Application Resource Requirement Estimation in a Parallel-Pipeline Model of Execution

Jirada Kuntraruk[1], William M. Pottenger[1] and Andrew M. Ross[2]
Computer Science & Engineering Department[1]
and Industrial & Systems Engineering Department[2]
Lehigh University
Bethlehem, PA 18015 USA
jak5@sci.ubu.ac.th, {billp,amr5}@lehigh.edu

Abstract

We propose a massively parallel framework termed a parallel-pipeline model of execution that can be employed on a homogeneous computational cluster. We show that speedups near-linear in the number of processors are achievable for applications involving reduction operations based on a novel, parallel-pipeline model of execution. As computational clusters become viable alternative platforms for solving large computational problems, the research community acknowledges that the cluster environment can be used effectively when adaptive resource management is employed. This requires the ability to estimate the resource requirements of applications before scheduling decisions are made. We propose a resource estimation model for applications executed in the parallel-pipeline model of execution. We develop a performance model that predicts the resource utilization (i.e., computation and communication complexity) for applications executing under the parallel-pipeline model on a homogeneous computational cluster. This performance prediction model can provide information to a cluster scheduler.

I. Introduction

In [28] Pottenger developed a framework for understanding parallelism in a program based on the associativity of operations which accumulate, aggregate or *coalesce* a range of values of various types into a single conglomerate. The framework for understanding such parallelism is based on an approach that models loop bodies as *coalescing loop operators*. Within this framework the author distinguished between associative coalescing loop operators and associative and commutative coalescing loop operators. He identified coalescing loop operators that are associative in nature in a variety of different applications [29]. A number of these cases involve programs previously considered difficult or impossible to parallelize; however, the framework provides the necessary theoretical foundations for performing the analysis needed to prove these loops parallel and transform them into a parallel form.

Due to the wide variety of applications that can be parallelized within this theoretical framework, in follow-on work we developed a parallel-pipeline model of execution for computational clusters. This parallel-pipeline model of execution provides an execution framework within which applications involving associative operations can achieve near-linear speedups on homogeneous computational clusters. One of our target applications is feature extraction, an important task in mining textual data. We will discuss

feature extraction along with two other applications, numerical sorting and query lookup, in detail in this article. These applications are parallelizable based on the existence of associative operators.

In fact, as available computing power increases because of faster processors and faster networking, the computational cluster is becoming a viable alternative platform for executing distributed jobs to solve computational problems. It is recognized that a cluster can be effectively shared when adaptive resource management is employed. This implies an ability to estimate the resource requirements of any given run before a scheduling decision is made.

This article thus addresses the problem of developing a resource estimation model for applications executed within our parallel-pipeline model of execution on a homogeneous computational cluster. In [21], we report preliminary results that demonstrate that our parallel-pipeline model of execution achieves near-linear speedups on such a platform. The primary goal of this article is to develop a practical performance model that predicts the resource utilization (i.e., computation and communication complexity) for applications executing under our parallel-pipeline model on a homogeneous computational cluster.

A related purpose of this article is to determine the efficiency of the parallel-pipeline model compared to other models of execution. The parallel-pipeline model is intended to be a generalized framework that works well for a wide range of applications in which the underlying operation is associative. Thus, the third purpose of the article is to evaluate the scalability of the parallel-pipeline model across multiple applications that involve associative operations.

II. Related Work

A number of research projects [9], [14], [1], [8], [26], [25], [23], [31] have contributed useful results to the performance prediction of parallel computation in dedicated homogeneous environments. These prediction models are categorized as “classical” performance evaluation. The models treat two main components: the computation and the communication time. The work in [14] and [31], for example, provides a very simple communication delay model. It is, however, insufficient for our purposes. The models LogP [9], LogGP [2], BDM [17], BSP [34], and QSM [16] include some terms for network-related delays; they focus on upper bounds, and assume upper bounds for network delay are available without considering in detail how to derive them. Since we aim to provide data to schedulers as to *expected* job duration, especially in the presence of other jobs (and thus traffic) on the cluster, we must employ more sophisticated network contention models than employed in these previous works.

The communication delay model presented in [18] includes a network contention factor. Kleinrock [19] also introduced a method for applying queueing theory to model network contention in communication

delays. These two efforts focused only on the communication time - just one of the two main components of parallel computation complexity. We have developed performance models for a parallel-pipeline model of execution that model both the computation and the communication complexity. We also characterize network contention in our performance models. We base our communication delay models on [18] and [19].

To the best of our knowledge, no performance models have been developed for a parallel-pipeline model of execution. In this regard, we conclude that our contribution is novel.

III. Parallel-Pipeline Model of Execution

We have developed a parallel-pipeline model of execution that performs a parallel reduction over a large set of distributed processors [21]. Reduction in this sense means a combining operation - for example, merging two sorted arrays in a mergesort. The ability to perform a reduction in parallel relies on the fact that the target application involves one or more associative operations and can, as a result, be parallelized [28]. Therefore, theoretically, any application that involves an associative operation (e.g., a reduction) can be executed under our model. Computationally, the application can be modeled as two different tasks: first, a computational task, and second, a parallel merge as discussed in [21]. Of these two tasks, the parallel merge forms the reduction stage of the computation. In a distributed environment, communication takes place during the reduction. This communication is represented by the arrows in an example reduction pictured in Figure 1. The complexity of each step is modeled as *cost = computation time + communication time*.

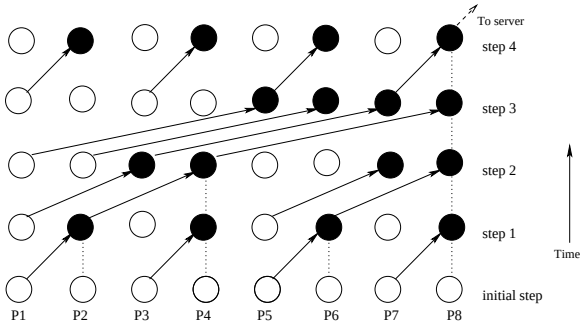

Fig. 1. Parallel-Pipeline Reduction Model. The white nodes represent the execution of the application task and the black nodes represent the merging operation. This figure depicts execution on eight processors. The arrow edges represent the communication that takes place. The dotted lines and the arrow edges together form a reduction tree.

Figure 1 depicts an example of our parallel-pipeline model of execution on eight processors. During the initial step (step 0) every processor executes the application task. Then, starting with step one, a reduction is completed every $\lg P$ steps[1]. The system reaches a state of equilibrium after $\lg P$ steps. At each step

[1]We make the simplifying assumption that the number of processors P is a power of 2. Note that $\lg P = \log_2 P$.

afterwards, there are $\frac{P}{2}$ processors performing the application task and $\frac{P}{2}$ processors performing merges. There are $\lg P$ merge stages for each set of P processors in the parallel-pipeline because a complete binary tree with P leaves has a depth of $\lg P$. Since we model communication stages as part of the pipeline, the number of stages becomes $2 * \lg P$. Adding the initial stage pictured in Figure 1 yields, in this case, $2 * 3 + 1 = 7$ stages for a parallel-pipeline created from eight processors. This forms, in essence, a pipelined, parallel reduction consisting of $2 * \lg P + 1$ stages in which new input is continually being processed in the application task, and pipelined to the $2 * \lg P + 1$ stages of the reduction tree[2]. The lengths of the $2 * \lg P + 1$ stages in the pipeline are constrained such that all stages are equal, thus guaranteeing the optimality of the pipelined reduction [27].

