Scalable Computing:
Practice and Experience

Scientific International Journal
for Parallel and Distributed Computing

ISSN: 1895-1767

Universitatea de Vest din Timişoara

Volume 16(2) June 2015

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 16, Number 2, June 2015

TABLE OF CONTENTS

DOI 10.12694/scpe.v16i2.1084
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. iii–iv. http://www.scpe.org

INTRODUCTION TO THE SPECIAL ISSUE ON SCALABLE AND DISTRIBUTED APPLICATIONS

Dear SCPE readers,

The special issue of SCPE covers issues of Scalable and Distributed Applications that cope with increased demand for data-intensive computations, deployed on GPUs, Grids, Clusters, supercomputers, cloud-based or other virtualized resources. Speeding up the computations and increasing the bandwidth require many techniques like algorithm optimizations, load balancing, parallelization, resource brokering, reducing communication, etc.

This special issue provides a selection of articles that concern large scale computations and scalable computing in distributed systems. The idea was born at the 2014 Fedcsis conference, particularly at the 7th workshop on Large Scale Computations on Grids (LaSCoG) and 2nd workshop on Scalable Computing in Distributed Systems, but the call was open to any submission.

Two papers were selected out of seven for our current special issue. The first paper deals with GPU implementation of a multiple string matching. This paper evaluates the speedup of the basic parallel strategy and the different optimization strategies for parallelization of Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms on a GPU. The authors obtain speed up between 2.5 and 10.9 over the equivalent sequential version of the algorithms, and a variety of speedup factors between 11 and 34 for parallel implementations of the analyzed algorithms.

The second paper presents a scalable and distributed solution version of a Broker As A Service for the SLA-based brokering of Cloud resources and for Multi-Cloud brokering. It describes the design of a service that takes as input a set of service requirements and a set of resource providers, and finds the best (or at least a good) combination of providers. The authors discuss how their architecture of a distributed broker overcomes the problems of the centralized broker allowing a Multi-User and a Multi-Cloud utilization. Their prototype implementation is a scalable solution, by distributing smaller tasks among independent agents, whose population dynamically scale together with the computing infrastructure.

We would like to thank the editorial board of SCPE for the chance of arranging this special issue, and all the reviewers for their hard work.

David Anderson, University of California, Berkeley, USA
Marjan Gusev, University Sts Cyril and Methodius, Skopje, Macedonia

DOI 10.12694/scpe.v16i2.1085
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 121–137. http://www.scpe.org

MULTIPLE STRING MATCHING ON A GPU USING CUDA

CHARALAMPOS S. KOUZINOPOULOS*, PANAGIOTIS D. MICHAILIDIS† AND KONSTANTINOS G. MARGARITIS‡

Abstract. Multiple pattern matching algorithms are used to locate the occurrences of patterns from a finite pattern set in a large input string. Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG, five of the most well known algorithms for multiple matching require an increased computing power, particularly in cases where large-size datasets must be processed, as is common in computational biology applications. Over the past years, Graphics Processing Units (GPUs) have evolved to powerful parallel processors outperforming CPUs in scientific applications. This paper evaluates the speedup of the basic parallel strategy and the different optimization strategies for parallelization of Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms on a GPU.

Key words: multiple pattern matching, parallel computing, many-core computing, GPU, CUDA

AMS subject classifications. 68W32, 68W10, 68N19

1. Introduction. Multiple pattern matching is an important kernel in computer science and it occurs in many security and network applications including information retrieval, web filtering, intrusion detection systems, virus scanners and spam filters [3, 30]. More specifically, many well known tools utilize multiple pattern matching such as: Snort [36] to perform intrusion detection and GNU grep, fgrep and egrep programs support multi-pattern matching through the -f option [12, 8, 43, 44]. Moreover, in recent years there is an interest in string matching problems as a powerful tool to locate nucleotide or Amino Acid sequence patterns in biological sequence databases. The multiple pattern matching problem can be defined as [30]: Given an input string $T = t_0 t_1 \dots t_{n-1}$ of size n and a finite set of d patterns $P = p^0, p^1, \dots, p^{d-1}$, where each p^r is a string $p^r = p_0^r p_1^r \dots p_{m-1}^r$ of size m over a finite character set Σ and the total size of all patterns is denoted as $|P|$, the task is to find all occurrences of any of the patterns in the input string. More formally, for each p^r find all i where $0 \leq i < n - m + 1$ such that for all j where $0 \leq j < m$ it holds that $t_i = p_j^r$.

Numerous sequential algorithms to this problem have been proposed [30]. In general, most of the multiple string matching algorithms consist of two phases. The first phase is a preprocessing of the set of patterns and the second phase is searching of multiple patterns in the input string. During the preprocessing phase a data structure X is constructed, X is usually proportional to the length of the pattern set and its details vary in different algorithms. The search phase uses the data structure X and it tries to quickly determine if any pattern occurs in the input string. A naive solution to the multiple string matching problem is to perform d separate searches in the input string with a single string matching algorithm [30] leading to a worst-case complexity of $\mathcal{O}(|P|)$ for the preprocessing phase and $\mathcal{O}(n|P|)$ for the search phase. While frequently used in the past, this technique is not efficient, especially when a large pattern set is involved. There are some efficient and practical algorithms have been presented for multiple string matching in terms of preprocessing and searching time for different types of data, such as: Aho-Corasick [1], Set Horspool [30], Set Backward Oracle Matching [30], Wu-Manber [44] and SOG [34].

As multiple pattern matching algorithms are often used to process huge amounts of data with a size that is increasing almost exponentially in time, their sequential execution on conventional computer systems is often impractical. Multiple pattern matching algorithms have nearly all the characteristics that make them suitable for parallel execution on graphics processing units (GPUs); they involve sequential accesses to the memory, parallelism can easily be exposed through data partitioning while the device on-chip shared memory and caches can be utilized to further decrease their execution time. The use of GPUs to accelerate the performance of multiple pattern matching algorithms has been studied in the past [15, 14, 23, 24, 32, 38, 39, 40, 42, 45]. Similar parallelization efforts for multiple string matching have been presented for multi-core and cluster platforms using different programming frameworks (i.e. POSIX threads, OpenMP and MPI) [4, 21, 33, 37]. Most of these research studies have focused on parallelization of the Aho-Corasick and Wu-Manber algorithms.

*CERN, (charalampos.kouzinopoulos@cern.ch).
†Department of Balkan, Slavic and Oriental Studies, University of Macedonia, (pmichailidis@uom.gr).
‡Department of Applied Informatics, University of Macedonia (kmarg@uom.gr).

This paper focuses on the parallelization of the Aho-Corasick [1], Set Horspool [30], Set Backward Oracle Matching [30], Wu-Manber [44] and SOG [34] multiple pattern matching algorithms through their implementation on a GPU using the Compute Unified Device Architecture (CUDA) programming model. These multiple string matching algorithms were chosen since they are practical and are frequently encountered in other research papers. The performance of the parallel implementations is evaluated after applying different optimization techniques and their execution time is compared to the time of the sequential implementations. This paper differ from previous research studies in that the Set Horspool, Set Backward Oracle Matching and SOG algorithms have never been implemented before on a GPU using the CUDA API (Application Programming Interface).

The rest of the paper is organized as follows: in Section 2, we give an overview of the most significant sequential algorithms for multiple pattern matching, the GPU architecture and the CUDA programming model and the state-of-the-art multiple pattern matching algorithms on GPU. In Section 3, we present the basic parallel implementation and optimization techniques of multiple pattern matching algorithms on the GPU architecture. In Section 4 we show the performance results of the proposed implementations. Finally, in Section 5, we draw conclusions.

2. Literature review.

2.1. Sequential multiple pattern matching algorithms. The aim of all multiple pattern matching algorithms is scan the input string in a single pass to locate all occurrences of the patterns. These algorithms are often based on single string matching algorithms with some of their functions generalized to process multiple patterns simultaneously during the preprocessing phase. Based on the way the multiple patterns are represented and the search is performed, the algorithms can generally be classified in to one of the four following approaches.

- **Prefix algorithms** The prefix searching algorithms use a *trie* to store the patterns, a data structure where each node represents a prefix u of one of the patterns. For a given position i of the input string, the algorithms traverse the trie looking for the longest possible suffix u of $t_0...t_i$ that is a prefix of one of the patterns. One of the most well known prefix multiple pattern matching algorithms is Aho-Corasick [1], an efficient algorithm based on Knuth-Morris-Pratt algorithm [19] that preprocesses the pattern in time linear in $|P|$ and searches the input string in time linear in n in the worst case. Multiple Shift-And, a bit-parallel algorithm generalization of the Shift-And algorithm for multiple pattern matching was introduced in [30] but is only useful for a small size of $|P|$ since the pattern set must fit in a few computer words.
- **Suffix algorithms** The suffix algorithms store the patterns backwards in a suffix trie, a rooted directed tree that represents the suffixes of all patterns. At each position i of the input string the algorithms compute the longest suffix u of the input string that is a suffix of one of the patterns. Commentz-Walter [8] combines a suffix trie with the *good suffix* and bad character shift functions of the Boyer-Moore algorithm [6]. A simpler variant of Commentz-Walter is Set Horspool [30], an extension of the Horspool algorithm [13] that uses only the bad character shift function. Suffix searching is generally considered to be more efficient than prefix searching since on average more input string positions are skipped following each mismatch.
- **Factor algorithms** The factor searching algorithms build a *factor oracle*, a trie with additional transitions that can recognize any substring (or factor) of the patterns. Dawg-Match [10] and MultiBDM [11] were the first two factor algorithms, algorithms complicated and with a poor performance in practice [30]. The Set Backward Oracle Matching and the Set Backward Dawg Matching algorithms [30] are natural extensions of the Backward Oracle Matching [2] and the Backward Dawg Matching [9] algorithms respectively for multiple pattern matching.
- **Hashing algorithms** The algorithms following this approach use hashing to reduce their memory footprint, usually in conjunction with other techniques. Wu-Manber [44] is one such algorithm that is based on the Horspool algorithm. It reads the input string in blocks to effectively increase the size of the alphabet and then applies a hashing technique to reduce the necessary memory space. Zhou et al. [46] proposed an algorithm called MDH, a variant of Wu-Manber for large-scale pattern sets. Kim and Kim introduced in [18] a multiple pattern matching algorithm that also takes the hashing approach. The Salmela-Tarhio-Kytöjoki [34] variants of the Horspool, Shift-Or [5] and BNDM [29] algorithms can locate *candidate* matches by excluding positions of the input string that do not match

any of the patterns. They combine hashing and a technique called q-grams to increase the alphabet size, similar to the method used by Wu-Manber.

2.1.1. Aho-Corasick algorithm. Aho-Corasick is an extension of the Knuth-Morris-Pratt algorithm [19] for a set of patterns P. It uses a deterministic finite state pattern matching machine; a rooted directed tree or *trie* of P with a *goto* function g and an additional *supply* function *Supply*. The *goto* function maps a pair consisting of an existing state q and a symbol character into the next state. It is a generalization of the *next* table or the *success* link of the Knuth-Morris-Pratt algorithm [19] for a set of patterns where a parent state can lead to its child states by σ where σ is a matching character. The *supply* function of Aho-Corasick is based on the *supply* function of the Knuth-Morris-Pratt algorithm [19]. It is used to visit a previous state of the automaton when there is no transition from the current state to a child state via the *goto* function. The *goto* function and the *supply* function are constructed during the preprocessing phase. In the searching phase of algorithm, each character in the input string is scanned from left to right with the trie that tracks partial matched patterns through state transitions. If one of terminal states is visited indicates that an occurrence of one of the pattern strings was found. The trie of P can then be built for all m patterns in $\mathcal{O}(|\Sigma|m^2)$ time, with a total size of $\mathcal{O}(|\Sigma|m^2)$. The time to pass through a transition of the *goto* function is $\mathcal{O}(1)$ in the worst and average case, while the search phase has a cost of $\mathcal{O}(n)$ in the worst and average case.

2.1.2. Set Horspool algorithm. The Set Horspool algorithm combines a deterministic finite state pattern matching machine with the *shift* function of the Horspool algorithm [13] to search for the occurrence of multiple patterns in the input string in sublinear time on average. The pattern matching machine used is a trie with a *goto* function g, created from each pattern $p^r \in P$ in reverse. The *goto* function and the *shift* function are computed during the preprocessing phase. The search for the occurrences of the patterns is then performed backwards similar to Horspool. When a mismatch or a complete match occurs, a number of input string positions can be safely skipped based on the bad character shift of the Horspool algorithm generalized for a set of patterns. The construction of the trie and the shift function requires $\mathcal{O}(|\Sigma||P|)$ time and space while the search phase of the algorithm is $\mathcal{O}(nm)$ worst case time or sublinear on average.

2.1.3. Set Backward Oracle Matching algorithm. The Set Backward Oracle Matching algorithm [2] extends the Backward Oracle Matching string matching algorithm to search for the occurrence of multiple patterns in the input string in sublinear time on average. It uses a *factor oracle*, a deterministic acyclic automaton created from each pattern $p^r \in P$ in reverse that is based on the notion of weak factor recognition. The automaton consists of a *goto* function g and at most m^2 additional external transitions such that at least any factor of a pattern can be recognized, similar to the *factor oracle* of Backward Oracle Matching. The *goto* function is constructed during the preprocessing phase from the set of the reversed patterns, similar to the automaton of the Set Horspool algorithm. During the search phase, the algorithm reads backwards with the factor oracle. If the oracle fails to recognize a factor at a given position, we can shift the pattern beyond that position. The oracle is created during the preprocessing phase in $\mathcal{O}(|\Sigma||P|)$ for all d patterns of the pattern set using a size of $\mathcal{O}(|\Sigma||P|)$. The search phase complexity of the algorithm is $\mathcal{O}(n|P|)$ worst case time or sublinear in average time.

2.1.4. Wu-Manber algorithm. Wu-Manber is a generalization of the Horspool algorithm [13] for multiple pattern matching. To improve the efficiency of the algorithm, Wu-Manber considers the characters of the patterns and the input string as blocks of size B instead of single characters. As recommended in [44], a good value for B is $log_\Sigma 2|P|$ although usually B could be equal to 2 for a small pattern set size or to 3 for a large pattern set size. During the preprocessing phase, three tables are built from the patterns, the *SHIFT*, *HASH* and *PREFIX* tables. *SHIFT* is the equivalent of the bad character shift of the Horspool algorithm and is used to determine the number of characters that can be safely skipped based on the previous B characters on each text position. The *PREFIX* table stores a hashed value of the B-characters prefix of each pattern while the HASH table contains a list of all patterns with the same prefix. During the searching phase, the algorithm is searching for the occurrences of all patterns in the input text with the assistant of the three tables that have been created by the previous state. Firstly, a hash value (h) for the block of B characters is calculated into the current search window and the shift value for that is checked ($SHIFT[h]$). If the shift value is greater than zero, then the current search window is shifted by $SHIFT[h]$ positions or else there is a candidate match and the

hashed value of the previous B characters of the input string is then compared with the hashed values stored at the *PREFIX* table to determine if an exact match exists. For the experiments of this paper, the algorithm was implemented with a block size of $B = 2$ and $B = 3$. To calculate the values of the *SHIFT*, *HASH* and *PREFIX* tables during the preprocessing phase, the algorithm requires an $\mathcal{O}(|P|)$ time. The worst case searching time of Wu-Manber is given in [7] as $\mathcal{O}(n|P|\log_{|\Sigma|}|P|)$. In [28] the lower bound for the average time complexity of exact multiple pattern matching algorithms is given as $\Omega(n/m \cdot \log_{|\Sigma|}(|P|))$ and according to [7] the searching phase of the Wu-Manber algorithm is optimal in the average case for a time complexity of $\mathcal{O}(n/m \cdot \log_{|\Sigma|}(|P|))$. In [25] the average time complexity of Wu-Manber was also estimated as $\mathcal{O}\big(n/\big[(m-B+1)\cdot(1-\frac{(m-B+1)\cdot d}{2|\Sigma|^B})\big]\big)$.

2.1.5. SOG algorithm. SOG extends the Shift-Or string matching algorithm [5] to perform multiple pattern matching in linear time on average. Similar to Shift-Or, SOG is a bit-parallel algorithm that simulates a nondeterministic automaton. Moreover, it acts as a character class filter; it constructs a generalized pattern that can simultaneously match all patterns from a finite set. The generalized pattern accepts classes of characters based on the actual position of the characters in the patterns. The searching phase of the algorithm consists of a filtering phase and a verification phase. To improve the efficiency of the verification, a two-level hashing technique is used as detailed in [27]. When a candidate match is found at a given position of the input string during the filtering phase, the patterns are verified using a combination of hashing and binary search to determine if a complete match of a pattern occurs. When the pattern set has a relatively large size, every position of the generalized pattern will accept most characters of the alphabet. In that case, false candidate matches will occur in most positions of the input string. To overcome this problem, SOG increases the alphabet size to $|\Sigma|^B$ by processing the characters of the input string and the patterns in blocks of size B, similar to the methodology used by the Wu-Manber algorithm. For the experiments of this paper, SOG was implemented using a block size of $B = 3$. The preprocessing time of the SOG algorithm is $\mathcal{O}(|P|)$ with an $\mathcal{O}(|\Sigma|^B + |P|)$ space. The combined filtering and verification phase is then $\mathcal{O}(n|P|)$ when all pattern rows have the same hash value and $\mathcal{O}(nm)$ otherwise in the worst case and linear in n in the average case.

For further details and pseudocode about the preprocessing and searching phase of the above sequential algorithms are presented in [20].

2.2. GPU architecture and CUDA programming model. Nvidia developed an unified GPU architecture that supports both graphics and general purpose computing. In general purpose computing, the GPU is viewed as a massively parallel processor array contains many processor cores. The GPU consists of an array of streaming multiprocessors (SMs). Each SM consists of a number of streaming processors (SPs) cores. Each SP core is a highly multithreaded, managing a number of concurrent threads and their state in hardware. Further, each SM has a SIMD (Single-Instruction Multiple-Data) architecture: at any given clock cycle, each streaming processor core performs the same instruction on different data. Each SM consist of four different types of on-chip memory, such as registers, shared memory, constant cache and texture cache. Constant and texture caches are both read-only memories shared by all SPs. On the other hand, the GPU has a off-chip memory, namely global memory (to which Nvidia refers as the device memory) in which has long access latency. More details for these different types of memory and GPU architecture can be found in [31].

For our numerical experimets, we take NVIDIA GTX 280 which is a compute capability 1.3 GPU from the GT200 series of NVIDIA's Tesla hardware. It has $1GB$ of GDDR3 global memory, $602MHz$ Graphics clock rate, $1.3GHz$ Processor clock tester rate and $1.1GHz$ memory clock rate. The GPU consists of 30 SMs. Each SM has 8 SPs for a total of 240 SPs, $16KB$ of on-chip shared memory, and a $16KB$ 32-bit register file. Each thread block can have a maximum of 512 threads while each SM supports up to 1024 active threads and up to 8 active blocks. Some of the related issues that affect the GPU performance such as non-coalesced memory accesses, shared memory bank conflicts, control flow divergence and occupancy are described in [20].

Nvidia's CUDA is one of the most widely adopted programming model that enables developing GPU-based applications using C programming language. A unified program C for CUDA in which consists of host code running on CPU and kernel code running on a device or GPU. The host code is a simple code C that implements some parts of an application that exhibit little or no data parallelism. The kernel code is a code using CUDA keywords that implements some parts of an application exhibit high amount of data parallelism. The kernel code implements the computation for a single thread and is usually executed on GPU by a set of parallel

threads. All parallel threads execute the same code of the kernel with different data. Threads are organized in a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate via barrier synchronization and access to a shared memory which is only visible to the thread block. Each thread in a thread block has a unique thread ID number *threadIdx*. Thread ID can 1, 2, or 3 dimensional. Usually there can be 512 or 1024 threads per block, depending on the compute capability of the device. Thread blocks are grouped into a grid and are executed independently. Each block in a grid has a unique block ID number *blockIdx*. Block ID can be 1 or 2 dimensional. Usually there are 65535 blocks in a GPU. The programmer invokes the kernel, specifying the number of threads per block and the number of blocks per grid. We must note that prior the invoking the kernel, all the data required for the computations on GPU must be transferred from the CPU memory to GPU (global) memory using CUDA library functions. Invoking a kernel will hand over the control to the GPU and the specified kernel code will be executed on this data [26].

2.3. Existing approaches on GPU. Several implementations of multiple pattern matching algorithms running on GPUs have been introduced during the last years, offering a substantial performance increase compared to their sequential versions.

The Aho-Corasick algorithm was implemented in [40] using the CUDA API to perform network intrusion detection. The Aho-Corasick trie was represented by a two-dimensional *state_transition* array; each row of the array corresponded to a state of the trie, each column to a different character of the alphabet Σ while the cells of the array represented the *next* state. The array was precomputed by the CPU and was then stored to the texture memory of the GPU. The input string (in the form of network packets) was stored in buffers allocated using *pinned* host memory using a double buffering scheme and was copied to the global memory of the GPU as soon as each buffer was full. Since the input string was in the form of network packets, two different parallelization approaches were considered; with the first, each packet was processed by a different warp of threads while with the second, each packet was processed by a different thread. A speedup of 3.2 comparing to the respective sequential implementation was reported of the parallel implementation of Aho-Corasick when executed using an NVIDIA GeForce G80 GPU. A similar implementation of the Aho-Corasick algorithm was used in [41] to perform heavy-duty anti-malware operations.

The Parallel Failureless Aho-Corasick algorithm (PFAC), an interesting variant of the Aho-Corasick algorithm on a GPU was presented in [24]. It is significantly different than the rest of the parallel implementations of the same algorithm in the sense that each character of the input stream was assigned to a different thread of the GPU. Since each thread needs only to examine if a pattern exists *starting* at the specific character and no back-tracking takes place, the *supply* function of Aho-Corasick was removed. The *goto* function was mapped into a two-dimensional *state transition* array. The *state transition* array was then stored in shared memory by grouping the patterns based on their prefixes and distributing these groups into different multiprocessors. The original paper claimed a 4000 speedup of the PFAC algorithm when executed using an NVIDIA GeForce GTX 295 GPU comparing to the sequential implementation of Aho-Corasick.

In [39], the Aho-Corasick algorithm was implemented using the CUDA API. The Aho-Corasick trie was represented using a two-dimensional *state transition* array that was precomputed on the CPU and stored to the texture memory of the GPU. Instead of storing a map of the final states in another array, the final states that corresponded to a complete pattern were flagged directly in the *goto* array by reserving 1 bit. Bitwise operations were then used to check its value. The parallelization of the algorithm was achieved by assigning different characters of the input string to different threads of the GPU and letting them perform the matching by accessing the shared *state transition* array.

The work presented in [45] focused on the implementation of the Aho-Corasick algorithm on a GPU. The Aho-Corasick trie was precomputed on the CPU and was stored in the texture memory of the GPU but it is not clear the way it was represented. The input string was stored in the global memory of the GPU, partitioned to blocks and assigned to different threads in order to achieve parallelization. The threads were then responsible to process the input string and create an output array of the states of the Aho-Corasick trie that corresponded to each character position. The paper utilized a number of optimizations in order to further improve the performance of the algorithm implementation; casting the input string from *unsigned char* to *uint4* to ensure that each thread will read 16 input string characters from the global memory instead of 1. To further improve the bandwidth utilization, the accesses of multiple threads inside the same half-warp were coalesced by reading

the required data to process an input string block to the shared memory of the device. Finally, to avoid shared memory bank conflicts, the threads of a half-warp were accessing memory from different banks. The experimental results for the parallel implementation of the Aho-Corasick algorithm on an NVIDIA Tesla GT200 GPU reported a speedup of up to 9.5 relatively to the sequential implementation of the same algorithm.

An implementation of the Aho-Corasick algorithm was also presented in [14] using the CUDA API. Similar to the methodology used in previously published research papers, the trie of the algorithm was represented using a two-dimensional *state transition* array and was stored compressed in the texture memory of the GPU while the input string was stored in global memory. Kargus was introduced in [16], a software that utilizes the Aho-Corasick algorithm to perform intrusion detection on a hybrid system consisting of multi-core CPUs and heterogeneous GPUs. The trie of Aho-Corasick was represented in the GPU using a two-dimensional *state transition* array. The *state transition* array was created in the CPU and was stored in the GPU. The network packets that comprise the input string were stored in texture memory. To increase the utilization of the memory bandwidth of the GPU, the input string was cast to *uint4* using a technique similar to [45].

In [32], the Aho-Corasick and Commentz-Walter algorithms were used to perform virus scanning accelerated through a GPU. The Aho-Corasick and Commentz-Walter tries were represented in the GPU using stacks. The *goto* and *supply* functions were substituted with offsets, essentially serializing the trie in a continuous memory block. The stacks were precomputed in the CPU and were then transferred to the GPU. Parallelization was achieved using a data-parallel approach.

Simplified Wu-Manber (SWM), a variant of the Wu-Manber algorithm was implemented in [42] using the OpenCL API for network intrusion detection. The algorithm was modified by using smaller values of B as well as avoiding hashing and direct comparison of the patterns to the input string. With these changes, the shift tables became smaller and thus they were able to fit in the shared memory of the GPU cores. Moreover, multiple pattern comparisons didn't occur and thus thread divergence between cores was avoided. Finally, the hash calculations were entirely removed, further improving the performance of the algorithm implementation. The shift tables of the Wu-Manber variant were calculated by the CPU and were then transferred to the shared memory of the GPU. The input string in the form of network packets was stored in *pinned* host memory and was mapped to the global memory of the GPU. To achieve parallelization, different threads were assigned to non-adjacent sections of the network packets.

A group of researchers proposed a GPU version of the Wu-Manber multiple pattern matching algorithm [15]. The algorithm was to be used in network intrusion detection, and was implemented under OpenGL in order to take the advantage of an NVIDIA GeForce 7600 GT card. The experiments proved to be two times faster than the existing optimized version that was used in Snort [36].

An optimized version of the agrep algorithm was presented [23], which is based on Wu-Manber algorithm using the CUDA API and taking advantage of a GeForce GTX285, for approximate nucleotide sequence matching. The performance of the implementation was evaluated for sequences of genomes, comparing an OpenMP implementation to the CUDA implementation of the algorithm, and proved to exhibit 70-fold and 36-fold performance speedups, for pattern sizes of 30 and 60 respectively.

Moreover, a modified version of the Wu-Manber algorithm for approximate matching was presented in [38]. The implementation was simplified to run on a NVIDIA GeForce 480 using the OpenCL API, and managed to achieve 62-fold speedups.

3. Parallel implementations using CUDA. In this section, we present a basic data-parallel implementation strategy for the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG multiple pattern matching algorithms, analyzes the characteristics of the algorithm implementations that leverage the capabilities of the device, discusses the flaws that affect their performance and addresses them using different optimization techniques.

3.1. Basic parallelization strategy. Table 3.1 lists the notation for the parallel implementation of the algorithms with the CUDA API that is used for the rest of this paper.

To expose the parallelism of the multiple pattern matching algorithms, the following basic data-parallel implementation strategy was used. The preprocessing phase of the algorithms was performed sequentially on the host CPU. The input string and all preprocessing arrays, including the arrays that represent the trie of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms as discussed later in this

TABLE 3.1
GPGPU implementation notation

$numBlocks$	The number of thread blocks
$blockDim$	The size in threads of each block
$threadId$	The unique ID of each thread of a block
$blockId$	The unique ID of each block
S_{chunk}	The size in characters of each chunk
S_{thread}	The number of characters that each thread processes
$S_{memsize}$	The size in bytes of the shared memory per thread block

paper, were copied to the global memory of the device. The input string was subsequently partitioned into $numBlocks$ character chunks, each with a size S_{chunk} of $\frac{n}{numBlocks}$ characters. The chunks were then assigned to $numBlocks$ thread blocks. Each chunk was further partitioned into $blockDim$ sub-chunks, that in turn were assigned to each of the $blockDim$ threads of a thread block. Each thread uses two auxiliary variables, *start* and *stop* were used to indicate the input string positions where the search phase begins and ends respectively. More specifically, the variable *start* is $(blockId \times n)/numBlocks + (n \times threadId)/(numBlocks \times blockDim)$ and the variable *stop* is $start + n/(numBlocks \times blockDim)$. To ensure the correctness of the results, $m-1$ overlapping characters were used per thread. These characters were added to the *stop* variable of Aho-Corasick and SOG, algorithms that perform forward searching and to the *start* variable of Set Horspool, Set Backward Oracle Matching and Wu-Manber, algorithms that scan the input string backwards. Therefore, each thread processed $S_{thread} = \frac{n}{numBlocks \times blockDim} + m - 1$ characters for a total of $(m-1)(numBlocks \times blockDim - 1)$ additional characters. In other words, each thread executes the corresponding search phase of the multiple string matching algorithms on the device as a CUDA kernel. Furthermore, an array *out* with a size of $numBlocks \times blockDim$ integers was used to store the number of matches per thread. To avoid extra coding complexity it is assumed that n is divisible by both $numBlocks$ and $blockDim$. Since the character chunks have to overlap, the fewer possible thread blocks should be used to reduce the redundant characters as long as the maximum possible occupancy level is maintained per SM. A pseudocode of the host function for the parallel implementation of the algorithms is given in Figure 3.1.

```
Main procedure
main()
{
   1. Execute the preprocessing phase of the algorithms;
   2. Where relevant represent tries using arrays;
   3. Allocate one-dimensional and two-dimensional arrays in the global memory
      of the device using the cudaMalloc() and cudaMallocPitch() functions respectively;
   4. Copy the data from the host arrays to the device arrays using
      the cudaMemcpy() and cudaMemcpy2D() functions;
   5. Launch CUDA kernel;
   6. Copy the results array back to host memory;
   7. Calculate the results;
}
```

FIG. 3.1. *Pseudocode of the host function*

Three tables were constructed during the preprocessing phase of the Aho-Corasick algorithm implementation. *State_transition* that corresponds to the *goto* function is a two-dimensional array where each row corresponds to a state of the trie, each column to a different character of the alphabet Σ and the cells of the array represent the next state. To ensure that alignment requirements are met on each row, *state_transition* was allocated as pitched linear device memory using the *cudaMallocPitch()* function. *State_supply* that corresponds to the *supply* function is one-dimensional array, allocated using the *cudaMalloc()* function. Each column of the array corresponds to a different state of the trie while the cells represent the supply state of a given state and the information whether that state is final or not respectively. Since the Aho-Corasick trie can have a maximum of $m \times d + 1$ trie states, a value that for the experiments of this paper was typically equal to more than 65536, each state was represented using a 4-byte integer. In this implementation, there can be divergence among the execution paths of the threads when previous states are visited using the *supply* function of the algorithm.

The Set Horspool implementation used identical version to the *state_transition* table of the Aho-Corasick implementation to locate all the occurrences of any pattern in the input string during the search phase in reverse. Moreover, *state_shift* was used, a one-dimensional array allocated using the *cudaMalloc()* function with a size of $|\Sigma|$ that represents the bad character shift of the Horspool algorithm. Each row of the array corresponds to a different state of the trie, each column to a different character of the alphabet Σ while each cell holds the number of input string character positions that can be safely skipped during the search phase. There can be divergence in the execution path of the threads in the basic implementation of Set Horspool when the next states are visited using the *goto* function of the algorithm.

To represent the factor oracle of the Set Backward Oracle Matching algorithm, a modified version of the *state_transition* array was used with an equal size to the versions used by the Aho-Corasick and Set Horspool implementations. The cells of *state_transition* correspond not only to the next state for each character of the alphabet Σ but also contain information for the external transitions of the trie. The *state_transition* array was allocated as pitched linear device memory using the *cudaMallocPitch()* function to ensure that alignment requirements are met in the global memory of the device. There is the possibility for the threads of every half-warp to diverge from their execution path, causing individual threads to branch out and execute independently. This can happen when there are potential matches of some patterns from the pattern set since they have to be verified directly to the input string.

The Wu-Manber algorithm uses the one-dimensional *SHIFT* table, allocated using the *cudaMalloc()* function and the two-dimensional *HASH* and *PREFIX* tables allocated as pitched linear memory using *cudaMallocPitch()*. Each row of the *HASH* and *PREFIX* represents a different pattern from the pattern set while the columns represent different hash values. The relevant data structures were copied to the global memory of the device with no modifications. As with the implementation of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms, there can be divergence among the threads of the same-half warp during the verification of potential matches. The value of the *out* array is set during the verification phase.

Finally, SOG computes the V, *hs_array* and *hs'_array* one-dimensional tables during the preprocessing phase, allocated as linear memory using the *cudaMalloc()* function. V is a bit vector that can store the hash value h for each different B-character block, *hs_array* holds the hash value hs for each pattern from the pattern set while *hs'_array* stores the two-level hs' hash values. The implementation of the SOG algorithm exhibits a significant level of thread divergence that is expected to affect its performance. As with Wu-Manber, the value of the *out* array is also set during the verification phase.

