CHARACTERIZING DISTRIBUTED XML PROCESSING
Moving XML Processing from Servers to Networking Nodes

Yoshiyuki Uratani
Graduate School of CS and Systems Engineering, Kyushu Institute of Technology, Iizuka, 820-8502 Fukuoka, Japan

Hiroshi Koide
Faculty of Computer Science of Systems Engineering, Kyushu Institute of Technology, Iizuka, 820-8502 Fukuoka, Japan

Dirceu Cavendish, Yuji Oie
Network Design Research Center, Kyushu Institute of Technology, Iizuka, 820-8502 Fukuoka, Japan

Keywords: Distributed XML processing, Task scheduling, Pipelining and parallel processing.

Abstract: This study characterizes distributed XML processing on networking nodes. XML documents are sent from a client node to a server node through relay nodes, which process the documents before arriving at the server. When the relay nodes are connected tandem, the XML documents are processed in a pipelining manner. When the relay nodes are connected parallel, the XML documents are processed in a parallel fashion. Well-formedness and grammar validation pipelining and parallel processing characterization reveals inherent advantages of the parallel processing model.

1 INTRODUCTION

XML technology has become ubiquitous on distributed systems, as it supports loosely coupled interfaces between servers implementing Web Services. Large XML data documents, requiring processing at servers, may soon require distributed processing, for scalability. Recently, distributed XML processing has been proposed and studied from an algorithmic point of view for well-formedness, grammar validation, and filtering (Dirceu Cavendish, 2008). In that work, a Prefix Automata SyStem is described, where (PASS) nodes opportunistically process fragments of an XML document travelling from a client to a server, as a data stream. PASS nodes can be arranged into two basic distributed processing models: pipelining, and parallel model. The problem of allocating specific fragments of an XML document to PASS nodes, in a pipeline manner, has been addressed in (Yoshiyuki Uratani, 2009). We leverage their results into an efficient task allocation method in this current work. Moreover, the problem of scheduling tasks on streaming data that follows parallel paths has been addressed in (Kazumi Yoshinaga, 2008). In that work, we have also studied the migration of tasks between nodes depending on network congestion.

In this paper, we evaluate the two distributed XML processing models - i) pipelining, for XML data stream processing systems; ii) parallel, for XML parallel processing systems - with regard to processing efficiency, correlating performance with XML document structure and processing task for well-formedness and grammar validation tasks. The paper is organized as follows. In section 2, we describe generic models of XML processing nodes, to be used in both pipelining and parallel processing. In section 3, we describe a task scheduler used to assign XML document fragments to XML processing nodes. In section 4, we describe a PC based prototype system that implements the distributed XML processing system, and characterize XML processing performance of the pipeline and parallel computation models. In section 5, we address related work. In section 6, we summarize our findings and address research directions.

Uratani Y., Koide H., Cavendish D. and Oie Y..
CHARACTERIZING DISTRIBUTED XML PROCESSING - Moving XML Processing from Servers to Networking Nodes.
DOI: 10.5220/0003338300410050
In *Proceedings of the 7th International Conference on Web Information Systems and Technologies* (WEBIST-2011), pages 41-50
ISBN: 978-989-8425-51-5

2 XML PROCESSING ELEMENTS

Distributed XML processing requires some basic functions to be supported:

- **Document Partition.** The XML document is divided into fragments, to be processed at processing nodes.
- **Document Annotation.** Each document fragment is annotated with current processing status upon leaving a processing node.
- **Document Merging.** Document fragments are merged so as to preserve the original document structure.

XML processing nodes support some of these tasks, according to their role in the distributed XML system.

2.1 XML Processing Nodes

We abstract the distributed XML processing elements into four types of nodes: StartNode, RelayNode, EndNode, and MergeNode. The distributed XML processing can then be constructed by connecting these nodes in specific topologies, such as pipelining and parallel topologies.

`StartNode`. StartNode is a source node that executes any pre-processing needed in preparation for piecewise processing of the XML document. This node also starts XML document transfer to relay nodes, for XML processing. We show components of the StartNode in Figure 1. The StartNode has one or more threads of the type Read/SendThread. In Figure 1, StartNode has three next nodes. The Read/SendThreads each reads part of an XML document per line, and add checking and processing information. The checking information is used for tag check processing. Processing information describes which RelayNode shall process which parts of the XML document. Each Read/SendThread reads part of the document roughly of the same size. Then, each Read/SendThread sends the processed data to next nodes. Read/SendThreads run concurrently.

`RelayNode`. RelayNode executes XML processing on parts of an XML document. It is placed as an intermediate node in paths between the StartNode and the EndNode. We show components of the RelayNode in Figure 2. The RelayNode has three types of threads: ReceiveThread, TagCheckThread and SendThread. The ReceiveThread receives data which contains lines of an XML document, checking and processing information, storing the data into a shared buffer. The TagCheckThread attempts to process the data, if the data is assigned to be processed at the node. SendThread sequentially sends data to a next node.