A. Model Optimality

Due to the nature of the binary reduction tree, message size reaches a bound of $O(\frac{P}{2})$ when the computation reaches step $\lg P$ for many applications of interest. As noted previously, the system reaches a state of equilibrium that optimally uses the processors and communication resources given certain constraints on the target application. This optimal use of resources depends on the $2 * \lg P + 1$ stages being equal in length. These stages consist of T_{Comp}, $(\lg P - 1) * T_{Merge}$, $\lg P * T_{Comm}$ and $T_{CommServer}$ as depicted in Figure 2, where T_{Comp} is the time to perform the application task and T_{Merge} is the merge time for adding the result from a new task to the existing result.

As noted, the parallel-pipeline is optimal when all stages in the pipeline are equal in length and bounded above by T_{Comp}. To achieve this, for example, the number of processors participating in the merge operation in the parallel-pipeline can be used to control T_{Merge}. Similarly, T_{Comm} is dependent on message size, and for many applications T_{Merge} drives the size of messages (because greater fan-in during merge operations results in a larger output), and therefore T_{Merge} drives T_{Comm}. Consequently, the number of processors participating in a merge can be used to control T_{Comm} as well. Since T_{Comp}, T_{Merge} and T_{Comm} depend on the particular application, users can vary the number of processors participating in the parallel-pipeline in order to keep the pipeline stages balanced. A practical example of determining the number of processors needed to balance the pipeline stages is discussed in detail in Section VI-B.

[2]Note that unlike a hardware pipeline, the communication between stages in the reduction tree is significant and as a result is modeled as $\lg P$ of the $2 * \lg P + 1$ stages.

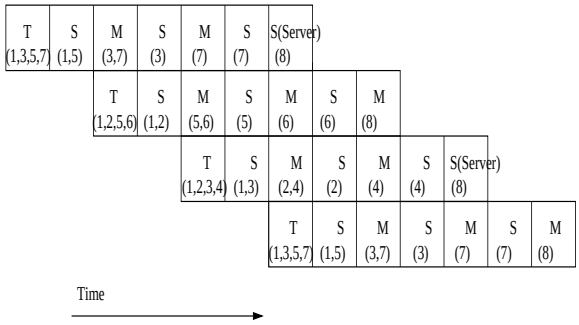

Fig. 2. Parallel Pipeline. The parallel reduction pipeline of seven stages used in execution on eight processors. T is the application task executed on four processors. S is a send, M is a merge. Processors are numbered in parentheses in each stage of the pipeline.

B. The Speedup Model

In this sub-section we present the central theorem for the theoretical maximum performance of the parallel-pipeline model of execution.

Theorem 1: The parallel-pipeline model of execution achieves a near-linear speedup.

Proof: Let N be the number of tasks and P be the number of processors. Let T_{Comp} be the execution time for one task, T_{Merge} be an upper bound on the merge time, and T_{Comm} be an upper bound on the communication time for one or more tasks. We assume that $T_{Comp} \approx T_{Merge}$ during sequential execution and that $T_{Comp} \approx T_{Merge} \approx T_{Comm}$ during parallel execution (i.e., all pipeline stages are approximately equal[3]).

The speedup model incorporates speedups due to both parallel and pipelined execution as depicted in Figure 2 for an eight processor example. The sequential execution time is

$$T_{Seq} = N \cdot T_{Comp} + (N-1) \cdot T_{Merge} \tag{1}$$

The parallel execution time for one set of N tasks is

$$T_{Par} = \frac{N}{P/2} \cdot T_{Comp} + \frac{N}{P/2} \cdot (T_{Merge} + T_{Comm}) \cdot \lg P \tag{2}$$

The first term, $\frac{N}{\frac{P}{2}} \cdot T_{Comp}$, represents the execution time to produce results from N tasks using $\frac{P}{2}$ processors. The second term, $\frac{N}{\frac{P}{2}} \cdot (T_{Merge} + T_{Comm}) \cdot \lg P$, represents the reduction (combination) of the results from the $\frac{N}{\frac{P}{2}}$ sets of tasks. Each reduction of $\frac{P}{2}$ results in a set takes $\lg P - 1$ merges and $\lg P$

[3]Note that the constraint of equal pipeline stages is required for optimality of the pipeline operation as discussed previously.

communications on a single set of $\frac{P}{2}$ processors, plus an additional merge or send-to-server, so the total reduction time for each set of $\frac{P}{2}$ tasks is $(T_{Merge} + T_{Comm}) \cdot \lg P$. Note that here we represent the final merge or send-to-server as a merge.

Generalizing from Figure 2 we have

$$Pipeline\ depth = 2 \cdot \lg P + 1 \tag{3}$$

This derives directly from the model. However, the actual maximum theoretical speedup is $2 \cdot \lg P$ due to a functional hazard in the first two stages of the pipeline. For example in Figure 2 processors 1,3,5,7 perform the application task in pipeline stage one, then 1,5 send results to 3,7, so none of these four processors are free until the end of stage two and no other processors are available because they are being used in other (e.g., reduction) operations when the pipeline is full.

The speedup due to parallel execution of one set of tasks on a single set of $\frac{P}{2}$ processors is

$$\begin{aligned} S_{Par} &= \frac{T_{Seq}}{T_{Par}} \\ &= \frac{P}{2 \cdot \lg P + 1} \end{aligned} \tag{4}$$

The overall speedup is thus

$$\begin{aligned} S_{overall} &= S_{par} \cdot S_{Pipeline} \\ &= \frac{T_{Seq}}{T_{Par}} \cdot Pipeline\ depth \\ &= \frac{P}{2 \cdot \lg P + 1} \cdot 2 \cdot \lg P \end{aligned} \tag{5}$$

Note that the $2 \cdot \lg P$ in the numerator is approximately equal to $2 \cdot \lg P + 1$ in the denominator. Thus, $S_{overall} \approx\leq P$, a near-linear speedup. This completes our proof.

IV. A Performance Prediction Model for the Parallel-Pipeline Model of Execution

Although our original complexity model sketched in [21] is able to predict the behavior of our parallel-pipeline model of execution, it introduces unnecessary complexity. As a result, herein we develop a simplified complexity model, thereby making it easier to use the model for scheduling.

Our new model is composed of the two components that play an important role in parallel program execution time, communication and computation. Figure 1 in Section III depicts the parallel-pipeline

model on eight processors, where the white nodes represent the execution of the application task and the black nodes represent merge operations. During the initial step of execution in the parallel-pipeline, there are P tasks (i.e., input data items) processed on P processors, one task per processor. After this initial step, there are only $\frac{P}{2}$ tasks processed, since half of the processors are merging data received from the previous step. Therefore the number of steps required to process N input items (not including the initial step) is $\frac{N-P}{\frac{P}{2}}$. It takes lgP additional steps to drain the pipeline when using a binary reduction tree in the parallel-pipeline model. The computation that takes place in these last lgP steps is the merge operation. In this analysis, we ignore the last stage of the pipeline (the send-to-server). From Figure 1, it can be seen that almost every T_{Comp} or T_{Merge} stage has a matching communication stage (a send). The only exception is the final merge that takes place when the pipeline is drained. Therefore there is one less T_{Comm} stage than T_{Comp}/T_{Merge} stages in the parallel-pipeline.

Assume that we want to process N data input items (e.g., N single-dimensional arrays for sorting). The total time to process N input items is thus:

$$T_{Total} = (\frac{N-P}{\frac{P}{2}} + 1) \cdot T_{Comp} + lgP \cdot T_{Merge} + \frac{N-P}{\frac{P}{2}} \cdot T_{Comm} + lgP \cdot T_{Comm} \quad (6)$$

Per the optimality model presented in Section III-A, T_{Merge} is bounded above by T_{Comp}. Therefore, we replace T_{Merge} with T_{Comp} in Equation 6. We do not, however, replace T_{Comm} with T_{Comp} even though the same optimality constraints hold. This is because T_{Comm}, as we will see, varies widely depending on the application, and the prediction model accuracy is improved by modeling T_{Comm} separately using queueing theory. This is the topic of the following section. Finally, as noted previously, the remaining lgP steps in Equation 6 comprise the pipeline drain time for both merge and communication stages.