3.2. Implementation Limitations and Optimization Techniques. We know that accesses to global memory for compute capability 1.3 GPUs by all threads of a half-warp are coalesced into a single memory transaction when all the requested words are within the same memory segment. The segment size is 32 bytes when 1-byte words are accessed, 64 bytes for 2-byte words and 128 bytes for words of $4, 8$ and 16 bytes. With the basic implementation strategy, each thread reads a single 1-byte character on each iteration of the search loop; in this case the memory segment has a size of 32 bytes. When $S_{thread} > 32$, each thread accesses a word from a different memory segment of the global memory. This results to uncoalesced memory transactions, with one memory transaction for every access of a thread. Since the maximum memory throughput of the global memory is 128 bytes per transaction, the access pattern of the threads results in the utilization of only the $\frac{1}{128}$ of the available bandwidth.

3.2.1. Coalescing Memory Accesses. To work around the coalescing requirements of the global memory and increase the utilization of the memory bandwidth, it is important to change the memory access pattern by reading words from the same memory segment and subsequently store them in the shared memory of the device. This involves the partition of the input string into $\frac{n}{S_{memsize}}$ chunks and the collective read of $S_{memsize}$ characters from the global into the shared memory by all $blockDim$ threads of a thread block. For each 16 successive characters from the same segment then, only a single memory transaction will be used. This technique results in the utilization of the $\frac{1}{8}$ of the global memory bandwidth, improved by a factor of 16. The threads can subsequently access the characters stored in shared memory in any order with a very low latency.

Using the shared memory to increase the utilization of the memory bandwidth has two disadvantages. First, a total of $\frac{n}{S_{memsize}} \times (blockDim - 1) \times (m - 1)$ redundant characters are used that introduce significantly more work overhead when compared to the basic data-parallel implementation strategy. Second, using the shared

memory effectively reduces the occupancy of the SMs. As the size of the shared memory for each SM of the GTX 280 GPU is $16KB$, using the whole shared memory would reduce the occupancy to one thread block per SM. Partitioning the shared memory is not an efficient option since it was determined experimentally that it further increases the total work overhead.

3.2.2. Packing Input String Characters. The utilization of the global memory bandwidth can also increase when the threads read 16-byte words instead of single characters on every memory transaction. For that, the built-in *uint*4 vector can be used, a C structure with members x, y, z, and w that is derived from the basic integer type. This way, each thread accesses an 128-bit *uint*4 word that corresponds to 16 characters of the input string with a single memory transaction while at the same time the memory segment size increases from 32 to 128 bytes. By having each thread read 128-bit *uint*4 words from different memory segments results in the utilization of the $\frac{1}{8}$ of the global memory bandwidth similar to the coalescing technique above. The input string array stored in global memory can be casted to *uint*4 as follows:

```
uint4 *uint4_text = reinterpret_cast < uint4 * > ( d_text );
```

The two previous techniques can be combined; reading 16 successive 128-bit words or 256 bytes in the form of 16 *uint*4 vectors from global to shared memory can be done with just two memory transactions, fully utilizing the global memory bandwidth. The input string characters are then extracted from the uint4 vectors as retrieved from the global memory and are subsequently stored in shared memory on a character-by-character basis. To access the characters inside a uint4 vector, the vector can be recasted to *uchar*4:

```
uint4 uint4_var = uint4_text[i];

uchar4 uchar4_var0 = *reinterpret_cast < uchar4 * > ( &uint4_var.x );
uchar4 uchar4_var1 = *reinterpret_cast < uchar4 * > ( &uint4_var.y );
uchar4 uchar4_var2 = *reinterpret_cast < uchar4 * > ( &uint4_var.z );
uchar4 uchar4_var3 = *reinterpret_cast < uchar4 * > ( &uint4_var.w );
```

The drawback of casting input string characters to *uint*4 vectors and recasting them to *uchar*4 vectors is that it can be expensive in terms of processing power.

3.2.3. Texture Binding. The preprocessing arrays of the algorithms are relatively small in size while at the same time they are frequently accessed by the threads. In the case of the Set Backward Oracle Matching, Wu-Manber and SOG algorithms where a character-by-character verification of the patterns to the input string is required, the pattern set array is also accessed often. The performance of the parallel implementation of the algorithms should then benefit from the binding of the relevant arrays to the texture memory of the device. The linear memory region was bound to the texture reference using *cudaBindTexture()* for one-dimensional arrays and *cudaBindTexture2D()* for two-dimensional arrays. The textures were then accessed in-kernel using the *tex1Dfetch()* and *tex2D()* functions. Arrays accessed via textures not only take advantage of the texture caches to minimize the memory latency when cache hits occur but also bypass the coalescing requirements of the global memory.

3.2.4. Avoiding Bank Conflicts. The shared memory of the GTX 280 GPU consists of 16 memory banks numbered $0 - 15$. The banks are organized in such a way that successive 32-bit words are mapped into successive banks with word i being stored in bank i mod 16. Bank conflicts occur when two or more threads of the same half-warp try to simultaneously access words $i, j, \ldots z$ when $i \bmod 16 = j \bmod 16 = \ldots = z \bmod 16$. When the memory coalescence optimizations described above are used, it is challenging to avoid bank conflicts when the 16 characters of a *uint*4 vector are successively stored to the shared memory and when are retrieved from shared memory by the threads of the same half-warp during the search phase. Storing the input string characters in shared memory results in a 4-way bank conflict. An alternative would be to cast each *uint*4 vector to 4 *uchar*4 vectors and store them in shared memory in a round-robin fashion:

```
int tid16 = threadIdx.x % 16;

if ( tid16 < 4 ) {

        uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;
```

```
            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;
            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;

    } else if ( tid16 < 8 ) {

            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;
            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;
            uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;

    } else if ( tid16 < 12 ) {

            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;
            uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;
            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;

    } else {

            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;
            uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;
            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;
            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
    }
```

s_array represents an array stored in the shared memory of the device with a size of $S_{memsize}$. This technique was not used since in practice the performance of the implementations did not improve. Although it was conflict-free when storing the vectors, it resulted in a 4-way thread divergence that caused the serialization of accesses to shared memory, the same effect that the example code was trying to avoid. The modulo operator is very expensive when used inside CUDA kernels and had a significant impact therefore in the implementations' performance.

For further details and pseudocode about the basic and optimized parallel implementations of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms are presented in [20].

4. Experimental results. The performance of the parallel implementations of the algorithms was evaluated by comparing the running time of the GPU implementations to that of the sequential version. The computing platform in which the experiments of these implementations were executed is composed by an Intel Xeon CPU with a $2.40GHz$ clock speed and $2GB$ of memory which was used as a host and a GTX 280 GPU which was used as the device. The sequential algorithms were implemented using the ANSI C programming language and were compiled using the GCC compiler with the "-O2" and "-funroll-loops" optimization flags. The parallel GPU implementations of the algorithms were compiled using the NVCC 5.0 compiler of the CUDA API with the "-O2" optimization flag.

For the comparison of the parallel implementations we used the running time and speedup as measures. Running time is the total execution time of an implementation including the preprocessing time and the searching time. Preprocessing time is the time in seconds an algorithm uses to preprocess the pattern / pattern set while searching time is the total time in seconds an algorithm uses to locate all occurrences of any pattern in the input string. The running time of the CUDA implementations was measured using the CUDA event API. To decrease random variation, all time results were averages of 100 runs. Speedup is defined as (Sequential running time on a specific CPU) / (Parallel running time on a specific GPU).

The parameters that affect the performance of parallel multiple pattern matching algorithms are the size of the input string n, the size of the pattern set d, the size of the patterns m and the size $|\Sigma|$ of the alphabet used.

The data set used for the experiments was similar to the sets used in [17, 22, 35]. It consisted of randomly generated input strings of a binary alphabet, the genome of Escherichia coli from the Large Canterbury Corpus, the Swiss Prot. Amino Acid sequence database, the FASTA Amino Acid (FAA) and FASTA Nucleic Acid (FNA) sequences of the A-thaliana genome and natural language input strings of the English alphabet:

- Randomly generated input strings of size $n = 4,000,000$ with a binary alphabet. The alphabet used was $\Sigma = \{0, 1\}$.
- The genome of Escherichia coli from the Large Canterbury Corpus with a size of $n = 4,638,690$ characters and the FASTA Nucleic Acid (FNA) of the A-thaliana genome with a size of $n = 116,237,486$

characters. The alphabet $\Sigma = \{a, c, g, t\}$ of both genomes consisted of the four nucleobases of the Deoxyribonucleic Acid (DNA).

- The FASTA Amino Acid (FAA) of the A-thaliana genome with a size of $n = 10,830,882$ characters and the Swiss Prot. Amino Acid sequence database with a size of $n = 177,660,096$ characters. The alphabet $\Sigma = \{a, c, d, e, f, g, h,$
  $i, k, l, m, n, p, q, r, s, t, v, w, y\}$ used by the databases consisted of 20 different characters.
- The CIA World Fact Book from the Large Canterbury Corpus. The input string had a size of $n = 1,914,500$ characters and an alphabet of size $|\Sigma| = 128$ characters.

The pattern set for each execution was created from subsequences of the corresponding input string, consisting of $1,000$ and $8,000$ patterns with each pattern having a size of $m = 8$ and $m = 32$ characters.

The performance of the algorithm implementations is evaluated when a number of optimizations are applied for different sets of data and is compared to the performance of the sequential implementations. Each implementation stage also incorporates the optimizations of the previous stages.

1. The first stage of the implementation was unoptimized. The input string, the pattern set, the preprocessing arrays of the algorithms and the *out* array that holds the number of matches per thread, were stored in the global memory of the device. Each thread accessed input string characters directly from the global memory as needed and stored the number of matches directly to *out*. At the end of the algorithms' search phase, the results array was transferred to the host memory and the total number of matches was calculated by the CPU.
2. The second stage of the implementation involved the binding of the preprocessing arrays and the pattern set array in the case of the Set Backward Oracle Matching, Wu-Manber and SOG algorithms to the texture memory of the device.
3. In the third stage, the threads worked around the coalescing requirements of the global memory by collectively reading input string characters and storing them to the shared memory of the device.
4. The fourth stage of the implementation was similar to the third but in addition, each thread read simultaneously 16 input string characters from the global memory by using a *uint*4 vector. The characters were extracted from the vectors using *uchar*4 vectors and were subsequently stored to the shared memory.

For the experiments of this paper, 30 thread blocks were used, one per each SM, with 256 threads per block. As discussed in section 3, the shared memory was utilized to work around the coalescing requirements of the global memory during the third implementation stage and it was determined experimentally that further partitioning the shared memory was not an efficient option as the total work overhead increases.

Figures 4.1 to 4.5 depict the speedup of different implementations of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms for randomly generated input strings, biological sequence databases and English alphabet data on a GTX280 GPU. As can generally be seen, the performance of the algorithms increased significantly when implemented in parallel on a GPU using the CUDA API, even before applying any optimization. When the presented optimization techniques were used though, the implementations had a significantly higher speedup over the sequential versions, although different algorithms were affected in different ways.

When the preprocessing arrays and the pattern set in the case of some algorithms were bound to the texture memory of the device, the performance of the implementations increased considerably. Also in most cases, the implementations exhibited a faster running time when the accesses to the input string for the threads of the same half-warp were coalesced with the use of the shared memory. On the other hand, it can be observed that the performance of the algorithm implementations did not improve significantly in the fourth implementation stage due to the high cost involved in casting the input string characters to *uint*4 vectors and recasting them to *uchar*4 vectors as discussed in section 3.

For the Aho-Corasick algorithm, the experimentally measured speedup for the basic parallel implementation of the Aho-Corasick algorithm over the sequential implementation ranged from 3.2 to 10.9. For the second and third implementation stages, the overhead that was caused by the frequent accesses of the threads to the global memory of the device was partially reduced with an additional performance increase of up to 3.5 times. The speedup of the final, optimized kernel ranged between 10 and 18.5 comparing to the sequential implementation,

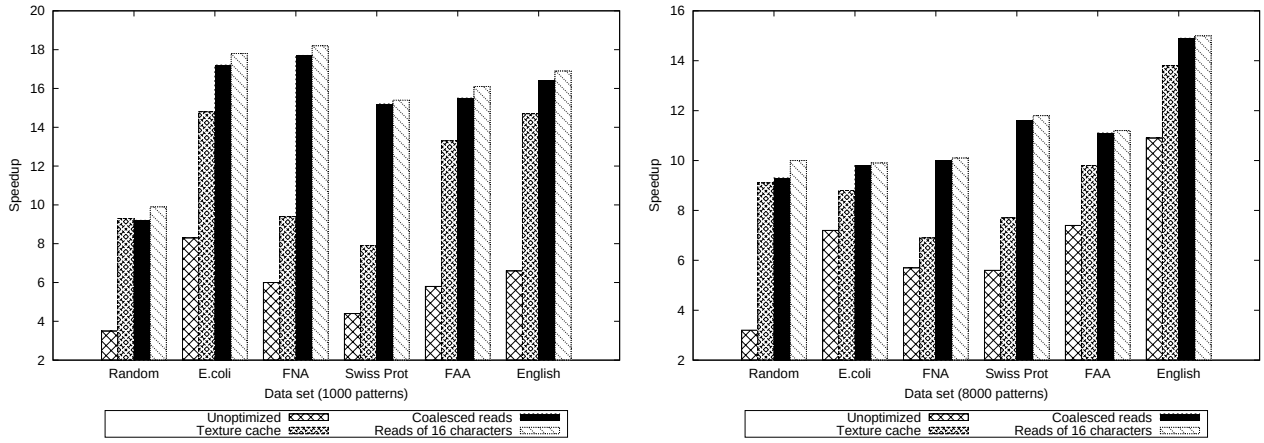


FIG. 4.1. *Speedup of different Aho-Corasick implementations for randomly generated input strings, biological sequence databases and English alphabet data*

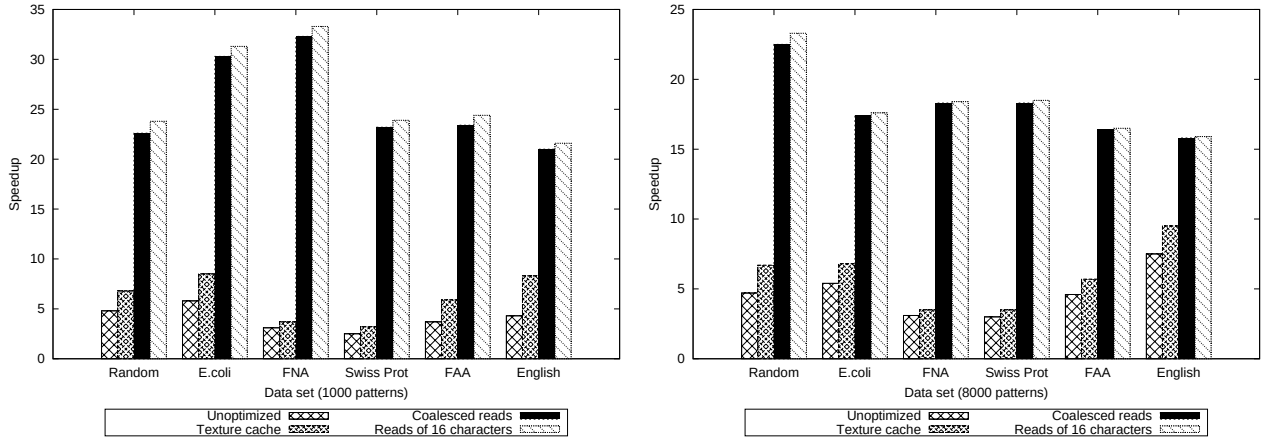


FIG. 4.2. *Speedup of different Set Horspool implementations for randomly generated input strings, biological sequence databases and English alphabet data*

slightly faster than the reported speedup in [40, 45]. This was expected since for the experiments of this paper a more recent GPU was used to evaluate the performance of the algorithm implementations. The highest speedup was generally observed when the E.coli genome and the FASTA Nucleic Acid (FNA) of the A-thaliana genome were used, together with sets of $1,000$ patterns.

The unoptimized implementation of the Set Horspool algorithm had a speedup between 2.5 and 7.5 over the sequential version. As with the implementation of the Aho-Corasick algorithm, the parallel searching time of Set Horspool also benefited when the preprocessing arrays were bound to the texture memory of the device in the second stage of the implementation, although to a lesser degree; the speedup increased between 1.2 and 1.9 times comparing to the basic parallel implementation for sets of $1,000$ patterns and between 1.1 and 1.4 times for sets of $8,000$ patterns. The performance of the implementation increased to a much greater extent during the third implementation stage, with a speedup increase over the second implementation stage of up to 8.7. The speedup of the optimized kernel was between 22.1 and 33.7, significantly higher than the corresponding speedup of the Aho-Corasick algorithm. Again, the highest speedup was achieved when the E.coli genome and the FASTA Nucleic Acid of the A-thaliana genome were used, together with sets of $1,000$ patterns.

As can be seen in Figure 4.3, the performance of the Set Backward Oracle Matching algorithm had similar

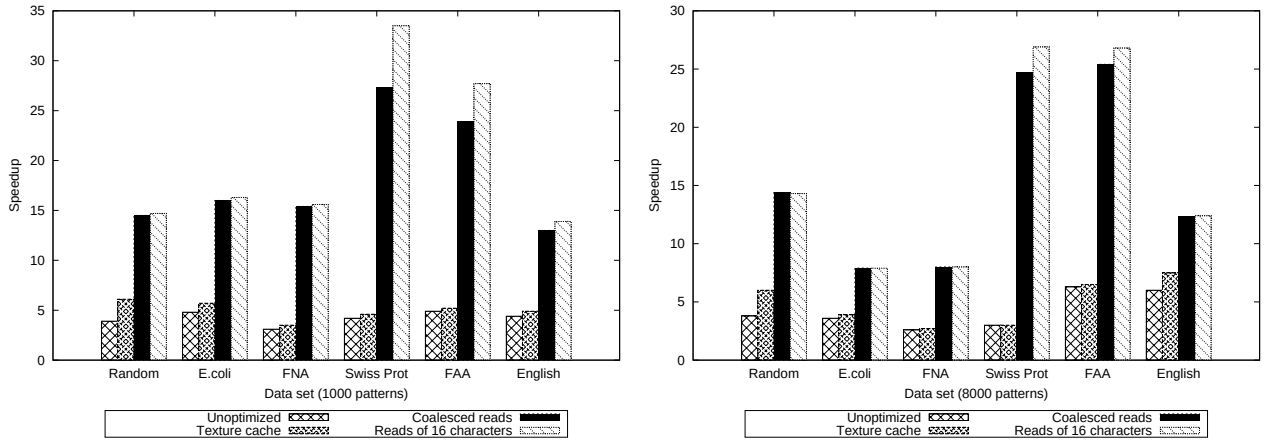


FIG. 4.3. *Speedup of different Set Backward Oracle Matching implementations for randomly generated input strings, biological sequence databases and English alphabet data*

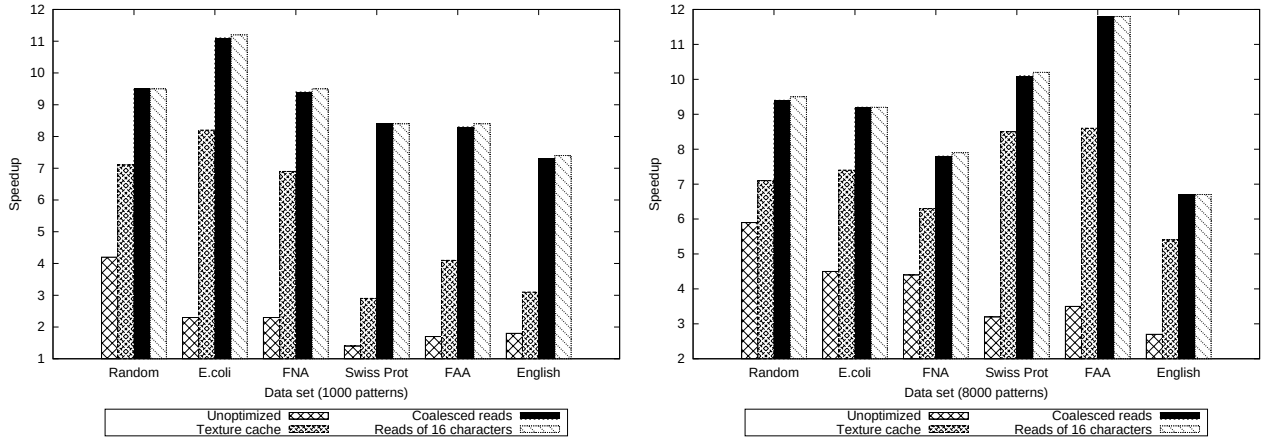


FIG. 4.4. *Speedup of different Wu-Manber implementations for randomly generated input strings, biological sequence databases and English alphabet data*

characteristics to that of Set Horspool. The speedup of the implementation ranged between 2.6 and 6.3 over the sequential version of the algorithm and further improved by up to 1.6 times when the preprocessing arrays and the pattern set were bound to the texture memory of the device. When the accesses for the threads of the same half-warp were coalesced, the performance of the algorithm implementation improved by up to an additional 8.2 times. The speedup of the optimized kernel of the algorithm ranged between 16.1 and 33.7. The highest speed was achieved when the Swiss Prot. Amino Acid sequence database and the FASTA Amino Acid (FAA) of the A-thaliana genome were used, for sets of either $1,000$ or $8,000$ patterns.

The unoptimized implementation of Wu-Manber was up to 5.9 times faster than its sequential implementation. When the pattern set as well as the *SHIFT*, *HASH* and *PREFIX* tables that were created during the preprocessing phase of the algorithm, were bound to the texture memory of the device and the shared memory was used to bypass the coalescing requirements of the global memory during the second and third implementation stages, the speedup of the implementation increased by up to an additional 6.1 times. The performance of the final, optimized implementation of Wu-Manber was up to 11.8 times faster than the sequential implementation and was not significantly affected by the type of the dataset used.

The parallel implementation of SOG exhibited the lowest speedup comparing to the rest of the presented

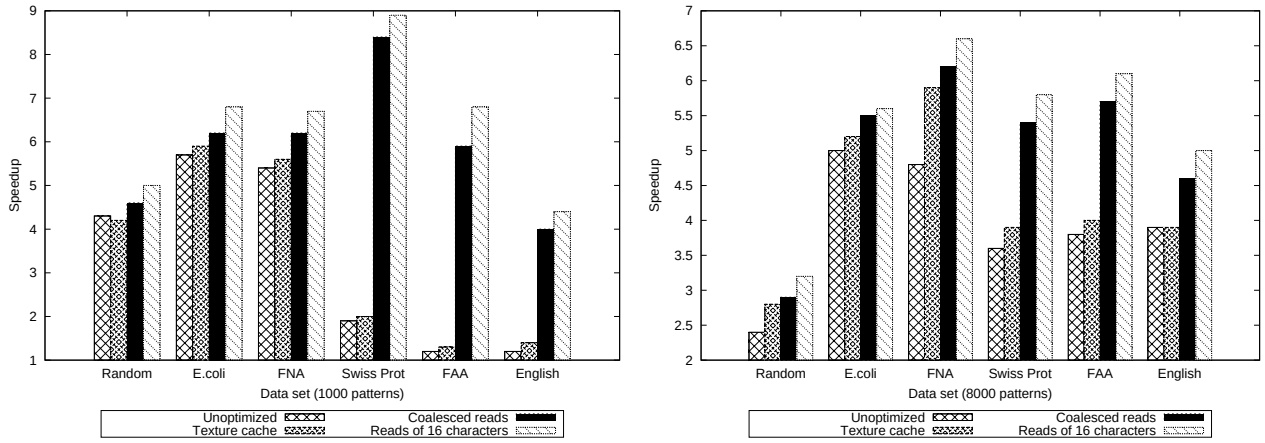


FIG. 4.5. *Speedup of different SOG implementations for randomly generated input strings, biological sequence databases and English alphabet data*

multiple pattern matching algorithms. The performance of the unoptimized kernel increased by up to 5.7 times, comparing to the sequential implementation. As can be seen in Figure 4.5, the algorithm implementation had a significant speedup increase when the preprocessing arrays were bound to the texture memory and the input string was copied to the shared memory of the device. The increase was more evident for the Swiss Prot. Amino Acid sequence database, the FASTA Amino Acid of the A-thaliana genome and for the English language datasets, all of them data with a large alphabet size. The optimized algorithm implementation had a speedup of up to 8.9 that was observed when the Swiss Prot. Amino Acid sequence database was used together with sets of $1,000$ patterns.

From the results, we can also see that there are two trends in the performance of the CUDA implementations when the number of patterns is scaled. The speedup of the Aho-Corasick, Set Backward Oracle Matching, Wu-Manber and SOG algorithms had a rising tendency as the patterns increased for Random, FAA, FNA/Swiss Prot and FNA/FAA/English databases respectively, in contrast with the other cases where the speedup of the algorithms had a decreasing tendency. This decreasing tendency for large pattern sets is due to the fact that the preprocessing phase of the algorithms was performed sequentially on the CPU. More specifically, the preprocessing phase of the algorithms imposes a relatively high cost in terms of time to setup the necessary data structures and the time to preprocess the pattern set increases linearly with the number of patterns d. Furthermore, the searching phase on the GPU becomes slower for large pattern sets because of the unfavorable GPU memory hierarchy access times. This fact has been alleviated in more recent versions of GPU devices.

Although the performance of the CUDA implementations increased significantly when different optimization techniques applied, the speedup of the implementations is not proportional to the number of cores in GPU. This fact is due to three reasons: One reason is that the implementations of the algorithms require the frequent access of the threads to the GPU memory hierarchy of the device. Another limiting factor is the significant overhead of the thread divergence, the serial execution of threads within the same warp that execute different instructions as a result of *if-else* and *while* statements in the kernel. Furthermore, the optimized implementations of each algorithm require increased register usage per thread to execute out of the 16KB registers per block that the GTX 280 provides. More specifically, the algorithms Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG require 18, 19, 22, 22 and 21 registers per thread, respectively. The number of registers per thread is a limiting factor on the number of concurrent threads the hardware is capable of executing. The proposed implementations could be scaled up by using a large number of GPUs by means of cuda-aware MPI infrastructure and by applying suitable modifications of the data structures used in preprocessing phase of the algorithms so that pattern set can be partitioned and the GPU memory hierarchy is better utilized.

5. Conclusions. In this paper, we presented details on the design and the implementation of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG multiple pattern matching algorithms in parallel on a GPU using the CUDA API. Different optimizations were used to further increase the performance of the parallel implementations. Based on the experimental results, it was concluded that the speedup of the basic implementation of the algorithms ranged between 2.5 and 10.9 over the equivalent sequential version of the algorithms. It was also shown that by applying the optimizations discussed in this paper, the parallel implementation of the Aho-Corasick algorithm was up to 18.5 times faster than the corresponding sequential implementations, the implementations of the Set Horspool and the Set Backward Oracle Matching algorithms were up to 33.7 times faster, the parallel implementation of the Wu-Manber algorithm was up to 11.8 times faster while the implementation of SOG was up to 8.9 times faster.

It was concluded that the parallel implementations of the Set Horspool and the Set Backward Oracle Matching algorithms exhibited higher speedup than other three parallel implementations. Note that the experimental results presented herein for Aho-Corasick algorithm achieve a high speedup of 18.5, which compares favorably with the results presented in the previous research studies [40, 45]. Further, the speedup presented herein for parallel version of the Wu-Manber (WM) algorithm is relatively better than the parallel version of the WM algorithm that has been implemented using OpenGL and – achieved twice the performance of the corresponding WM algorithm used in Snort [15]. The speedups reported in those articles are comparable to the speedups of this paper because the aforementioned research proceed in a fashion similar to our work, i.e. they parallelize the same original Aho-Corasick and WM algorithms. On the other hand, two approximate string matching implementations, based on a modified Wu-Manber algorithm [23] and the bit-parallel BPR algorithm [38] achieve better results than those presented herein. The speedups reported are 76 and 30, in the former reference and 62, in the latter. Furthermore, some other GPU implementations [24, 39, 14, 42] seem to perform better than those presented herein. These studies have incorporated significant modifications of the original Aho-Corasick and Wu-Manber algorithms, such that the resulting GPU implementations avoid important problems such as memory accesses of threads, thread divergence between cores and hashing comparisons/calculations. Therefore, the resulting speedups are not directly comparable to the reported speedups of ours or other approaches, since the sequential algorithm, i.e. the basis of comparison has been modified significantly. Furthermore, in all the previous cases, the direct comparison of absolute speedups is not possible, since the experimental setups used in different experiments, that is host machines and GPU devices, are not directly comparable.

Acknowledgments. The authors are thankful to the reviewers for the useful comments and suggestions, which improved the presentation of this paper.

REFERENCES

[1] A. Aho and M. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Communications of the ACM, 18 (1975), pp. 333–340.
[2] C. Allauzen, M. Crochemore, and M. Raffinot, *Factor Oracle: A New Structure for Pattern Matching*, SOFSEM99: Theory and Practice of Informatics, 1725 (1999), pp. 758–758.
[3] A. Apostolico and Z. Galil, *Pattern Matching Algorithms*, Oxford University Press, 1997.
[4] S. Arudchutha, T. Nishanthy, and R. Ragel, *String matching with multicore cpus: Performing better with the aho-corasick algorithm*, in Industrial and Information Systems (ICIIS), 2013 8th IEEE International Conference on, Dec 2013, pp. 231–236.
[5] R. Baeza-Yates and G. Gonnet, *A New Approach to Text Searching*, Communications of the ACM, 35 (1992), pp. 74–82.
[6] R. Boyer and J. Moore, *A Fast String Searching Algorithm*, Communications of the ACM, 20 (1977), pp. 762–772.
[7] X. Chen, B. Fang, L. Li, and Y. Jiang, *WM+: An Optimal Multi-pattern String Matching Algorithm Based on the WM Algorithm*, Advanced Parallel Processing Technologies, (2005), pp. 515–523.
[8] B. Commentz-Walter, *A String Matching Algorithm Fast on the Average*, in Proceedings of the 6th Colloquium, on Automata, Languages and Programming, 1979, pp. 118–132.
[9] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, *Speeding Up Two String-matching Algorithms*, Algorithmica, 12 (1994), pp. 247–267.
[10] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter, *Fast Practical Multi-pattern Matching*, Information Processing Letters, 71 (1999), pp. 107 – 113.
[11] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, Inc., 1994.
[12] G. Grep, *Webpage containing information about the gnu grep search utility.* Website, 2012. http://www.gnu.org/software/grep/.