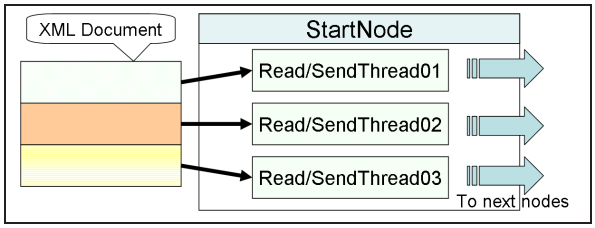

Figure 1: Components of StartNode (three next nodes case).

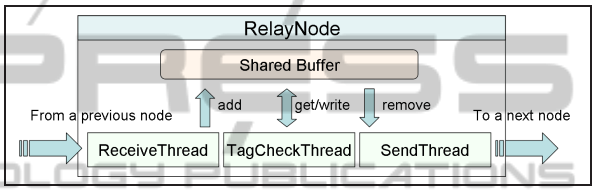

Figure 2: Components of RelayNode.

`EndNode`. EndNode is a destination node, where XML documents must reach, and have their XML processing finished. This node receives data which contains the XML document, checking information and processing information, from a particular previous node. If the tag checking has not been finished yet, the EndNode processes all unchecked tags, in order to complete XML processing of the entire document. In addition, the EndNode strips the document from any overhead information, so as to restore the document to its original form. Components of the EndNode are similar to the RelayNode, except that the EndNode has DeleteThread instead of SendThread. The DeleteThread cleans the data from processing and checking information, and deletes the data from the shared buffer.

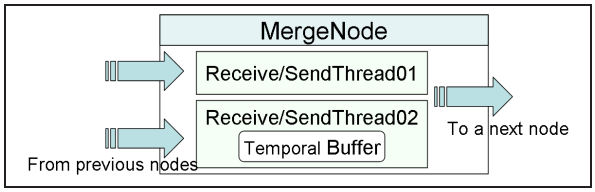

Figure 3: Components of MergeNode (two previous nodes).

`MergeNode`. The MergeNode receives data from multiple previous nodes, serializes it, and sends it to a next node, without performing any XML

processing. The MergeNode is used only in parallel XML processing topologies. We show a component of the MergeNode in Figure 3. The MergeNode has more than one thread of the type Receive/SendThread. Each thread receives data from a previous node. The MergeNode further sends the data in order, so some threads may need to wait previous data to be sent before sending its part of data. For instance, in Figure 3, Receive/SendThread02 waits until Receive/SendThread01 finishes sending its data related to a previous part of the XML document, before sending its own data, related to a subsequent part of the document.

2.1.1 Node Tag Checking Method

XML document processing involves stack data structures for tag processing. When a node reads a start tag, it pushes the tag name into a stack. When a node reads an end tag, it pops a top element from the stack, and compares the end tag name and the popped tag name. If both tag names are the same, the tags match. The XML document is well-formed when all tags match. In case the pushed and popped tags do not match, the XML document is declared ill formed.

Checking information is added to the document, for processing status indication: already matched; unmatched; or yet to be processed.

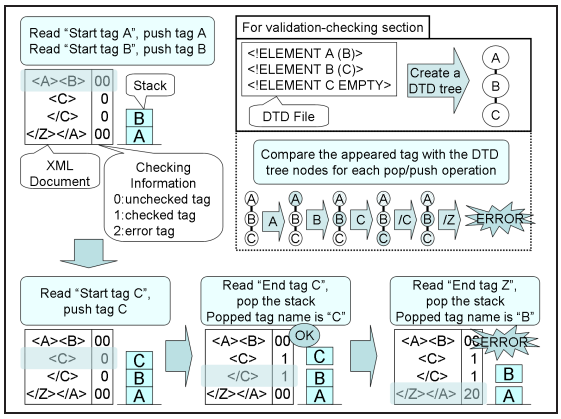

Figure 4: A Node Tag Checking Example.

In Figure 4, we show a simple example for well-formedness checking and validation checking. When the distributed XML processing system checks well-formedness only, the upper right box of the figure is not executed. When the distributed XML processing system also executes grammar validation, each node processes validation and well-formedness at the same time. In this case, each processing node has available DTD (Document Type Definition) files for grammar validation, defining the grammar that XML documents must comply with. Each node executing grammar validation reads DTD files, and generates grammar rules for validation checking. We represent these rules as a tree in the figure. Each processing node pops/pushes the tags from/to the stack, as represented in the lower part of Figure 4. If grammar validation is also executed, the node also consults the grammar rules, in order to evaluate whether the tag is allowed to appear at a specific place in the document, according to the grammar.

2.1.2 Node Allocation Patterns

As mentioned earlier, the distributed XML system can execute two types of distributed processing: pipeline and parallel processing. We show pipeline processing in Figure 5, and parallel processing in Figure 6.

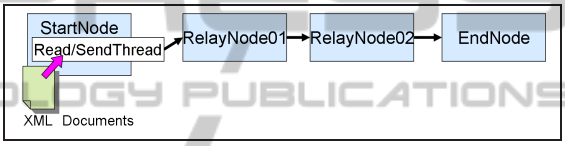

Figure 5: Pipelining Processing (two stages instance).