A. A Delay Model using Queueing Theory

For the purposes of exemplifying our approach, the complexity model presented herein is developed specifically for Myrinet [4], a high speed interconnection network developed by Myricom, Inc. In this communication complexity model, queueing theory is applied to model the network contention in order to predict T_{Comm}. We will use the results of classical M/M/1 queueing theory to suggest functional forms for predicting communication delays. The classical queueing model assumes a Poisson stream of arriving messages requesting transmission over communication links, where each message has a length which is exponentially distributed with a mean of L bytes. The arrival stream in our applications will probably not

be Poisson, but the M/M/1 formula may still give useful answers; thus, we will use it even though its assumptions may not be met. Let ρ denote the system utilization factor, then $\rho = \lambda \cdot \frac{L}{C}$ where λ is the arrival rate and C is the channel capacity. The mean response time of the system, as a function of L, is denoted $D(L)$ and is given by

$$D(L) = \frac{\frac{L}{C}}{1-\rho} + \tau \tag{7}$$

where τ is the propagation delay (i.e., the channel latency in seconds). In our situation, as in [19], the channel latency is negligible compared to $\frac{L}{C}$, so we set $\tau = 0$

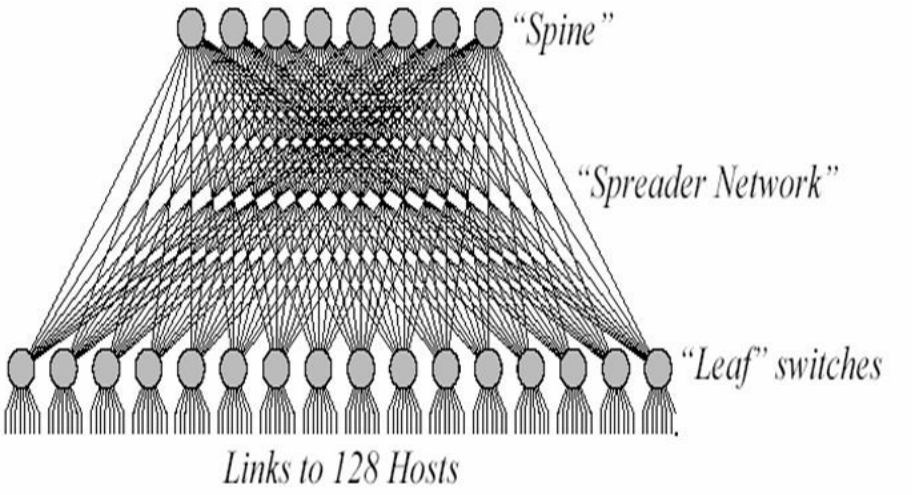

Fig. 3. The topology of the Myrinet network switch. Figure courtesy Myricom, Inc.

Let us consider sending a message over a Myrinet network. The current topology of a Myrinet switch is shown in Figure 3. Each message travels through three nodes in a 16-port switch: a leaf node, a spine node, and back through another leaf node. Thus, we model the communication delay in three steps.

First the message travels from a PC (personal computer, or host node) through a leaf node to the spine, denoted as $D(L)_{leaf1}$. We model $D(L)_{leaf1}$ using an M/M/1 framework. The arrival rate at a given leaf node is derived directly from the topology of the Myrinet network. From Figure 3, we can see that there are eight links from the hosts to each leaf node. Assuming that all the eight hosts connected to the leaf node are transmitting messages simultaneously, there will be $8 \cdot \frac{1}{T_{Comp}}$ messages arriving at the leaf node per second. Therefore, the maximum arrival rate of 'upstream' messages to a given leaf node is $\lambda = 8 \cdot \frac{1}{T_{Comp}}$ messages/second and

$$D(L)_{leaf1} = \frac{\frac{L}{C}}{1 - (8 \cdot \frac{1}{T_{Comp}} \cdot \frac{L}{C})} \tag{8}$$

Then messages travel through the spine back to another leaf node, denoted as $D(L)_{spine}$. There is only one possible path to the destination, so as before we model $D(L)_{spine}$ using an M/M/1 system. Again, we derive the arrival rate at the spine nodes from the Myrinet topology. From Figure 3, we see that there are sixteen links to each spine node from the leaf nodes beneath them. In the limit, if messages are routed from the leaf nodes to a single spine node, $16 \cdot \frac{1}{T_{Comp}}$ messages arrive at the spine node at the same time. Therefore, the arrival rate of the 'upstream' messages to a given spine node is $\lambda = 16 \cdot \frac{1}{T_{Comp}}$ messages/second and

$$D(L)_{spine} = \frac{\frac{L}{C}}{1 - (16 \cdot \frac{1}{T_{Comp}} \cdot \frac{L}{C})} \tag{9}$$

Lastly, messages travel through leaf nodes to destination hosts, denoted as $D(L)_{leaf2}$. Again there is only one possible path and we model $D(L)_{leaf2}$ using an M/M/1 system. We also derive the arrival rate at the leaf nodes directly from the Myrinet topology. From Figure 3, we see that there are eight links from the spine nodes to each leaf node. We assume again that all eight spine nodes route messages to the same leaf node at the same time, and thus $8 \cdot \frac{1}{T_{Comp}}$ messages arrive at the leaf node. Therefore, the arrival rate of 'downstream' messages at the leaf node is $\lambda = 8 \cdot \frac{1}{T_{Comp}}$ messages/second and

$$D(L)_{leaf2} = \frac{\frac{L}{C}}{1 - (8 \cdot \frac{1}{T_{Comp}} \cdot \frac{L}{C})} \tag{10}$$

Therefore the overall communications complexity is

$$T_{Comm} = D(L)_{leaf1} + D(L)_{spine} + D(L)_{leaf2} \tag{11}$$

V. Application and Implementation

Our first target application is feature extraction. Feature extraction is an important task in mining distributed textual data. We implement our parallel-pipeline model of execution using the feature extraction algorithms in HDDI [30], Hierarchical Distributed Dynamic Indexing, as our application. Our results confirm that the performance of the parallel-pipeline model of execution achieves a near-linear speedup on a homogeneous cluster. A second target application is Latent Semantic Indexing [10], an approach to information retrieval. We provide an overview of the two target applications in the following sections.

A. Feature Extraction in HDDI

In this sub-section, we review the three functional parts of the HDDI feature extraction process: input, part-of-speech tagging, and concept extraction.

1) Input: Since a collection can originate from any source, we need to handle different input formats including SGML and various subsets such as HTML and XML. In addition, the feature extraction process requires us to identify particular fields of data in the input collection that are of interest (e.g., the title of an item). In order to accomplish these tasks we employed an extensible, reusable object-oriented input parser. See [3] for details.

2) Part-of-Speech Tagging: After identifying fields of interest, our feature extraction algorithms perform part-of-speech tagging. The part-of-speech tagger is a rule-based system for tagging English parts of speech. This system is based on [5], [6], [7]. The tagger uses three levels of rule sets to determine the part of speech of each word, and tags words with their English part-of-speech tag, as specified in the Brown tagset [15].

3) Feature Extraction: A key part of textual data mining is feature or concept extraction. For this purpose, we employed a sophisticated English language noun phrase extractor. Our premise is that maximal length noun phrases are high quality discriminators and should therefore be used as keyword features for indexing purposes by the HDDI textual data mining system. In order to identify maximal length noun phrases from the tagged text, a finite state machine capable of handling complex noun phrases was employed [3].