[13] R. Horspool, *Practical Fast Searching in Strings*, Software: Practice and Experience, 10 (1980), pp. 501–506.
[14] L. Hu, Z. Wei, F. Wang, X. Zhang, and K. Zhao, *An Efficient AC Algorithm with GPU*, Procedia Engineering, 29 (2012), pp. 4249–4253.
[15] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai, *A GPU-based multiple-pattern matching algorithm for network intrusion detection systems*, in Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on, March 2008, pp. 62–67.
[16] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, *Kargus: a Highly-scalable Software-based Intrusion Detection System*, in Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2012.
[17] P. Kalsi, H. Peltola, and J. Tarhio, *Comparison of Exact String Matching Algorithms for Biological Sequences*, Communications in Computer and Information Science, 13 (2008), pp. 417–426.
[18] S. Kim and Y. Kim, *A Fast Multiple String-pattern Matching Algorithm*, Proceedings of the 17th AoM/IAoM Inernational Conference on Computer Science, (1999), pp. 1–6.
[19] D. Knuth, J. Morris, and V. Pratt, *Fast Pattern Matching in Strings*, SIAM Journal on Computing, 6 (1977), pp. 323–350.
[20] C. Kouzinopoulos, *Parallel and Distributed Implementations of Two-Dimensional and Multiple Pattern Matching Algorithms*, PhD thesis, Department of Applied Informatics, University of Macedonia, 2013.
[21] C. Kouzinopoulos, P. Michailidis, and K. Margaritis, *Performance Study of Parallel Hybrid Multiple Pattern Matching Algorithms for Biological Sequences*, in International Conference on Bioinformatics-Models, Methods and Algorithms, 2012, pp. 182–187.
[22] T. Lecroq, *Fast Exact String Matching Algorithms*, Information Processing Letters, 102 (2007), pp. 229–235.
[23] H. Li, B. Ni, M.-H. Wong, and K.-S. Leung, *A fast CUDA implementation of agrep algorithm for approximate nucleotide sequence matching*, in Application Specific Processors (SASP), 2011 IEEE 9th Symposium on, June 2011, pp. 74–77.
[24] C. Lin, S. Tsai, C. Liu, S. Chang, and J. Shyu, *Accelerating String Matching using Multi-Threaded Algorithm on GPU*, in Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE, 2010, pp. 1–5.
[25] P. Liu, Y. Liu, and J. Tan, *A Partition-based Efficient Algorithm for Large Scale Multiple-strings Matching*, in String Processing and Information Retrieval, Springer, 2005, pp. 399–404.
[26] P. D. Michailidis and K. G. Margaritis, *Accelerating kernel density estimation on the GPU using the CUDA framework*, Applied Mathematical Sciences, 7 (2013), pp. 1447–1476.
[27] R. Muth and U. Manber, *Approximate Multiple String Search*, in Combinatorial Pattern Matching, Springer, 1996, pp. 75–86.
[28] G. Navarro and K. Fredriksson, *Average Complexity of Exact and Approximate Multiple String Matching*, Theoretical Computer Science, 321 (2004), pp. 283–290.
[29] G. Navarro and M. Raffinot, *A Bit-parallel Approach to Suffix Automata: Fast Extended String Matching*, Lecture Notes in Computer Science, 1448 (1998), pp. 14–33.
[30] ———, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, 2002.
[31] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 5.5*, 2013. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.
[32] C. Pungila and V. Negru, *A Highly-Efficient Memory-Compression Approach for GPU-Accelerated Virus Signature Matching*, Information Security, (2012), pp. 354–369.
[33] Y. Qi, Z. Zhou, Y. Wu, Y. Xue, and J. Li, *Towards high-performance pattern matching on multi-core network processing platforms*, in Proceedings of the Global Communications Conference, 2010. GLOBECOM 2010, 6-10 December 2010, Miami, Florida, USA, 2010, pp. 1–5.
[34] L. Salmela, J. Tarhio, and J. Kytöjoki, *Multipattern String Matching with q-grams*, Journal of Experimental Algorithmics, 11 (2006), pp. 1–19.
[35] S. Sheik, S. Aggarwal, A. Poddar, B. Sathiyabhama, N. Balakrishna, and K. Sekar, *Analysis of String-searching Algorithms on Biological Sequence Databases*, Current Science, 89 (2005), pp. 368–374.
[36] Snort, *Webpage containing information on the snort intrusion prevention and detection system.* Website, 2010. http://www.snort.org/.
[37] G.-M. Tan, P. Liu, D.-B. Bu, and Y.-B. Liu, *Revisiting multiple pattern matching algorithms for multi-core architecture*, Journal of Computer Science and Technology, 26 (2011), pp. 866–874.
[38] T. Tran, M. Giraud, and J.-S. Varr, *Bit-parallel multiple pattern matching*, in Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waniewski, eds., vol. 7204 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 292–301.
[39] A. Tumeo, S. Secchi, and O. Villa, *Experiences with String Matching on the Fermi Architecture*, Architecture of Computing Systems-ARCS 2011, (2011), pp. 26–37.
[40] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, *Gnort: High Performance Network Intrusion Detection using Graphics Processors*, Proceedings of RAID, 5230 (2008), pp. 116–134.
[41] G. Vasiliadis and S. Ioannidis, *Gravity: A Massively Parallel Antivirus Engine*, in Recent Advances in Intrusion Detection, Springer, 2010, pp. 79–96.
[42] L. Vespa and N. Weng, *SWM: Simplified Wu-Manber for GPU-based Deep Packet Inspection*, in Proceedings of the 2012 International Conference on Security and Management, 2012.
[43] S. Wu and U. Manber, *Agrep - A Fast Approximate Pattern-Matching Tool*, In Proceedings of USENIX Technical Conference, (1992), pp. 153–162.
[44] ———, *A Fast Algorithm for Multi-pattern Searching*, (1994), pp. 1–11. Technical report TR-94-17.

[45] X. Zha and S. Sahni, *Multipattern String Matching on a GPU*, in Computers and Communications (ISCC), 2011 IEEE Symposium on, IEEE, 2011, pp. 277–282.
[46] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li, *MDH: A High Speed Multi-phase Dynamic Hash String Matching Algorithm for Large-Scale Pattern Set*, Information and Communications Security, 4861 (2007), pp. 201–215.

Edited by: Marian Gusev
Received: Dec 21, 2014
Accepted: Mar 30, 2015

DOI 10.12694/scpe.v16i2.1086
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 139–151. http://www.scpe.org

A SCALABLE AND DISTRIBUTED CLOUD BROKERING SERVICE

ALBA AMATO* AND SALVATORE VENTICINQUE†

Abstract. Cloud computing can be considered the key of the development of ICT systems business. It allows to work with the basic tools, but in a dynamic, mobile and technologically advanced way. Taking a good choice of Cloud provider, is the key to safely embracing the Cloud and the benefits that it provides. To do that is necessary to identify the Cloud service needs based on the application requirements. Some applications will be more critical than others. Moreover it is necessary to select the provider which best fits those requirements. In fact, as happens to a more traditional delivery model in which organizations use different product packages from different providers because no single supplier can meet all of their needs, there are different Cloud providers that offer different features and perks, and many of them are not directly comparable. In this paper a scalable distributed multi-users version of a Broker As A Service solution is proposed, mainly suitable for exploiting the capability of a distributed environment and for addressing the related issues. The idea is to decompose global broker into a set of distributed brokers that cooperate and dynamically scale, together with the computing infrastructure, to support unforeseeable workloads produced by the interactions with large groups of users. The brokering problem is divided into simpler tasks, which are distributed among independent agents, whose population dynamically scales together with the computing infrastructure, to support unforeseeable workloads produced by the interactions with large groups of users. This solution is also suitable in case of Multi-Cloud brokering. Several experimental results have been performed to verify the effectiveness of the proposed system.

Key words: Intelligent Agents; Distributed Brokering; Cloud Computing

1. Introduction. In the IT world as we know it today computers become exponentially more powerful and cost per unit of resources is rapidly decreasing, so that IT can be considered a commodity. On the other hand IT becomes more and more pervasive in organizations and the increasing complexity in managing the entire infrastructure, made up of different software architectures within which the information is distributed, has led companies to spend a growing amount in IT. The diffusion of the phenomenon of Cloud Computing has generated a strong interest in the companies, in fact Cloud computing can be considered the key to the development of ICT systems business. It is a fundamental revolution in the way Information Technology (IT) services are created, developed, deployed, updated, maintained and paid for.

In particular, among the benefits of adopting Cloud models, elasticity plays a relevant role. In fact it allows for providing the required resources to the deployed applications according to the current workload and paying just for their usage.

Especially in a service oriented context the exploitation of such a facility is desired. A service must support access to multiple users simultaneously ensuring always the required level of availability and performance. The application workload can vary during the day, with regular or unforeseeable bursts on special periods. Moreover the service can be invoked also by applications and robots and not only by human users.

However scalability of a service with its changing workload is not enough to guarantee the exploitation of Cloud elasticity. Providing a service with self-adaptivity to scale dynamically with a changing computing infrastructure is not trivial for any new developed applications or legacy ones when they are ported in the Cloud.

This paper discusses this issues presenting a scalable distributed Broker As A Service solution that has been designed for exploiting the elasticity of a Cloud computing infrastructure. Such a broker service supports the users to take a good choice among the many alternatives of Cloud services offered by the increasing number of different providers [19].

The idea is to decompose a global broker into a set distributed brokers, which cooperate and dynamically scale, together with the computing infrastructure, to support unforeseeable workloads produced by the interactions with large groups of users. This solution is suitable also in case of Multi-Cloud brokering, where the optimal choice is a collection of services provided by heterogeneous providers. The remainder of this paper is structured as follows. Some related approaches are introduced in 2. Section 3 motivates, introduces, and

*Department of Industrial and Information Engineering, Second University of Naples, Aversa, Italy (alba.amato@unina2.it).
†Department of Industrial and Information Engineering, Second University of Naples, Aversa, Italy (salvatore.venticinque@unina2.it).

TABLE 2.1
Brokering Solutions

Reference	Properties	Limitations
RightScale	select, migrate and monitor different Clouds	missing negotiation and optimization features
CloudSwitch	make simpler the process of migrating an application or workload to the Cloud	missing proper brokerage service
JamCracker	aggregates and distributes on-demand services through a global ecosystem of Service Providers	missing a proper brokerage service
[9]	based on a Cloud service register catalog where it performs a search of the desired proposal	missing a proper brokerage service
[12]	focused on the scheduler	only cost optimization
OPTIMIS	under development	only cost optimization
[14]	optimize placement of VMs across multiple cloud providers	dynamic cloud scheduling not supported

analyzes the problem, whose solution's requirements, approach and implementation are then investigated in Sections 4, 5 and 6. Sections 7 and 8 concludes with an overview and interpretation of the results.

2. Related Work.

2.1. Background. In recent years, Gartner has analyzed in great detail the role of the Cloud Service Brokers [6]. In the first place, Gartner defines Cloud Services Brokers (CSB) it as an IT role and a business model where a company or other entity adds value to one or more Cloud services (public or private) on behalf of one or more consumers of this service through three roles primary: aggregation, integration and customization. The activity of the CSB has become necessary in recent years for the diffusion of solutions of hybrid Cloud involving multiple Cloud services. Furthermore, the role of the CSB has a positive impact both for business and IT operations. In fact organizations are increasingly adopting Cloud solutions, so the supply of services is increasing more and more. Companies must therefore address issues of greater complexity in the adoption and must be able to handle a large number of providers.

2.2. Brokering Solutions. For those reasons several companies offers brokering solutions which are summarized in Table 2.1. RightScale [17] Self-Service is a solution that enables IT users to deliver self-service access to a curated catalog of commonly used components, stacks, and applications and an easy-to-use interface for developers and other Cloud users to request, provision, and manage these services across public Cloud, private Cloud, and virtualised environments. RightScale Self-Service includes a Cloud service catalog that enables users to design a set of Cloud applications that can be configured with the application versions, security patches, and software configurations that match company standards. It also includes built-in cost management features that enable users to set usage quotas in order to stay within set project budgets. Nevertheless it is focused on select, migrate and monitor different Clouds from a single management environment and does not provide negotiation and optimization features.

CloudSwitch [18] is a Cloud broker software vendor that helps enterprises to move Cloud data more easily. It advertises itself as a the enterprise gateway to the Cloud and provides the ability to integrate public application program interfaces (APIs) with Cloud providers to make the process of migrating an application or workload to the Cloud as simple as a drag-and-drop action in a web browser. Nevertheless it does not really provide a proper brokerage service but it gives customers an interface to port applications to several different providers.

The JamCracker system, available in [7], provides a platform where it aggregates and distributes on-demand services through a global ecosystem of Service Providers, Resellers, System Integrators, and ISVs but it does not provide a proper brokerage service whereby an entity looks at the actual QoS requirements of the service under question, the various IPs that could potentially meet them, rank them against parameters like cost, trust,

eco-efficiency, risk etc. and provide functionality to on board these applications in to the various IPs finally selected as stated in [15]. The brokering problem is also investigated in several scientific communities and European Project.

The Cloud-service broker, presented in [9] as an emerging technology, intermediates heterogeneous multiple Cloud services for both providers and consumers. The Cloud-service-broker portal enables the Cloud service providers to specify that their services are available. In addition, the Cloud-service-consumers may find the most suitable services by negotiating the agreements on the services. It is based on a Cloud service register catalog where it performs a search of the desired proposal. Nevertheless it does not assist users with decision making to evaluate and select a Cloud vendor or solution based on specific requirements.

A Cloud broker architecture for deploying virtualised servers across available Clouds is presented in [12]. The paper is focused specially on the scheduler, that is one of the three components. Other components are the VM manager and the Cloud manager. In particular the paper addresses the placement challenge by using different algorithms developed for optimizing the cost of the required infrastructure. Specifically, it utilizes a prediction model which takes into account the historical prices of available Cloud providers in order to calculate, for the next hour deployment, the best Cloud provider to move the VMs to. Nevertheless it is focused on cost optimization and does not consider the other different criteria.

In the scope of the EU FP7 project OPTIMIS, an architectural design of a framework capable of powering the brokerage of Cloud services is proposed and is under development. It is described in [13] that introduces the problem and the architectural design, but it does not provide an implementation or algorithms to achieve the brokering. The main features are to ensure data confidentiality and integrity to service customers, to match the requirements of Cloud consumer with the service provided by the provider, to negotiate with service consumers over Service Level Agreements (SLAs), to maintain performance check on these SLA's and take actions against SLA violation. Moreover it aims to effectively deploy services provided by the Cloud provider to the customer, to manage the API so that provider does not learn anything about the identity of the service consumer, and to manage security problems.

The paper [14] proposes an architecture for cloud brokering and multi-cloud VM management, describes algorithms for optimized placement of applications in multi-cloud environments. The proposed model incorporates price and performance, as well as constraints in terms of hardware configuration, load balancing, etc. An evaluation against commercial clouds demonstrates that compared to single-cloud deployment, a multi-cloud placement algorithms improve performance, lower costs, or provide a combination thereof. It becomes evident that capabilities of optimization and decision making to evaluate and select a Cloud vendor or solution based on specific requirements, which is the focus of the proposed approach have little or no support in Cloud service brokerage available today.

In preliminary work [3, 4], the authors present the architecture of the Broker Agent and its implementation within the EU FP7 mOSAIC project for provisioning of brokering service at Cloud platform level. The proposed Cloud broker provides several facilities in order to optimize different variables like overall infrastructure cost, service performance, or others, depending on the strategies selected by the user. Moreover it gives the possibilities to set different optimization criteria and several restrictions to support user in the choice of the provider that is more adapt to their applications. According to [16], the agility and flexibility are the main elements that encourage people to adopt the Cloud, so the Cloud broker will assume a key role in the future deployment of solutions as a service. But this kind of multi-criteria optimization problems are hard to compute, so it is necessary a scalable, multi-user, distributed solution. In this paper is described a new scalable and distributed approach that overcomes the performance limitations of paradigm as a service. Some experimental results are also presented.

3. Problem Statement. Deployment of Cloud services makes it necessary to select the provider that best fits the application requirements. Usually it is not a trivial task because there are different Cloud providers that offer different features and perks, and many of them are not directly comparable. Besides it could be necessary to consider composition of services from different providers, as it happens when organizations use different product packages because no single supplier can meet all of their needs.

The broker realized in the mOSAIC FP7 project [1] is designed and developed to support decision making by a Cloud deployer. It is centralized on a single brokering process, which takes care of evaluating all available

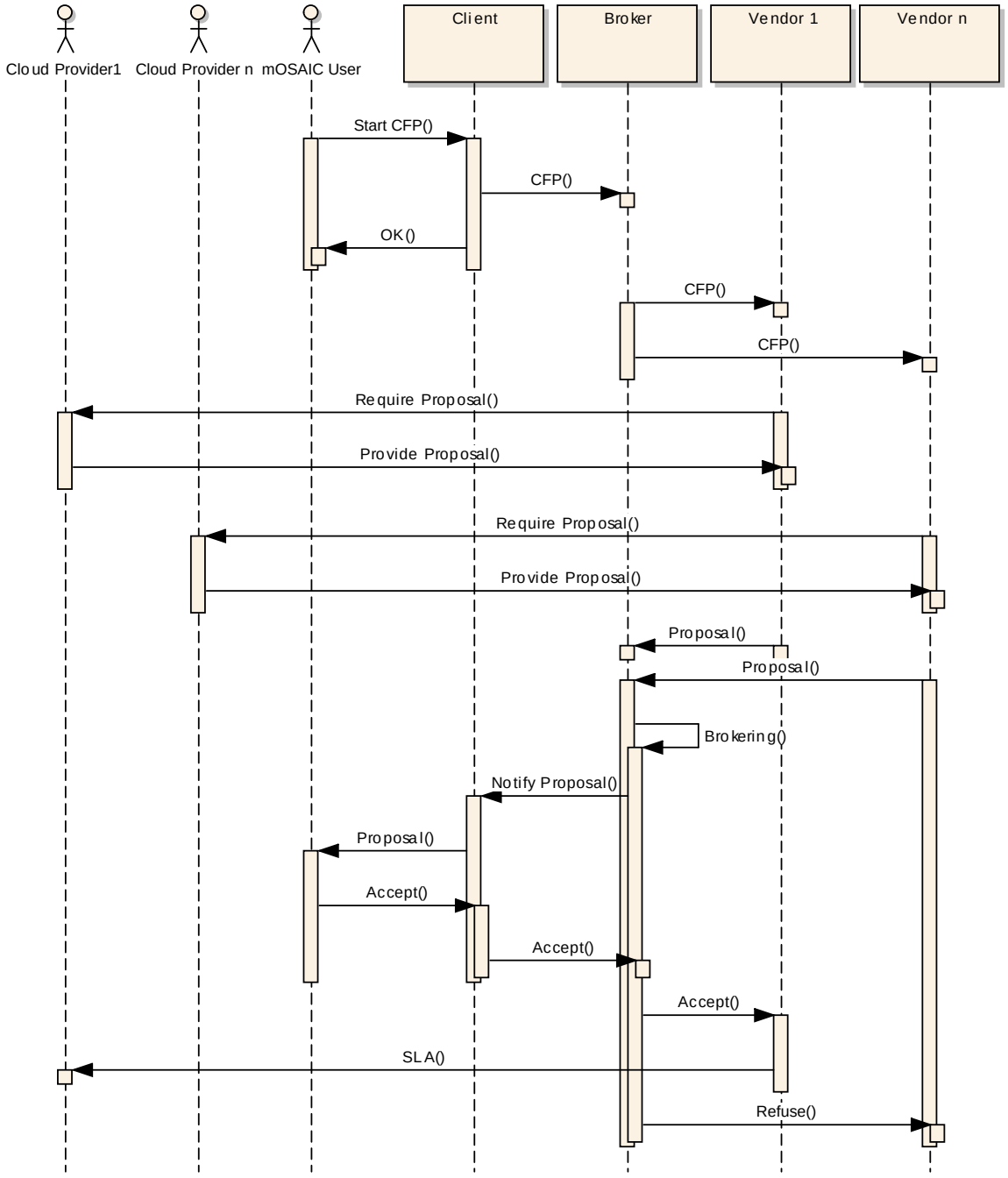


FIG. 3.1. *mOSAIC Broker*

alternatives limited to a single user's preferences. Moreover in this project the brokering of service composition has not been addressed.

The mOSAIC Broker is shown in Figure 3.1. For each received call for proposal (CFP), cloud agency creates a broker that searches for vendors that can offer resources with the required QoS (Quality of Services). As shown in Figure 3.1, the broker collects a number of proposals described in a vendor agnostic way and chooses the

best one(s) according to the brokering rules.

To provide such an application *as a service*, with the aim to overcome the single-user delivery and the atomic service brokering, new requirements and challenges will be introduced. The former limitation deals with the need of handling increasing and dynamically changing workloads. The latter constraint affects the response time and the complexity of a single brokering problem.

With this premise a centralized approach is not feasible anymore. But Cloud allows to scale the computing resources according to the measured workload, so improving also their utilization. The idea is to exploit such elastic computing model for building a distributed Cloud broker over a Cloud infrastructure. But the application must be able to reconfigure itself autonomically to keep the QoS level above the desired threshold. Due to lack of control both of the network and of the compute utility, not only performance issues should be addressed, but also reliability and availability when the service execute over such kind of distributed platform.

At a glance it is possible to identify three different scenarios for our brokering solution with improving capabilities.

1. A *mOSAIC Broker* is a *single-user application* that gets *one CFP* and returns *many atomic proposals*.
2. A *brokering As A Service* is a *multi-user application* that gets *many CFP* and returns *many atomic proposals*.
3. A *multi-Cloud brokering As A Service* is a *multi-user application* that gets *many CFP* and returns *many compositions of proposals*.

Moving from the first scenario to the second ones implies the handling of parallel requests from multiple users. The service must be capable to distribute the workload among many worker nodes. A trivial solution would be the execution of brokering processes shown in Figure 3.1 across the nodes according to a task parallel approach. This solution would limit a fine grain balancing of the workload among working nodes because different brokering problems could have different completion time. In addition it prevents the utilization of the distributed resources to solve a single brokering problem, that can become computationally hard when we move from the second to the third scenario.

4. Requirements for an elastic brokering and design guidelines. Addressing the design of a Cloud service two different requirements have to be taken into account: the interaction with the service requestor at front-end and the processing of service requests at back-end. Service levels at front-end are measured by the availability and responsiveness. The first one regards the capability to accept service requests from requestors. It could be independent from the processing capability at back-end, above all if the results must not be provided in real time. The second one is about as much interactive should be the service. It affects in advance the quality perceived by the requestor beyond the time within the response is expected to be available. In the presented case the idea is to accept the brokering requests and to provide information to the requestor about the status of the job, or about intermediate results with a certain level of responsiveness. Design solutions for exploitation of elasticity at front-end are quite common. Availability is the first service level that every Cloud provider grants into the SLA. Moreover most of Cloud services are conceived to provide web interface and interactive interfaces.

The design guidelines suggest to implement a synchronous front-end that is deployed on multiple virtual machines. The Cloud infrastructure will monitor the resource utilization of each virtual machine and will start/stop new instances. A load balancer will distribute the requests between the running replicas. The development of a stateless pool of web services is a mandatory choice to allows for a simple automatic scaling of the front-end. Besides the status of the web session is shared by Cloud data services. Distributed cache will provide reliability, allow transparency between service instances and load balancer, improving responsiveness.

Service levels at back-end are throughput and the completion time of each job. Here it is important that the utilization of the computing resources is maximized. The effective scheduling for load balancing of jobs and their fair allocation is the most relevant issue for scalability. A stateless solution is again desired for allowing any new idle worker to get any waiting job and to execute it using available resources. The usage of NOSQL solutions is quite common to partition the data providing availability, and facilitating the scheduling. Key value stores, queues or big tables [10] are example of services that store the application data. Queue services are used for synchronization and to route the message among workers. Asynchronous solutions improve utilization and facilitate the scheduling. The routing algorithm of the queue can be used to add intelligence to the scheduling by distributing the workload.

In the presented case it is important to scale the system to collect as many brokering requests (call for proposals) as they come. The only limitation will be the number of virtual machine instances to use and pay for, both to run the service front-end and the service back-end. Although many load balancing schemes have been presented in Cloud computing, there is no scheme providing the elasticity and adaptive adjustment in Cloud computing. This allows us to address fine-grained load balancing and scalability at design time and at component level. It is possible to model the brokering as a data parallel problem. The pool of stateless worker will be composed of vendors, brokers and SLA managers. Vendors will run on behalf of providers. They receive CFPs and are in charge to generate correspondent proposals. Brokers will evaluate the score of a proposal comparing it with the correspondent CFP, without caring about the specific transaction and about who is the requestor or the vendor, producing a new SLA candidate. SLA manager compare the score of each new produced SLA candidate with the ones belonging to the same transaction, filtering only the best ones.

In this scenario the data flow generates events which trigger asynchronous workers. A new CFP trigger idle vendors. New proposals triggers idle brokers and new SLA candidates activate SLA managers.

To conclude this section it is necessary to discuss about real time requirements, which can become relevant for such a kind of service. Let us suppose that we must integrate services form different providers, when constraints and preferences are not satisfied by only one. In this case it is necessary to consider all the possible compositions of services, being aware about how the Service Levels and cost of the composite service change. The brokering problem can become computationally hard as it is shown in [2] because of the exploration of a wide solution space. In fact the number of solutions to be evaluated grows exponentially and an exhaustive evaluation of all combination can become unfeasible. Heuristic approaches such as multi-objective genetic algorithms can reduce the computational requirements and the execution time, but introduce a loss of optimality. In this case it is possible to provide a sub-optimal solution within time constraints that can be acceptable for the user.

To integrate such approach within our Cloud service it is necessary to add an expiration time to the proposals and the capability for the SLA manager to trigger a new CFP that allows the vendors to produce the next generation of proposals.

This solution have been described in [5] and [11] where also experimental results have been shown. Experimental results show that Multi-Cloud brokering service can reduce the computational requirements and the execution time, but introduce a loss of optimality. The optimality is affected by the specific genetic algorithm, by its parameters such as the crossover, by the number of individuals evaluated (that means the execution time), the number of objectives to be minimized/maximized, the kind of problem. Performance evaluations showed the feasibility of the approach and that solutions approximated well the true Pareto front in all the cases at least in some condition. In any case the generality of the discussion about the scalability of the Cloud brokering service is not affected by the typology of solution.

5. System Overview. The front-end architecture is composed of a web service that allows for

- Registration: Users can create a profile and have a unique personal account having the possibility to choose his credentials (username and password).
- CFP submission: the logged users can insert one or more Call For Proposal. Users that have inserted a CFP can run a new request, in a completely asynchronous way.
- Proposals upload: Cloud Providers can insert proposal and update the portfolio of their offer.
- Brokering results: Users can consult their requests, the status of the requests and the available results.

Figure 5.1 shows the front end section with several service instances that receive requests from end users and access in-memory shared information. The in-memory session manager will allow for the exploitation of elastic capability of the Cloud infrastructure. The warm copy of the session manager allows for improving reliability.

The back-end architecture is composed of stateless and asynchronous agents whose tasks are scheduled by service queues. Those agents have to be able to run on every available computing resources and perform the brokering. Nevertheless the front-end must be implemented by a service interface that accepts synchronous requests and forward them to an asynchronous back end. Those new pending tasks are stored in order to be handled by pools of asynchronous working agents that will return the current status of service elaboration.

As it is shown in Figure 5.1 agents will implement the back-end of the new agency, that is completely free from management of the interactions with clients. Cloud storages (database ad queues) keep persistent information of the distributed applications and implement communication channel between synchronous front-

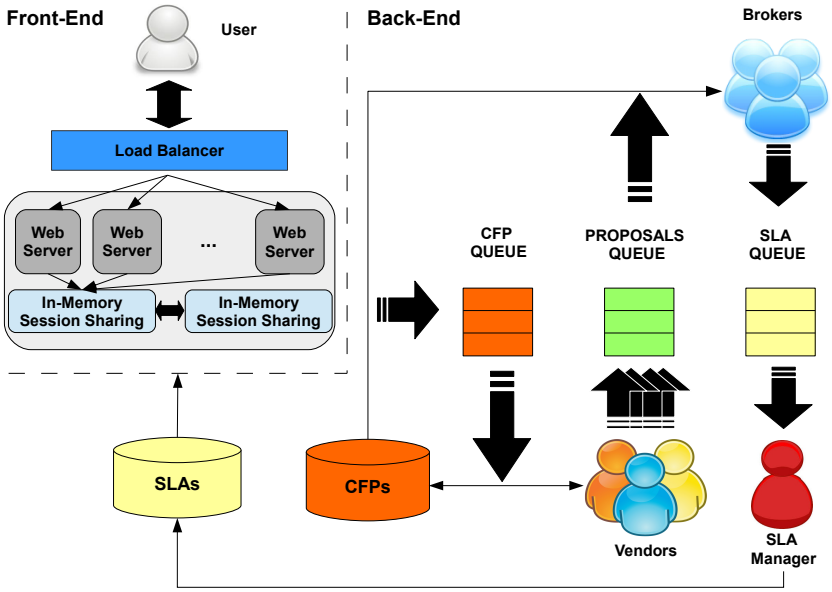


FIG. 5.1. *Distributed architecture*

end handlers and back-end workers. This design choice has been done in order to obtain an easy distribution of the workload using a task parallel programming model.

Stateless agents get new problems by a common Bag-of-Tasks in a concurrent and parallel way, executing wherever there are available computing resources, and updating the computing results if they complete successfully. If there are unsolved problems which remain in the Bag-of-Tasks because of any failures or delay, they are re-scheduled, so ensuring reliability.

The vendors agent sign up to the CFP_QUEUE to receive the CFPs received from users. Each of them submits its proposal to the PROPOSALS_QUEUE. Idle brokers are waiting for proposals. A single proposal is dispatched to one broker that executes its matching with the correspondent CFP for evaluation purpose. The matching results is stored into the SLA_QUEUE if it belongs to the Pareto front of optimal solutions. ActiveMQ has been used as queue services for communication and synchronization. Brokering results are stored also into an in-memory session, together with information of each related user's request.

6. Prototype Implementation. The implementation of a prototype shows the utilization of open-source tools which results in a rapid implementation with minimal or no software input cost.

Users are provided with the web interface shown in Figure 6.1. It allows for composing the CFP and listing, for each session, the best SLAs according to different brokering objectives. Users log into a web page, the web page makes a request to a REST service and submits the call for proposals.

The Call for Proposals are included, along with information of the session and the user, in CFP queue. The characteristic of the CFP queue is that all consumers that join the queue, receive all the submitted CFPs. Apache Tomcat has been used as web and application server to run the web service at front-end. It has been specifically configured for working with the Terracotta Framework for the transparent distribution and sharing of web sessions. Jersey API have been used to implement the RESTFull service that provide methods for authentication, CFP submission and SLA retrieval.

Meanwhile their requests are pending, users can wait for the result of brokering or may poll periodically to get the status of their request. When one of the brokers has found a feasible proposal for a certain request, the current results are updated so that user can get it. The brokering terminates when there are no more proposal

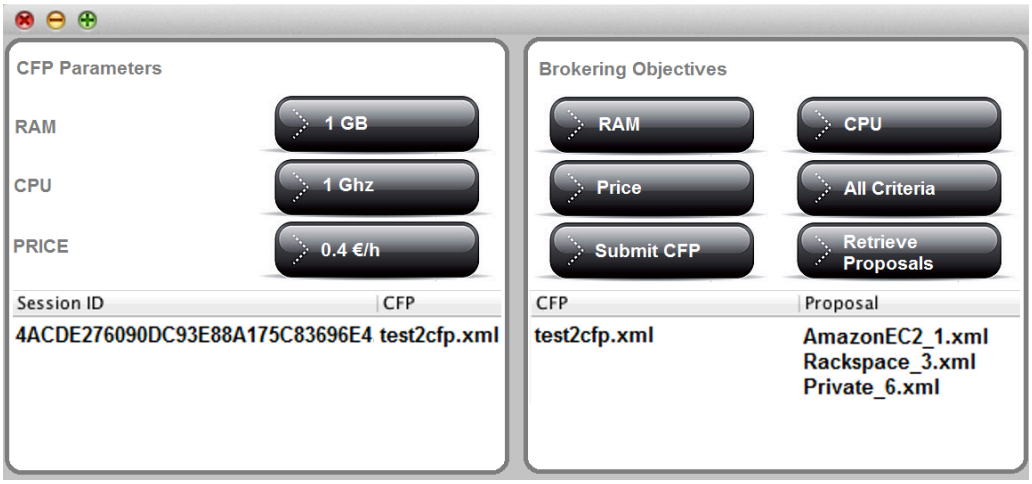


FIG. 6.1. *Web GUI*

suitable for optimizing the user's query to be evaluated. Then the termination is notified to the user.

CFPs and SLAs are stored in the RDBMS Mysql databases, that is used as persistent storage.

The Web and Application Server chosen to run the web service at front-end is Apache Tomcat that is the most popular open-source Java Web application server [20]. Tomcat, typifying object-based servers, has some unique runtime properties including intensive threading, high read/write ratios, extensive object sharing via collections framework and fine-grained irregular object access patterns [8].

Terracotta [21] is a JVM-level clustering product. It works within an aspect-oriented programming (AOP) framework and has to instrument product-specific classes. Users need to manually specify shared classes as distributed shared objects (DSOs) and their cluster-aware concurrency semantics.

Tomcat has been specifically configured for working with the Terracotta Framework for the transparent distribution of web sessions.

Jersey API, that allow the creation of Asynchronous RESTFul web service, have been used by the service requestor and by vendor agents to handle HTTP service requests and responses. The relational database management system (RDMS) is implemented by Apache Derby. Finally Apache ActiveMQ provided us a distributed open source queue service. The queue service is very important because the scalability of the proposed architecture is inherently tied to the assumption that the message queue can scale perfectly. Figure 6.2 shows the sequence diagram of the system. The first action that is performed is the login phase. As one can see, the client sends an http request (post) to the server, which creates an instance of access for managing users and their profiles. After the authentication phase, users can enter their CFPs. In particular, the client sends a request for insert a CFP to the server, which creates an instance of Services class that uses the Producer class to access a Queue. Similarly, the client sends a request for insert a CFP to the server, which creates an instance of VendorServices class that uses the Producer class to access a Queue. A generic client sends a request for insert a proposal to the server, which creates an instance of VendorServices class that uses the Producer class to access a Queue. Indexing agents and broker agents are created to get elements respectively from the PROPOSAL QUEUE and from the CFP QUEUE as they have been populated by vendors and broker clients. In particular, a consumer process (which runs in the background) withdraws proposals from the proposals queue and inserts them from time to time in a database. The Consumer send a getProposal() message to Queue and obtains a proposal to be sent to the DB for store it. Brokers operate in background and send requests to the class queue to get a CFP and class DB to get the proposals inserted. After that, they start the brokering and send the result to the Session class that, using a database, returns the result to the user with the corresponding ID.