In Figure 5, the distributed system has two stages pipeline. All data is transferred from StartNode to EndNode via two RelayNodes. The StartNode reads the XML document per line, add some information, and sends the resulting document to the RelayNode01. The RelayNode01 and the RelayNode02 process parts of the received data, according to allocation determined by the scheduler. The EndNode receives data from the RelayNode02, processes all unchecked data, and produces the final XML processing result.

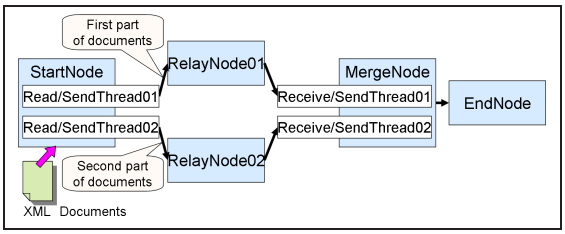

Figure 6: Parallel Processing (two path instance).

In Figure 6, the distributed system has two routes from StartNode to EndNode. The route via RelayNode01 relays a first part of documents, and the route via RelayNode02 relays a second part of documents. The RelayNode01 and the RelayNode02 process parts of the received data, according to the scheduler's allocation. MergeNode receives data from the RelayNodes, merges it, and sequentially sends it to

the EndNode. The EndNode receives data from the MergeNode, processes all unchecked data, and produces the final result. Parallel processing models have an extra node, the MergeNode, as compared with the pipeline processing models. Notice that the MergeNode executes only extra processing overhead, needed for merging parts of the document, in the parallel architecture, not executing any XML processing. We could also have integrated such merging capability to the EndNode, an hence obtaining an exact same number of nodes between pipeline and parallel models. We have decided to create a specific MergeNode in order to keep the EndNode the same across both architectures, and the number of nodes executing XML processing the same in both processing models.

3 TASK SCHEDULING

Task Scheduling System is a platform for parallel distributed processing. It is implemented in Java™ and is constructed by several modules, as illustrated in Figure 7.

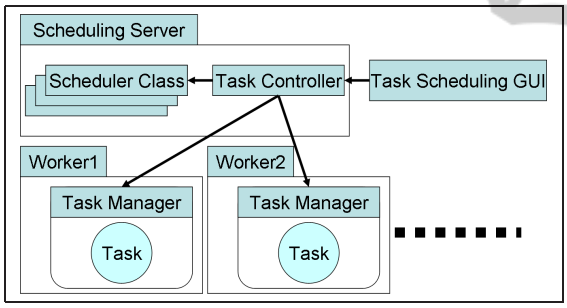

Figure 7: Task Scheduling System.

`Task Controller Module.` Task Controller module runs on the scheduler node. This module allocates tasks to Task Manager Modules, which run at worker machines, executing them. The Task Controller implements a scheduler for decision where tasks should be allocated, and when they should be executed.

`Task Manager Module.` The Task Manager module works on each worker machine. This module executes tasks assigned by the Task Controller module.

`Scheduler Class.` Scheduler Class implements a scheduling algorithm, which decides allocation of tasks and timing when tasks start. We can install not only static scheduling algorithms such as (James E. Kelley Jr, 1959) or (Tarek Hagras, 2004) but also dynamic scheduling algorithms such as (Kazumi Yoshinaga, 2008) or (Manimaran G., 1998), before starting distributed program execution. However, we use a static scheduling algorithm which assigns all tasks beforehand to avoid the affect of scheduling algorithms in this paper. Tasks communicate with each other via streaming connections in the experiments of this paper (Section 4). The tasks are allocated to workers by the scheduler beforehand. Once the tasks start, they will run until the processing or all data is finished. Also the scheduler does not migrate tasks to other workers, even if some worker is idle.

`Task Scheduling GUI Module.` Task Scheduling GUI is the GUI module front-end of the Task Scheduling System (Figure 8). We can easily define a specific distributed system and edit nodes configurations by using this module. In Figure 8, boxes represent specific tasks. Arrows between boxes represent connections between tasks. Tasks communicate data such as arguments and processing results.

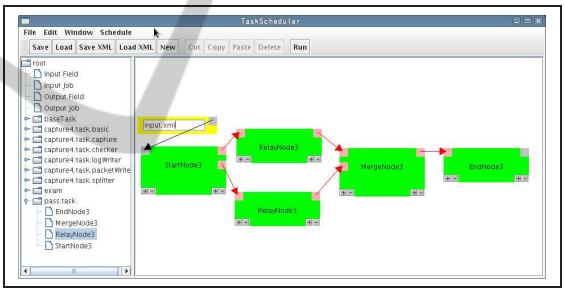

Figure 8: A Screenshot of Task Scheduling GUI.

4 DISTRIBUTED XML CHARACTERIZATION

In this section, we characterize distributed XML well-formedness and grammar validation processing.

4.1 Experimental Environment

We use a Sun SPARC Enterprise T5440 Server (Oracle, 2010) server machine as an experimental environment. Server specification is described in Table 1. This server has 4 CPU sockets, a total of 32 cores, and can concurrently run up to 256 threads. In this server, we configure hard disk storage as RAID-0 striping.

We also use Solaris Container, which is an operating system-level virtualization technology, to implement distributed XML processing nodes. These vir-

Table 1: T5440 Server Specification.