Concurrently with the extraction of noun phrases, other information that is used later in the HDDI model building stage is extracted and preserved. For example, a frequency of occurrence is calculated for each concept in each item as well as the character offset of each concept in the original item. Also, the field in which the concept occurred (e.g., title) is preserved.

Overall, the computation forms a global dictionary of noun phrase features. This is a reduction operation in three senses: first, the various sets of features are combined lexicographically in the merge stage. Simultaneously, occurrence frequencies for identical phrases (possibly from multiple documents) are reduced to a single frequency. Likewise, offsets for phrases occurring in multiple documents are merged. This creation of a global lexicon with occurrence frequencies and offsets is an associative reduction operation [29].

B. Latent Semantic Indexing

In [20], Kontostathis and Pottenger define Latent Semantic Indexing as follows: "Latent Semantic Indexing (LSI) [10] is a well-known technique used in information retrieval. LSI has been applied to a wide variety of learning tasks, such as search and retrieval [10], [11], classification [35] and filtering [12], [13]. LSI is a vector space approach for modeling documents, and many have claimed that the technique brings out the 'latent' semantics in a collection of documents [10], [11]. LSI is based on a mathematical technique called Singular Value Decomposition (SVD). The SVD process decomposes a term by document matrix[4] into three matrices: a term by dimension matrix, T, a singular value matrix, S, and a document by dimension matrix, D. The number of dimensions is $min(t, d)$ where t = number of terms and d = number of documents. The original matrix can be obtained through matrix multiplication of TSD^T. In the LSI system, the T, S and D matrices are truncated to k dimensions. Dimensionality reduction reduces 'noise' in the term-document matrix resulting in a richer word relationship structure that reveals latent semantics present in the collection. Queries are represented in the reduced space by $T_k^T q$, where T_k^T is the transpose of the term by dimension matrix, after truncation to k dimensions. Queries are compared to the reduced document vectors, scaled by the singular values $(S_k D_k)$ by computing the cosine similarity, which provides a natural ranking for the document set for each query."

In this information retrieval application, queries are compared to multiple sets of document vectors in parallel. The rankings that results are combined in the merge stage of the parallel-pipeline. In so doing, a global ranking results. Overall, this process of computing a global ranking is an associative reduction operation.

C. Implementation

In this sub-section we outline pseudo-code for the core computation and communication pattern of the implementation of our parallel, pipelined reduction model. The *while* loop in the code in Figure 4 implements continuous, never-ending execution of the core task as discussed in Section III-B for feature extraction. The *for* loop and the parameter *blksize* control the communication pattern described and depicted in Section III (Figure 1). The *if* clause determines whether a processor sends or receives a message. It is these loops that are (software) pipelined and executed in parallel.