7. Experimental Results. In order to evaluate performances of the proposed approach the following testbed were set up.

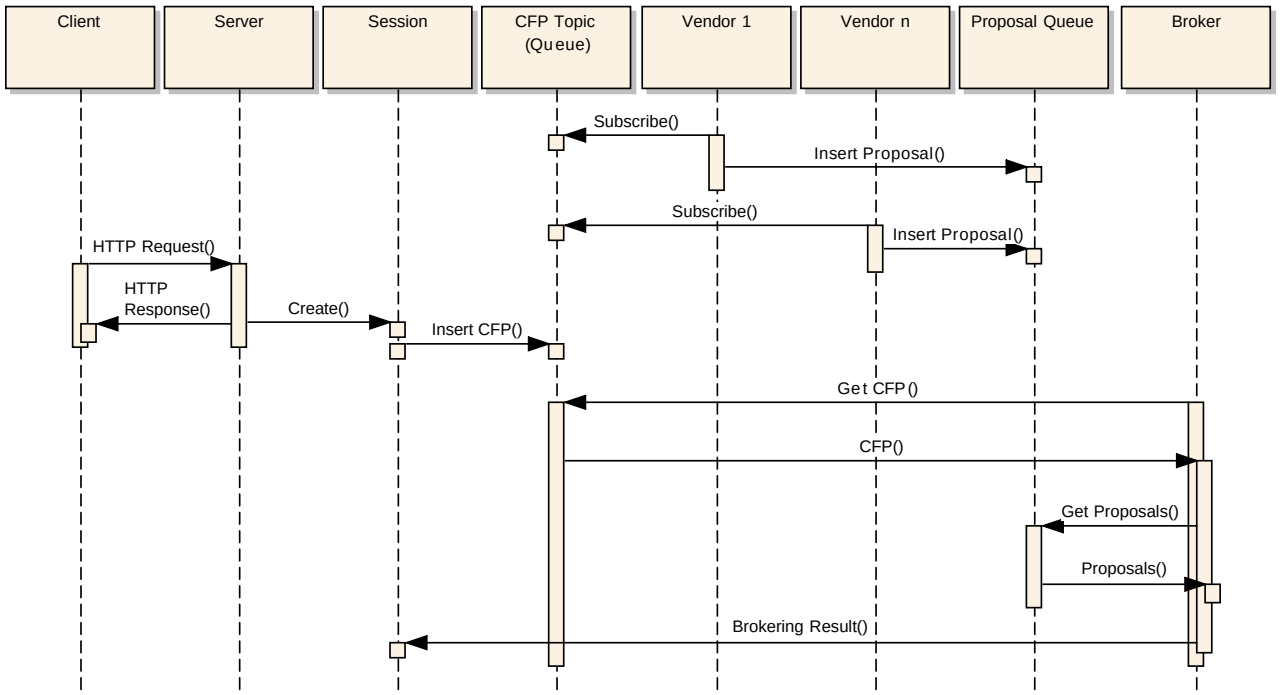


FIG. 6.2. *Diagram*

7.1. Description of Testing Environment and Test Cases. A Linux physical machine hosts the ActiveMQ 5.6 service, with a Topic, named CFP_QUEUE, that receives CFPs from concurrent clients, which run on a different physical machine in the same 1GB Ethernet local network. The server is 64bit Intel Core^{TM}2 Quad Processor Q9300 (6M Cache, 2.50 GHz, 1333 MHz FSB) with 4GB RAM. Oracle Java7 is the runtime environment. Concurrent clients send 10 CFPs, each one, according to a Poisson process with different mean time of arrivals. Vendors run on the server and wait for incoming CFPs. All vendors get the same CFP and generate their proposal, which is sent to the PROPOSALS_QUEUE. A Poisson distribution has been used to generate a synthetic workload. The average queue time has been measured by configuring the native trace log facility of Active MQ and analyzing the output. The measurements run on at least three machines. The database is only used at the start-up to load the current offers by the vendors and to save request and response (CFPs and SLAs). All the brokering computation does not use the RDBMS Mysql databases but information is stored in memory and in the queues.

7.2. Scenarios and Test Data. In a first scenario all brokers run on the server itself. They receive a different proposal from the QUEUE and evaluate the compliance with the correspondent CFP. The result is sent to an SLA_QUEUE from which only the best ones are notified to the clients. The performance of such configuration have been evaluated changing the number of clients, the number of vendors and the number of brokers. All the measures are taken at server side. In the following the most significant figures are explained. Figure 7.1 presents the number of evaluated proposals per millisecond for each experiment.

The first series of Figure 7.1 shows the case of 2 clients that send CFPs with a mean time of arrivals of 2 seconds to the server that hosts 4 vendors and only one broker. It is possible to see that to evaluate 80 proposals the server takes about 24 seconds in this case.

7.3. Results. It is possible to conclude that the application scales well in all the cases where the workload does not exceed the server capability.

The second series represents the same scenario, but with an mean time of arrivals that is 1 second. In this case it is straightforward to observe that only one broker is able to process all the requests in about 13 seconds.

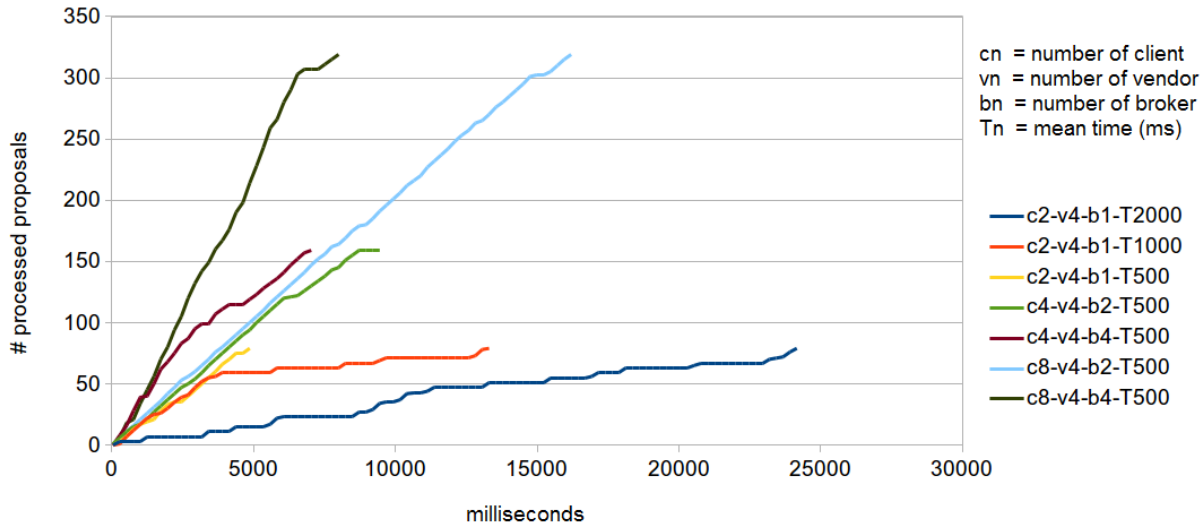


FIG. 7.1. *Proposals processing*

The slope of the line shows that the proposals are processed faster as they arrive with greater frequency and the workload is below the capability of the server. The same happens in about 5 seconds when the mean time of arrivals is 0.5 seconds (third series).

In the fourth series the number of clients and the number of brokers have been doubled. So there are 160 CFPs, which arrive with a doubled rate, but the throughput of the server does not change as the number of broker threads have been doubled. Just in the end it is possible to observe a slightly shift of the line because the clients stopped and the queue are going to be empty.

In the fifth series the number of brokers have been doubled resulting in a faster throughput at the beginning, if compared with the previous case, but after that the slope of the line follows again the trend of the previous case.

Finally, in the case of 8 clients with 2 and 4 brokers (sixth and seventh series) it is possible to observe that the slope of the line depends on the number of brokers again and it is constant till when the queue has proposals ready to be dispatched or the overhead makes the machine slow.

A loss of performance is observed either when the number brokers is greater than 4, as they exceed the number of cores of our server, and when the number of clients double, as the workload overcome the capability of our machine. Figure 7.2 shows the maximum throughput in the case of 8 clients and 4 brokers.

The effect of the overhead, due both to the need of handling the high rate of incoming CFPs and to the schedule of a number of broker greater than the number of cores can be observed. In particular 2 clients never overload the system. The workload increases both because the high number of concurrent clients and because of the scheduling overhead. For this reasons there is a random delay for both the traced events starting from when the server reaches the overload condition.

A noticeable effect of this behavior is the average waiting time of a proposal into the queue, that affects directly the response time of the system and service level perceived by the client.

Moreover it is possible to notice that, with the same mean time of arrivals, configuration with 8 clients and 4 brokers performs better than configuration with 8 clients and 8 brokers, and similar combinations. It happens because CPU exceeds the number of cores because of the overhead due to the process scheduling and to the context switch. In fact the Intel Q9300 quad processor has a single quad core CPU that does not support hyper threading. When the number of threads becomes 8 two of them share the same cache and the same core, affecting the performance.

In Figure 7.3 the occurrence of the overload condition can be observed, with 8 clients and 4 vendors varying

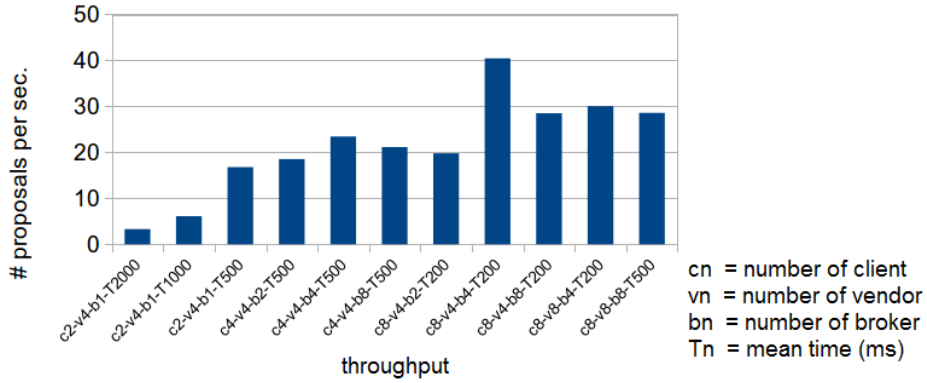
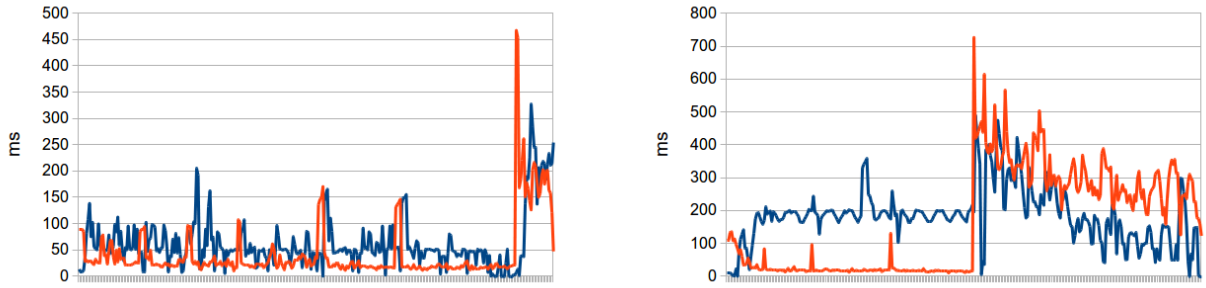


FIG. 7.2. *Broker throughput*



(a) c8-v4-b4-T500

(b) c8-v4-b8-T500

FIG. 7.3. *Mean time of arrivals vs service time of proposals*

the number of brokers. The x-axis shows the different proposals in order of arrival. The blue series shows the elapsed time between the arrival of a proposals into the queue and its withdrawal by an idle broker. The red series shows the time between receiving of a proposal by the broker and the next arrival into the SLA_QUEUE. In Figure 7.3 (a) 4 brokers pull proposals from the queue, keeping the processing time comparable with the enqueued time. At the end of the time series the rate of incoming proposals, which is slightly higher that the brokers' throughput, is going to introduce an overhead that affects the performance. However the overhead rapidly decreases because brokers have processed almost all proposals when clients finished to send their requests. In Figure 7.3 (b) the number of brokers exceeds the number of supported threads. Their scheduling, over a limited amount of resources, causes a waiting time of proposals greater than their processing time. The processing time is not affected till when the high rate of incoming proposals overloads the system. Experimental results show that the time elapsed from the beginning of the experiments and the overload condition is almost the same, as it depends on arrival rate of incoming requests. However in the first case almost all the requests have been processed, but in the second experiment many requests still need to be consumed and their processing time is much affected by the overload condition. These results demonstrate that when the rate of incoming requests grows it is necessary to look for a distributed solution that scales well when new computing resources are added.

Table 7.1 shows the average enqueued time of a proposals in the case of the server. It is much more than the mean time needed by a broker to process the proposal that is about 55 ms. It is possible to observe that before the performance degradation the speedup is 2.38 with two brokers and 2.52 with 3 brokers.

In order to improve this parameter, and to address the problems related to the overload caused by a too high arrival rate of requests, the possibility to offload part of the workload using a Cloud Infrastructure have

TABLE 7.1
Average enqueued time on server

configuration	8c-8v-1b	8c-8v-2b	8c-8v-4b	8c-8v-8b
average time	428ms	180ms	170ms	171ms

been investigated. An OpenStack installation in the same local network have been used. This private Cloud provided a Linux virtual machines. The Linux OS sees just one processor with a 64bit virtual Intel Core 2 Duo P9xxx (Penryn Class Core 2) 2.5GHz with 2K L1 cache and 2GB RAM. In order to estimate the processing capability of such computing resource tests have been performed on the scenario defined above with no brokers running on the server and some brokers executing only on the virtual machine. The estimated processing time for a proposal evaluation is equals to 100ms when only one broker is running. Table 7.2 shows that the average enqueued time for a proposals improves when two brokers are used and goes worse when 4 threads run concurrently. Also in this case the reason of the performance degradation is a number of thread greater than the number of cores, as the virtual processor does not support hyper threading. The speedup is 1.55 in the case of two brokers.

TABLE 7.2
Average enqueued time on VM

configuration	8c-8v-1b	8c-8v-2b	8c-8v-4b
average time	1430ms	920ms	11296ms

Cloud support have been tested to evaluate the improvement of the best working configuration presented before, that has 8 clients, 4 vendors and 4 brokers on the server. Table 7.3 shows that it is possible to improve the performances when 2 broker threads run on the VM. About the distribution of workload between machines, it has been delegated to the ActiveMQ service that distributes the messages to the consumers according to their capability to consume. It has not been studied further. Using heterogeneous machines it is not relevant the speedup as scalability measure. Moreover the benefits of using only one less powerful machine provide a not relevant improvement. The availability of many nodes, even if with reduced computational resource could be useful.

TABLE 7.3
Average enqueued time on hybrid computing infrastructure

configuration	8c-8v-4b	8c-8v-4b-1b	8c-8v-4b-2b	8c-8v-4b-4b
average time	170ms	165.578ms	160ms	167ms

8. Conclusion. Cloud Computing is an emerging trend that will become even more widespread especially in companies and enterprises but some changes are needed to allow users to take advantage of these innovations. When moving to the Cloud it is important that the requirements for the move are understood and that the Cloud services are selected to meet these needs.

8.1. Background. The paper presents a scalable and distributed solution for the SLA-based brokering of Cloud resources from different Cloud vendors and for Multi-Cloud brokering. It also describes the system architecture and implementation details of a prototype of this distributed broker that overcomes the problems of the centralized broker so allowing a Multi-User and a Multi-Cloud utilization.

8.2. Main Findings and Future Works. The great advantages of the proposed approach is the scalability, obtained dividing the brokering problem into simpler tasks, which are distributed among independent agents, whose population dynamically scale together with the computing infrastructure, to support unforeseeable workloads produced by the interactions with large groups of users. Besides it allows the concurrency, is open and available to enable easy extensions of existing components and to add new components. A study of

the scalability of the system and of the behavior of its components is described. In future work, we plan to perform a more extensive performance study in a wider range of scenarios.

REFERENCES

[1] A. Amato, B. Di Martino, and S. Venticinque, *Agents based multi-criteria decision-aid*, Journal of Ambient Intelligence and Humanized Computing, 5 (2014), pp. 747–758.

[2] ———, *Multi-objective genetic algorithm for multi-cloud brokering*, in Euro-Par 2013: Parallel Processing Workshops, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. Scott, and J. Weidendorfer, eds., vol. 8374 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 55–64.

[3] ———, *Evaluation and brokering of service level agreements for negotiation of cloud infrastructures*, in ICITST, 2012, pp. 144–149.

[4] A. Amato and S. Venticinque, *Multi-objective decision support for brokering of cloud sla*, in The 27th IEEE International Conference on Advanced Information Networking and Applications (AINA-2013), Barcelona, Spain, March 25-28 2013, IEEE Computer Society.

[5] ———, *Modeling, Design and Evaluation of Multi-Objective Cloud Brokering*, In: International Journal of Web and Grid Services (IJWGS), Inderscience, Vol. 11, No. 1, 2015, ISSN: 1741-1114

[6] M. Cantara, *Hype cycle for cloud services brokerage*, 2013.

[7] Jamcracker, *Jamcracker cloud management.*

[8] K. T. Lam, Y. Luo, and C.-L. Wang, *A performance study of clustering web application servers with distributed jvm*, in Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on, Dec 2008, pp. 328–335.

[9] J. Lee, J. Kim, D.-J. Kang, N. Kim, and S. Jung, *Cloud service broker portal: Main entry point for multi-cloud service providers and consumers*, in Advanced Communication Technology (ICACT), 2014 16th International Conference on, Feb 2014, pp. 1108–1112.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber , *Bigtable: A Distributed Storage System for Structured Data*, in OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006.

[11] A. Amato, B. Di Martino, F. Xhafa, and S. Venticinque, *Brokering of Cloud Infrastructures driven by Simulation of Scientific Workloads*, in Proceedings of the XI Conference of the Italian Chapter of AIS, (2014).

[12] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, *Cost optimization of virtual infrastructures in dynamic multi-cloudscenarios*, Concurrency and Computation: Practice and Experience, (2012).

[13] S. K. Nair, S. Porwal, T. Dimitrakos, A. J. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. U. Khan, *Towards secure cloud bursting, brokerage and aggregation*, in Proceedings of the 2010 Eighth IEEE European Conference on Web Services, ECOWS '10, Washington, DC, USA, 2010, IEEE Computer Society, pp. 189–196.

[14] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, I. M. Llorente, *Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers*, in Future Generation Computer Systems, Volume 28, Issue 2, February 2012, pp. 358367.

[15] Optimis, *Optimis-project*, 2013, URL: www.optimis-project.eu/.

[16] S. Ried, *Cloud broker a new business model paradigm*, 2010, URL: https://www.forrester.com/go?docid=57809.

[17] RightScale, *Cloud portfolio management by rightscale*, 2013, URL: http://www.rightscale.com/cloud-portfolio-management/benefits.

[18] Verizon, *Cloudswitch*, 2011, URL: https://home.cloudswitch.com/.

[19] Talkin' Cloud, *2014 Talkin' Cloud 100: Top Cloud Service Providers, Aggregators and Brokers*, 2014, URL: http://talkincloud.com/tc100.

[20] A. Zeichick, *Tomcat, eclipse named the most popular in SDTimes study.*, 2013, URL: http://www.sdtimes.com/content/.

[21] A. Zilka, *Terracotta - jvm clustering, scalability and reliability for java.*, 2013, URL: http://www.terracotta.org.

Edited by: Marian Gusev
Received: Dec 18, 2014
Accepted: Mar 31, 2015

DOI 10.12694/scpe.v16i2.1087
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 153–162. http://www.scpe.org

A SCALABILITY STUDY USING SUPERCOMPUTERS FOR HUGE FINITE ELEMENT VARIABLY SATURATED FLOW SIMULATIONS*

FRED T. TRACY† THOMAS C. OPPE‡ WILLIAM A. WARD§ AND MAUREEN K. CORCORAN¶

Abstract. This paper describes the challenges and scalability results of running a large finite element model of variably saturated flow in a three-dimensional (3-D) levee on a large high performance, parallel computer using a mesh with more than a billion nodes and two billion elements. MPI (Message Passing Interface) was used for the parallelization. The original finite element model consisted of 3,017,367 nodes and 5,836,072 3-D prism elements. The model exhibited three characteristics which made the problem difficult to solve. First, the different soil layers had soil properties that differed by several orders of magnitude. Secondly, there existed a 5 ft × 6 ft × 6 ft region at the toe of the levee where the mesh was refined using 1 in × 1 in × 1 in 3-D prism elements having randomly generated soil properties. Thirdly, variably saturated flow in levees is governed by the highly nonlinear Richards' equation.

A utility program was written to increase the size of the original problem by an arbitrarily large factor by replicating the original mesh in the y direction. A factor of two, for instance, would exactly double the number of elements and double the number of nodes less the interface nodes connecting the two pieces. The original data set was run using 32, 64, 96, and 256 MPI processes (one core per process was used throughout this study) with time to solution taken for each of these process counts. The data set was then magnified by 2 and runs for 64, 128, 192, and 512 processes were made with time to solution again recorded. This procedure was repeated for different numbers of processes and magnification values. The largest data set was generated from a magnification of 350 yielding a mesh of 1,044,246,303 nodes and 2,042,625,200 3-D prism elements. The Cray XE6 and Cray XC30 computers were used in this study. A tabulation of results is presented and analyzed, as well as the significant challenges that occurred in scaling up the problem size. Weak and strong scalability results are also presented in this paper.

Key words: high performance computing, finite element method, variably saturated seepage flow modeling

AMS subject classifications. 35J66, 65Y05, 76S05

1. Introduction.

The maturing of large parallel supercomputers makes it possible to solve problems never before attempted. It is a trivial matter to ask for 100,000 or more cores in a batch submission script, but doing so without experimenting with gradually scaled-up problem sizes often leads to unexpected parallel inefficiencies, waste of resources, or even outright failure. This paper describes the challenges presented and the efforts made to run ever larger finite element groundwater models using up to a billion nodes and two billion 3-D prism elements. Weak and strong scaling were used to measure the success of the effort.

The problem chosen is a finite element simulation of variably saturated flow in porous media [1] where there are significant heterogeneities in the soil and difficult nonlinearities in the governing partial differential equation of Richards' equation [2]. Large systems of linear, simultaneous equations must also be solved hundreds of times using solvers such as the conjugate gradient method [3]. The matrices are often stored in a sparse matrix format leading to matrix-vector operations that involve irregular data patterns and indirect addressing. Clearly, the chosen problem is a difficult real-world problem to tackle.

2. Description of the problem.

The problem consists of steady-state flow through a levee as shown in Fig. 2.1 and idealized in Fig. 2.2 where there are several soil layers with soil properties differing by 5 to 6 orders of magnitude (e.g., from clay with a low hydraulic conductivity to sand with a high hydraulic conductivity). Fig. 2.3 shows a portion of the 3-D mesh of the levee system before a tree with its root system was added at the toe. More details are given in [4]. To model the tree root at the toe of the levee, a 5 ft × 6 ft × 6 ft heterogeneous zone was added (see Fig. 2.4) in which the mesh was refined using 1 in × 1 in × 1 in 3-D prism elements. To simulate heterogeneities, a randomly generated hydraulic conductivity was assigned to each element in this zone. The resulting mesh consisted of 3,017,367 nodes and 5,836,072 3-D prism elements.

*This work was supported in part by a grant of computer time from the Department of Defense High Performance Computing Modernization Program (HPCMP).

†Information Technology Laboratory (ITL), Engineering Research and Development Center (ERDC), Vicksburg, MS, USA.
‡ITL, ERDC, Vicksburg, MS, USA.
§HPCMP, ITL, ERDC, Vicksburg, MS, USA.
¶Geotechnical and Structures Laboratory, ERDC, Vicksburg, MS, USA.

FIG. 2.1. *River side of a levee with trees.*

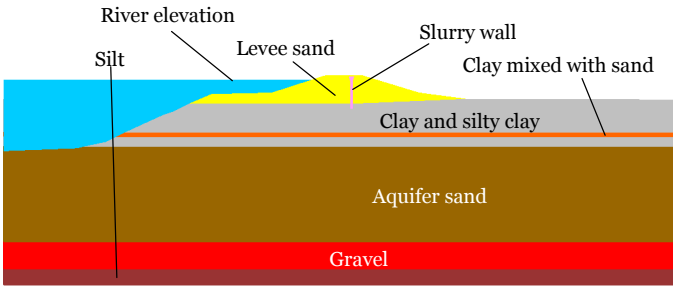


FIG. 2.2. *Cross section of a levee with material types and elevation of the river.*

3. Computational challenges. The many computational challenges inherent in this application were as follows:

- *Complicated geometry.* The geometry of the levee system contained several regions having different soil properties.
- *Wide-ranging material property values.* The soils in the levee ranged from sand to silt to clay, and the associated material properties such as hydraulic conductivity spanned several orders of magnitude [5].
- *Nonlinear system of equations to solve.* The system of equations resulting from the finite element discretization of the levee seepage flow problem was nonlinear because the flow was partially saturated and partially unsaturated. This required the repeated solution of a simultaneous, linear system of equations resulting from either a Picard or Newton linearization in conjunction with line search algorithms [6], [7], and [8]. As the number of nodes and elements increased, this computation became increasingly difficult.
- *Ill-conditioned system of linear equations.* When the soil is partially saturated, material properties such as relative hydraulic conductivity and moisture content become several orders of magnitude smaller as the soil becomes less and less moist. This adds additional stress to solving the linear system of equations at each nonlinear step. A study of preconditioners and solvers for the variably saturated problem is given in [9].
- *Sparse matrix format.* The coefficients of the matrices used in solving the linear system of equations at each nonlinear iteration were stored in sparse matrix format with indirect addressing. This irregular pattern of data decreased the efficiency of the computer for such things as vectorization of the matrix-vector calculations.

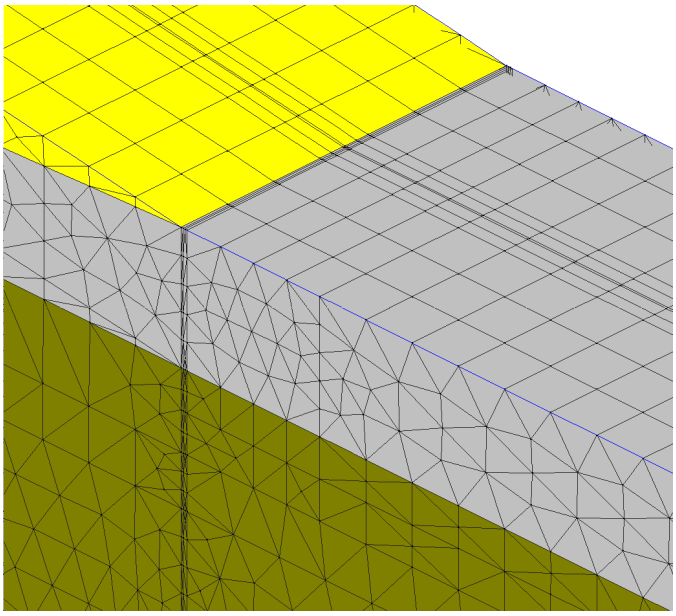


FIG. 2.3. *Portion of the 3-D mesh before the root zone was added.*

4. High performance parallel computing. A 3-D seepage/ground-water finite element program was parallelized using MPI [10] on Garnet, the Cray XE6 at ERDC [11], and on Lightning, the Cray XC30 at the Air Force Research Laboratory, Aberdeen, MD, USA [12]. Garnet consists of 4,716 dual-socket compute nodes with each socket populated with a 2.5 GHz 16-core AMD 6200 Opteron (Interlagos) processor. Each node has 64 GB memory (60 GB user-accessible) or an average of 1.875 GB memory per core. The interconnect type is Cray Gemini in a 3-D torus topology. Garnet is rated at 1.5 peak PFLOPS or 10 GFLOPS per core. Garnet has a large Lustre file system that is tuned for parallel I/O. Lightning consists of 2,360 dual-socket compute nodes with each socket populated with a 2.7 GHz 12-core Intel Xeon E5-2697v2 (Ivy Bridge) processor. Each node has 64 GB memory (63 GB user-accessible) or an average of 2.625 GB memory per core. The interconnect type is Cray Aries in a Dragonfly topology. Lightning is rated at 1.2 peak PFLOPS or 21.6 GFLOPS per core. Lightning has a large Lustre file system that is tuned for parallel I/O.

The parallelization of the 3-D seepage/groundwater program was broken into four separate parts, and each part was a stand-alone FORTRAN or C computer program. One MPI process was placed on each core of a compute node. The four programs will now be described.

- *Partitioner.* The mesh is partitioned using the Parallel Graph Partitioning and Fill-reducing Matrix Ordering program, ParMETIS [13], to divide the mesh into approximately equal pieces among the processes. This is illustrated in Fig. 4.1 where three parallel processes have the nodes of the mesh divided among them. The output of the first version of ParMETIS used was a single file containing a line for each finite element node stating to which process the given node belongs. As the problem size grew, the version of ParMETIS that distributes the output file over the processes in a block partition style was used. As seen in Fig. 4.1, some elements contained nodes belonging to different processes. Each element is assigned to that process that owns the first node of the element. Nodes of an element belonging to a particular process that are not owned by that process are called ghost nodes. Also some elements have nodes belonging to a particular process but that element does not belong to that process. These are called ghost elements, and they create even more ghost nodes.
- *Preparer.* The next piece of the solution creates a file for each process that contains the owned and ghost nodes and elements for that process using local numbers (see Sect. 5.2.2), a subset of the boundary condition file and initial condition file belonging to that process, a list of nodes where data are to be sent to every other process, a list of nodes where data are to be received from every other process, and

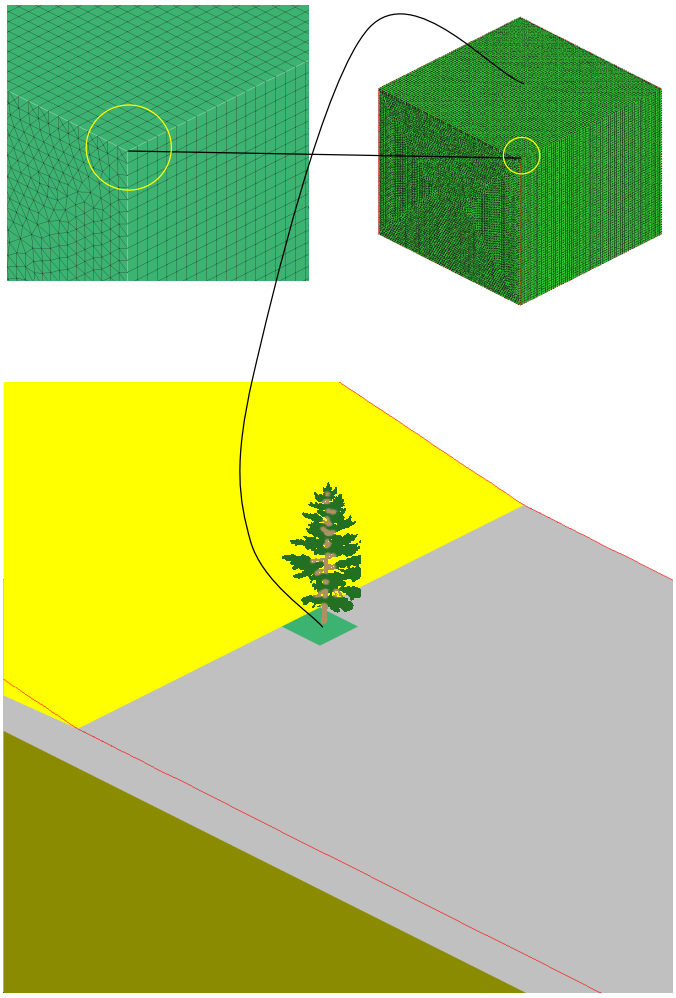


FIG. 2.4. *Heterogeneous zone representing the roots of a tree.*

global node and element numbers for the owned and ghost nodes and elements on that process.

- *Finite element program.* This part does the finite element computations with resulting output files containing results for each owned node of that process. This is the primary focus of the time-to-solution results presented later in this paper.
- *Postprocessor.* This final part is a postprocessor program that combines all data from each process into the final output files.

5. Challenges.

5.1. Partitioner. This original version of this program (written in C) simply reads in the geometry data file and calls one of the ParMETIS programs to produce a single file with a line of data for each node that gives the process that owns the node. That technique was adequate for the original problem with 3,017,367 nodes and smaller versions of the bigger meshes. For example, when the original data set is doubled ($m = 2$), the mesh has 6,000,831 nodes, and the partitioner program took only 6.3 seconds to run. At this size of problem, it was possible to use global arrays to store the mesh. However, as the problem size grew, the geometry files also continued to grow, and the time for reading this file into the program continued to grow as well.

5.1.1. No global arrays. The finite element mesh data needed for the partitioner took 710 Mbytes of memory for the original problem. However, magnifying the problem to $m = 100$, for instance, required 68

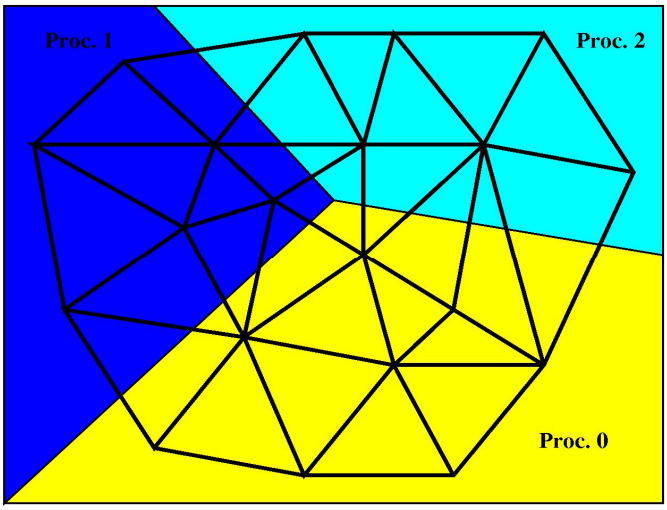


FIG. 4.1. *An example of dividing a finite element mesh into several partitions.*

Gbytes of memory to store this same data. This clearly illustrated why no global arrays can be allowed in any part of this application. To completely avoid this use of global arrays makes the process of parallelization much more difficult. It was decided to first read the mesh data and store it in the block partition format. Then the adjacency table routine [14] had to be modified to use this partitioned data. Finally, a different version of ParMETIS was called and the output file written to all the processes in block partition format.