CPU	Sun Ultra SPARC® T2 Plus (1.2GHz) × 4
Memory	128G bytes (FB-DIMM)
OS	Solaris™10
JVM	Java™1.5.0_17

tual environments are called zones. The zones can directly communicate with each other by a machine internal communication mechanism. We illustrate the structure of the experimental environment in Figure 9. The Solaris Container treats one operating system as a global zone which is unique in the system. We can dynamically allocate resources to the zones, such as CPU cores and memory space. We implement the scheduler at zone 01. Each RelayNode executes independently, since the number of CPU cores, 32, is larger than the number of nodes.

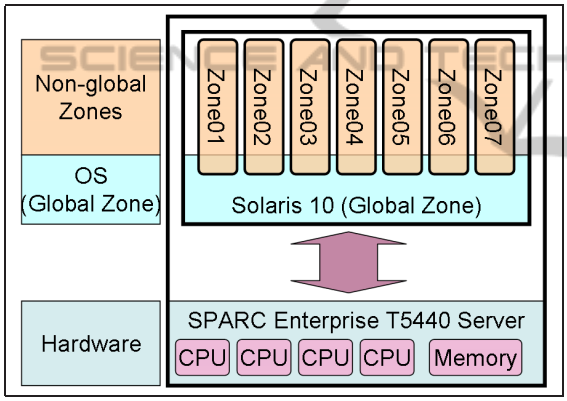

Figure 9: Zones in T5440 Server.

4.2 Node Allocation Patterns

We use several topologies and task allocation patterns, for characterizing distributed XML processing, within the parallel and pipelining models. We vary also the number of RelayNodes, within topologies, to evaluate their impact into processing efficiency.

- Two stages pipeline (Figure 10).
- Two path parallelism (Figure 11).
- Four stages pipeline (Figure 12).
- Four path parallelism (Figure 13).

In Figure 10–13, tasks are shown as light shaded boxes, underneath nodes allocated to process them.

For two RelayNode case, (Figure 10 and 11), we divide the XML documents into three parts: first two lines, fragment01 and fragment02. The first two lines contain a meta tag and a root tag. In Figure 10,

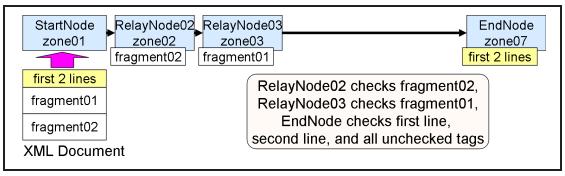

Figure 10: Two Stages Pipeline.

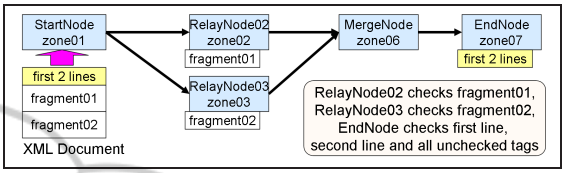

Figure 11: Two Path Parallelism.

we configure two stage pipeline topology. Data flow from StartNode to EndNode via two RelayNodes. RelayNode02 is allocated for processing fragment02, RelayNode03 is allocated for processing fragment01, and the EndNode is allocated for processing the first two lines, as well as processing all left out unchecked data. In Figure 11, we configure two path parallelism. The first two lines, fragment01 and related data flow from StartNode to EndNode via RelayNode01 and MergeNode. Fragment02 and related data flow from the StartNode to the EndNode via RelayNode02 and the MergeNode. RelayNode02 is allocated for processing fragment01, RelayNode03 is allocated for processing fragment02, whereas the EndNode is allocated for processing the first two lines, as well as processing all left out unchecked data.

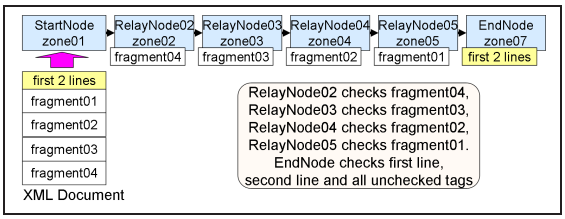

Figure 12: Four Stages Pipeline.

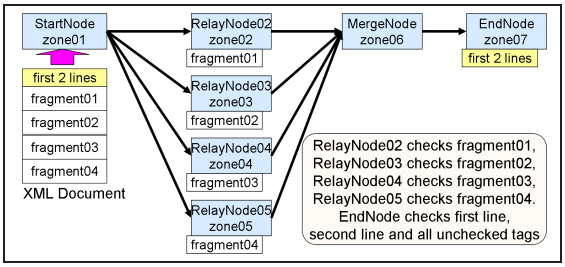

Figure 13: Four Path Parallelism.