[4]A term by document matrix is a matrix where columns represent documents and rows represent terms in the collection. The elements in the matrix represent a frequency of the terms in each document in the collection.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(&size);
P=size;
MPI_Comm_rank(&rank);
output=NP_extractor(url); or output=Sort(array); or output=Match(query);
while(true)
{
    blksize=2;
    for(i=1;i ≤ lg(P);i++)
    {
        if(rank%blksize>0 and rank%blksize≤blksize/2)
        {
            buf=output;
            dest=rank+blksize/2;
            MPI_Send(buf,dest);
            output=NP_extractor(url); or output=Sort(array); or output=Match(query);
        }
        else
        {
            source=rank-blksize/2;
            MPI_Recv(buf,source);
            list=buf;
            Merge(output,list);
        }
        blksize=blksize*2;
    }
}
```

Fig. 4. Pseudo-code for the Parallel-Pipeline Model implementation

VI. Results on the IA-32 Cluster at NCSA

In this section, we detail the results of the application of our complexity model in a parallel-pipeline on a homogeneous computational cluster. As noted, one of our target applications is feature extraction from textual documents, an important task in mining distributed textual data. We employed our parallel-pipeline model of execution using the feature extraction algorithms implemented in HDDI [30]. We also provide an empirical performance comparison as well as an isoefficiency comparison of our parallel-pipeline model with the master-slave and reduction tree paradigms.

A. Experimental Platform

The NCSA IA-32 cluster [33] is a cluster of 484 individual computing nodes, each with two CPUs per node, for a total of 968 processors. A high-speed, low latency Myrinet network interconnects the 484 compute nodes, and a Fast Ethernet network connects the cluster to file servers and the Internet.

B. Application of the Complexity Model

In this sub-section we detail the results of the application of our complexity model in a parallel-pipeline on a homogeneous computational cluster for our first target application, feature extraction from textual documents. As noted, this involves the creation of a global dictionary of lexicographically ordered features. Our results confirm that our performance prediction model is capable of estimating the resource requirements of this application when executed within our parallel-pipeline model of execution.

As discussed in Section III, in order to maximize performance within the parallel-pipeline model of execution, the stages in the pipeline must be nearly equal and bounded by T_{Comp}. The number of processors participating in the parallel-pipeline is one of the parameters of the parallel-pipeline model of execution. The depth of the pipeline, for example, can be controlled by varying the number of processors in the pipeline in order to achieve maximum performance of the model. In addition, as discussed previously, the number of processors in the parallel-pipeline can be varied to control the value of T_{Merge}[5]. For this particular test application (feature extraction), we determined that executing the parallel-pipeline using 16 processors results in all stages being bounded above by T_{Comp}, and as a result the model achieves maximum performance.

The estimation of the execution time T_{Total} of our application within our parallel-pipeline model is calculated using Equations 6 and 11 presented in Section IV. Thus, the computation time T_{Comp} in our test application is the time required to extract features from a single textual input document and produce a sorted list of features. During the initial step (step 0) depicted in Figure 1, every processor executes this feature extraction task. Following this, starting in step one in Figure 1, $\frac{P}{2}$ processors continuously perform feature extraction and $\frac{P}{2}$ processors perform merging. In the experiments reported herein, the average input document size was approximately 5KB and the average computation time T_{Comp} for feature extraction was measured empirically to be 0.15 seconds. As noted, for this particular application $P = 16$ processors in the parallel-pipeline yields values of T_{Merge} and T_{Comm} that are bounded above by T_{Comp}.

Based on the use of 16 processors in the parallel-pipeline, we have determined L, the average message size, to be approximately $23,720$ bytes in this case. This yields an average for the actual communication complexity. Based on the Myrinet interconnection network in the IA-32, the channel capacity C is equal to 1.28 Gbits/second.

To understand the application of the prediction model, we now discuss an example. In one of the

[5]This holds when the size of the data being merged grows with the depth of the pipeline (e.g., when merging multiple single-dimensional arrays during a parallel-pipelined sort).

experiments that we conducted we employed 128 processors in the parallel-pipeline model. These processors were divided into eight groups of 16 processors each. Assuming a total of 4096 input documents, $\frac{4096}{8}$ documents were distributed to each set of 16 processors. As a result, the total time to process 4096 documents on 128 processors is estimated as[6]:

$$\begin{aligned} T_{Total} &= (\frac{N-P}{\frac{P}{2}}+1)\cdot T_{Comp} + \frac{N-P}{\frac{P}{2}}\cdot T_{Comm} \\ &= (\frac{\frac{4096}{8}-16}{\frac{16}{2}}+1)\cdot 0.15 + \\ &\frac{\frac{4096}{8}-16}{\frac{16}{2}}\cdot[\frac{\frac{23720*8}{1.28G}}{1-(8\cdot\frac{1}{0.15}\cdot\frac{23720*8}{1.28G})} + \frac{\frac{23720*8}{1.28G}}{1-(16\cdot\frac{1}{0.15}\cdot\frac{23720*8}{1.28G})} + \frac{\frac{23720*8}{1.28G}}{1-(8\cdot\frac{1}{0.15}\cdot\frac{23720*8}{1.28G})}] \\ &= 9.45 + 0.16 = 9.61\ seconds \end{aligned} \tag{12}$$

It is clear that the communication time in this example (0.16 seconds) is of little engineering significance. Thus, our experiments with the feature extraction application serve mainly to test our predictions of T_{Total}. Later, in SectionVII, we discuss an application in which the communication time is large relative to the overall execution time. This allows us to separately investigate the two aspects of our prediction model. This example serves to demonstrate the use of the performance prediction model, and the results reported in what follows are based on this same method.

We first employed 16 nodes on the IA-32 cluster. The results are presented in Table I.

Input Size	Seq Time	Par Time	SP Par	Pred Time	SP Pred	SP from SP Model
4096	878	83	10	77	11	16
8192	1779	165	10	156	11	16
16384	3472	326	10	312	11	16

TABLE I

RESULTS FOR 16 PROCESSORS (IN SECONDS)

In the second set of experiments, we used 32 nodes on the IA-32. These nodes were divided into two groups (communicators) of 16 processors each. The input was divided and distributed to each group equally. These groups of nodes concurrently executed our parallel-pipeline model. The results are presented in Table II.

In the third set of experiments we employed 64 nodes. These nodes were divided into four groups

[6]For simplicity, we ignore the two lgP terms in Equation 6 that involve pipeline drain time.

Input Size	Seq Time	Par Time	SP Par	Pred Time	SP Pred	SP from SP Model
4096	878	43	20	38	23	32
8192	1779	83	21	77	23	32
16384	3472	166	20	156	22	32

TABLE II

RESULTS FOR 16-PROCESSOR SETS USING 32 PROCESSORS (IN SECONDS)

of 16 processors each. As before, the input was divided and distributed evenly and the groups of nodes concurrently executed our parallel-pipeline model. The results are presented in Table III.

Input Size	Seq Time	Par Time	SP Par	Pred Time	SP Pred	SP from SP Model
4096	878	23	38	19	46	64
8192	1779	43	41	38	46	64
16384	3472	84	41	77	45	64

TABLE III

RESULTS FOR 16-PROCESSOR SETS USING 64 PROCESSORS (IN SECONDS)

Lastly, we employed 128 nodes on the IA-32 cluster. As noted previously, these nodes were divided into eight groups of 16 processors each. Again, the input was divided and distributed to each group evenly and the groups of nodes concurrently executed our parallel-pipeline model. The results of the 128-node experiments are depicted in Table IV.

Input Size	Seq Time	Par Time	SP Par	Pred Time	SP Pred	SP from SP Model
4096	878	13	67	9	97	128
8192	1779	23	77	19	93	128
16384	3472	43	80	38	91	128

TABLE IV

RESULTS FOR 16-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

As can be seen from these four tables, the sequential and parallel execution time for each of three different sets of documents was measured. The speedups achieved based on the empirically observed wallclock execution times are presented in column four of each table. The predicted execution times based on our performance model are in column five of each table, and the speedups corresponding to

these estimated execution times are in column six of each table. The last column of each table shows the upper bound on the speedup as predicted by our speedup model.