5.1.2. No ASCII data file containing the entire mesh. The original version of this application had as one of its ASCII input files a file containing the nodes and elements. A node line began with the characters, GN, and an element line began with GE. Further, these type lines could be mixed in any order, requiring the reading and processing of one line at a time. As the problem size grew, the time to process the large geometry data files became prohibitive. For example, at $m = 2$, the entire partitioner took 6.3 sec on Garnet and for $m = 4$, the time was 11.8 sec on Garnet. However, at $m = 100$, the time to read this input file exceeded two hours.

Different solutions were suggested including rewriting the ASCII file as a binary fire. A better solution was to modify the serial program that generated the geometry file in the first place. Instead of writing one huge file, the new version of the serial program wrote node and element data for each process. Further, one file for nodes and one file for elements were written in a simpler format. In this new way, to create the data files for each process for $m = 100$ took 2544.6 sec on Garnet. However, since 3,200 processes were used for the seepage/groundwater program run, this is still less than one second per process. In the case of $m = 100$ in which the grid data could not be read into the partitioner program in two hours using the previous approach, the new approach required only 1.9 seconds on Garnet for the same task.

5.2. Preparation program. When this research project began, it was thought that the parallelization of the actual finite element program would present the greatest challenge. However, because of the restriction of no global arrays, the preparation program was by far the most troublesome and could be improved further from the modifications that were made. Given below is a summary of the challenges faced for the preparation program.

5.2.1. No ASCII data file containing the entire mesh. The same problem existed with the preparation program as with the partitioner in that the time reading the very large geometry file became overwhelming. The same technique of reading the individual node and element files for each process worked well here, too.

5.2.2. Global, local, and ghost nodes and elements. Each node has an original global node number assigned to it, and it is the same everywhere. When the nodes are first and temporarily distributed evenly in

the block partition style to the different processes, each node then has a local node number stating where it is placed on its process. Now ParMETIS dictates that a given node actually belongs to a different process than, in general, it now resides, giving rise to a second local node number. Also, elements are done exactly like the nodes; read in block partition format and then moved to their proper final process. As with the nodes, each element has two different local node numbers. As can be seen, managing all this data is very tedious.

5.2.3. Output files. The finite element program needs various files such that there is one for each process and all data use the final local node and element numbers. These files will now be briefly described.

- *Mesh file.* The list of local (owned plus ghost) nodes and elements are placed in this file. Before writing, the ghost nodes are sorted by increasing process number where the given node resides. For example, suppose for a small problem, process 0 has 10 owned nodes and 2 ghost nodes from process 1, 3 ghost nodes from process 4, and 1 ghost node from process 14. The ghost node numbers are assigned local node numbers 11 and 12 for the process 1 ghost nodes, 13, 14, and 15 for the process 4 ghost nodes, and 16 for the process 14 ghost node. Getting all these data properly numbered and sorted without any arrays containing the entire mesh is remarkably challenging.
- *Initial condition file.* The original initial condition file contains a value of total head for each node to give the finite element program a place to start for iterating to a solution. A value for all the local nodes and elements is now needed for each process.
- *Boundary condition file.* The original file contains parameter data that is simply reproduced for the file for each process. However, boundary conditions are also applied to individual nodes with one example being the total head resulting from the elevation of a river. The original data has for this type of boundary condition the global node number and the total head associated with it. An initial way of handling such a boundary condition was to send the global node number to all the processes and then have each process search through the global node numbers associated with each local node to see if the node is on that process. If the node is on that process, that boundary condition line is written to that process' boundary condition file with the global node number being replaced with its local node number. As the problem size increased, so did the number of such boundary condition nodes to consider. This simple search had to be replaced with a hash formulation [15]. For $m = 100$, the search time using the hash table was 7.5 sec on Garnet.
- *Combination of global node numbers, global element numbers, and ghost node data.* A file for each process containing (1) the global node number for each local node, (2) the global element number for each local element, (3) the number of values and starting memory location for data received from the other processes, and (4) the number and values of the local node numbers where data are to be sent to all the other processes.

5.3. Finite element program. This is the major piece of the seepage/ground-water suite of programs that was tested and timed as it is the primary computational engine. It was initially thought that this would cause the greatest difficulty in running huge problems with large numbers of processes. However, there was only one place where global arrays were being used, so only this algorithm had to be revised. This was the ghost node update portion of the conjugate gradient solver where the data created by the preparer program was stored more efficiently.

6. Results and Analysis.

6.1. Results. Tables 6.1–6.12 give the time to solution for the finite element program on the Cray XE6 and XC30 for different problem sizes (m values) and process counts. The PGI [16] compiler with the compiler option "-fastsse" was used in the computer runs. The original problem was run with 32 MPI processes. The "weak" scaling or speedup in the tables refers to increasing the number of MPI processes the same amount as the problem size is increased. Thus, in Table 6.2, $\frac{T_{m=1,p=32}}{T_{m=2,p=64}}$ means divide the time obtained from running the $m = 1$ problem with 32 MPI processes by the time it took to run the $m = 2$ problem with 64 MPI processes. As another example, when $m = 100$, the number of processes used was 3200 for the weak scaling. The ideal result of the weak scaling ratios is always 1.

"Strong" scaling in the paper refers to keeping the problem size the same while increasing the number of MPI processes and computing the ratio of the original and new running times. For example, $\frac{T_{m=1,p=32}}{T_{m=1,p=256}}$ in Table

TABLE 6.1
Time (sec) for the Cray XE6 and XC30 for $m = 1, nodes = 3017367, and\ elements = 5836072$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2195.9				
32	XC30	902.1				
	XE6	2195.8				
..	XC30	899.6				
	XE6	2199.7				
..	XC30	905.8				
	XE6	Avg: 2197.1				
..	XC30	Avg: 902.7			NA	NA
	XE6	1082.7				
64	XC30	458.0				
	XE6	1083.9				
..	XC30	461.3				
	XE6	1079.1				
..	XC30	458.7				
	XE6	Avg: 1081.9	$= 2.03$			
..	XC30	Avg: 459.3	$\frac{T_{m=1,p=32}}{T_{m=1,p=64}} = 1.97$	2		
	XE6	742.9				
96	XC30	320.5				
	XE6	739.1				
..	XC30	318.9				
	XE6	737.2				
..	XC30	313.7				
	XE6	Avg: 739.7	$= 2.98$			
..	XC30	Avg: 317.7	$\frac{T_{m=1,p=32}}{T_{m=1,p=96}} = 2.84$	3		
	XE6	353.1				
256	XC30	128.9				
	XE6	332.1				
..	XC30	122.8				
	XE6	330.1				
..	XC30	127.9				
	XE6	Avg: 338.4	$= 6.49$			
..	XC30	Avg: 126.5	$\frac{T_{m=1,p=32}}{T_{m=1,p=256}} = 7.14$	8		

6.1 refers to dividing the time it took to run the problem when $m = 1$ using 32 MPI processes by the time it took to run the same problem with 256 MPI processes. The ideal scaling or speedup value is always the ratio of the new number of MPI processes and the original number of MPI processes. In the above example, the ideal speedup is 8.

To test strong scaling, the process count was made 2, 3, and sometimes 8 times that of the original process count for given values of m. Values of $m = 2$, 3, 4, 5, 10, 20, 30, 100, 200, 300, and 350 were evaluated. The largest data set where $m = 350$ contains 1,044,246,303 nodes and 2,042,625,200 elements.

Fig. 6.1 shows the results of the weak scaling. It is important to note that between $m = 1$ and $m = 2$, the number of elements exactly doubles, but the number of nodes less than doubles (slightly) due to interface nodes. Thus the amount of computation per process may be slightly less in the $m = 2$ case for double the number of processes. Thus, weak scaling ratios may be slightly higher than they would be if a perfect doubling of the nodes was also done. Nevertheless, the ideal weak scaling is approximately 1. Figs. 6.2, 6.3, and 6.4 show plots of the strong scaling results. Table 6.13 shows a ratio of the running times for the XE6 and XC30 for different values of m. For multiple runs on the same machine using the same value of m, the elapsed times were averaged.

6.1.1. Analysis. The weak scaling was excellent throughout all the problem sizes and both computers with the XC30 results being better at the larger problem sizes. The strong speedup results were good on the XE6 until $m = 200$ and the process count was increased 3 times such that the process count was 19,200. Also, the XE6 did poorly at $m = 300$ when the process count was doubled. The XC30 consistently outperformed the

TABLE 6.2
Time (sec) for the Cray XE6 and XC30 for $m = 2, nodes = 6000831,$ and $elements = 11672144$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2259.2				
64	XC30	897.9				
	XE6	2196.8				
..	XC30	908.3				
	XE6	2202.5				
..	XC30	904.1				
	XE6	Avg: 2219.5			$= 0.99$	
..	XC30	Avg: 901.8			$\frac{T_{m=1,p=32}}{T_{m=2,p=64}} = 1.00$	1
	XE6	1097.4				
128	XC30	468.4				
	XE6	1093.5				
..	XC30	460.5				
	XE6	1103.1				
..	XC30	460.1				
	XE6	Avg: 1098.0	$= 2.02$			
..	XC30	Avg: 463.0	$\frac{T_{m=2,p=64}}{T_{m=2,p=128}} = 1.95$	2		
	XE6	743.1				
192	XC30	312.9				
	XE6	748.5				
..	XC30	316.4				
	XE6	756.2				
..	XC30	314.7				
	XE6	Avg: 749.3	$= 2.96$			
..	XC30	Avg: 314.7	$\frac{T_{m=2,p=64}}{T_{m=2,p=192}} = 2.85$	3		
	XE6	360.8				
512	XC30	130.4				
	XE6	348.5				
..	XC30	130.2				
	XE6	349.7				
..	XC30	130.5				
	XE6	Avg: 353.0	$= 6.29$			
..	XC30	Avg: 133.7	$\frac{T_{m=2,p=64}}{T_{m=2,p=512}} = 6.72$	8		

XE6 by a factor of approximately 2.5.

7. Consistency check. A system of simultaneous, linear equations is solved each time one of 300 nonlinear iterations is done with the number of unknowns equal to the number of node points. Some parallel programs struggle with getting the same answers when the number of processes is increased. This test is much more difficult in that the size of the problem and the number of processes are increased as m is increased. Table 7.1 gives pressure head results for the first few nodes and the last few nodes of the respective meshes for $m = 1$ and $m = 350$. Because of symmetry, these results should match. These numbers, in fact, match very well.

8. Conclusions. The following conclusions are made from this study:

- It is remarkable that even at over a billion nodes and two billion 3-D elements, the Cray XE6 and Cray XC30 can still be productively used as indicated by the speedup computations when comparing the times to solution of the original problem with data sets up to 350 times larger.
- The weak speedup values were excellent and about the same on both computers.
- The strong speedup capability began to fail for the XE6 at higher values of m. The XC30 values remained good for all problems that were run.
- The XC30 performed approximately 2.5 times better than the XE6.
- It is well known that reading ASCII files can be inefficient. For very large data sets, the problem becomes acute.
- For large problems, the use of global arrays in the source code becomes infeasible.

TABLE 6.3
Time (sec) for the Cray XE6 and XC30 for $m = 3, nodes = 8984295, and\ elements = 17508216$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2215.1				
96	XC30	906.1				
	XE6	2200.4				
..	XC30	898.0				
	XE6	2195.3				
..	XC30	901.8				
	XE6	Avg: 2203.6			$= 1.00$	
..	XC30	Avg: 902.0			$\frac{T_{m=1,p=32}}{T_{m=3,p=96}} = 1.00$	1
	XE6	1097.3				
192	XC30	462.0				
	XE6	1126.4				
..	XC30	462.6				
	XE6	1117.2				
..	XC30	460.6				
	XE6	Avg: 1113.6	$= 1.98$			
..	XC30	Avg: 461.7	$\frac{T_{m=3,p=96}}{T_{m=3,p=192}} = 1.95$	2		
	XE6	751.0				
288	XC30	312.6				
	XE6	747.4				
..	XC30	313.7				
	XE6	750.2				
..	XC30	314.8				
	XE6	Avg: 749.5	$= 2.94$			
..	XC30	Avg: 313.7	$\frac{T_{m=3,p=96}}{T_{m=3,p=288}} = 2.89$	3		
	XE6	313.7				
768	XC30	141.6				
	XE6	313.0				
..	XC30	132.5				
	XE6	310.5				
..	XC30	130.2				
	XE6	Avg: 312.2	$= 7.06$			
..	XC30	Avg: 134.8	$\frac{T_{m=3,p=96}}{T_{m=3,p=768}} = 6.69$	8		

- Preparing the different data files for the finite element program requires very tedious bookkeeping when using the traditional MPI paradigm of accumulating large amounts of data before sending them.
- Large nonlinear implicit algorithms for 3-D finite element variably saturated flow calculations can be performed with excellent scalability when increasing the problem size to ever larger numbers of elements.

REFERENCES

[1] J. ISTOK, *Groundwater modeling by the finite element method*, AGU, 1989.
[2] L. A. RICHARDS, *Capillary conduction of liquids through porous mediums*, J. of Physics, 1 (1931), pp. 318-333.
[3] Y. SAAD, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2003.
[4] M. CORCORAN, J. PETERS, J. DUNBAR, J. LLOPIS, F. TRACY, J. WIBOWO, J. SIMMS, C. KEES, S. MCKAY, J. FISCHENICH, M. FARTHING, M. GLYNN, B. ROBBINS, R. STRANGE, M. SCHULTZ, J. CLARKE, T. BERRY, C. LITTLE, AND L. LEE, *Initial research into the effects of woody vegetation on levees, volume I of IV: project overview, volume II of IV: field data collection, volume III of IV: numerical model simulation, and volume IV of IV: summary of results and conclusions*, U.S. Army Engineer Research and Development Center, Vicksburg, MS, 2011.
[5] A. W. WARRICK, *Soil Water Dynamics*, Oxford University Press, 2003.
[6] S. MEHL, *Use of Picard and Newton iteration for solving nonlinear ground water flow equations*, Ground Water, 44 (2006), pp. 583-594.
[7] F. T. TRACY, *Testing line search techniques for finite element discretizations for unsaturated flow*, Proc. of the 9th Int. Conf. in Computational Science, Baton Rouge, LA, May 25-27, 2009.
[8] C. T. KELLEY, *Solving nonlinear equations with Newton's method*, SIAM, 2003.
[9] H. V. NGUYEN, J. C. CHENG, AND R. S. MAIER, *Study of parallel linear solvers for three-dimensional subsurface flow problems*,

TABLE 6.4
Time (sec) for the Cray XE6 and XC30 for $m = 4, nodes = 11967759, and\ elements = 23344288$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2229.6				
128	XC30	905.1				
	XE6	2224.3				
..	XC30	910.0				
	XE6	2225.2				
..	XC30	914.6				
	XE6	Avg: 2226.4			$= 0.99$	
..	XC30	Avg: 909.9			$\frac{T_{m=1,p=32}}{T_{m=4,p=128}} = 0.99$	1
	XE6	1102.4				
256	XC30	474.6				
	XE6	1102.6				
..	XC30	471.9				
	XE6	1106.0				
..	XC30	467.6				
	XE6	Avg: 1103.7	$= 2.02$			
..	XC30	Avg: 471.4	$\frac{T_{m=4,p=128}}{T_{m=4,p=256}} = 1.93$	2		
	XE6	757.3				
384	XC30	332.9				
	XE6	757.2				
..	XC30	327.4				
	XE6	753.7				
..	XC30	316.1				
	XE6	Avg: 756.1	$= 2.94$			
..	XC30	Avg: 325.5	$\frac{T_{m=4,p=128}}{T_{m=4,p=384}} = 2.80$	3		
	XE6	312.0				
1024	XC30	143.8				
	XE6	311.8				
..	XC30	137.3				
	XE6	326.2				
..	XC30	140.4				
	XE6	Avg: 316.7	$= 7.03$			
..	XC30	Avg: 140.5	$\frac{T_{m=4,p=128}}{T_{m=4,p=1024}} = 6.48$	8		

Proc. of the 9th Int. Conf. in Computational Science, Baton Rouge, LA, May 25-27, 2009.
[10] MESSAGE PASSING INTERFACE FORUM, *MPI: A Message-Passing Interface Standard, Version 3.0*, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.
[11] ERDC DSRC, http://www.erdc.hpc.mil/hardware/index.html, Department of Defense Supercomputing Resource Center, Vicksburg, MS, 2014.
[12] AFRL DSRC, http://www.afrl.hpc.mil/index.html, Department of Defense Supercomputing Resource Center, Aberdeen, MD, USA 2014.
[13] G. KARYPIS, *ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering*, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview, 2014.
[14] D. EPPSTEIN, *Lecture notes: graph algorithms*, http://www.ics.uci.edu/eppstein/161/960201.html, ICS, 161 (1996).
[15] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms, 3rd ed.*, Massachusetts Institute of Technology, (2009), pp. 253-280.
[16] PORTLAND GROUP, INC., http://www.pgroup.com, 2014.

Edited by: Dana Petcu
Received: Dec 1, 2014
Accepted: Mar 19, 2015

TABLE 6.5
Time (sec) for the Cray XE6 and XC30 for $m = 5$, nodes $= 14951223$, and elements $= 29180360$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2230.5				
160	XC30	910.6				
	XE6	2222.8				
..	XC30	911.34				
	XE6	2222.8				
..	XC30	911.2				
	XE6	Avg: 2225.4			$= 0.99$	
..	XC30	Avg: 911.0			$\frac{T_{m=1,p=32}}{T_{m=5,p=160}} = 1.00$	1
	XE6	1105.6				
320	XC30	463.7				
	XE6	1105.4				
..	XC30	464.6				
	XE6	1105.3				
..	XC30	463.7				
	XE6	Avg: 1105.4	$= 2.01$			
..	XC30	Avg: 464.0	$\frac{T_{m=5,p=160}}{T_{m=5,p=320}} = 1.96$	2		
	XE6	762.5				
480	XC30	314.9				
	XE6	770.9				
..	XC30	317.4				
	XE6	761.8				
..	XC30	317.4				
	XE6	Avg: 765.1	$= 2.94$			
..	XC30	Avg: 316.6	$\frac{T_{m=5,p=160}}{T_{m=5,p=480}} = 2.88$	3		
	XE6	333.2				
1280	XC30	130.4				
	XE6	335.6				
..	XC30	133.5				
	XE6	354.5				
..	XC30	132.9				
	XE6	Avg: 341.1	$= 6.52$			
..	XC30	Avg: 132.3	$\frac{T_{m=5,p=160}}{T_{m=5,p=1280}} = 6.89$	8		

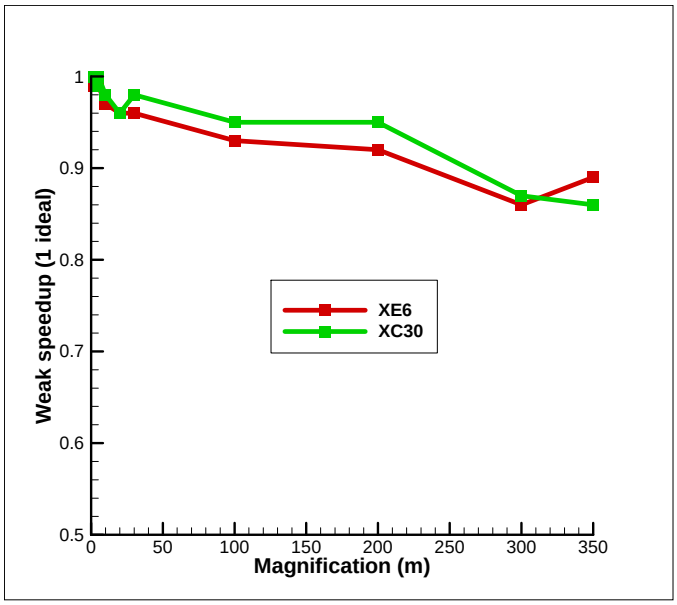


FIG. 6.1. *Weak speedup for different m values.*

TABLE 6.6
Time (sec) for the Cray XE6 and XC30 for $m = 10$, *nodes* $= 29868543$, *and elements* $= 58360720$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
320	XE6 XC30	2260.8 916.4				
..	XE6 XC30	2264.3 922.1				
..	XE6 XC30	2268.6 914.8				
..	XE6 XC30	Avg: 2264.6 Avg: 917.8			$= 0.97$ $\frac{T_{m=1,p=32}}{T_{m=10,p=320}} = 0.98$	1
640	XE6 XC30	1138.8 468.9				
..	XE6 XC30	1143.4 467.2				
..	XE6 XC30	1137.2 468.7				
..	XE6 XC30	Avg: 1139.8 Avg: 468.3	$= 1.99$ $\frac{T_{m=10,p=320}}{T_{m=10,p=640}} = 1.96$	2		
960	XE6 XC30	787.0 331.1				
..	XE6 XC30	774.6 335.9				
..	XE6 XC30	801.0 347.1				
..	XE6 XC30	Avg: 787.5 Avg: 338.0	$= 2.88$ $\frac{T_{m=10,p=320}}{T_{m=10,p=960}} = 2.72$	3		
2560	XE6 XC30	447.3 173.4				
..	XE6 XC30	404.7 156.2				
..	XE6 XC30	350.3 171.6				
..	XE6 XC30	Avg: 400.8 Avg: 167.1	$= 5.65$ $\frac{T_{m=10,p=320}}{T_{m=10,p=2560}} = 5.47$	8		

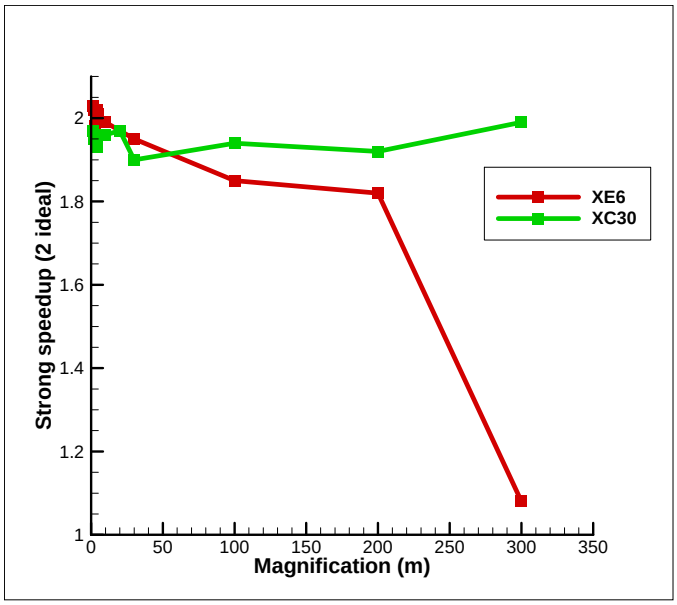


FIG. 6.2. *Strong speedup for twice the original number of processes for different* m *values.*

TABLE 6.7
Time (sec) for the Cray XE6 and XC30 for $m = 20$, *nodes* $= 59703183$, *and elements* $= 116721440$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
640	XE6 XC30	2276.2 918.5				
..	XE6 XC30	2287.2 968.4				
..	XE6 XC30	2277.4 920.0				
..	XE6 XC30	Avg: 2280.3 Avg: 902.7			$= 0.96$ $\frac{T_{m=1,p=32}}{T_{m=20,p=640}} = 0.96$	1
1280	XE6 XC30	1152.4 477.4				
..	XE6 XC30	1171.7 476.5				
..	XE6 XC30	1153.0 472.6				
..	XE6 XC30	Avg: 1159.0 Avg: 475.5	$= 1.97$ $\frac{T_{m=20,p=640}}{T_{m=20,p=1280}} = 1.97$	2		
1920	XE6 XC30	805.5 323.2				
..	XE6 XC30	812.3 329.3				
..	XE6 XC30	805.8 324.1				
..	XE6 XC30	Avg: 807.9 Avg: 325.5	$= 2.82$ $\frac{T_{m=20,p=640}}{T_{m=20,p=1920}} = 2.87$	3		
5120	XE6 XC30	410.8 141.1				
..	XE6 XC30	402.2 140.1				
..	XE6 XC30	408.7 140.9				
..	XE6 XC30	Avg: 407.2 Avg: 140.7	$= 5.59$ $\frac{T_{m=20,p=640}}{T_{m=20,p=5120}} = 6.65$	8		

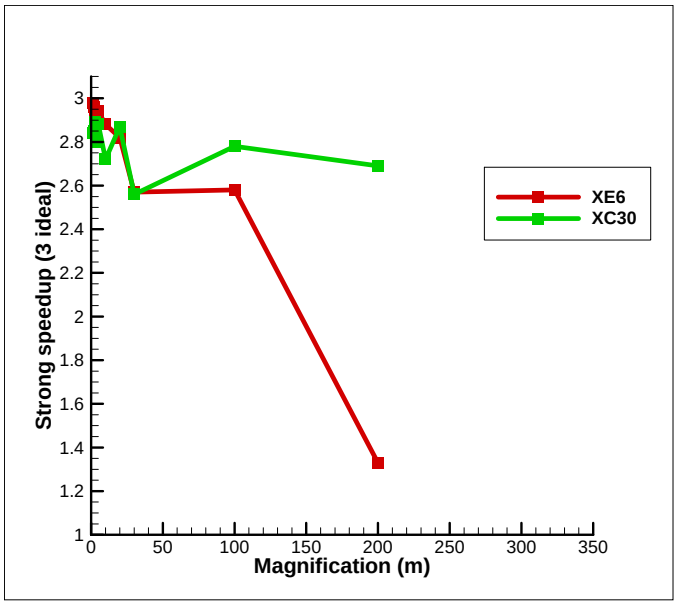


FIG. 6.3. *Strong speedup for three times the original number of processes for different* m *values.*

TABLE 6.8
Time (sec) for the Cray XE6 and XC30 for $m = 30$, *nodes* $= 89537823$, *and elements* $= 175082160$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
960	XE6 XC30	2276.5 924.1				
..	XE6 XC30	2278.5 922.7				
..	XE6 XC30	2277.9 923.7				
..	XE6 XC30	Avg: 2277.6 Avg: 923.5			$= 0.96$ $\frac{T_{m=1,p=32}}{T_{m=30,p=960}} = 0.98$	1
1920	XE6 XC30	1172.5 473.0				
..	XE6 XC30	1165.8 473.4				
..	XE6 XC30	1159.7 514.6				
..	XE6 XC30	Avg: 1166.0 Avg: 487.0	$= 1.95$ $\frac{T_{m=30,p=960}}{T_{m=30,p=1920}} = 1.90$	2		
2880	XE6 XC30	889.7 348.8				
..	XE6 XC30	895.0 364.8				
..	XE6 XC30	872.4 370.0				
..	XE6 XC30	Avg: 885.7 Avg: 361.2	$= 2.57$ $\frac{T_{m=30,p=960}}{T_{m=30,p=2880}} = 2.56$	3		
7680	XE6 XC30	398.7 139.9				
..	XE6 XC30	429.1 139.4				
..	XE6 XC30	447.1 138.2				
..	XE6 XC30	Avg: 425.0 Avg: 139.2	$= 5.36$ $\frac{T_{m=30,p=960}}{T_{m=30,p=7680}} = 6.72$	8		

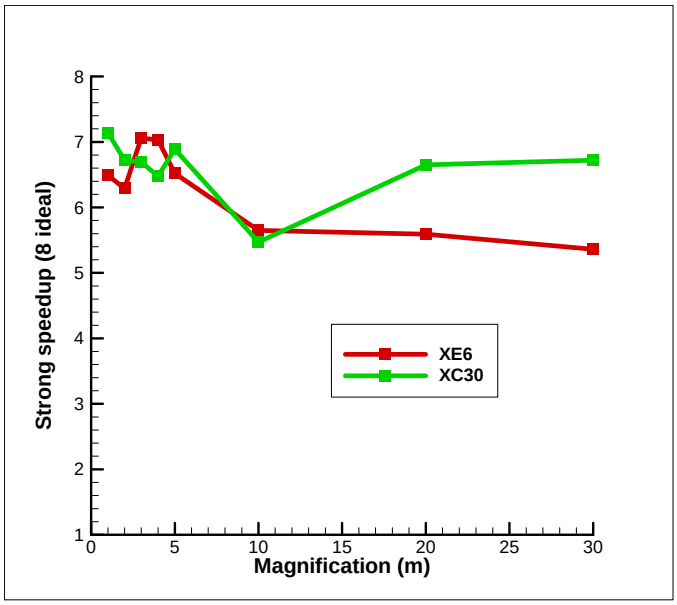


FIG. 6.4. *Strong speedup for eight times the original number of processes for different* m *values.*

TABLE 6.9
Time (sec) for the Cray XE6 and XC30 for $m = 100, nodes = 298380303, and\ elements = 583607200$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2369.4				
3200	XC30	969.6				
	XE6	2358.4				
..	XC30	934.2				
	XE6	2359.9				
..	XC30	961.3				
	XE6	Avg: 2362.6			$= 0.93$	
..	XC30	Avg: 955.0			$\frac{T_{m=1,p=32}}{T_{m=100,p=3200}} = 0.95$	1
	XE6	1228.4				
6400	XC30	479.2				
	XE6	1357.0				
..	XC30	499.5				
	XE6	1250.3				
..	XC30	498.7				
	XE6	Avg: 1278.6	$= 1.85$			
..	XC30	Avg: 492.5	$\frac{T_{m=100,p=3200}}{T_{m=100,p=6400}} = 1.94$	2		
	XE6	899.8				
9600	XC30	339.2				
	XE6	897.4				
..	XC30	352.4				
	XE6	952.1				
..	XC30	354.1				
	XE6	Avg: 916.4	$= 2.58$			
..	XC30	Avg: 348.6	$\frac{T_{m=100,p=3200}}{T_{m=100,p=9600}} = 2.78$	3		

TABLE 6.10
Time (sec) for the Cray XE6 and XC30 for $m = 200, nodes = 596726703, and\ elements = 1167214400$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2394.4				
6400	XC30	947.3				
	XE6	2391.1				
..	XC30	943.6				
	XE6	2392.4				
..	XC30	945.3				
	XE6	Avg: 2392.6			$= 0.92$	
..	XC30	Avg: 945.4			$\frac{T_{m=1,p=32}}{T_{m=200,p=6400}} = 0.95$	1
	XE6	1300.0				
12800	XC30	494.8				
	XE6	1292.8				
..	XC30	493.0				
	XE6	1340.4				
..	XC30	488.3				
	XE6	Avg: 1311.1	$= 1.82$			
..	XC30	Avg: 492.0	$\frac{T_{m=200,p=6400}}{T_{m=200,p=12800}} = 1.92$	2		
	XE6	1783.5				
19200	XC30	351.2				
	XE6	1782.0				
..	XC30	350.3				
	XE6	1839.1				
..	XC30	352.6				
	XE6	Avg: 1801.5	$= 1.33$			
..	XC30	Avg: 351.4	$\frac{T_{m=200,p=6400}}{T_{m=200,p=19200}} = 2.69$	3		

TABLE 6.11
Time (sec) for the Cray XE6 and XC30 for $m = 300, nodes = 895073103, and\ elements = 1750821600$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2671.7				
9600	XC30	1039.2				
	XE6	2442.3				
..	XC30	1041.0				
	XE6	2566.9				
..	XC30	1034.9				
	XE6	Avg: 2560.3			$= 0.86$	
..	XC30	Avg: 1038.4			$\frac{T_{m=1,p=32}}{T_{m=300,p=9600}} = 0.87$	1
	XE6	2351.4				
19200	XC30	512.3				
	XE6	2390.8				
..	XC30	517.1				
	XE6	2392.9				
..	XC30	527.2				
	XE6	Avg: 2378.4	$= 1.08$			
..	XC30	Avg: 522.2	$\frac{T_{m=300,p=9600}}{T_{m=300,p=19200}} = 1.99$	2		

TABLE 6.12
Time (sec) for the Cray XE6 and XC30 for $m = 350, nodes = 1044246303, and\ elements = 2042625200$.

