extracting performance permanent, but changed the slave nodes and the GPU threads used in each node. In fact, we used ten threads for structure extracting to keep the best extracting performance. Then we measured the time cost for processing a 967MB file, and the results is showed in Figure14.

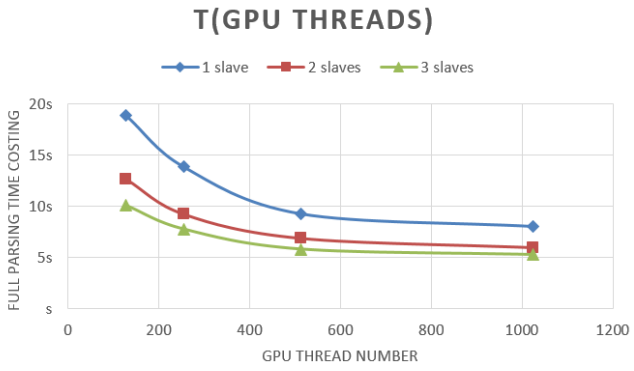

Fig. 14. Time cost for full parsing with using different number of slaves and different GPU threads in each slave. The x indicates the GPU threads number and the y indicates the time cost.

From Figure14 we saw that the time cost for processing file got less as the number of GPU threads increasing. The more

GPU threads used meant the better tags parsing performance, the better tags parsing performance caused better full parsing performance. More slave nodes would increase the full parsing performance, this could be explained by that more slave nodes increased the total GPU threads in tags parsing phase thereby improved the full parsing performance.

On the other hand, when we used about 512 threads in each GPU, the performance almost got best, and stopped improving obviously. This fit well to the GPU's cores number 448, which meant we did not waste GPU threads in prior experiment in which we used 1024 threads.

C. *System Breakdown*

Loading balance is an important element for a cluster system, so we measure the time breakdown of the system to test the cluster's load balance. On the master, we run the structure extracting for recognizing tags and document structure to produce parsing tasks for slaves. At the same time, slaves consume the parsing tasks and store the parsing results on the slaves. In theory, the master and slaves will work parallel, and the best case is that master and slaves start and stop almost at the same time. To balance the load, we can adjust data in every parsing task, the number of tags every parsing task contains, which will influence the tags parsing task speed. Moreover, because our parsing tasks are managed by a task pool, we get the peak system performance when the speed of producing tasks, which is performed by master, equals to the speed of consuming tasks, which is performed by slaves.

Here we tried to parse a 976MB file, and we set that one parsing task contains 1024 tags. The time breakdown about the two parsing phases is show in Figure 15.

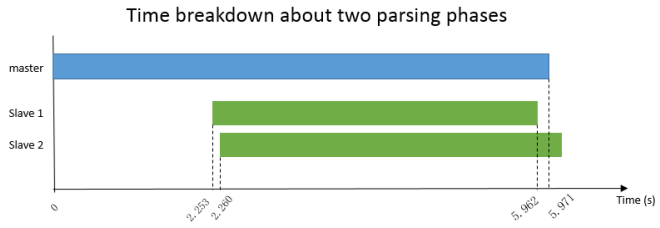

Fig. 15. System full parsing breakdown.

The slaves almost stopped at the same time with master in this experiment, this meant that we got a nice system load balance.

Compared to other approaches, our approach tries to parallelize the structure extracting stage to exploit multiple threads in this phase, as we discovered the structure extracting overhead is considerable especially after we improved the parsing performance by processing file on GPGPUs cluster. Our algorithm is task-based and each task contains thousands of tags, and these tags should be parsed by GPUs in slaves. The produce-consumer mode is used in the tasks management and therefore complementary to our approach. Moreover, our paper is the first one using the GPGPUs cluster to processing large-scale XML file. The performance evaluation results show the performance benefits from this system.

V. CONCLUSION

In this paper, parallel implementations were presented of using the CUDA toolkit. Both the serial and the parallel implementations were compared in terms of running time for different file sizes and number of threads. It was shown that the parallel implementation of the algorithms was up to 2 times faster than the serial implementation, especially when larger text and smaller pattern sizes where used. In addition, it was discussed that in order to achieve peak performance on a GPU, the hardware must be as utilized as possible and the shared memory should be used to take advantage of its very low latency.

Future research in the area of large-scale xml parsing and GPGPU parallel processing could focus on the file system study for the parallel implementation to store final parsing results in slaves, such as transferring the invert index into file format. Moreover, it would be interesting to examine the performance of the algorithms when executed on multiple GPUs and on hybrid MPI and CUDA clusters. Finally, further optimization of the parallel implementation of the algorithms could be considered to make better use of the GPUs capabilities, including loop unrolling, matrix reordering and register blocking in addition to smarter use of the shared memory.

REFERENCES

[1] Wei Lu, Kenneth Chiu, Yinfei Pan, "A Parallel Approach to XML Parsing," Grid Computing Conference 2006

[2] Yu Wu, Qi Zhang, Zhiqiang Yu, and Jianhui Li"A Hybrid Parallel Processing for XML Parsing and Schema Validation," Balisage: The Markup Conference 2008

[3] Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis, "String Matching on a multicore GPU using CUDA", Informatics, 2009. PCI '09. 13th Panhellenic Conference

[4] Zacharia Fadika, Michael R. Head, Madhusudhan Govindaraju, "Parallel and Distributed Approach for Processing Large-Scale XML Datasets", 10th IEEE/ACM International Conference on Grid Computing 2009

[5] Kevin Lü, Yuanling Zhu, Wenjun Sun, Shouxun Lin and Jianping Fan, "Parallel Processing XML Documents", Proceedings of the International Database Engineering and Applications Symposium (IDEAS'02) 2002

[6] Wei Zhang and Robert van Engelen,"A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services", IEEE International Conference on Web Services 2006

[7] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales and D. K. Panda, "Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-Process Communication", IEEE computer society 2012

[8] Sam White, Niels Verosky, Tia Newhall, "A CUDA-MPI Hybrid Bitonic Sorting Algorithm for GPU Clusters", 2012 41st International Conference on Parallel Processing Workshops.

[9] W3schools, XML Totutorial, http://www.w3schools.com/xml/default.ASP

[10] Andrew Lumsdaine, Joshua Hursey, Jeffrey M. Squyres and Abhishek Kulkarni, Open MPI Tutorial

[11] NVIDIA Inc. CUDA C Programming Guide

[12] NVIDIA Inc. CUDA, http://www.nvidia.com/object/cuda_home_new.html