In four RelayNode case (Figure 12 and 13), we divide the XML documents into five parts: first two lines, fragment01, fragment02, fragment03 and fragment04. Figure 12 shows a four stage pipeline topology. Data flow from StartNode to EndNode via four RelayNodes. RelayNodes fragment allocation is as shown in the figure. The EndNode is allocated for processing the first two lines, as well as processing all left out unchecked data. Figure 13 shows four path parallelism topology. The first two lines, fragment01 and related data flow from StartNode to EndNode via RelayNode02 and MergeNode. Fragment02 and related data flow from the StartNode to the EndNode via RelayNode03 and the MergeNode. Fragment03 and related data flow from the StartNode to the EndNode via RelayNode04 and the MergeNode. Fragment04 and related data flow from the StartNode to the EndNode via RelayNode05 and the MergeNode. RelayNodes fragment allocation is as shown in the figure. The EndNode is allocated for processing the first two lines, as well as processing all left out unchecked data. Notice that, even though the the parallel model has one extra node, the MergeNode, as compared with corresponding pipeline model, the MergeNode does not perform any XML processing per se. Hence, the number of nodes executing XML processing is still the same in both models.

4.3 Patterns and XML Document Types

The distributed XML processing system can execute two types of processing: well-formedness checking, and grammar validation checking of XML documents. Efficiency of these XML processing tasks may be related to: processing model, pipelining and parallel; topology, number of processing nodes and their connectivity; XML document characteristics.

We use different structures of XML documents to investigate which distributed processing model yields the most efficient distributed XML processing. For that purpose, we create seven types of XML documents, by changing the XML document depth from shallow to deep while keeping its size almost the same. We show the XML document characteristics in Table 2, and their structures in Figure 14. Each XML document has 10000 tag sets.

We combine four node allocation patterns, two processing patterns and seven XML document types to produce 56 types of experiments.

4.4 Performance Indicators

We use two types performance indicators: system performance indicators and node performance indicators.

Table 2: XML Document characteristics.

XML file	Count of lines	Width	Depth	File size [K bytes]
doc01	10002	10000	1	342
doc02	15002	5000	2	347
doc03	17502	2500	4	342
doc04	19902	100	100	343
doc05	19998	4	2500	342
doc06	20000	2	5000	342
doc07	20001	1	10000	342

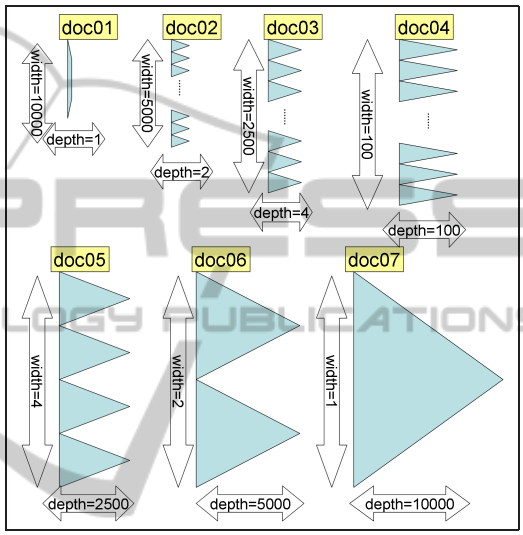

Figure 14: XML Document Structures.

System performance indicators characterize the processing of a given XML document. Node performance indicators characterize XML processing at a given processing node. The following performance indicators are used to characterize distributed XML processing:

- **Job Execution Time.** Job execution time is a system performance indicator that captures the time taken by an instance of XML document to get processed by the distributed XML system in its entirety. As several nodes are involved in the processing, the job execution time results to be the period of time between the last node (typically EndNode) finishes its processing, and the first node (typically the StartNode) starts its processing. The job execution time is measured for each XML document type and processing model.
- **Node Thread Working Time.** Node thread working time is a node performance indicator that captures the amount of time each thread of a node performs work. It does not include thread waiting time when blocked, such as data receiving wait

time. It is defined as the total file reading time, data receiving time and data sending time a node incurs. For instance, in the MergeNode, the node thread working time is the sum of receiving time and sending time of each Receive/SendThread. Assume the MergeNode is connected to two previous nodes, with two Receive/SendThreads. Furthermore, one Receive/SendThread spends 200 msec for receiving data and sending data, whereas the other Receive/SendThread spends 300 msec for receiving data and sending data. In this case, the node thread working time is 500 msecs, regardless whether the threads run sequentially on in a parallel manner. We also derive a **System thread working time** as a system performance indicator, as the average of node thread working time indicators across all nodes of the system.

- **Node Active Time.** Node active time is a node performance indicator that captures the amount of time each node runs. The node active time is defined from the first ReceiveThread starts receiving first data until the last SendThread finishes sending last data in the RelayNode or finishes document processing in the EndNode. Hence, the node active time may contain waiting time (e.g, wait time for data receiving, thread blocking time). Using the same MergeNode example as previously, assume two threads, one having 200 msec node thread working time, and another having 300 msec node thread waiting time. If one thread runs after the other, in a sequential fashion, the node active time results to be 500 msecs. However, if both threads run in parallel, the node active time results to be 300 msecs. We also define **System active time** as a system performance indicator, by averaging the node active time of all nodes across the system.
- **Node Processing Time.** Node processing time is a node performance indicator that captures the time taken by a node to execute XML processing only, excluding communication and processing overheads. We also define **System processing time** as a system performance indicator, by averaging node processing time across all nodes of the system.
- **Parallelism Efficiency Ratio.** Parallelism efficiency ratio is a system performance indicator defined as "*system thread working time* / *system active time*".