Processes (p)	Cray	Time (T)	Strong Scaling	Ideal	Weak Scaling	Ideal
	XE6	2464.6				
11200	XC30	1041.9				
	XE6	2510.2				
..	XC30	1048.6				
	XE6	2424.8				
..	XC30	1043.1				
	XE6	Avg: 2466.5			$= 0.89$	
..	XC30	Avg: 1044.5			$\frac{T_{m=1,p=32}}{T_{m=350,p=11200}} = 0.86$	1

TABLE 6.13
Ratio of running times for the XE6 and XC30 for values of m.

m	1	2	3	4	5	10
Ratio	2.45	2.46	2.39	2.34	2.45	2.41
m	20	30	100	200	300	350
Ratio	2.58	2.59	2.57	3.44	3.51	2.36

TABLE 7.1
Consistency check comparing values of pressure head from the original mesh and the mesh for $m = 350$ *for the first few nodes and last few nodes of each mesh.*

Node	$m = 1$	Node	$m = 350$
1	129.00000	1	129.00000
2	128.99678	2	128.99678
3	128.99345	3	128.99345
4	119.00000	4	119.00000
5	118.99735	5	118.99735
6	124.23808	6	124.23808
7	118.99464	7	118.99464
8	123.54520	8	123.54520
9	128.98993	9	128.98993
10	110.50000	10	110.50000
6000826	0.00000	1044246298	0.00000
6000827	0.041718481	1044246299	0.041718484
6000828	3.0365778	1044246300	3.0365778
6000829	0.036494874	1044246301	0.036494874
6000830	0.00000	1044246302	0.00000
6000831	0.00000	1044246303	0.00000

DOI 10.12694/scpe.v16i2.1088
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 169–185. http://www.scpe.org

FAULT TOLERANCE SCHEMES FOR GLOBAL LOAD BALANCING IN X10

CLAUDIA FOHRY, MARCO BUNGART, AND JONAS POSNER *

Abstract. Scalability postulates fault tolerance to be efficient. One approach handles permanent node failures at user level. It is supported by Resilient X10, a Partitioned Global Address Space language that throws an exception when a place fails.

We consider task pools, which are a widely used pattern for load balancing of irregular applications, and refer to the variant that is implemented in the Global Load Balancing framework GLB of X10. Here, each worker maintains a private pool and supports cooperative work stealing. Victim selection and termination detection follow the lifeline scheme. Tasks may generate new tasks dynamically, are free of side-effects, and their results are combined by reduction. We consider a single worker per node, and assume that failures are rare and uncorrelated.

The paper introduces two fault tolerance schemes. Both are based on regular backups of the local task pool contents, which are written to the main memory of another worker and updated in the event of stealing. The first scheme mainly relies on synchronous communication. The second scheme deploys asynchronous communication, and significantly improves on the first scheme's efficiency and robustness.

Both schemes have been implemented by extending the GLB source code. Experiments were run with the Unbalanced Tree Search (UTS) and Betweenness Centrality benchmarks. For UTS on 128 nodes, for instance, we observed an overhead of about 81% with the synchronous scheme and about 7% with the asynchronous scheme. The protocol overhead for a place failure was negligible.

Key words: Resilient X10, task pool, GLB, algorithmic resilience, lifeline scheme

AMS subject classifications. 68M15, 68W10, 68Q10

1. Introduction. Large-scale applications are likely to encounter hardware failures during their execution. Consequently, fault tolerance is of crucial importance. Checkpoint/restart is an established approach, but application-specific techniques may induce less overhead.

Resilient X10 [4] provides an interesting platform to experiment with user-level fault tolerance. This extension of the language X10 raises an exception in the event of a permanent place failure. Moreover, Resilient X10 provides an inquiry function to check a particular place's liveness. Failure notification to programs is quite a unique feature in current parallel programming systems [5].

The base language X10 [3] follows the Partitioned Global Address Space (PGAS) model, and provides a shared memory abstraction on top of distributed hardware. Nodes of a compute cluster are modeled as places, which comprise a set of processors and a memory partition. Access to local memory is faster than access to remote memory, and the difference is visible to the programmer.

On the algorithm side, we consider task pools, which are a widely used pattern for load balancing of irregular applications. Moreover, tasks are deployed by several modern parallel programming systems, where they replace or complement threads/ processes as a central construct for specifying parallelism. Examples include Cilk [6], OpenMP [25], Chapel [7], and X10 [3].

The use of tasks is promoted for reasons such as ease of programming and load balancing support. Beyond that, tasks provide the additional benefit of easy migration. Since tasks do not refer to a thread / process number in their code, they may be moved away from a faulty place, without having arranged for that in the application code.

Despite their importance, fault-tolerant task pools have received little attention in previous research. In contrast, much work has been conducted on fault tolerance for the related master/worker and parallel divide-and-conquer patterns. The former was chiefly considered in MapReduce systems such as Hadoop [8]. Unlike task pools, Hadoop considers the set of tasks to be fixed from the beginning, and a central master has an overview of all tasks. In divide-and-conquer algorithms, the overall result is computed along the call tree, which is traversed bottom-up after the computation of leaves. Consequently, children must preserve a link to their parents, and parents can recompute their children by need. Fault-tolerant divide-and-conquer algorithms also

*Research Group Programming Languages / Methodologies, University of Kassel, Kassel, Germany contact: {fohry@ | marco.bungart@ | jonas.posner@student.}uni-kassel.de

handle the case that subtasks are stolen away recursively [9, 10, 11]. The schemes perform well for divide-and-conquer, but would induce too much overhead in our setting. A third group of related work specifically deals with idempotence when recomputing tasks with side effects (e.g. [20]).

Unlike related work, we assume that tasks may generate other tasks dynamically, are free of side effects, and produce results that are combined by reduction. This task model is used by various search and optimization algorithms, as represented by the Unbalanced Tree Search [12] and Betweenness Centrality [13] benchmarks. It also underlies the Global Load Balancing framework GLB, which is part of the X10 standard library.

In this paper, we consider the particular type of task pool that is implemented in GLB. It stores tasks in a collection of private pools, each of which belongs to a single worker. A worker operates on its own pool most of the time. Only when this pool is empty, it asks a coworker for tasks. Victim selection and termination detection follow an efficient state-of-the-art scheme, called the lifeline algorithm [1].

We introduce two different fault tolerance schemes that extend GLB. Both regularly save the local task pool contents of one worker in the main memory of another. When a place fails, its backup partner takes over the lost tasks. In case of unfavourably correlated failures, such as simultaneous loss of a place and its backup partner, the program aborts with an error message. Successful completion, on the other hand, guarantees that the final result is correct, despite possible failures.

In both schemes, particular attention was paid to stealing, to avoid inconsistencies between victim and thief after a place failure. The first scheme, which has been introduced in [2], defines a conservative steal protocol. It mainly adopts synchronous communication and cautiously aborts the program in any potentially critical situation.

The second scheme improves on the first one wrt. efficiency and robustness, where robustness characterizes the number of situations that lead to program abort. The improvements have been achieved by a major redesign. Most importantly, the second scheme relies on asynchronous communication consistently. Thus, a worker may continue processing tasks while waiting for an outstanding communication reply. To keep complexity manageable, an actor-like communication structure has been devised. Moreover, the second scheme exploits the redundancy that is immanent in stealing. As compared to the first scheme, the second scheme furthermore reduces handshaking in the steal protocol, combines multiple steal requests in a transaction, and adopts several more minor changes.

In addition to the two schemes presented in this paper, we are currently working at a third scheme. That scheme may be even more efficient, but at the price of reduced flexibility. In particular, it can only be applied to task pools that are organized as a collection of stacks. The UTS benchmark considered in this paper, for instance, can not be handled with the third scheme. The second and third schemes share the idea of an actor-like communication structure. The third scheme has been outlined in [15], with focus on X10 language support, but its implementation is still ongoing.

The schemes described in this paper have been implemented in X10, by extending the GLB source code. Our own code which can be obtained for free from the second author's homepage.

Apart from the unfavourably correlated failures mentioned above, the two schemes can cope with any number of permanent place failures. To avoid unfavourable correlations, we allow only one worker per cluster node. This assumption partly stems from GLB, which permits one worker per PGAS place, but goes farther by requiring that only one place is mapped to each node.

Experiments have been run with the UTS and BC benchmarks. While we observed significant overheads with the first fault tolerance scheme, the second scheme was about as fast as the original GLB. The overhead for restore was low in all cases.

The paper starts with background on X10, task pools, and the GLB library in Sect. 2. Then, Sect. 3 introduces some basic concepts that are common to both fault tolerance schemes, and gives an overview of their respective approaches. Details of the synchronous scheme are presented in Sect. 4, and details of the asynchronous scheme in Sect. 5, respectively. Sect. 6 describes our experimental setting, reports performance numbers, and discusses results. Finally, Sects. 7 and 8 are devoted to related work and conclusions, respectively.

2. Background.

2.1. X10 and Resilient X10. X10 is a novel parallel language from IBM [3], which supports object orientation and exception handling in a similar way as Java. Following the Asynchronous PGAS (APGAS)

programming model, gives the programmer the view of a shared address space that is divided into disjoint partitions. The notion of *place* captures a memory partition and a set of processors, which have faster access to the local memory partition than to remote data.

The placement of data and computations is controlled by the programmer. Access to remote data requires to move the computation to the remote place. This is accomplished with the `at` keyword. In a place change, part of the data from the original place is transparently copied along. Parallelism is specified orthogonally. Using the `async` keyword, a programmer starts an asynchronous task on the current place. Tasks are called *activities*, and can be moved to a remote place by combining `async` and `at`.

Later in this paper, we will refer to remote computations launched via `at` or `at async` as messages. This notion reflects that, in both cases, a computation request, possibly accompanied by data, is sent to the remote place. There, it is queued until a remote thread is available for its execution. Messages sent with `at` are called *synchronous*, since the activity at the origin place is suspended until the remote activity returns, possibly with a result. Messages sent with `at async` are called *asynchronous*, since the activity at the origin place immediately continues with its sequential flow of control.

In asynchronous communication, the origin activity is not notified on completion. Spawning of activities can, however, be enclosed in a `finish` block. At the end of the block, the parent activity waits until all spawned activities and their descendants have terminated.

For place-internal load balancing, a work-stealing scheduler is built into the X10 runtime system. It assigns the activities to the available threads. This scheduler can not be used for global load balancing, since the assignment of tasks to places is under programmer control. Therefore, it is complemented by GLB, which belongs to the X10 library since version 2.4.2. GLB tasks are more heavy-weight than X10 activities. For their deployment, a GLB user must implement interfaces in a custom class.

Since version 2.4.1 of end-2013, X10 supports resilience in its runtime system. It is switched on by setting some environment variables. We deployed resilience mode 1, which has the limitation that place 0 must not fail, or otherwise the program aborts with an error message.

Resilient X10 provides two mechanisms for failure notification. First, a `DeadPlaceException` (DPE) is raised in the event of a failure. When a place fails before or during synchronous communication, the DPE is delivered to all parent activities instead of the regular reply. In asynchronous communication, the DPE is delivered to parent activities, as well, but usually they catch the exception only at the end of a surrounding `finish` block. Second, X10 provides an inquiry function `isDead(...)`, which may be called by any place to check any other particular place's liveness.

2.2. Task Pools and GLB. The paper refers to the particular type of task pool that is used by GLB. Tasks are assumed to be free of side effects, and may generate other tasks dynamically. The overall result is computed by reduction from individual results of each task.

The GLB task pool comprises a distributed data structure and a set of workers. The data structure is a collection of private pools, each of which belongs to a single worker and is stored at the worker's place. Initially, one or several workers may have tasks in their pool, which is referred to as dynamic or static initialization, respectively.

Workers are realized by activities. Each worker runs a loop, in which it repeatedly takes a task out of the local pool, processes it, and possibly inserts new tasks generated. When the local pool is empty, the worker contacts one or several coworkers, called victims, and asks them for tasks.

Victim selection and termination detection follow the lifeline scheme [1]. According to this scheme, a worker successively contacts up to w random coworkers and z lifeline buddies. The latter in their entirety form an appropriate graph. If a victim has no tasks to share, it rejects the request and, if it is a lifeline buddy, additionally stores it. If a lifeline buddy obtains tasks later, it shares them with the stored worker.

When all $w + z$ steal attempts failed, the worker activity ends. If a lifeline buddy sends tasks later, it restarts the worker by spawning a new activity at the corresponding place. Note that multiple lifeline buddies may have stored steal requests from our worker simultaneously, and may, thus, send tasks at the same time later. While only the first message arriving leads to restart, the worker may consequently receive tasks when its pool is non-empty. Similarly, it may receive tasks from a lifeline buddy while waiting for the answer to an outstanding steal request.

The computation terminates when all workers have ended. Then, the final result is computed by reduction from partial results, which are collected and combined by each worker during task processing. Termination is detected by an outer `finish` block, which surrounds all starts and restarts of worker activities.

GLB assumes that there is only one worker per place. Additionally, it precludes any parallel activities at this place. This is enforced by setting environment variables. The assumption is quite restrictive, but eliminates any need for place-internal synchronization. Thus, a worker may, without distruption, alternatingly process up to n tasks, receive incoming messages, and invoke local functions to answer the received steal requests. Recall that messages correspond to remote activities. In GLB, these activities are queued, but are not allowed to run until the worker calls `Runtime.probe()` to receive the messages. This call releases the worker's thread, and all pending activities are scheduled sequentially in any order. GLB uses the following types of messages:

- `give`: The worker receives tasks from a victim. This message may be the response to an outstanding steal request, or originate from an older request stored by a lifeline buddy. In either case, the corresponding activity integrates the tasks into the worker's pool by calling a `merge` function. If the worker has ended, it is additionally restarted.
- `tryStеal`: The worker is the victim of a steal request. If its pool is empty, the corresponding activity immediately replies with a `noTasks` message. Otherwise, it records the request at the worker's place.
- `noTasks`: This is the rejection of a former steal request.

When all messages have been received, the worker activity resumes and sends out tasks to the thiefs recorded. The first thief gets half of the tasks, the second a quarter, and so on. Any remaining thieves are sent a `noTasks` message. Since the requests are taken from a data structure, the worker is able to prefer random thieves over lifeline thieves.

Steal requests are sent asynchronously with `at async`, i.e., the thief spawns a remote activity and continues. So it remains responsive to requests from others. The victim sends its reply asynchronously, as well. Request and reply are related by a volatile variable, which is located at the thief place. The thief sets this variable when sending out the request, and the victim resets it later.

From a user's perspective, GLB defines a class and two interfaces that must be implemented. In particular, a GLB user must define a data structure and the following access functions for the private pools:

- `merge` and `split` integrate tasks and split the pool, respectively.
- `process(n)` takes n tasks out, processes them, and inserts any newly generated tasks. If less than n tasks were available, the function returns `false`.

Note that GLB does not restrict the data structure, except that the functions must be provided. While the fault tolerance scheme in [15] only allows stacks, the schemes introduced in the present paper do not restrict GLB's flexibility. As a minor restriction, we forbid use of the `yield()` function, which may be invoked by GLB user code to interrupt long-running tasks. A workaround is explained in Sect. 6.

3. General Structure of the Fault Tolerance Schemes. Our schemes handle the following issues:

- writing regular backups and maintaining a ring structure of backup places,
- recognizing node failure,
- keeping backups consistent during stealing, and
- returning to a consistent state after failures.

The first and second issues are handled in a similar way by the two schemes, and are discussed in this section. The third and forth issues are handled differently with quite complex protocols. Sect. 4 provides the details for the synchronous scheme, and Sect. 5 for the asynchronous one.

Scope of Fault Tolerance. Fault tolerance begins with writing the first backup right after task pool initialization. If a failure occurs earlier, the program can be restarted without notable loss of time.

The number of cases that lead to program abort is lower for the asynchronous than for the synchronous scheme, as captured by the term robustness. Both schemes guarantee that the program either outputs a correct result, or crashes.

Any fault tolerance scheme must find a compromise between overhead and robustness. Our schemes strive to hold each relevant data element at exactly two locations at any time. If the two locations fail simultaneously, the program is aborted. While the redundancy level could in principle be changed from two to some other value, a fair balance of redundancy among data elements is desirable to minimize the backup overhead for a

given level of robustness. The asynchronous scheme gets closer to a fair balance than the synchronous scheme, as explained in Sect. 5.1.

Our algorithms suppose a reliable communication layer, i.e., if both sender and receiver of a message are alive, the message must eventually be delivered. A timing guarantee is not needed, though. In particular, different messages from the same sender to the same receiver may overtake. This may happen since, at `Runtime.probe()`, pending activities are scheduled in any order.

Ring Structure. Let P denote the number of places. In both schemes, each worker regularly writes a backup to the main memory of another place. For that, workers are arranged in a ring, and numbered $0 \ldots P-1$. Outside failures, worker i regularly writes its backup to the main memory of worker $(i+1) \bmod P$, which is called its backup partner $\texttt{Back}(i)$. Vice versa, if $\texttt{Back}(i) = j$, we denote the predecessor as $i = \texttt{Forth}(j)$.

In case of failures, the gap is bridged, i.e., the next node alive along the ring takes the role of $\texttt{Back}(i)$. Re-establishing the ring structure is accomplished by a restore protocol. These protocols differ between the synchronous and asynchronous schemes and are explained in Sects. 4 and 5.

Logs. In both fault tolerance schemes, each worker keeps a log of the places that it has taken over due to restore. The relation of taking over is transitive, i.e., if worker $i-2$ restored $i-1$, and later i restored $i-1$, then we say that worker i has taken over both workers $i-2$ and $i-1$.

Logs are inspected when a worker receives a restore request and must decide whether the program can recover. Consider, for instance, the sequence of workers $i-2, i-1, i$. If $i-1$ fails first, and $i-2$ later, then i can perform $i-2$'s restore if and only if it has $i-1$ in its log and $\texttt{Forth}(i) = i-2$.

The synchronous scheme stores logs as a list of all workers restored. The asynchronous scheme, instead, only stores `Forth` as its log, relying on the invariant that each worker i must have taken over all workers $[\texttt{Forth}+1 \ldots i-1]$ (cyclically), or otherwise `Forth` is invalid. There must not be gaps between `Forth` and i, since any missing worker's data would be lost. In both schemes, the log is part of a worker's backup data.

Backup Writing. Regular backups are written every kn processing steps, called a *backup interval*. Here, n denotes the GLB parameter, and $k \geq 1$ is an additional parameter. To determine the length of a backup interval, each worker counts steps independently and, at the end, autonomously sends its backup. The backup comprises the current contents of the local task pool, the current value of the partial result, as well as the worker's log. These data are copied to a `val` variable and transparently sent by X10's `at` construct.

Our algorithm writes backups only right before a `process(n)` call. At these moments, the task pool contents, supplemented by the current value of the partial result, defines a worker's state in full.

Shadowing Results of partially finished Tasks. Sometimes, the execution of a single task takes too long to keep a worker responsive. Therefore, GLB defines a `yield()` method. A user can invoke this method inside `process(n)` to interrupt task execution and call `Runtime.probe()`. Our fault tolerance schemes do not support `yield()` for two reasons: First, `yield()` would complicate the program structure, especially for the asynchronous scheme, by introducing an additional `Runtime.probe()` call. Second, at a `yield()` call, a worker's state is not fully represented by the task pool contents.

One of our benchmarks (Betweenness Centrality) deploys long-running tasks, so we still had to find a workaround. Within the execution of a single task, BC performs a breadth-first search on a graph to find all shortest paths from one specific node to all others. Since the number of nodes is typically large, `yield()` should be called in-between. An alternative is provided by one of the BC sample codes that are supplied with GLB. This code does not use `yield()`, but instead decomposes each `process(n)` into several `step` calls.

After each `step` call, the worker may suspend to answer steal requests. To continue thereafter, it needs to save its internal state. This state is larger than the backup state, we denote it as *shadow*. The shadow is only visible to the particular worker, not to others. During each `process(n)`, the worker operates on the shadow, and at the end of `process(n)`, the shadow's result is combined into the partial result. When a backup is written during `process(n)`, it covers the state at the beginning of `process(n)`, but not the shadow. Maintenance of shadows induces additional overhead for copying after each `step`.

Communication Structure. As noted before, the major difference between our two fault tolerance schemes concerns communication. While the synchronous scheme uses X10's `at` construct in most cases, the

asynchronous scheme uses `at async` everywhere.

Asynchronous communication improves efficiency, since a worker may continue processing tasks while communicating with others. Moreover, it remains responsive this way.

On the backside, asynchrony complicates the program structure, since a worker may have to manage multiple outstanding requests at the same time. To reduce complexity, we designed an actor-like communication structure, in which a worker alternatingly processes tasks, receives messages, and carries out the actions required by these messages. The structure is inspired by GLB, but goes farther by consistently applying this three-phase structure to all types of communication. This allows us to process messages in a well-defined order, and to prioritize messages. The scheme resembles the actor model [16] insofar as a worker, except for processing tasks, is a passive entity, and only becomes active upon message receipt.

Failure Notification. As noted in Sect. 2.1, X10 supports failure notification by DPEs and by the `isDead(...)` function. DPEs are well-suited for the synchronous scheme. Throughout the corresponding program, for timely failure notification, all place changes are enclosed in `try-catch` blocks. When a place fails during a regular backup, its predecessor is informed immediately and can initiate the restore protocol (see Sect. 4.2). When a place fails during stealing, the DPE is received by a different worker, which broadcasts it to all other workers. The broadcast is an expensive, but timely way to reach all workers that are currently involved in a communication with the failed worker and whose identities may not be known to the sender.

Unfortunately, DPEs are not suited for the asynchronous scheme, since they are only caught by the outer `finish`, which comes too late. Therefore, we do not use DPEs in the asynchronous scheme, but instead rely on `isDead(...)`. This X10 issue is further discussed in [15].

We implemented a monitoring scheme, in which each place regularly inquires its backup place's liveness by calling `isDead(...)`. Monitoring is disturbed if a worker has ended and thus can not take the initiative for the regular calls. Here, monitoring deploys so-called *ghost activities*, which are temporal re-activations of an ended worker.

To state it in more detail, a worker regularly calls `isDead(...)`, as noted. If the successor is dead, the restore protocol is invoked. If it is alive, the worker sends a `monitor` message. The corresponding remote activity checks whether the successor has ended and, if so, starts a ghost activity. The ghost activity is responsible for 1) invoking `isDead(...)` on the successor's successor, and 2) calling `monitor` recursively, if needed. It can be easily verified, that any failure is recognized this way.

The scheme is speeded up by a second failure notification mechanism, called *timeouts*. Timeouts are deployed in the asynchronous protocols: When a worker has sent a message, it typically expects a reply, even though it does not explicitly wait for it in the actor scheme. These replies are stored in a list, together with some time limit. Occasionally, the worker runs through the list. For all replies that have exceeded their time limit, it invokes a synchronous communication to the respective communication partner. If it is dead, it takes appropriate action according to the respective protocol. In addition, it informs the place's `Forth` who will initiate the restore protocol.

4. Synchronous Fault Tolerance Scheme.

While the basic approach of our fault tolerance schemes has been discussed in Sect. 3, steal and restore are more complex. Since the approaches in the synchronous and asynchronous schemes are fundamentally different, we handle the synchronous approach in this section, and the asynchronous approach in Sect. 5.

4.1. Steal Protocol.

Stealing requires special arrangements to avoid inconsistencies between thief place `F`, victim place `V`, as well as their backups at `Back(F)` and `Back(V)`. Otherwise tasks may be computed twice or not at all. The protocol is conservative in that the program is cautiously killed in any possibly inconsistent situation, to guarantee that a computed result is always correct.

Figure 4.1 depicts the steal protocol, which was introduced in [2], with time advancing from top to bottom. In the figure, a wavy line indicates that the place is processing tasks (if available), and a solid line marks actions that are part of the protocol. At the backup places, wavy lines are omitted for clarity before and after their involvement. A thick dotted line represents the queueing of a request, i.e., the request has to wait until the worker calls `Runtime.probe()`. An asterisk denotes an asynchronous message sent via `at async`, whereas the other arrows correspond to synchronous communication via `at`. To state it in more detail, `V` invokes three `at`s:

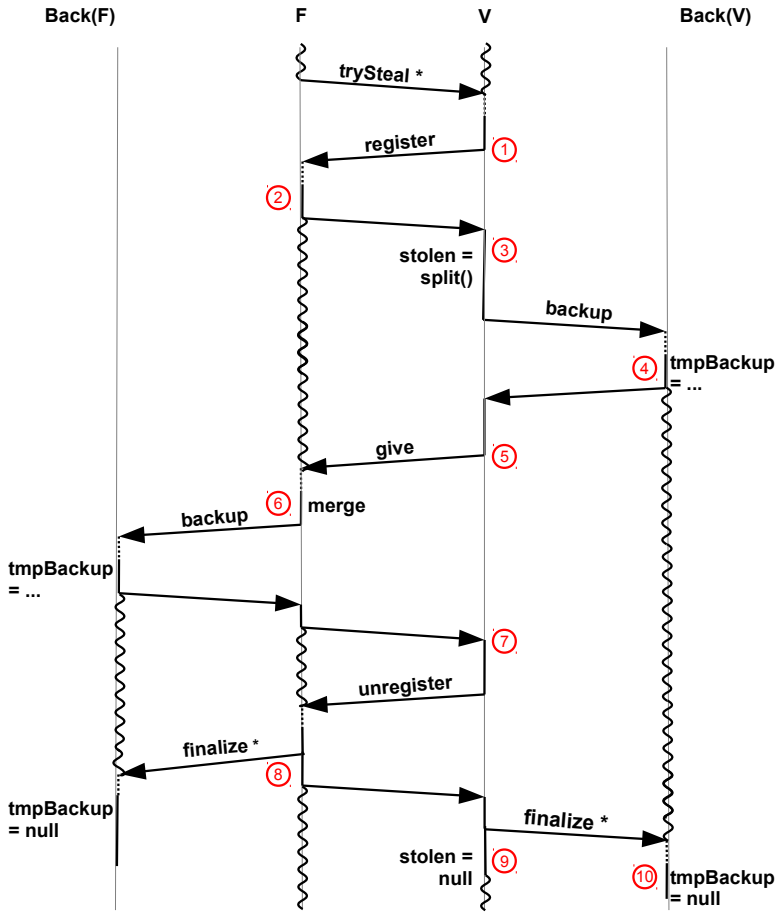


FIG. 4.1. *Steal Protocol of the Synchronous Scheme*

from ① to ③, from ⑤ to ⑦, and from about ⑦ to about ⑨. During the second `at`'s operation at `F`, another `at` to `Back(F)` is launched.

The protocol starts with a `trySteal` request. If `V` has no work, it responds as explained in Sect. 2.2, and no further action is taken. Otherwise, `V` registers at place `F`. From ② to ⑧, `F` is not allowed to accept other register requests.

After registration, `V` splits the pool by calling the user-provided `split` function. Then, it saves the tasks to be stolen and writes a backup without them. While `tmpBackup` $\neq$ `null`, there may be an inconsistency, since `Back(V)` does not know whether `F` and `Back(F)` have already taken over the tasks.

After the backup, `V` sends the tasks to `F`, called `give` in the figure. On arrival, `F` merges the tasks into its own pool. The pool is not necessarily empty, since section ② to ⑧ of the protocol may be repeated if different lifeline buddies send work at about the same time.

Analogously to `V`, thereafter `F` writes a backup and starts processing the tasks just received. The remaining arrows display some handshaking to let all places know when the protocol has finished. Then, variables `tmpBackup` and `stolen` may be reset, and `F` and `V` may accept other requests.

Looking back, there are two situations in which backups are written: regularly every kn steps, and during stealing. To reduce overheads, we restart the kn period after each steal-related backup. Moreover, a place first checks for steal requests before writing a regular backup.

The protocol in Fig. 4.1 considers the base case of a single steal request. Other cases are reduced to this

one as follows:

- If V receives other steal requests between ① and ⑨, they are rejected or delayed until ⑨.
- If V receives a `register` from F between ① and ③, a place number-based ordering guarantees that only one of the crossing `register`s between F and V passes. The other is delayed until ⑨. Crossing `register`s may occur when F is V's lifeline partner and wants to answer a steal request from V that it has recorded long ago.
- If V receives other register requests between ① and ⑨, they are delayed until ⑨
- If F receives `trySteal`s or `register`s between ② and ⑧, they are rejected or delayed until ⑧.
- If F receives other `register`s before ②, section ② to ⑧ is run for the first request arriving, and the others are delayed until ⑧. Hence, for "our" request, the protocol corresponds to the base case, except for a larger delay between ① and ③ in this rare case.
- At F and V, backup requests from `Forth(F)` and `Forth(V)` are processed as usual, concurrently to the steal protocol.
- During the protocol, regular backups of F and V are skipped and replaced by the steal backups.
- If places F and `Back(V)`, or places V and `Back(F)` coincide, the respective roles are pursued concurrently.

4.2. Restore Protocol. The restore protocol is depicted in Fig. 4.2. After `Forth(P)` has been notified of the failure, as explained in Sect. 3, it locates `Back(P)` as being the next place alive in the ring. Then, `Forth(P)` sends a `restore(P)` message to `Back(P)`. Upon receipt, `Back(P)` inspects its log, as further explained below. If recovery is possible, `Back(P)` merges the tasks from the backup into its own local pool, combines the partial results, and inserts P into its log. Thereafter, it sends a backup of its new state to `Back(Back(P))`. This backup is called a taken-over backup, or shortly `TO-backup`.

After the `TO-backup`, `Back(P)` reports completion to `Forth(P)`, which is denoted by `RSTack` in the figure. Thereupon, `Forth(P)` sends a backup to `Back(P)`. It is called an inauguration backup, or shortly `IA-backup`, since `Back(P)` is `Forth(P)`'s new backup place. Note that the protocol uses synchronous communication, therefore the arrows labelled `RSTack` and `IAack` correspond to implicit returns from `at` calls.

To explain the usage of the log, consider the case that `Back(P)` fails at (R3). Although `Forth(P)` recognizes `Back(P)`'s failure, it has no clue when the failure occured. Therefore, `Forth(P)` looks up `Back(Back(P))` and requests P(!)'s backup, as it would do if `Back(P)` died long ago. In our setting, `Back(Back(P))` has received `Back(P)`'s backup, including P's data, as indicated by the log. Therefore, `Back(Back(P))` transforms the request to restore P into a request to restore `Back(P)`, and, on completion, reports success to `Forth(P)`. Because of the ring structure, the scheme extends to any number of failed places.

If `Back(P)` fails before (R2), P is not contained in the log, and therefore `Back(Back(P))` aborts the program. Between (C1) and (C2), `Forth(P)`'s data are unsecured. Thus, if `Forth(P)` dies during this time, `Forth(Forth(P))` will neither find `Forth(P)`'s backup place, nor a place that has `Forth(P)` in its log. This is recognized by the first place alive, which aborts the program.

4.3. Recovery and Correctness. The following items show that, during the steal protocol, a failure of F and/or V can either be recovered successfully, or halts the program:

- Failure of F before sending the ② → ③ message: Because of the register call, V is notified of the failure. Since the stealing has not yet begun, recovery corresponds to the base case.
- Failure of F after ② but before sending the message into ⑦: V re-merges the stolen tasks into its queue and directs `Back(V)` to reset `tmpBackup`. Depending on the state of F's backup, `Back(F)` either restores the old backup of F, or kills the program. When the program goes on, all places consistently see the tasks in question at V.
- Failure of F right before or during V's `unregister` call: V continues with the protocol, finalizing its backup at `Back(V)`. If `Back(F)` has a temporary backup from F, it kills the program. Otherwise it restores the new backup, which includes the stolen tasks.
- Failure of V: Depending on time, `Back(V)` restores the old state, restores the new state, or kills the program.

Killing the program when `tmpBackup` $\neq$ `null` can be avoided by some additional handshaking between `Back(V)` and F, or between `Back(F)` and V, in the event of a failure. This is done in our asynchronous fault tolerance scheme, which is described in Sect. 5. In the synchronous scheme, we stayed with the simpler protocol.

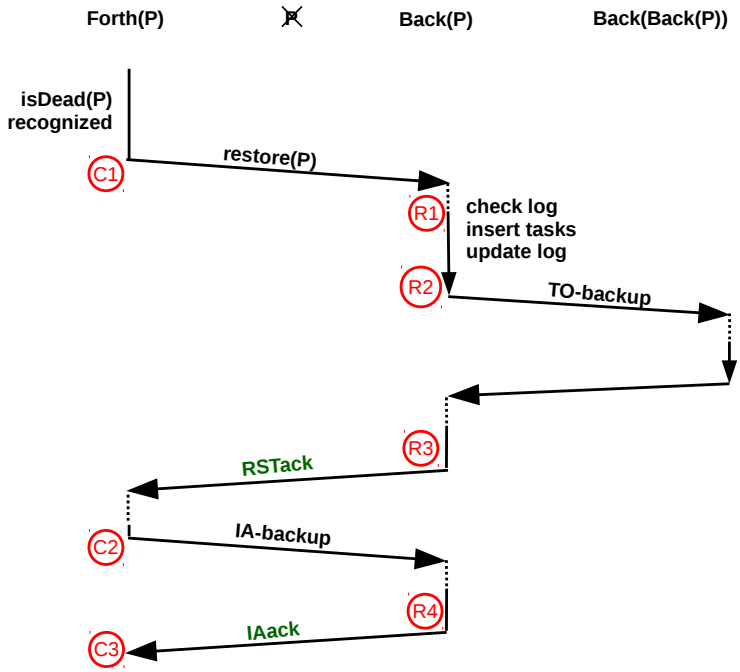


FIG. 4.2. *Restore Protocol of the Synchronous Scheme*

Depending on context (e.g. overall running time), the lower robustness of this scheme may be sufficient since typical steal rates are low [17], and place failures are rare, such that their coincidence is unlikely.