Table V summarizes the relative errors from the predicted time vs. actual execution time. The observed variations of the actual execution time from 30 runs are small enough to ignore (under 2%). The analysis reveals that the worst underestimate is 30.7%, which is also the worst overall. The average relative error is -11.5%. The analysis also reveals that this model does not produce an overestimation.

Num Proc	Input Size	Actual Time	Pred Time	Rel Error(%)
16	4096	83	77	−7.22
16	8192	165	156	−5.45
16	16384	326	312	−4.29
32	4096	43	38	−11.6
32	8192	83	77	−7.22
32	16384	165	156	−6.02
64	4096	23	19	−17.3
64	8192	43	38	−11.6
64	16384	83	77	−8.33
128	4096	13	9	−30.7
128	8192	23	19	−17.3
128	16384	43	38	−11.6
Average Relative Error				−11.5
Average Absolute Relative Error				11.5
Maximum Relative Error				−4.29
Maximum Absolute Relative Error				30.7
Minimum Relative Error				−30.7
Minimum Absolute Relative Error				4.29

TABLE V

RELATIVE ERROR ON IA-32

Figure 5 depicts the same relative error in graphical form.

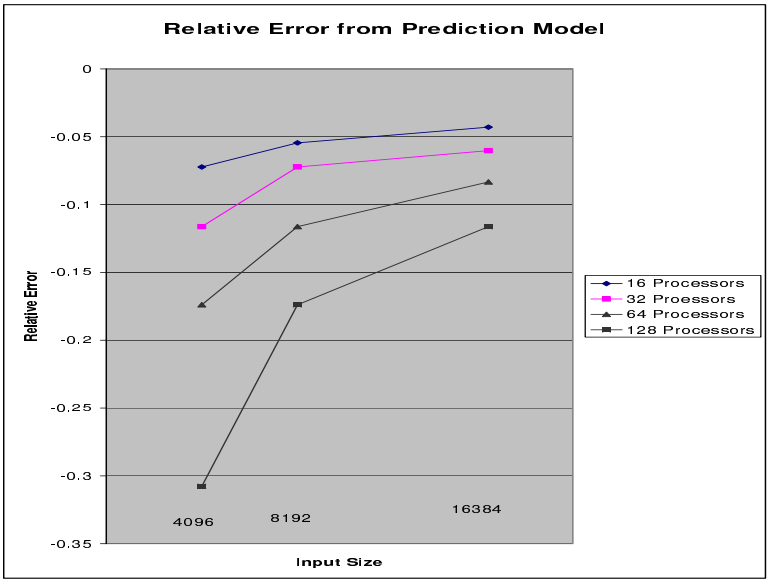

Fig. 5. Relative Error on IA-32

C. Performance Comparison of the Parallel-Pipeline vs. the Master-Slave Model

In this sub-section, we present a performance comparison of our parallel-pipeline model of execution with the master-slave paradigm. The test application is feature extraction. In the master-slave paradigm, the slave processors retrieve documents and process them. The output, which is the list of features, is sent to the master where all the features are merged into a global dictionary.

We employed 128 nodes of eight-processor sets on the IA-32 cluster in these experiments. In other words, the nodes were divided into groups of eight processors each. Table VI presents the execution time of the application in our parallel-pipeline model and in the master-slave paradigm. Both the parallel-pipeline runs and the master-slave runs are based on eight-processor sets. The input was divided and distributed to each group equally. The results demonstrate that our parallel-pipeline model outperforms the master-slave model as expected. Table VII and Table VIII present the results from experiments conducted in the same fashion on 16-processor sets and 32-processor sets respectively.

The master-slave paradigm can achieve computational speedups but has a limited degree of scalability [32]. For a large number of processors the centralized control of the master process can become a bottleneck. Mathis et al. [24] studied the performance of the master-slave model and reported that as they scaled the number of processors while holding the amount of computation per processor constant, they found the overall execution time increased gradually due to increased communication costs and the master process bottleneck factor. It is possible to enhance the scalability of the master-slave paradigm by extending a single master to a set of masters, each of them controlling a different group of process slaves [32]. In our experiments, however, we implemented a single master in the master-slave model to highlight the bottleneck factor in the master-slave paradigm.

Input Size	Seq Time	PP	MS	Improvement
16384	3560	42	65	1.54
32768	7113	83	128	1.54
65536	14239	166	254	1.53
131072	28192	331	505	1.52
262144	57251	659	1008	1.52

TABLE VI

RESULTS FOR 8-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

As we scaled the number of processors, we observed that our parallel-pipeline outperformed the master-slave paradigm by larger and larger factors. This is because as the number of processors grow, a

Input Size	Seq Time	PP	MS	Improvement
16384	3560	43	233	5.41
32768	7113	84	450	5.35
65536	14239	166	891	5.36
131072	28192	330	1782	5.40
262144	57251	654	3574	5.46

TABLE VII

RESULTS FOR 16-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

Input Size	Seq Time	PP	MS	Improvement
16384	3560	54	825	15.27
32768	7113	105	1649	15.70
65536	14239	209	3307	15.82
131072	28192	411	6616	16.09
262144	57251	818	13252	16.20

TABLE VIII

RESULTS FOR 32-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

bigger bottleneck develops at the master processor in the master-slave paradigm and degrades the overall performance. Meanwhile, the parallel-pipeline shows consistent performance.

For many parallel algorithms, speedup declines when the problem size is fixed and the number of processors is increased and speedup increases when the number of processors is fixed and the problem size increases. The *scalability* of a parallel algorithm is used to refer to the change in performance of a parallel algorithm as the problem size and number of processors increase. Intuitively, a parallel algorithm is scalable if its performance continues to improve as we scale (i.e., increase) the size of the system (i.e., problem size as well as number of processors). The results in this section empirically demonstrate that our parallel-pipeline model exhibits scalability while the master-slave model does not. For a theoretical analysis that reaches the same conclusion, please see [22].

D. Performance Comparison of Parallel-Pipeline vs. Binary Reduction Tree

In this sub-section, we present a performance comparison of our parallel-pipeline model of execution with a binary reduction tree. Again, the test application is feature extraction. In the binary reduction tree implementation, all processors initially process documents. A binary reduction tree is formed to merge the output up to the root of the reduction tree. During the merge, some of the processors will be idle until

the reduction completes. Then all processors start processing the next set of documents until the task is completed.

We employed 128 nodes split into eight-processor sets on the IA-32 cluster in these experiments. Table IX presents the execution time of the application in our parallel-pipeline model versus using a binary reduction tree. Both models employed eight-processor sets. The input was divided and distributed to each set evenly. As before, the results demonstrate that our parallel-pipeline model outperforms the binary reduction tree model. Table X and Table XI present the results from experiments conducted in the same fashion on 16-processor sets and 32-processor sets respectively.

Input Size	Seq Time	PP	BT	Improvement
16384	3560	42	48	1.14
32768	7113	83	90	1.08
65536	14239	166	178	1.07
131072	28192	331	355	1.07
262144	57251	659	708	1.07

TABLE IX

RESULTS FOR 8-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

Input Size	Seq Time	PP	BT	Improvement
16384	3560	43	73	1.69
32768	7113	84	140	1.66
65536	14239	166	278	1.67
131072	28192	330	556	1.68
262144	57251	654	1107	1.69

TABLE X

RESULTS FOR 16-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

Again, as we scaled the number of processors, we observed that our parallel-pipeline outperformed the binary reduction tree paradigm. This is because our parallel-pipeline model makes use of the idle processors while the binary reduction tree does not.

The results in this section empirically demonstrate that our parallel-pipeline model exhibits a better scalability than the binary reduction tree.

Input Size	Seq Time	PP	BT	Improvement
16384	3560	54	123	2.27
32768	7113	105	243	2.31
65536	14239	209	482	2.30
131072	28192	411	968	2.35
262144	57251	818	1928	2.35

TABLE XI

RESULTS FOR 32-PROCESSOR SETS USING 128 PROCESSORS (IN SECONDS)

VII. RESULTS FOR END-TO-END SYSTEMS AND FOR A SORTING APPLICATION

The parallel-pipeline model is intended to be a generalized framework that works well for a wide-range of applications. In this section, we report the results of two end-to-end applications executed in our parallel-pipeline model on a homogeneous computational cluster. End-to-end in this context means, for example, the entire process of retrieving web pages, extracting features, and storing them on a server. The first application is feature extraction running on a 48-node homogeneous cluster at Lehigh University. We chose LSI (Section V-B) as our second target end-to-end application for execution in the parallel-pipeline framework. As noted, LSI takes query phrases as input, matches the queries and retrieves the documents in a collection that match the queries. We have incorporated the send-to-server stage to send results to a central server in both of these applications. The LSI application was also executed on Lehigh's cluster.

Finally, we report the results from a third target application, sorting arrays of numbers. The sorting experiments were conducted on the IA-32 cluster.

A. Experimental Platforms

Lehigh's cluster (Fire) is a 48-node Beowulf cluster constructed entirely from personal computer technology to create a high-performance, parallel computing environment. The processors are connected by a 100 Mbit Ethernet. Nodes on Fire run the Linux operating system.

B. Speedup Results for Feature Extraction in an End-to-end System