4.5 Experimental Results

For each experiment type (scheduling allocation and distributed processing model), we collect performance indicators data over seven types of XML document instances. Figures 15 and 16 report job execution time; Figures 17 and 18 report system active time; Figures 19 and 20 report system processing time; Figures 21 and 22 report system parallelism efficiency ratio. On all graphs, X axis describes scheduling and processing models, as well as well-formedness and grammar validation types of XML document processing. X axis legend is as follows:

`PIP_wel` : Pipeline and Well-formedness checking.

`PAR_wel` : Parallel and Well-formedness checking.

`PIP_val` : Pipeline and Validation checking.

`PAR_val` : Parallel and Validation checking.

Y axis denotes each performance indicator as averaged over 22 XML document instance processing.

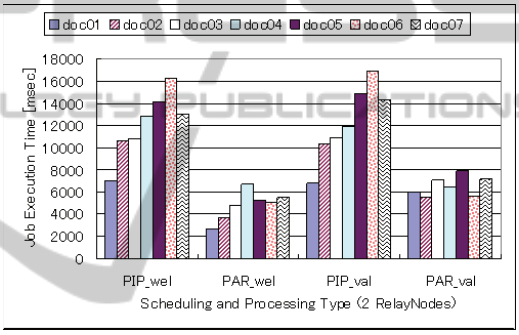

Figure 15: Job Execution Time (Two RelayNodes).

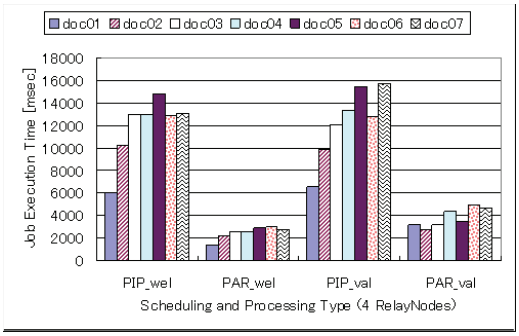

Figure 16: Job Execution Time (Four RelayNodes).

Regarding job execution time (Figures 15 and 16), parallel processing is faster than pipeline processing for all docs. Moreover, comparing Figs. 15 with 16, we see that job execution time speeds up faster with increasing the number of relay nodes in parallel processing than in pipeline processing. In parallel processing, StartNode reads concurrently parts of XML documents and sends them to next nodes. Hence, each RelayNode receives/processes/sends only part

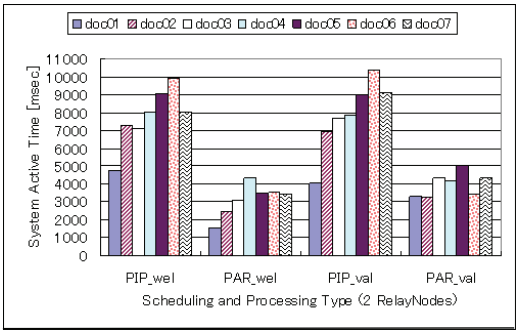

Figure 17: System Active Time (Two RelayNodes).

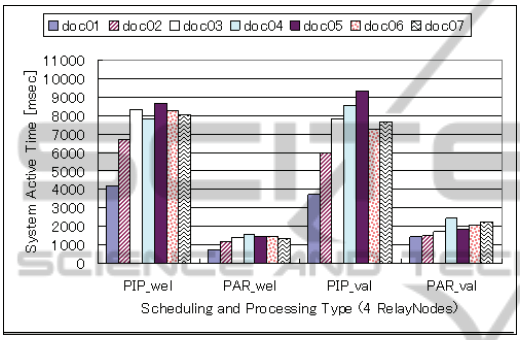

Figure 18: System Active Time (Four RelayNodes).

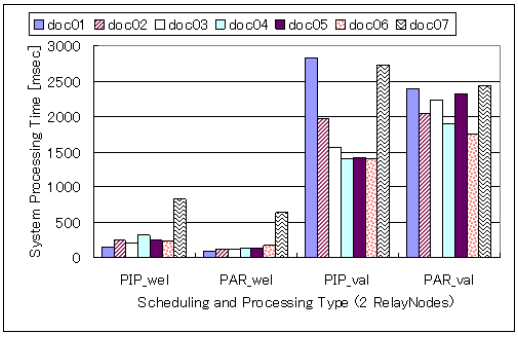

Figure 19: System Processing Time (Two RelayNodes).

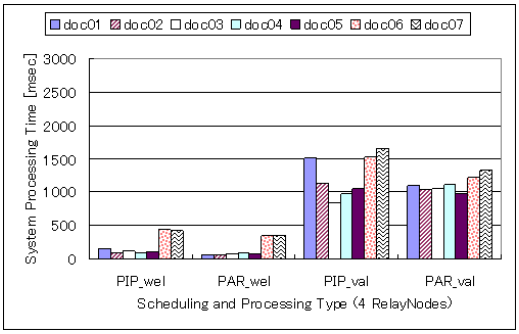

Figure 20: System Processing Time (Four RelayNodes).