Remark on Implementation. In a few cases, the synchronous scheme uses asynchronous messages, similar to the original GLB. The original GLB marks these messages with `@Uncounted`, such that `finish` ignores these activities in its bookkeeping to save time. Unfortunately, `@Uncounted async` discards exceptions [18]. Since we need exceptions for fault tolerance, we had to eliminate some of the `@Uncounted` annotations.

5. Asynchronous Fault Tolerance Scheme. As noted in Sect. 3, each worker has a three-phase structure, which we have implemented by the following loop:

```
while (nTasks > 0 || stealFailed) {
  processUpToNTasks();
  Runtime.probe();
  processRecorded();
}
```

Here, `stealFailed` is true iff all $w + z$ steal attempts failed. At the call to `Runtime.probe()`, all pending remote activities are run. They may record requests, e.g. steal requests, in the worker's data structures. Recorded requests are carried out by the worker in `processRecorded`.

5.1. Steal Protocol. The asynchronous steal protocol is depicted in Fig. 5.1. A victim `V` may have more than one thief, but for clarity, only one thief `F` is depicted here. As compared to the protocol in Sect. 4, this protocol has been designed to strictly minimize communication. This manifests in minimalist handshaking and handling multiple steal requests together. These aspects will be elaborated later.

First, let us look at a positive side effect of stealing: redundancy. When tasks are sent from `V` to `F`, they are copied, and thus exist twice. Inspired by resilient divide-and-conquer algorithms [9, 10], we exploit the redundancy for fault tolerance. Thus, the backups are not updated to account for task movement, or at least not immediately. Instead, the stolen tasks are kept at `V`, and only a link to `V` is sent to `Back(F)`. If `F` dies, `Back(F)` obtains the tasks from `V`. This way, the backup volume is reduced.

Nevertheless, excessive scattering of a worker's data must be avoided, since it would increase the probability

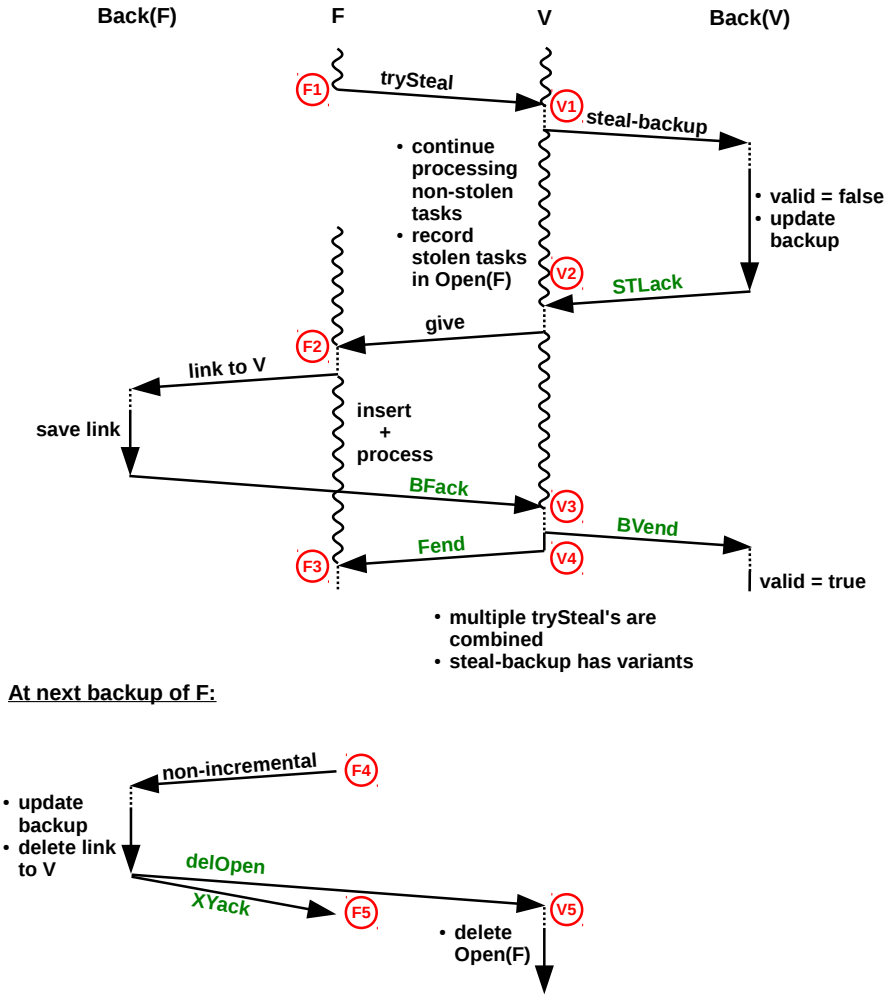


FIG. 5.1. *Steal Protocol of the Asynchronous Scheme*

of coincident failures. Therefore, as shown at the bottom of Fig. 5.1, the links are replaced by real tasks at any next backup. The backup, which may be scheduled at `F` for whatever reason, incorporates the tasks behind the links (if not finished yet). It is called an *afterMerge*-backup. After this backup, a `delOpen` message is sent to `V`, to release the tasks.

Because of the algorithm's three-phase structure, it is well possible that multiple steal requests are received at `V` at the same time. Thus, in Fig. 5.1, a victim `V` may have more than one thief, but for clarity only one of them is depicted. To reduce the number of messages, multiple steal requests are handled together in a so-called *transaction*. Transactions are numbered by *tan*s, which are assigned consecutively by each worker i. Consequently, a pair (i, tan) is system-wide unique.

First, `V` runs through the thieves in some particular order, which is defined later. For each thief `F`, it determines the tasks to be sent. According to the lifeline scheme, the first thief gets about half of the tasks, the second a quarter, and so on. Any remaining requests are rejected. The transaction is formed from those thieves that get tasks. Their identities are kept in a list at `V` during the transaction.

The tasks destined for the different thieves are removed from the local pool, and inserted into a local data structure at `V`, called `Open`. It is a P-size array of lists, where each entry `Open(F)` holds the tasks to be sent from `V` to `F`. Additionally, references to the task groups of the current transaction are saved in another list, to

speed up access.

Before giving the tasks to thieves, V informs its backup partner. In Fig. 5.1, this is called steal-backup. Note that a single backup is sufficient for all steal requests of a transaction. A steal-backup differs from other types of backup in that the tan and a list of thieves are included.

Note that the stolen tasks are omitted from the steal-backup, which reflects our approach of relying on the redundancy of stealing. Since at least half of the tasks are stolen, the backup volume is significantly reduced this way.

A steal-backup may be an *afterMerge* backup. After the steal-backup, V and its backup are synchronized. Thus, the counter for the kn time period may be reset, which relativizes the steal backup's expense.

During the steal-backup, V goes on processing tasks, to avoid loosing time while waiting. When the backup is finished, as signalled by the receipt of the STLack message, V gives the task groups to the respective thieves. This operation runs fast, since the task groups have already been recorded before. Then, V returns to work, again, to not loose time waiting.

At this point, V can safely give away the tasks, since Back(V) is able to and will take care of consistency. Briefly stated, if V dies, Back(V) will contact all thieves of the current transaction (whose identities have been stored), and make sure they have received their tasks. Usually they have, or will after some short waiting (except if there is a second failure). Nevertheless, the time between receiving the steal backup and the BVend message is somewhat critical for Back(V), since it does not hold V's state completely. Therefore, we denote this time period as *queasy* and mark it by valid=false.

Note the asymmetry between the victim and thief sides: F is involved in a single give of the transaction, whereas V may give tasks to multiple thieves. As stated before, F sends the steal requests consecutively, but can nevertheless receive multiple give messages at about the same time. They are handled in any order. It is not necessary to wait for the final Fend message of the first give before processing the next. Thus, there may be multiple outstanding Fend messages. F keeps them in a list, called OpenFend. Only when this list is empty, a backup may be sent from F to Back(F). The resulting delay is limited, since F may receive tasks from at most z workers.

Bookkeeping of all participants makes use of the tans. They are included in all messages of the steal protocol, and saved wherever appropriate. For instance, at V, they are added to each task group. Unlike the other participants, F saves the tans of merged task groups permanently. It needs the information since, in case V dies after (V4), it must be able to answer Back(V)'s inquiry. Fortunately, there is an efficient scheme for permanent storage: Since F's at most $w + z$ steal requests go to different victims, subsequent transactions with the same victim are processed in the order of their tans. Therefore, it is sufficient to keep each V's most recent received tan.

For each give, F first sends the respective link to Back(F), and then inserts the tasks. Right afterwards, F starts processing the tasks. There is no problem about returning to work since, if F dies before (F4), Back(F) resets to a previous state anyway, and if F reaches (F4), the results of work are correctly reflected in the new backup.

As noted above, the handshaking in Fig. 5.1 has been designed to be minimalistic. In particular, Back(F) only reports to V, but not to F. When V receives a BFack message, it immediately sends Fend to the respective thief. Moreover, it removes F from the thieves-list of this transaction. It easily recognizes when all F's have sent their BFack. Then, the transaction is over, and V signals that to Back(V) via a single BVend message.

5.2. Restore Protocol. Restore presupposes timely failure notification, which has been explained in Sect. 3. In the following, we assume that Forth(P) has recognized P's failure. The restore protocol is depicted in Fig. 5.2. Note that Forth(P), P, etc. need not have successive numbers, but workers in-between may have been restored.

The protocol is time-critical between (C1) and (R4), since Forth(P)'s data are unsecured. Therefore, Forth(P) and Back(P) run the protocol in so-called *emergency mode*. In this mode, processing of tasks pauses, for higher reactivity. Moreover, only urgent and short messages are handled. The other messages must be received, but they are only stored and handled later. The loss in efficiency due to emergency mode can be neglected, since we assume that failures are rare.

As shown in Fig. 5.2, the protocol resembles the restore protocol of the synchronous fault tolerance scheme,

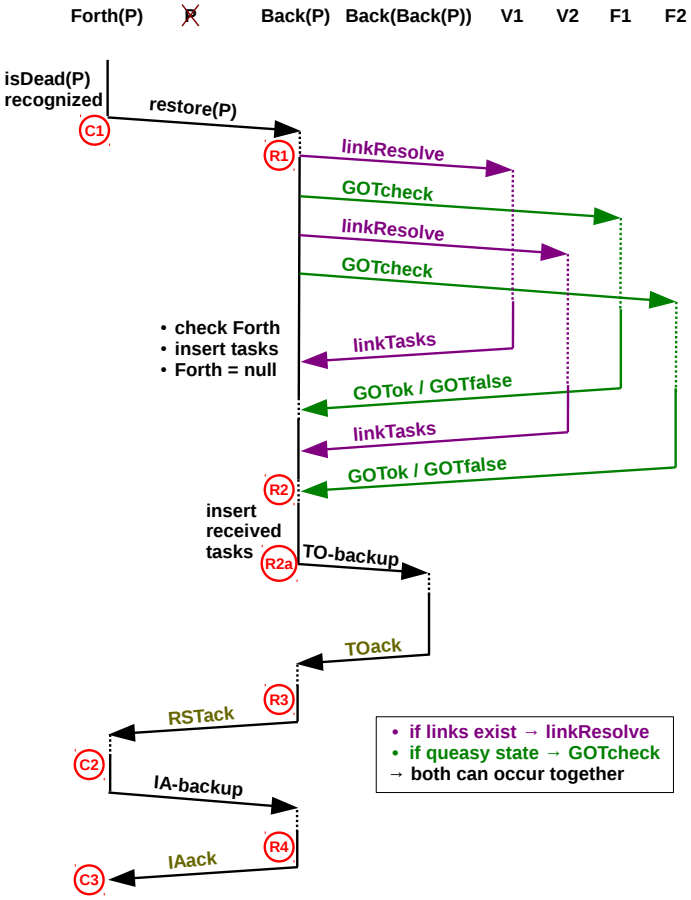


FIG. 5.2. *Restore Protocol of the Asynchronous Scheme*

but accounts for the fact that tasks referenced by links must be collected. Also, to improve robustness, failure of `V` in queasy state does not automatically lead to program abort. Instead, `Back(V)` checks whether the tasks have been taken over by the thief side. Only if the tasks are lost, the program is killed.

The protocol begins with a restore message from `Forth(P)` to `Back(P)`. Upon receipt, as explained at the end of Sect. 3, `Back(P)` makes sure that `P = Forth`, and that the source equals `Forth(Forth(P))`. Since failures may happen at any time, three particular situations must be taken care of:

1. P fails between (V1) and (V4) in the steal protocol, i.e., `Back(P)` is in queasy state.
2. P fails between (F2) and (F5), and `Back(P)` has saved links.
3. P fails when `Forth(P)` and/or `Back(P)` are already involved in another restore protocol.

Situations 1 and 2 may appear together, since a worker is allowed to receive tasks between (V2) and (V3). In situation 1, `Back(P)` contacts all thieves of the current transaction, called `GOTcheck` in Fig. 5.2. They check whether they have received tasks for the corresponding tan, otherwise wait for a moment in case the message is late, and then respond (called `GOTok/GOTfalse` in the figure). The restore can only be performed if all answers are `GOTok`. Otherwise, `Back(P)` aborts the program.

In situation 2, `Back(P)` sends a `linkResolve` message to each former victim. Usually, the victim finds the respective tasks in `Open(P)` and sends them back in a `linkTasks` message. Occasionally, though, the victim has recognized P's failure before `Forth(P)` did (from a timeout on an outstanding `BFack`). In this case, the victim has already taken back the tasks, i.e., it re-inserted them into its own pool and removed them from `Open(P)`. Consequently, the tasks are not available anymore. Interestingly, such orphaned links are no problem: The victim just sends back `null` instead of a task group, and therewith informs `Back(P)` that it has taken care of

the tasks. One may ask why the victim takes back the tasks when observing timeout. The reason is that `F` may have failed before (F2), and, thus, there is no link.

Situation 3 stands for several sub-cases, which will be discussed below. Now, let us look at a normal protocol run. After having sent the `linkResolve` and `GOTcheck` messages, `Back(P)` waits for answers. During that time, in fact, it performs the `Forth` checks discussed earlier. Moreover, it inserts the tasks from the saved backup into its own pool, as it does with the tasks arriving via `linkResolve` in course of time.

Sending out the messages before deciding whether to perform the restore at all may appear counter-intuitive. This order is more economical, though, since it avoids loosing the waiting time, and time-critical messages are sent earlier this way. Changing the order does not compromise correctness: If `Back(P)` remains alive, the order does not matter. If `Back(P)` dies before performing the `Forth` checks, two successive workers are gone, and `Back(Back(P))` kills the program anyway. While details are omitted for brevity, considerations like that determine the ordering of actions throughout our algorithm.

When all outstanding `linkTasks` and `GOTok` messages have been received and the tasks have been inserted (denoted (R2A) in the figure), `Back(P)` has successfully restored P. This corresponds to (R2) in Fig. 4.2. Like there, the protocol finishes with `TO-` and `IA-backup`s.

The most important sub-cases for situation 3 are handled as follows:

- `Back(P)` dies during the protocol:After recognizing the failure,`Forth(P)` figures out that `Back(Back(P))` is the next place alive along the ring, and sends a restore message. If `Back(Back(P))` has already received `Back(P)`'s `TO-backup`, it performs the restore, otherwise it aborts the program.
- `Forth(P)` receives a `restore` between (C1) and (C2): `Forth(P)` kills the program.
- `Back(P)` recognizes failure of `Back(Back(P))`: `Back(P)` kills the program.

In the second and third cases, the program could in principle be continued by nesting protocol execution. This would, however, complicate the program. Since we have assumed unfavourably correlated failures to be unlikely, we instead abort the program.

5.3. Communication Structure. As noted before, the actor scheme allows us to handle incoming messages in a predefined order. Moreover, messages can be easily prioritized or delayed until the next iteration of the main loop. The order is accurately defined for all message types of our algorithm but, for brevity, the following list is restricted to a few typical representatives:

- Handshaking and link management messages are handled immediately (e.g. `noTasks`, `BFack`, receipt of a link, `linkResolve` and `GOTcheck`)
- Arrival of a `TO-backup` or `IA-backup` has the highest priority of recorded requests.
- Arrival of `restore` comes thereafter, since a `TO-backup` may increase the chances for successful restore.
- Upon arrival of `give`, the link is sent immediately, but the (more time-consuming) merge action is recorded.
- Steal requests come after merge, so that the steal backup replaces the links, and the received tasks can be distributed.
- Regular backups have lowest priority.

6. Experiments. Experiments were conducted on an Infiniband-connected cluster, where each node comprises two 8-core Intel Xeon E5-2760 CPUs as well as 32 GB main memory. All experiments used one place per node on up to 128 nodes. We deployed X10 version 2.5.2 (SVN, revision 29421) and GCC 4.8.4 to compile the programs. Resiliency was only switched on for the fault-tolerant program versions.

We used two benchmarks: Unbalanced Tree Search (UTS) and Betweenness Centrality (BC). Both are included as samples in the X10/GLB distribution. UTS is a well-known benchmark that counts the number of nodes in a highly irregular tree, which is constructed on the fly from node descriptors [12]. Each descriptor encodes a subtree and naturally corresponds to a task. We used geometric distribution for the tree shape. BC considers the set of all shortest paths in a graph, and for each node computes the number of shortest paths running through it [13]. GLB uses UTS as an example for dynamic task pool initialization, and BC for static initialization.

For the fault-tolerant frameworks, the sample codes of UTS and GLB were slightly adapted. In particular, the BC example was extended by shadows, as explained in Sect. 3.

TABLE 6.1
Experimental results for the Parameter n and k

Benchmark	Configuration	Framework	determined n and k
UTS	small	GLB	$n = 512$
		FTGLB	$n = 4\,096$, $k = 65\,536$
		FTGLB-Actor	$n = 8\,192$, $k = 65\,536$
	large	GLB	$n = 16\,384$
		FTGLB	$n = 32\,768$, $k = 65\,536$
		FTGLB-Actor	$n = 65\,536$, $k = 65\,536$
BC	small	GLB	$n = 512$
		FTGLB	$n = 65\,536$, $k = 512$
		FTGLB-Actor	$n = 32\,768$, $k = 1\,024$
	large	GLB	$n = 16\,384$
		FTGLB	$n = 65\,536$, $k = 512$
		FTGLB-Actor	$n = 65\,536$, $k = 512$

Both UTS and BC were run with a small and a large configuration, to account for different computation-to-communication ratios. In the following, b denotes the branching factor, d the tree depth, s a random seed, and N the number of graph nodes:

- small UTS: $d = 13$, $b = 4$, $s = 19$
- large UTS: $d = 17$, $b = 4$, $s = 19$
- small BC: $N = 2^{14}$, $s = 2$
- large BC: $N = 2^{16}$, $s = 2$

We compared three program versions:

- the original GLB code from the X10 distribution (GLB),
- our synchronous GLB version from Sect. 4 (FT-GLB), and
- our asynchronous GLB version from Sect. 5 (FTGLB-Actor).

Experiments were grouped into two stages. In the first stage, we determined the optimal n and k values for the `process(n)` calls and the kn backup intervals. In the second stage, we measured performance with the best n and k values obtained before. All experiments were repeated three times, and the average was taken.

In the first stage, we started the small UTS and BC configurations on 8 nodes, and the large configurations on 32 nodes, varying n. These place numbers have been observed before to lead to good performance. Although [1] reports on scalability to many more places, we were not able to achieve such speedups with our configurations and hardware.

The best n and k values are depicted in Table 6.1. The fault-tolerant program versions prefer higher n, due to their increased overhead. Although steal rates are similar for all programs, these versions need to perform additional actions to, e.g., write backups. The k values are not important for UTS, since steals, and therefore steal backups, are frequent. Therefore, any sufficiently large k value performs well. For BC, we chose k such that a regular backup is written roughly every 10 seconds.

Results of stage 2 are depicted in Figs. 6.1 and 6.2 for the small configurations, and in Figs. 6.3 and 6.4 for the large configurations.

For a deeper analysis, we divided a typical run into three phases:

- Setup and initial work distribution: Each place writes its initial backup. Dynamic work distribution requires stealing.
- Steady state: The nodes work mostly independently, the steal rate is low.
- Final stage: More and more places run out of work and steal from each other, increasing communication.

For the small configurations, the frameworks spent a high portion of time in the final stage, which led to a high communication-to-computation ratio. The huge overhead of up to 655% for FTGLB as compared to GLB in Figs. 6.1, 6.2 and 6.4 indicates that the synchronous scheme has deficiencies. They have been largely resolved with the FTGLB-Actor framework, which exhibits almost the same performance as GLB in the UTS runs. In

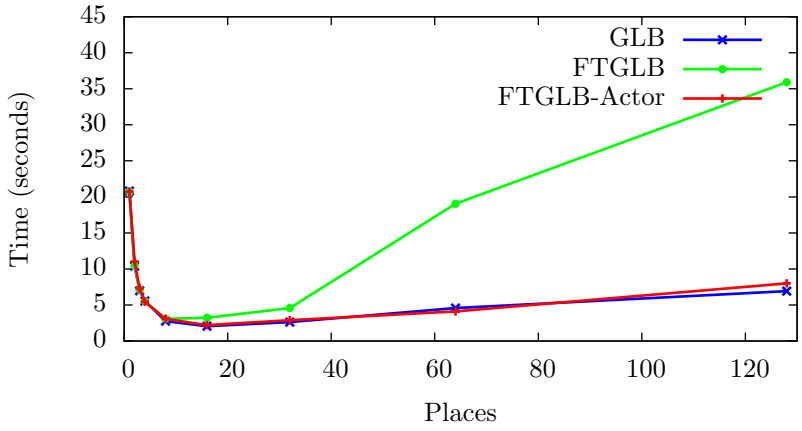


FIG. 6.1. *Experimental results for the small UTS configuration.*

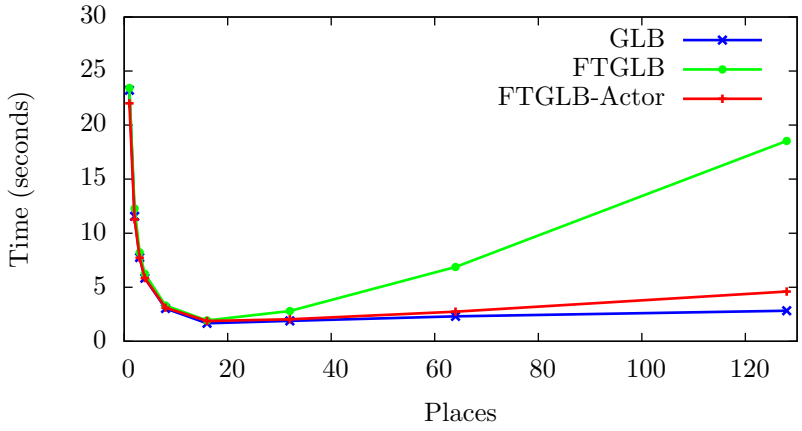


FIG. 6.2. *Experimental results for the small BC configuration.*

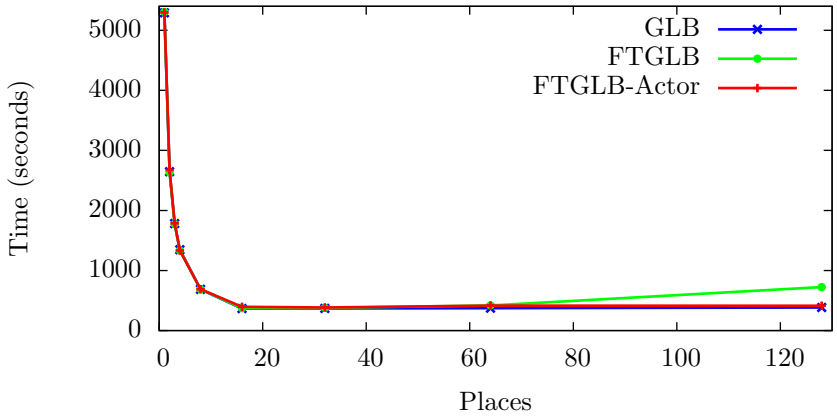


FIG. 6.3. *Experimental results for the large UTS configuration.*

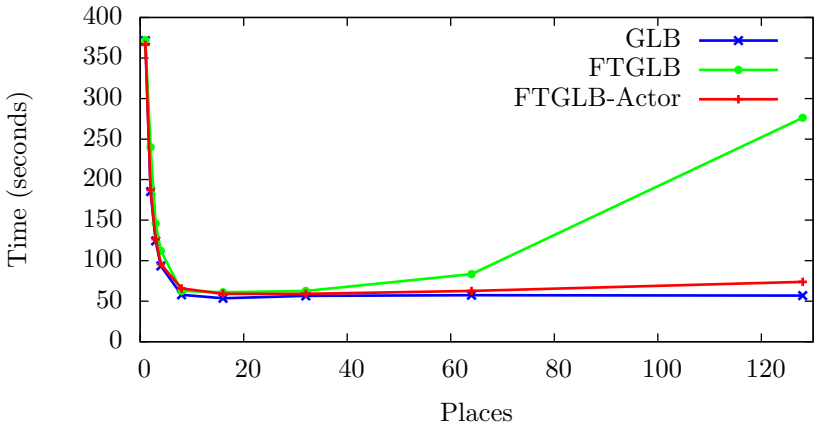


FIG. 6.4. *Experimental results for the large BC configuration.*

the BC runs, there is a notable difference of up to 62.5% between GLB and FTGLB-Actor in Figs 6.2 and 6.4. It can be explained by the need to manage shadows.

To measure the overhead of our restore protocols, we started three runs on 5 and 9 nodes for each configuration and benchmark, crashed one place shortly after the start of the task execution, and compared the execution time with that on 4 and 8 places, respectively. The overhead was about 6% in all cases.

7. Related Work. As already stated in the introduction, related work differs from ours in the task model used. First, MapReduce systems such as Hadoop [8] assume the set of tasks to be fixed. Ciel [19] extends MapReduce by permitting tasks to generate new tasks, but still task management is centralized at a master process. Second, divide-and-conquer algorithms exploit references between stolen children and their parents for fault tolerance [9, 10], whereas we discard parents and maintain only one partial result per worker. Third, side effects in tasks can be tackled by tracking operations [20].

Outside fault tolerance, load balancing and task pools are a long-standing area of intensive research, e.g. [1, 6, 14, 21]. In [22], an X10 keyword for mobile activities is suggested that brings activities and global load balancing together.

Resilient X10 has been deployed in other applications [4]. For instance, a fault-tolerant `DistArray` data structure [23] bases recovery on regular backups, similar as we do. Less closely related to our work, transient faults in task pools can be detected and handled by replication and voting [24].

8. Conclusions. This paper has introduced two different task pool algorithms that can tolerate any number of permanent place failures, except failure of the first place. The algorithms are based on backups of the local task pool contents, which are regularly written to the main memory of a neighboured place. Important aspects have been failure notification, as well as steal and restore protocols. Especially the protocols differ significantly between the algorithms. Most importantly, the first algorithm uses synchronous, and the second asynchronous communication. To reduce complexity, the second algorithm deploys an actor-like communication structure. Moreover, it exploits stealing-inherent redundancy, and requires less handshaking. Both algorithms are conservative in that a computed result is guaranteed to be correct, at the price of halting the program in a few rare cases. There are more such cases for the first than for the second algorithm.

The algorithms have been implemented in the GLB framework of X10. In experiments with UTS and BC, we observed significant overheads for the first algorithm, but the second algorithm's performance was close to that of non-fault-tolerant GLB.

There are several directions for future research. First, we will finish the implementation of the third scheme, which was mentioned in Sect. 1. Beyond that, it would be interesting to apply our approach to other task pool algorithms, especially to algorithms that allow multiple workers per place and deploy concurrent data structures. Also, it would be interesting to implement the fault tolerance schemes in other programming systems, e.g. in

MPI with user level failure mitigation [26].

REFERENCES

[1] V. Saraswat, P. Kambadur, S. Kodali *et al.*, "Lifeline-based global load balancing," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, 2011, pp. 201–212.

[2] M. Bungart, C. Fohry, and J. Posner, "Fault-tolerant global load balancing in X10," in *Proc. SYNASC Workshop on HPC Research Services*, 2014, pp. 471–478.

[3] *X10 Homepage.* [Online]. Available: x10-lang.org

[4] D. Cunningham, D. Grove, B. Herta *et al.*, "Resilient X10: Efficient failure-aware programming," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2014, pp. 67–80.

[5] W. Bland, P. Du, A. Bouteiller *et al.*, "A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI," in *Proc. Euro-Par.* Springer LNCS 7484, 2012, pp. 477–488.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM SIGPLAN Notices (PLDI)*, vol. 33, no. 5, pp. 212 – 223, 1998.

[7] *Chapel Homepage.* [Online]. Available: chapel.cray.com/

[8] *Hadoop Homepage.* [Online]. Available: http://hadoop.apache.org/

[9] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *Proc. USENIX Annual Technical Symp.*, 1997.

[10] G. Wrzesinska, R. V. V. Nieuwpoort, J. Maassen, and H. E. Bal, "Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid," in *Proc. Int. Parallel and Distributed Processing Symp.*, 2005.

[11] G. Wrzesiska, A.-M. Oprescu, T. Kielmann *et al.*, "Persistent fault-tolerance for divide-and-conquer applications on the grid," in *Proc. Euro-Par.* Springer LNCS 4641, 2007, pp. 425–436.

[12] S. Olivier, J. Huan, J. Liu *et al.*, "UTS: An Unbalanced Tree Search benchmark," in *Proc. Workshop on Languages and Compilers for High-Performance Computing.* Springer LNCS 4382, 2006, pp. 235–250.

[13] *HPCS Scalable Synthetic Compact Applications #2: Graph Analysis.* [Online]. Available: http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.0.pdf

[14] U. A. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," *ACM SIGPLAN Notices (PPoPP)*, vol. 48, no. 8, pp. 219–228, 2013.

[15] C. Fohry, M. Bungart, and J. Posner, "Towards an efficient fault-tolerance scheme for glb," 2015, to appear.

[16] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263–1277, Sep. 1999.

[17] V. Kumar, D. Frampton, S. M. Blackburn *et al.*, "Work-stealing without the baggage," *ACM SIGPLAN Notices (OOPSLA)*, vol. 47, no. 10, pp. 297–314, 2012.

[18] *X10 Distribution.* [Online]. Available: http://sourceforge.net/p/x10/mailman/message/24998422/

[19] D. G. Murray, M. Schwarzkopf, C. Smowton *et al.*, "CIEL: a universal execution engine for distributed data-flow computing," in *Proc. USENIX Conf. on Networked Systems Design and Implementation*, 2011.

[20] W. Ma and S. Krishnamoorthy, "Data-driven fault tolerance for work stealing computations," in *Proc. ACM Int. Conf. on Supercomputing*, 2012, pp. 79–90.

[21] J. Dinan, D. B. Larkins, S. Krishnamoorthy *et al.*, "Scalable work stealing," in *Proc. SC Conf. on High Performance Computing, Networking, Storage and Analysis*, 2009.

[22] J. Paudel, O. Tardieu, and J. N. Amaral, "On the merits of distributed work-stealing on selective locality-aware tasks," in *Proc. Int. Conf. on Parallel Processing*, 2013, pp. 100–109.

[23] K. Kawachiya, "Writing fault-tolerant applications using resilient X10," IBM Research Tokyo, Tech. Rep. RT0960, Apr. 2014.

[24] Y. Wang, W. Ji, F. Shi, and Q. Zuo, "A work-stealing scheduling framework supporting fault tolerance," in *Proc. Design, Automation and Test in Europe.* EDA Consortium / ACM DL, 2013.

[25] *OpenMP Homepage.* [Online]. Available: http://openmp.org

[26] *Process Fault Tolerance (Unofficial Draft for MPI-Standard)*, 2014. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/raw-attachment/ticket/323/ticket323.pdf

Edited by: Dana Petcu
Received: May 20, 2015
Accepted: Jun 24, 2015

DOI 10.12694/scpe.v16i2.1089
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 187–204. http://www.scpe.org

OPEN SOURCE SOLUTIONS FOR BUILDING IAAS CLOUDS

AMINE BARKAT*, ALYSSON DINIZ DOS SANTOS† AND SONIA IKKEN‡

Abstract. Cloud Computing is not only a pool of resources and services offered through the internet, but also a technology solution that allows optimization of resources use, costs minimization and energy consumption reduction. Enterprises moving towards cloud technologies have to choose between public cloud services, such as: Amazon Web Services, Microsoft Cloud and Google Cloud services, or private self built clouds. While the firsts are offered with affordable fees, the others provide more privacy and control. In this context, many open source softwares approach the buiding of private, public or hybrid clouds depending on the users need and on the available capabilities. To choose among the different open source solutions, an analysis is necessary in order to select the most suitable according with the enterprise's goals and requirements. In this paper, we present a depth study and comparison of five open source frameworks that are gaining more attention recently and growing fast: CloudStack, OpenStack, Eucalyptus, OpenNebula and Nimbus. We present their architectures and discuss different properties, features, useful information and our own insights on these frameworks.