In this sub-section we present speedup results for a complete end-to-end system that includes real-time download of documents from the web, feature extraction, and storage to an SQL server all operating within our parallel-pipeline. Our purpose in doing so is to substantiate our claim that the parallel-pipeline model of execution can be used in real-world applications that involve a reduction in the input prior to

execution. As a result, in this analysis we do not apply our performance prediction model. Instead, we show that the application achieves speedups near the upper bound predicted by the speedup model presented in Section III. (Recall that the theoretical upper bound on the speedup achieved in the parallel-pipeline is P, where P is the number of processors.) We have implemented a document downloading process and integrated it into the feature extraction application. In essence it is a multi-threaded web crawler that retrieves documents from the web and feeds them to the parallel-pipeline that executes the feature extraction application. We have also implemented the final stage in the pipeline, send-to-server. The last processor in each reduction tree in the parallel-pipeline sends the lexicographically ordered dictionary of extracted features to an SQL server for storage in a global dictionary.

We first employed eight nodes on Fire. The results are presented in Table XII.

Input Size	Runtime on one processor	Runtime on eight processors	Speedup	SP from SP Model
4096	4308	841	5.12	8
8192	8702	1980	4.39	8
16384	17491	3161	5.53	8

TABLE XII

FEATURE EXTRACTION SPEEDUP RESULTS FOR EIGHT PROCESSORS (IN SECONDS)

In the second set of experiments, we used 16 nodes on Fire. These nodes were divided into two groups (communicators) of eight processors each. The results are presented in Table XIII.

Input Size	Runtime on one processor	Runtime on 16 processors	Speedup	SP from SP Model
4096	4308	367	11.73	16
8192	8702	852	10.21	16
16384	17491	1632	10.71	16

TABLE XIII

FEATURE EXTRACTION SPEEDUP RESULTS FOR EIGHT-PROCESSOR SETS USING 16 PROCESSORS (IN SECONDS)

In the third set of experiments, we used 32 nodes on Fire. These nodes were divided into four groups (communicators) of eight processors each. The results are presented in Table XIV.

As can be seen from these results, for at least one application, feature extraction, the input reduction problem is tractable and the parallel-pipeline achieves speedups greater than $\frac{P}{2}$ in a real-world end-to-end

Input Size	Runtime on one processor	Runtime on 32 processors	Speedup	SP from SP Model
4096	4308	217	19.85	32
8192	8702	412	21.12	32
16384	17491	718	24.36	32

TABLE XIV

FEATURE EXTRACTION SPEEDUP RESULTS FOR EIGHT-PROCESSOR SETS USING 32 PROCESSORS (IN SECONDS)

system.

C. Speedup Results for LSI with an End-to-end System

In this sub-section we present speedup results for a complete end-to-end system that performs LSI based retrieval and stores the results on an SQL server. As with the feature extraction end-to-end system, in this analysis we do not apply our performance prediction model. Instead, we show that the application achieves speedups near the upper bound predicted by our speedup model presented in Section III. As before, we implemented the final stage in the pipeline, send-to-server. The last processor in each reduction tree in the parallel-pipeline sends the query results from LSI to an SQL server for storage.

We employed eight processors on Fire. The results are presented in Table XV. These results confirm that our parallel-pipeline framework scales across multiple applications in an end-to-end system.

Input Size	Runtime on one processor	Runtime on eight processors	Speedup	SP from SP Model
4096	2383	538	4.42	8
8192	4783	1055	4.53	8
16384	9617	2103	4.57	8
32768	19331	4184	4.62	8

TABLE XV

LSI SPEEDUP RESULTS USING EIGHT PROCESSORS (IN SECONDS)

D. Performance Prediction for Sorting on the IA-32 Cluster

In this sub-section we explore an application for which message sizes grow exponentially with the depth of the pipeline. For example, in a binary tree based pipeline, message sizes double at each stage of

communication. The purpose of these experiments is to explore the accuracy of our performance prediction model under these extreme conditions of exponential growth in message size. Our application is sorting: the quicksort is used to sort pseudo-randomly generated integers. A sequential merge is performed to merge sorted lists in merge stages of the pipeline. The parallel-pipeline was executed on the IA-32 cluster in these experiments.

The estimation of the execution time T_{Total} of the sorting application within our parallel-pipeline model is calculated based on Equations 6 and 11 presented in Section IV. The computation time T_{Comp} in our test application is the time required to sort a single input array of numbers using quicksort. During the initial step depicted in Figure 1, every processor executes quicksort to sort pseudo-randomly generated integers. Following this, starting in step one in Figure 1, $\frac{P}{2}$ processors continuously perform quicksort and $\frac{P}{2}$ processors perform the sequential merge, doubling the size of the result at each merge node in the parallel-pipeline. This also doubles message sizes since each merge node sends the result up the binary reduction tree that is embedded in the pipeline. The average computation time of quicksort on an array of one million integers was measured empirically to be 1.39 seconds. Using quicksort as our application and input arrays of one million integers, we determined the optimum number of processors needed to maximize performance to be eight. In other words, $P = 8$ ensures that T_{Comp} bounds T_{Merge} and T_{Comm} from above. In this particular application, sorting integers from multiple different arrays is an associative reduction operation.

Based on the use of eight processors in the parallel-pipeline, we have determined L, the average message size, to be approximately 8MB in this case. This yields an average for the actual communication complexity. Based on the Myrinet interconnection network, the channel capacity C is equal to 1.28 Gbits/second.

To understand the application of the prediction model, we now provide an example. In one of the experiments discussed herein we employed eight processors in the parallel-pipeline model with 512 arrays as input to be sorted. We thus compute the total predicted time to process 512 arrays as[7]:

$$\begin{aligned} T_{Total} &= (\frac{512-8}{4}+1)\cdot 1.39 + \\ &\quad \frac{512-8}{4}\cdot[\frac{\frac{8M*8}{1.28G}}{1-(8\cdot\frac{1}{1.39}\cdot\frac{8M*8}{1.28G})} + \frac{\frac{8M*8}{1.28G}}{1-(16\cdot\frac{1}{1.39}\cdot\frac{8M*8}{1.28G})} + \frac{\frac{8M*8}{1.28G}}{1-(8\cdot\frac{1}{1.39}\cdot\frac{8M*8}{1.28G})}] \\ &= 176.53 + 31.5 = 208.03\ seconds \end{aligned} \tag{13}$$

[7]As before, we ignore the two lgP terms in Equation 6 that involve pipeline drain time.

This example serves to demonstrate the use of the performance prediction model in the sorting application, and the results reported in what follows are based on this same method.

We tested our performance prediction model with the sorting application running on the IA-32 cluster on eight nodes. The results are presented in Table XVI.

Input Size	Seq Time	Par Time	SP Par	Pred Time	SP Pred	SP from SP Model
512	886	181	4.89	208	4.25	8
1024	1770	363	4.87	417	4.24	8
2048	3544	721	4.91	837	4.23	8
4096	7254	1430	5.07	1676	4.32	8

TABLE XVI

SORTING RESULTS FOR EIGHT PROCESSORS (IN SECONDS)

Table XVII summarizes the relative errors from the predicted time vs. actual execution time. The analysis reveals that the worst overestimate is 17.20% and the average relative error is 15.77%.

Num Proc	Input Size	Actual Time	Pred Time	Rel Error(%)
8	512	181	208	14.91
8	1024	363	417	14.87
8	2048	721	837	16.08
8	4096	1430	1676	17.20
Average Relative Error				15.77
Average Absolute Relative Error				15.77
Maximum Relative Error				17.20
Maximum Absolute Relative Error				17.20
Minimum Relative Error				14.87
Minimum Absolute Relative Error				14.87

TABLE XVII

RELATIVE ERROR ON IA-32 FOR SORTING

Figure 6 graphically depicts the relative error from the predicted vs. actual execution time.

In summary, in this section, we reported results for end-to-end systems for feature extraction and LSI information retrieval executed in our parallel-pipeline model on Lehigh's Cluster. The results have shown that our parallel-pipeline offers a near-linear speedup in applications in an end-to-end system. We also conducted experiments for an application in which message sizes grow exponentially with the depth of the

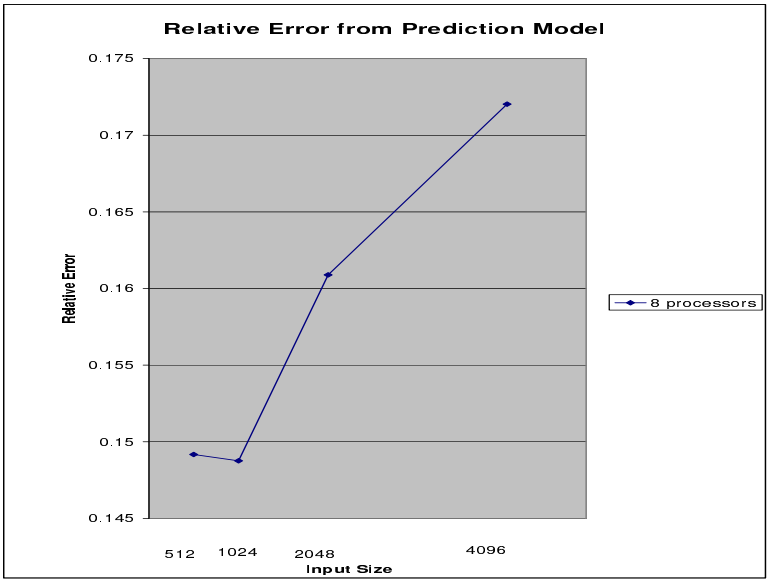

Fig. 6. Relative Error on IA-32 for Sorting

pipeline and compared the results with our performance prediction model. The prediction model predicts execution time within about 15% error.

VIII. Conclusion and Future Work