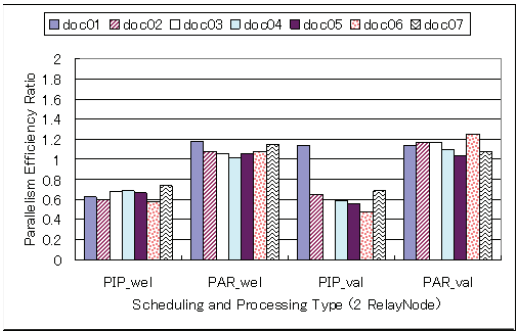

Figure 21: Parallelism Efficiency Ratio (Two RelayNodes).

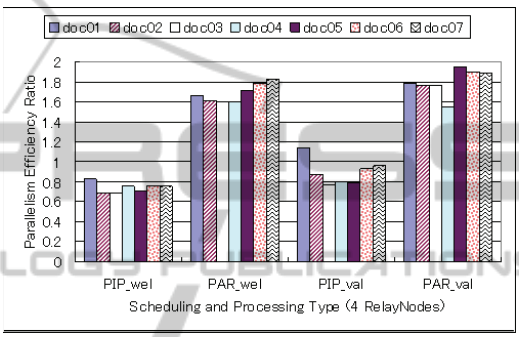

Figure 22: Parallelism Efficiency Ratio (Four RelayNodes).

of the XML document. Likewise, MergeNode receives only part of the XML document at each Receive/SendThread (Figure 3) and sends them in order to a next node. Each thread in these nodes run concurrently, resulting in reduced system active times for parallel processing in Figs. 17 and 18.

Regarding system active time and document type, the higher the document depth is, the larger the system active time. Grammar validation task is less sensitive to document depth for parallel processing. Useless processing is more pronounced at each RelayNode for documents with higher depth, because the RelayNodes are not able to match too many tags within the document part allocated to them. Hence, the EndNode is left with a large amount of processing to be done. We may reduce useless processing if we divide the document conveniently according to the document structure and grammar checking rules. In addition, node activity is more sensitive to the number of RelayNodes in parallel processing than in pipeline processing. Regarding task complexity, node activity results are similar in pipeline processing regardless of the task performed. Parallel processing induces less node activity if the task is simpler, i.e., for well-formedness checking. Figure 23 and 24 further show node active time for each node in the system when it processes doc01. In these graphs, "**SN**" means StartNode, "**RN**" means RelayNode, "**MN**" means

MergeNode and "**EN**" means EndNode. Figure 23 shows two RelayNode processing case, and Figure 24 shows four RelayNode processing case. We can see that the active time of each node is smaller in parallel as compared with pipeline processing.

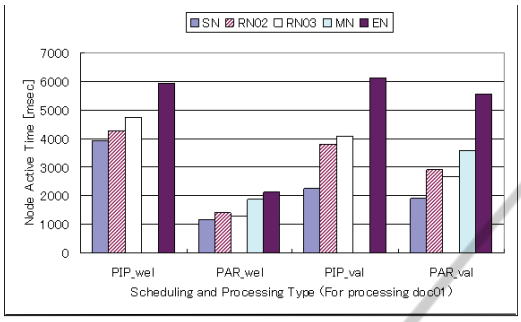

Figure 23: Node Active Time for Processing doc01 (Two RelayNodes).

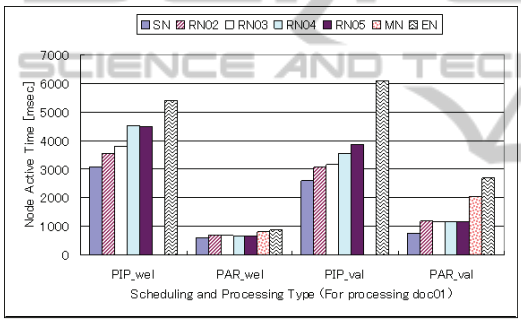

Figure 24: Node Active Time for Processing doc01 (Four RelayNodes).

Processing time is similar for both parallel processing and pipeline processing (Figure 19 and 20), which shows that the extra activity time in the pipeline processing is due to extra sending/receiving thread times. In addition, well-formedness checking has less processing time than validation checking, which is expected. Also, the average processing time is much affected by whether we can allocate XML data efficiently or not. Efficient scheduling of XML document parts to a given processing model and topology requires further investigation.

Regarding parallelism efficiency ratio (Figures 21 and 22), parallel processing is more efficient than pipeline in every case, because in parallel case, more threads are operating concurrently on different parts of the document.

Figure 25 and 26 show node thread working time when the system processes doc01 for two and four RelayNodes, respectively. In these graphs, we can see that there is more XML processing in validation than in well-formedness. Moreover, comparing

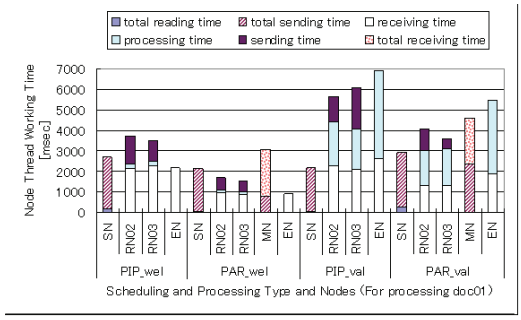

Figure 25: Node Thread Working Time for Processing doc01 (Two RelayNodes).