In this research we have developed a framework that combines the speedup achieved from both parallel and pipelined execution in one model and hence per our theoretical result, achieves a near-linear speedup for parallelized associative operations. In order to achieve the optimum performance offered by our parallel-pipeline model, pipeline stages must be approximately equal in length. In [22], we also proposed a multi-ary reduction tree for the parallel-pipeline model and proved a corollary to our primary theorem. We have tested the model on multiple applications and determined that the parallel-pipeline model achieves near-optimal efficiency as predicted by our theoretical analysis for several applications involving associative operations. We have also shown that the parallel-pipeline model outperforms the traditional master-slave and reduction tree paradigms for our application suite.

One of the open problems that we addressed is the input reduction problem - bringing the input data to the nodes involved in the parallel-pipeline computation. To address this issue we have developed a framework for ‘just-in-time’ retrieval of the data needed for computation. This in effect adds another stage to the parallel-pipeline, and can also be modeled as part of the computational stage. Following completion of this effort, we implemented the final step, the send-to-server pipeline stage, in end-to-end systems for feature extraction and query processing.

We have also developed a performance model that predicts the resource utilization (i.e., computation and communication complexity) for applications executing under our parallel-pipeline model of execution on a homogeneous computational cluster. We demonstrated the accuracy of these resource estimation models for a variety of processing environments and applications. Such models can provide information to a scheduler for a homogeneous computational cluster.

The performance of our prediction model varied. For feature extraction on the IA-32 cluster, the best predictions yielded an average error of about 10%. For the sorting application, the best predictions yielded an average error of about 15%.

One might ask if errors in the range of 10% to 20% are acceptable. It very much depends on the domain. We will take two examples from the world of queueing theory as guideposts:

1. Predicting the duration of a phone call to a call center is difficult; the variance is often quite large relative to the mean, so one would be happy with a prediction that was within 50%.

2. Predicting the duration of a manufacturing step, such as milling, is easier; one would only be happy with a prediction that was within 5%.

In our domain of parallel-pipeline execution, no previous studies have been performed on predicting run times. Therefore, our results in the 10% to 20% range are by default the state of the art. Predictions of this accuracy place the problem of scheduling multiple jobs somewhere between the world of M/M/c queues, with highly variable durations, and the world of job-shop scheduling, with durations of low variability.

We plan to continue to tune our resource estimation models by applying them to predict the resource utilization in end-to-end systems for information retrieval and feature extraction. This may require us to develop more sophisticated models of communication complexity at both ends of the parallel-pipeline. In addition, we plan to scale our parallel-pipeline model of execution and our resource estimation models to additional applications that involve associative operations and the use of multi-ary reduction trees. Finally, we expect to modify the feature extraction application so that it can merge or split input documents as required to balance the pipeline stages.

IX. Acknowledgements

Author Jirada Kuntraruk gratefully acknowledge the financial support of the Royal Thai Government. Co-authors William M. Pottenger and Andrew M. Ross express their deep gratitude for their salvation to their Lord and Savior, Jesus Christ. In addition, we are quite grateful to Todd Fisher and Lars Holzman for their diligent aid in coding.

REFERENCES

[1] V. Adve. *Analyzing the Behavior and Performance of Parallel Programs.* PhD thesis, Univ. of Wisconsin-Madison, December 1993.

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing,* 44(1), 1997.

[3] R.H. Bader, M.R. Callahan, D.A. Grim, J.T. Krause, N. Miller, and W.M. Pottenger. The Role of the HDDI™ Collection Builder in Hierarchical Distributed Dynamic Indexing. In *Proceedings of Textmine '01 Workshop, First SIAM International Conference on Data Mining,* April 2001.

[4] N.J. Boden and et al. Myrinet-A Gigabit per second Local Area Network. *IEEE Micro.*, 15, 1995.

[5] E. Brill. A Simple Rule-based Part of Speech Tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing.* ACL, 1992.

[6] E. Brill. *A Corpus-based Approach to Language Learning.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1993.

[7] E. Brill. Some Advances in Rule-based Part of Speech Tagging. In *Proceedings of the Twelfth National Conference on Artificial Intelligence,* 1994.

[8] M. Clement and M. Quinn. Analytical Performance Prediction on Multicomputers. In *Supercomputing 93,* 1993.

[9] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R.Subramonian, and T.von Eicken. LogP: A Practical Model of Parallel Computation. *Commun. ACM,* 39(11), 1996.

[10] S. Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *J. of the American Society for Information Science,* 41(6), 1990.

[11] S.T. Dumais. LSI meets TREC: A Status Report. In *The First Text REtrieval Conference,* 1993.

[12] S.T. Dumais. Latent Semantic Indexing (LSI) and TREC-2. In *The Second Text REtrieval Conference,* 1994.

[13] S.T. Dumais. Using LSI for Information filtering: TREC-3 experiments. In *The Third Text REtrieval Conference,* 1995.

[14] H.P. Flatt and K. Kennedy. Performance of Parallel Processors. *Parallel Computing,* 12(1), 1989.

[15] W.N. Francis and H. Kucera. Brown corpus manual. Department of Liguistics, Brown University, 1979 revision, http://www.hit.uib.no/icame/brown/bcm.html.

[16] P.B. Gibbons, Y. Matias, and V. Ramachandran. Can A Shared-Memory Model Serve as a Bridging Model for Parallel Computation? In *Proceedings of the 9th ACM Symp. on Parallel Algorithms and Architectures,* 1997.

[17] J.F. JaJa and K.W. Ryu. The Black Distributed Memory Model. *IEEE Trans. Parallel And Distributed Systems,* 7(8), 1996.

[18] Sang Cheol Kim and Sunggu Lee. Measurement and Prediction of Communication Delays in Myrinet Networks. *J. Parallel and Distributed Computing,* 61, 2001.

[19] L. Kleinrock. On the Modeling and Analysis of Computer Networks. In *Proceedings of IEEE,* volume 81(8), 1993.

[20] A. Kontostathis and W.M. Pottenger. Assessing the Impact of Sparsification on LSI Performance. In *Proceedings of the 2004 Grace Hopper Celebration of Women in Computing Conference,* 2004.

[21] J. Kuntraruk and W.M. Pottenger. Massively Parallel Distributed Feature Extraction in Textual Data Mining Using HDDI™. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing,* August 2001.

[22] Jirada Kuntraruk. *Application Resource Requirement Estimation in a Parallel-Pipeline Model of Execution on a Computation Grid*. PhD thesis, Lehigh University, Department of Computer Science and Engineering, 2003.

[23] V. Mak. Predicting Performance of Parallel Computations. *IEEE Trans. Parallel Distributed Systems,* 1(3), July 1990.

[24] M.M. Mathis, D.J. Kerbyson, and A. Hoisie. A Performance Model of non-Deterministic Particle Transport on Large-Scale Systems. In *Proceeding of International Conference on Computational Science,* volume 2659, 2003.

[25] A. Menasce and L. Barroso. A Methodology for Performance Evaluation of Parallel Applications on Multiprocessors. *J. Parallel Distributed Computing,* 14(2), 1992.

[26] J. Mohan. *Performance of Parallel Programs: Models and Analyses.* PhD thesis, Carnegie Mellon Univ., July 1984.

[27] D.A. Patterson and J.L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan-Kaufmann, 1996.

[28] William M. Pottenger. The Role of Associativity and Commutativity in the Detection and Transformation of Loop-Level Parallelism. In *Proceedings of the 12th ACM International Conference on Supercomputing,* July 1998.

[29] W.M. Pottenger. *Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1997.

[30] W.M. Pottenger, Y. Kim, and D.D. Meling. *Data Mining for Scientific and Engineering Applications,* chapter HDDI™: Hierarchical Distributed Dynamic Indexing. Kluwer Academic Publishers, 2001.

[31] J. Schopf. Structural Prediction Models for High-Performance Distributed Applications. In *Cluster Computing Conference 97,* 1997.

[32] L.E. Silva and R. Buyya. *High Performance Cluster Computing: Programming and Applications,* volume 2, chapter Parallel Programming Models and Paradigms. Prentice Hall, 1999.

[33] IA-32 Linux Supercluster. http://www.ncsa.uiuc.edu/userinfo/resources/hardware/ia32linuxcluster/.

[34] L.G. Valiant. A Bridging Model for Parallel Computation. *Communication of the ACM,* 33(8), 1990.

[35] S. Zelikovitz and H. Hirsh. Using LSI for Text Classification in the Presence of Background Text. In *Proceeding of 10th ACM International Conference on Information and Knowledge Management,* 2001.