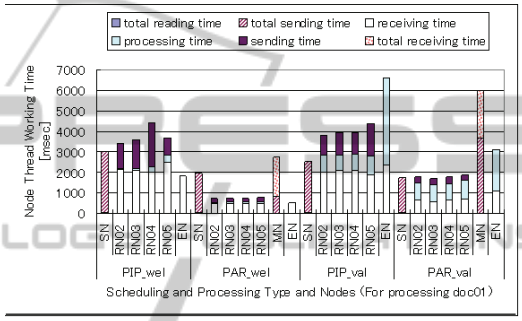

Figure 26: Node Thread Working Time for Processing doc01 (Four RelayNodes).

PIP_val with **PAR_val**, there is more processing at the EndNode in case of pipelining processing. So pipelining is less efficient than parallel in distributed processing XML documents of type 01. This is also true for other types of documents, whose results are omitted for space's sake.

For convenience, we organize our performance characterization results into a summary Table 3.

5 RELATED WORK

XML parallel processing has been recently addressed in several papers. (Wei Lu, 2007) proposes a multi-threaded XML parallel processing model where threads steal work from each other, in order to support load balancing among threads. They exemplify their approach in a parallelized XML serializer. (Michael R. Head, 2007) focuses on parallel XML parsing, evaluating multi-threaded parsers performance versus thread communication overhead. (Michael R. Head, 2009) introduces a parallel processing library to support parallel processing of large XML data sets. They explore speculative execution parallelism, by decomposing Deterministic Finite Automata (DFA) processing into DFA plus Non-

Table 3: Distributed XML Processing Characterization Summary.

	Two RelayNodes		Four RelayNodes		RelayNode increase	
	Pipeline	Parallel	Pipeline	Parallel	Pipeline	Parallel
Job execution time	Parallel is better				No change	Reduces
System active time	Well-formedness and validation are similar	Active time of validation does not depend on document depth	Similar to two RelayNode	Smaller than two RelayNode	No effect	Reduces
System processing time	Parallel is better				Reduces	
Parallelism efficiency ratio	Parallel is better		Parallel is significantly better		Slightly better	Significantly better

Deterministic Finite Automata (NFA) processing on symmetric multi-processing systems. To our knowledge, our work is the first to evaluate and compare parallel against pipelining XML distributed processing.

6 CONCLUSIONS

In this paper, we have studied two models of distributed XML document processing: parallel, and pipelining. In general, pipeline processing is less efficient, because parts of the document that are not to be processed at a specific node needs to be received and relayed to other nodes, increasing processing overhead. Regardless the distributed model, efficiency of distributed processing depends on the structure of the XML document, as well as its partition: a bad partitioning may result in inefficient processing. Optimal partition of XML document for efficient distributed processing is part of ongoing research. So far, we have focused on distributed well-formedness and validation of XML documents. Other XML processing, such as filtering and XML transformations. We are also planning on experimenting with realistic distributed XML processing systems, e.g., real nodes connected via local area network. A future research direction is to process streaming data at relay nodes (Masayoshi Shimamura, 2010). In such scenario, many web servers, mobile devices, network appliances, are connected with each other via an intelligent network, which executes streaming data processing on behalf of connected devices.

ACKNOWLEDGEMENTS

Part of this study was supported by a Grant-in-Aid for Scientific Research (KAKENHI:18500056).

REFERENCES

Dirceu Cavendish, K. S. C. (2008). Distributed xml processing: Theory and applications. *Journal of Parallel and Distributed Computing*, 68(8):1054–1069.

James E. Kelley Jr, M. R. W. (1959). Critical-path planning and scheduling. *IRE-AIEE-ACM '59 (Eastern)*, pages 160–173.

Kazumi Yoshinaga, Yoshiyuki Uratani, H. K. (2008). Utilizing multi-networks task scheduler for streaming applications. *International Conference on Parallel Processing - Workshops*, pages 25–30.

Manimaran G., M. C. S. R. (1998). An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel Distributed System*, 9(3):312–319.

Masayoshi Shimamura, Takeshi Ikenaga, M. T. (2010). Advanced relay nodes for adaptive network services - concept and prototype experiment. *Broadband, Wireless Computing, Communication and Applications, International Conference on*, 0:701–707.

Michael R. Head, M. G. (2007). Approaching a parallelized xml parser optimized for multi-core processors. *SOCP'07*, pages 17–22.

Michael R. Head, M. G. (2009). Performance enhancement with speculative execution based parallelism for processing large-scale xml-based application data. *HPDC'09*, pages 21–29.

Oracle (2010). Sun SPARC Enterprise T5440 Server. http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/031585.htm.

Tarek Hagras, J. J. (2004). A static task scheduling heuristic for homogeneous computing environments. *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*, pages 192–198.

Wei Lu, D. G. (2007). Parallel xml processing by work stealing. *SOCP'07*, pages 31–37.

Yoshiyuki Uratani, H. K. (2009). Implementation and evaluation of a parallel application which processes streaming data on relay nodes. *IEICE Technical Report*, 109(228):133–138.