Key words: Cloud Computing, IaaS, OpenStack, CloudStack, Eucalyptus, OpenNebula, Nimbus

AMS subject classifications. 68M14

1. Introduction. Open source cloud platforms were born as a response to the necessity of Infrastructure as a Service (IaaS) solutions to provide privacy and control over virtualized environments. Therefore the open source cloud platforms were primely used to build private clouds. Eventually, these open source solutions can be used to set up public clouds, private clouds or a mix of them, *i.e.*, hybrid clouds. With the emergence of different open source cloud solutions, the decision to choose the most suitable one becomes a confusing task, given the specific characteristics of each platform [18]. Moreover, since hybrid clouds are the most widely used nowadays, surveying open source middlewares that simplify cluster management is an important matter. In this context, several papers analyzed and compared different platform, trying to establish a starting point to look when deciding which open source cloud technology should be adopted.

Diverse researches detail comparison among cloud solution platforms. Table 1.1 shows related studies that compare cloud solutions, highlighting the analyzed solutions and their main focus/limitation.

Some studies are generalists on the approach and intend to provide a wide view on cloud solutions. Therefore they briefly present a higher number of solutions, with no comparison - the case in [33, 34] - or with a simple general feature comparison, as in [35].

More specific studies tend to deeper detail the analyzed solutions, considering a specific point of view. [11] focus on appearance design and novel features, while [32] focus on the scalability comparison of the platforms and [37] on placement policies, platform architecture and networking of the analyzed solutions. [13] details OpenStack and CloudStack solutions, but with deeper details only for OpenStack.

[28] provide a comparison of Eucalyptus, OpenNebula and Nimbus systems. The authors describe the different features and design, accurately highlighting tendencies in the focus of each of these products. Nevertheless, the study is outdated and missing two important open source solutions that gained importance since the publication of the paper, OpenStack and CloudStack.

In this context, this work is focused in IaaS open source cloud solutions. It presents in deep details the newest versions of OpenStack, CloudStack, OpenNebula, Eucalyptus and Nimbus and compare their general features and important properties, trying to provide useful information for users that need to choose an open source cloud software. This work is a continuation of the work done before [12], which compared only CloudStack and Openstack.

The paper is organized as follows: Sections 2, 3, 4, 5 and 6 describe respectively the architecture of OpenStack, CloudStack, Eucalyptus, OpenNebula and Nimbus and their important properties; Section 7 performs

*University of Bejaia in Algeria, Politecnico Di Milano in Italy (amine.barkat@polimi.it)
†Universidade Federal do Ceara in Brazil, Politecnico Di Milano in Italy (alysson@virtual.ufc.br)
‡University of Bejaia in Algeria, Telecom Sud Paris in France (sonia.ikken@telecom-sudparis.eu)

TABLE 1.1
Cloud solutions comparison studies

Study	Analyzed platforms	Main restriction
Voras et. al. (2011) [33]	OpenNebula, Eucalyptus, Ubuntu Enterprise Cloud, openQRM, Abiquo, Red Hat Cloud Foundations, OpenStack, Nimbus and mOSAIC	Brief introduction on any solution, no comparison provided
Zeng et. al. (2012) [34]	Amazon EC2, IBM smart cloud, Google App Engine, Windows Azure, Hadoop, Eucalyptus, OpenNebula and Nimbus	Brief introduction on any solution, no comparison provided
Endo et. al. (2010) [35]	XCP, Nimbus, OpenNebula, Eucalyptus, TPlatform, ECP (Enomalys Elastic Computing Platform) and Apaches VCL	Brief introduction on any solution, brief comparison provided
Amrani et. al. (2012) [11]	Eucalyptus, OpenNebula and Nimbus	Restricted to appearance design and new features description
Von Laszewski et. al. (2012) [32]	Eucalyptus, OpenNebula, Nimbus and OpenStack	Focus on scalability
Cordeiro et. al. (2010) [37]	XCP, Eucalyptus and OpenNebula	Focus on placement policies, platform architecture and networking
Baset (2012) [13]	OpenStack and CloudStack	Further details only on OpenStack
Sempolinsky and Thain (2010) [28]	Eucalyptus, OpenNebula and Nimbus	Outdated, missing OpenStack and CloudStack

comparisons between the platforms in terms of generalities, functionalities and proprieties. Conclusions are detailed on Section 8.

2. OpenStack. OpenStack is a cloud software that offers capability to control large pools of compute, storage and networking resources. It also empowers users providing on-demand resources [29]. Starting from 2010, OpenStack was developed by Rackspace Hosting and NASA [6] aimed to provide open source cloud solution to build public or private clouds. The mission of OpenStack is to enable any organization to create and offer cloud computing services running on standard hardwares. Provisioned as open source solution, OpenStack is built keeping these core principles in mind: *(i)* open source: all code will be released under the Apache 2.0 license allowing the community to use it freely; *(ii)* open design: every 6 months the development community hold a design summit to gather requirements and write specifications for the upcoming releases; *(iii)* open development: maintains a publicly available source code repository through the entire development process; and *(iv)* open community: produces a engaged development and user community through an open and transparent process.

The first version of OpenStack was entitled Austin and was released in October of 2010. Since then OpenStack adopts a policy of two major releases per year, totalizing 11 releases so far (the next one, entitled Liberty, expected to October of 2015). The Austin release of OpenStack had only the object storage (Swift) and compute (Nova) components, and presented important restrictions, like the support of objects limited to 5GB. The support to large files was introduced in the second release, entitled Bexar, together with the image registry and delivery service (Glance). New modules were introduced only on the fifth release, entitled Essex, with the graphical user interface (Horizon) and the security component (Keystone). The sixth release, Folsom, introduced the network as a core OpenStack project (then code-named Quantum, actually Neutron) and the block storage component (Cinder). Folsom was then, the first release to embody the OpenStack three more important modules, Swift, Nova and Neutron. Since this version, all the existent modules were fixed and improved and new shared services were added, such as: metering (Ceilometer) and orchestration (Heat) on the

eighth release, Havana; the database service (Trove) on the ninth release, Icehouse; the data processing (Sahara) on the tenth release, Juno; and the bare metal service (Ironic) on the newest version, Kilo. The following Section 2.1 details the components that compound the general architecture of OpenStack.

2.1. General Architecture. As in any cloud platform, the infrastructure underneath OpenStack is standard hardware, which can contain any pieces of physical devices such as servers, disks or network devices. In order to provide cloud services, OpenStack develops virtualization layers, promoting the abstract view of physical infrastructure to end users. These virtualization layers are built up in a multicomponent architecture.

The OpenStack architecture consists of three main components: Compute (Nova), Network (Neutron) and Storage (Swift). Beside these three pillars, OpenStack has been developing many other services, each of those designed to work together to provide a complete IaaS solution. The integration of these services is facilitated through public application programming interfaces (APIs) offered by each service [6].

In the following, the detailed description of each component is provided.

2.1.1. Compute (Nova). The Compute component, codenamed Nova and written in Python, is the computing fabric controller responsible for managing large networks of virtual machines (VMs), and eventually to properly schedule VMs among available physical machines (PMs) [6]. Compute is a distributed application that consists of six components: Nova-api, Message Queue, Nova-Compute, Nova-Network, Nova-Volume and Nova-Scheduler. Nova supports the complete life-cycles of an instance in the cloud, starting from the request to initialize a VM until its termination. It follows this architecture:

- Nova-api: accepts and responds to end user compute API calls. Beside providing its own OpenStack Compute API, Nova-api is compatible with Amazon EC2 API, offering the potential to integrate with Amazon cloud services. It has another special Admin API reserved for privileged users to perform administrative actions. This component also handles the orchestration activities, such as: running an instance and the enforcing policies (*e.g.* quota checks).
- Nova-compute: is primarily a worker daemon that creates and terminates VM instances via hypervisor APIs. In order to do so, it accepts actions from the queue and performs system commands to fulfill them, while updating the database state accordingly. OpenStack supports several standard hypervisors (listed in Section 7) while keeping the openness that allows to interface other hypervisors through its standard library.
- Nova-volume: manages the creation, attachment and detachment of persistent volumes to compute instances. There are two types of block devices supported by a VM instance: (1) Ephemeral storage, which is associated to a single unique instance. One ephemerally stored block device life-cycle is coupled with the instance life-cycle, *i.e.*, when the instance is terminated, data on this storage will also be deleted; (2) Volume storage is persistent and independent from any particular instance. This storage can be used as external disk device where the data stored on it still remains even when the instance is terminated.
- Nova-network: is a worker daemon that handles network-related tasks. It accepts and performs networking tasks from the queue. Task examples include setting up bridging interfaces or changing iptable rules.
- Nova-schedule: handles the scheduling of VMs among PMs. While the scheduling algorithms can be defined by users, Nova-schedule supports by default three algorithms: (1) Simple: attempts to find least loaded host, (2) Chance: chooses random available host from service table, (3) Zone: picks random host from within an available zone. By allowing users to define their own scheduling algorithms, this component is important for building fault tolerant and load-balanced systems.
- Queue: provides a central hub for passing messages between daemons. This is usually implemented with RabbitMQ, but can support any AMPQ message queue.
- Database: stores most of the build-time and run-time state of a cloud infrastructure. For example, it provides information of the instances that are available for use or in use, networks availability or storage information. Theoretically, OpenStack Nova can support any SQL-based database but the most widely used databases currently are sqlite3, MySQL and PostgreSQL.

All the components of the OpenStack architecture follow a shared-nothing and messaging-based policy. Shared-nothing means that each component or each group of components can be installed on any server, in a

distributed manner; while the messaging-based policy ensures that the communication among all components is performed via Queue Server.

2.1.2. Network (Neutron). The Network component of any cloud platform has important attributions: *(i)* to offer accessibility to resources and services; *(ii)* to provide address binding between different services (essential to support multi-tier applications) and *(iii)* to automatically configure the network (fundamental in auto-scaling scenarios).

The OpenStack Networking component gives operators the ability to leverage different network technologies to power their cloud networking. It does so through a rich set of APIs, multiple networking models (*e.g.*, flat or private network) and flexible plug-in architecture. Especially, the plug-in architecture - with the plug-in agent - enables, not only the usage of various network technologies, but also the ability to handle user workloads. In OpenStack, at network level, developers can implement their own load balancing algorithms and plug it in the platform to achieve better workload control.

The Network architecture consists of four distinct physical data center networks:

- Management network: used for internal communication between OpenStack components. The IP addresses on this network should be reachable only within the data center.
- Data network: used for VM data communication within the cloud deployment. The IP addressing requirements of this network depend on the OpenStack Networking plug-in in use.
- External network: used to provide VMs with Internet access in the deployment scenarios. The IP addresses on this network should be reachable by anyone on the Internet.
- API network: exposes all OpenStack APIs, including the OpenStack Networking API, to tenants. The IP addresses on this network should be reachable by anyone on the Internet.

2.1.3. Storage. The Storage component, one of three main pillars of OpenStack architecture, is used to manage stored resources. OpenStack has support for both Object Storage and Block Storage, with many deployment options for each, depending on the use case.

Object Storage (codename Swift) is a scalable object storage system. It provides a fully distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving, and data retention [6]. In Object Storage, data are written to multiple hardware devices, with the OpenStack software responsible for ensuring data replication and integrity across clusters. Object storage clusters are scaled horizontally while adding new nodes. If a node fails, OpenStack replicates its content from other active nodes. Because OpenStack uses software logic to ensure data replication and distribution, inexpensive commodity hard drives and servers can be used instead of expensive equipments. Therefore, Object Storage is ideal for cost effective, scale-out storage [6].

Block Storage (codename Cinder) allows block devices to be exposed and connected to compute instances for expanded storage, better performance and integration with enterprise storage platforms, such as NetApp, Nexenta and SolidFire [6]. By managing the storage resources in blocks, Block Storage is appropriate for performance sensitive scenarios, such as: database storage, expandable file systems, or the provision of a server with access to raw block level storage.

2.1.4. User interface - Dashboard. The OpenStack dashboard is a graphical user interface, to both administrators and users, that controls compute, storage and networking resources. Through the dashboard, administrators can also manage users and set limits on resources access for each user.

2.1.5. Shared Services. OpenStack Shared services are a set of several services that span across three pillars of compute, storage and networking, facilitating the cloud management operations. These services include the identity, image, telemetry, orchestration and database services [6]:

Identity Service (code-named Keystone) is the security service to protect resources access and usage. This service provides a central directory management, mapping users to OpenStack accessible services. It acts as a common authentication system across the cloud operating system. It also supports multiple forms of authentication including standard username and password credentials, token-based systems and AWS-style logins.

Image Service (code-named Glance) is the repository for virtual disk and server images used by the VMs. In OpenStack, user can copy or snapshot a server image and immediately store it away. Stored images can be

used as a template to get new servers up and running quickly and consistently.

Telemetry Service (code-named Ceilometer) aggregates resources usage and performance data of the services deployed in OpenStack cloud. This capability provides visibility into the usage of the cloud infrastructure and allows cloud operators to view metrics globally or individually.

Orchestration Service (code-named Heat) is a template-driven engine that allows application developers to describe and automate the deployment of the cloud infrastructure. It also enables detailed post-deployment activities of infrastructure, services and applications.

Database Service (code-named Trove) allows users to utilize the features of a relational database. Cloud users and database administrators can create and manage multiple database instances as needed.

2.2. Properties. Provisioned as IaaS, OpenStack is built following an open philosophy: to avoid technology lock-ins by not requiring specific technologies and to provide user freedom to choose the best slot that matches its needs [6]. In this section, we will analyze some important properties of OpenStack.

- Live migration: OpenStack supports two types of live migration: shared storage based and block live migration. The former supports live migration scenarios where the source and destination hypervisors have access to the shared storage, while the latter does not require shared storage.
- Load balancing: OpenStack supports load balancing at different scales. First of all, the supporting feature of live migration has enabled system administrators to distribute application workloads among physical servers by means of adjusting VM placement. Moreover, it is possible to control application workloads at VM level, service provided by OpenStack Network layer, controlled by Neutron component. This component, with a flexible plug-in architecture allows the development of run-time custom algorithms to distribute workloads among VMs. Indeed, OpenStack has an on-going project called Load Balancing as a Service (LBaaS) that is aimed to provide load balancing service to end users. This service has monitoring feature to determine whether the VMs are available to handle user requests and take routing decisions accordingly. Several routing policies are supported, such as: round robin (*i.e.*, rotates requests evenly between multiple instances), source IP (*i.e.*, requests from a unique source IP address are consistently directed to the same instance) and least connections (*i.e.*, allocate requests to the instance with the least number of active connections).
- Fault tolerance: Fault tolerance can also be handled at different levels, depending on the way the IaaS system is configured and deployed. At the VMs level, in order to prevent failures, users can develop scheduling algorithms (besides the three already supported algorithms by OpenStack) for placing the VMs that best fits to his use cases. Some scheduling algorithms have been designed at the present time, such as: group scheduling (*i.e.*, VMs that provide the same functionalities are grouped and placed to separate PMs) and rescheduling (*i.e.*, rescheduling of VMs from failed host to surviving hosts using live-migration). At storage or database level, fault tolerance is achieved by using replication and synchronization to ensure that a failure occurred at one device will not break the whole system.
- Availability: In OpenStack, high availability can be achieved through different setups depending on services types, stateless or stateful services. Stateless services can provide answer to a request without requiring further information of other services or historical data. OpenStack stateless services include nova-api and nova-scheduler. For these services, high availability is achieved by providing redundant instances and load balancing them. In the opposite, stateful services are ones that require other information to answer a request, which difficult the high availability achievement. These services, *e.g.*, database or storage, can be highly available by using replication, with the adequate synchronization between the main version and replicated versions in order to keep the system consistency [23].
- Security: OpenStack has a separated service (Identity service) which provides a central authentication management across the cloud operating system and users. The possibility to set up VPNs and firewalls is available.
- Compatibility: OpenStack is compatible with Amazon EC2 and Amazon S3 and thus client applications written for Amazon Web Services can be used with OpenStack with minimal porting effort [6]. In terms of hypervisors, OpenStack supports multiple hypervisors, *e.g.*, Xen, KVM, HyperV, VMWare, etc. Other hypervisors with existing standard drivers can also be interfaced with OpenStack through standard library, *e.g*, libvirt library.

3. CloudStack. CloudStack [4] is an open source software platform, written in Java, designed for development and management of cloud Infrastructure as a Service. It aggregates computing resources for building private, public or hybrid clouds. CloudStack brings together the "Stack" of features requested by companies and users, like data centers orchestration, management and administration of users and NaaS (Network as a Service).

The start of CloudStack was with Cloud.com in 2008. In May 2010, it was open source under GNU General Public License. Citrix bought CloudStack in July 2011, then in April 2012, Citrix donated CloudStack to Apache Software Foundation (ASF) where it was relicensed under Apache 2.0 and accepted as an incubation project. Since March 2013, CloudStack became a Top Level Project of Apache. Many companies are basing on CloudStack for building and managing their cloud infrastructures. Among these companies, there are: Nokia, Orange, Apple, Disney and many others.

The first major release of CloudStack since its graduation from the Apache Incubator is 4.1.0, major changes were realized on the platform with modifications in the codebase to make its development easier, Maven was used as a build tool, and for creating RPM/Debian packages a new structure was used. The second major release was 4.2, many features were born due to cooperations with different industries, including an integrated support of Cisco UCS compute chassis, SolidFire storage arrays, and the S3 storage protocol. Next CloudStack releases until the last one 4.5.1, mainly include new features, bug fixes and improvement of the interaction with different hypervisors.

3.1. General Architecture. In CloudStack, physical resources are organized and managed in a hierarchical structure. The lowest level contains the computational devices, *i.e.* hosts and primary storage. Different hosts accessing a shared storage form a cluster. Clusters combined by a layer 2 switch form a pod. Pods are grouped together with secondary storage by a layer 3 switch to form a zone. At the highest level, zones are grouped to create a region. All these resources are managed by a Management Server. In the following, we describe in details each component that forms the whole architecture.

3.1.1. The architectural levels. The architectural levels are the core of the CloudStack architecture. These levels are: the lowest level, clusters, pods, zones and regions.

A *host* represents a physical computational machine that contains local storage. The physical hosts are virtualized by hypervisors. CloudStack supports many hypervisors for VMs management such as Xen, KVM, vSphere, Hyper-V, VMWare, etc. as well as bare metal provisioning. All hosts within a cluster must be homogeneous in terms of the hypervisor, with the possibility of having heterogeneous hypervisors in different clusters.

Within a *cluster*, hosts are tied together into the same computational pool with the primary storage. In a cluster all hosts share the same IP subnet. The primary storage can be any kind of storage supported by the hypervisor and one cluster can have multiple primary storage devices.

A *pod* is a collection of different clusters linked with a layer 2 switch. Hosts in the same pod are in the same subnet. A pod is not visible to the end user.

A *zone* contains pods that are attached to the secondary storage using a layer 3 switch. Zones are visible to the end user and they can be private or public. Public zones are visible to all users in the cloud, while private zones are visible only to users from a particular domain. Zones main benefits are isolation and redundancy. Often, a zone corresponds to a data center; although if a data center is large enough, it can have multiple zones.

A *region* is the largest organizational unit in CloudStack. A region contains multiple zones distributed in geographic locations close to each other.

3.1.2. Management Server. A Management Server is used to manage all resources in cloud infrastructure through APIs or UI. One management server can support around 10K hosts and can be deployed either on a physical server or on a VM. In case of the existence of more than one management server, user interaction to either of them will return the same result. This ensures high availability of CloudStack.

A database is required for management servers to be persistent. In order to prevent single point of failure, we can have one primary database and several database replicas always synchronized with the primary copy.

3.1.3. Storage. In addition to the host local storage, CloudStack manages two main types of storage: primary storage and secondary storage.

- Primary storage: is a storage associated with a cluster or a zone, with the possibility of a single cluster to deploy multiple primary storages. This kind of storage is basically used to run VMs and to store application data. Since this storage interacts directly with applications deployed in VMs, it can be expensive in terms of I/O operations, reason why it is placed physically near to the hosts.
- Secondary storage: supports two different types, NFS and Object Storage. It is used to store ISO images, templates and snapshots.
  - ISO image: is used when user wants to create a VM.
  - Template: is the base operating system image that the user can choose when creating new instance. It may also include additional configuration information such as installed applications.
  - Snapshot: is used as backup for data recovery service. CloudStack supports two types of snapshot: individual snapshot and recurring snapshot. The former is one-time full snapshot, while the latter is either one-time full snapshot or incremental snapshot.

3.1.4. Networking. CloudStack supports the use of different physical networking devices (*e.g.* NetScaler, F5 BIG-IP, Juniper SRX, etc). In CloudStack, users have the ability to choose between two types of network scenarios: basic and advanced. The basic scenario is for an AWS-style networking. It provides a single network where guest isolation is done through the layer 3 switch. The advanced scenario is more flexible in defining guest networks [4]. For example, the administrator can create multiple networks to be used by the guests. CloudStack provides many networking services. Examples include:

- Isolation: CloudStack assures the isolation of networks, by allowing the access to the isolated network only by virtual machines of a single account.
- Load Balancing: to balance the traffic in the cloud, the user can create a rule to control and distribute the traffic and apply it to a group of VMs. Within the defined rule, user can choose a load balancing algorithm among the supported ones.
- VPN: to access a VM using a CloudStack account, users can create and configure VPNs. Each network has its own virtual router, so VPNs are not shared across different networks. Using VPN tunnels, hosts in different zones are allowed to access each other.
- Firewall: one host can access others in the same zone without passing through the firewall. Users can use external firewalls.

3.2. Properties. The main properties of CloudStack are [4]:

- Live migration: A live migration of running VMs between hosts is allowed in CloudStack through the Dashboard. Depending on the VMs hypervisor, migration conditions can be different. For example, live migration using KVM hypervisor will not support the use of local disk storage, and source and destination hosts have to be in the same cluster; while Xen and VMWare support local disk storage and allow to migrate between different clusters [3].
- Load balancing: a load balancer is an optional component of CloudStack that allows traffic distribution among different management servers [5]. In addition to creating rules and using load balancing algorithms, CloudStack offers the possibility to integrate with external load balancers such as Citrix NetScaler [27].
- Fault tolerance: in CloudStack, fault tolerance is achieved at different scales, *e.g.* management, database and host levels. In order to prevent failures of management server, the server can be deployed in multi-node configuration. If one management node fail, other nodes can be used without affecting the functioning cloud. Failures at database level are handled by using one or more replication of the database linked to the management server. For host's fail-over, CloudStack recovers the VM instances by taking the images from secondary storage and using application data in primary storage.
- Availability: CloudStack ensures high availability of the system by using multiple management server nodes which may be deployed with load balancers.
- Security: in addition to isolation using different accounts, VPNs and firewalls, CloudStack offers the isolation of traffic using the strategy of security groups. These are sets of VMs that filter the traffic on the basis of configuration rules. CloudStack provides a default security group with predefined rules, that can be modified if necessary.
- Compatibility: the pluggable architecture of CloudStack allows one cloud to support different hypervisor

implementations, including: Hyper-V, KVM, LXC, vSphere, Xenserver , Xen Project and also bare metal provisioning. Moreover, CloudStack is compatible with Amazon API and enables the integration of these two platforms.

- Scalability: CloudStack can manage thousands of servers distributed in different data centers and different locations, thanks to the management server capability (one management server node can manage a big pool of physical resources), and the possibility of using multiple management servers for reducing VMs downtime.
- API extensibility: The CloudStack APIs are very powerful and allow developers to create new command line tools and UIs, and to plug them into CloudStack architecture. If the developer wants to use a new hypervisor, new storage system or new networking service, he just needs to write a new plug-in in Java and integrate it.

4. Eucalyptus. Eucalyptus is a popular open source IaaS product, provided by Eucalyptus Systems [7]. It is used to implement, manage, and maintain private and hybrid clouds (but cannot build public clouds) with a main key design feature, which is Amazon Web Services (AWS) API compatibility. Many programming languages can be used to code Eucalyptus including: Java, C, Groovy, Shell, Perl, Python [7].

Originally it was designed at the University of California, Santa Barbara, as set of services that could emulate AWS on a different site apart from Amazon servers, with the aim of linking together AWS, supercomputer centers at National Science Foundation and several university sites [20]. It became a for-profit organization in 2009. In 2012, Eucalyptus started a partnership with AWS that allowed to develop more AWS-compatible environments, and to create hybrid clouds by facilitating movements of instances -created from stored Operating System Images- between an Eucalyptus private cloud and Amazon Elastic Compute Cloud (EC2). In September 2014 HP acquired Eucalyptus and lunch it under the Helion Eucalyptus name.

Beside its high AWS compatibility that allows running an application on AWS and Eucalyptus without any modifications, Eucalyptus is characterized by its high availability configuration, easy installation and simple user interface. Its main clients include: AppDynamics, Nokia, NASA, Puma and others [7].

Eucalyptus since its first releases contains its main feature which is compatibility with Amazon Web Services. Many announced releases were just maintenance releases and do not contain new features. Eucalyptus Cloud User Console was included in the release 3.2.0 which enabled self-service provisioning of different resources for cloud users. Eucalyptus 3.3.0 introduced many important features such as auto-Scaling, Load Balancing, CloudWatch and new VM Types, and this make it one of the most important releases of Eucalyptus. Many things were changed in release 4.0.0, including changes in the whole architecture and how components are installed and registered, changes in storage architecture, changes in networking mode and a new administrator roles were set. The current release 4.1.1 (11/5/2015) is a maintenance release for 4.1.0 in which a support for CloudFormation was add with new Management Console features and new instance status checks.

4.1. General Architecture. Eucalyptus has a highly modular, hierarchical and distributed architecture [22]. Users familiar with AWS do not find any difficulties with Eucalyptus because it replicates the same interaction tools and interfaces used in AWS, such as: euca2ools - the command-line tool - or the Eucalyptus User Console - a GUI based tool. Eucalyptus architecture is characterized by five main components: Cloud Controller, Cluster Controller, Storage Controller, Node Controller and Scalable Object Storage, and one optional component: VMware Broker. These components are grouped in three different logical levels: Cloud Level, Cluster Level and Node Level [7]. In the following sections we describe each logical level and the associated components.

4.1.1. Cloud Level. Contains two components: The Cloud Controller and the Scalable Object Storage.

The *Cloud Controller* (CLC) is a Java program that provides the interface to the cloud, and each cloud contains only one. CLC contains query interfaces and a EC2-compatible SOAP. It handles users requests and provides high level authentication, quota management, accounting and reporting. CLC does also meta-schedule and manages different cloud resources after collecting information provided by the Cluster Controller.

The *Scalable Object Storage* (SOS) is an Eucalyptus service that allows the use of external (open source or commercial) storage solutions. SOS is equivalent to AWS Simple Storage Service (S3). In case of small deployments Eucalyptus has a basic storage implementation called Walrus that has two main functionalities:

(i) storage of system files that could be accessible from different nodes (it may contain: VM images, volumes, snapshots, Linux kernel images, Root filesystem), and (ii) be used as Storage as a Service to store users data and applications.

4.1.2. Cluster Level. Each cluster is formed by a group of nodes linked with a LAN network. This level contains two main components: Cluster Controller and Storage Controller, and one optional: VMware Broker. Clusters are under subnets with a specific range of IP addresses, what compromises the flexibility and is a disadvantage of Eucalyptus.

The *Cluster Controller* (CC) is a C program that collects information, manages the execution of the virtual instances and virtual network and verifies the respect of Service Level Agreement. Multiple Cluster Controller could exist within one cloud. It works as an intermediate between the Cloud Controller, the Storage Controller and the Node Controller. CC is equivalent to AWS availability zone.

The *Storage Controller* (SC) is a Java program developed to have the same features as AWS Elastic Block Store (EBS). It manages the snapshots and volumes of a specific Cluster and controls the block-access network storage. The SC also communicates with different storage systems (NFS, iSCSI, SAN,...), with the Node Controller and the Cluster Controller.

The *VMware Broker* (VB) is an optional component available only in Eucalyptus version with VMware support. It provides an AWS compatible interface for VMware environments that allows the deployment of VMs on VMware infrastructure elements. VB directly manages the communication between the CC and the VMware hypervisors (ESX/ESXi), or it is possible that it passes through VMware vCenter.

4.1.3. Node Level. This level is composed by a single component, the Node Controller (NC), a C program that hosts VMs and their associated services, and manages the endpoint of the virtual network. NC interacts with the hypervisor and the hosted operating system to control VMs life-cycle, their creation and termination. It collects and sends information about VMs and their associated physical resources to the Cluster Controller to make high level decisions, like when to proceed with the load balancing.

4.2. Properties.

- Compatibility: it is the main feature of Eucalyptus. AWS APIs are built on top of Eucalyptus tools which simplifies intercommunication between both [25]. AWS APIs supported by Eucalyptus are: Elastic Compute Cloud (EC2), Elastic Block Storage (EBS), Amazon Machine Image (AMI), Simple Storage Service (S3), Identity and Access Management (IAM), Auto Scaling, Elastic Load Balancing, CloudWatch and CloudFormation.
- Live migration: one of the weak points of Eucalyptus is the absence of a live migration feature. Nevertheless many external approaches was proposed to add this feature, like the one described in [19].
- Load balancing and Fault tolerance: Eucalyptus does not contains an implicit load balancing mechanism for the low level of VMs, nor for the high level of user requests, but it could be achieved using Elastic Load Balancing of AWS. Elastic Load Balancing is a service that provides fault tolerance by distributing the incoming service requests and users traffic among different Eucalyptus instances. It automatically detects overloaded instances, and redirects the traffic to more available ones. Load balancers are the core of Elastic Load Balancing, they are special Eucalyptus instances created from specific Eucalyptus VMs images. If there is a problem in one of the instances or if it is removed due an internal error, the system stops rerouting traffic to that instance until it is restored or a new one is created. Load balancing could be managed within the same or among different clusters depending on the users need. Elastic Load Balancing contains also a health check system that routinely sends check requests to instances. It uses latency, RequestCount and HTTP response code counts to verify if the instance is responding in time to users requests.
- Scalability: as mentioned before, Eucalyptus is by nature distributed, what makes it highly scalable. There is also an auto scaling feature that allows to add and remove instances and VMs depending on traffic increase, the available resources and with respect to the Service Level Agreement. Using this feature developers can scale resources up or down depending on the need. There are three main component in Auto Scaling which are:
  - Auto Scaling group: is the main component of Auto Scaling, it defines for each user the minimum

and the maximum number of scaling instances, and the related parameters. In case the user did not select any specifications, it chooses the default parameters, which are equal to the minimum number of instances to use.
  - Launch configuration: it contains the information needed to make the scaling, including instance type, VM image ID, security groups, and many others.
  - Scaling plan (policy): it defines the manner of doing the auto scaling. It could be manually or automatically, responding to CloudWatch alarms.
- Cloud Watch: is an Eucalyptus service that collects raw data from different cloud resources (instances, elastic block store volumes, auto scaling instances and load balancers) and generates and records performance metrics. This allows users to make operational business decisions based on the historical records. Cloud Watch also configures alarms based on data from user filled metrics.
- Availability: Eucalyptus contains high availability as a feature since version 3. If an individual node or even a rack fails, Eucalyptus put other nodes into use immediately or moves to another rack in case of rack failure. This is achieved through a service that is running concurrently on physical machines which is "hot spare". The information failure is quickly diffused internally, without any signs to the users, and the failure is recovered with respect to SLA (service level agreement). In [30] an evaluation tool was proposed for testing availability in IaaS platforms. This tool was tested on Eucalyptus by faults injections in an Eucalyptus cloud testbed. This paper shows that software repairs were more often than hardware repairs.
- Security: Eucalyptus manages the access to the cloud using policies related to users, groups and accounts. A group is a set of users within an account, which have the authorization to access to a specified pool of resources. A user can belong to different groups. Security groups are also used by defining firewalls that should be applied to the set of VMs within a group. The security policies could be managed by users using the command line Euca2ools.

5. OpenNebula. OpenNebula [2] is an open source cloud platform developed by Universidad Complutense of Madrid UCM (Under the Apache 2.0 License) that delivers a simple, but feature rich, solution to build enterprise clouds and virtualized data centers. OpenNebula provides a complete toolkit to centrally manage heterogeneous virtual infrastructure, which is compatible with conventional hypervisors: VMware, Xen, KVM. It operates as a scheduler of storage layers, network, supervision and security. It is an appropriate solution for the conversion of a virtual infrastructure to IaaS platform. The centralized orchestration of hybrid environments is the heart of the tool. OpenNebula also utilizes Cloud Computing Interface (OCCI) and support to Amazon Elastic Cloud Compute (EC2) in order to expand the resources connected to it in order to form a hybrid cloud [2].

OpenNebula project started in 2005, has delivered its first version in 2008 and remains active since. Many releases have achieved today significant functional changes on the support of the storage nodes, high availability environments and ergonomics of the administrative interfaces. OpenNebula has also a wide user base that includes leading companies in banking, technology, telecommunications, and research and supercomputing centers. At present, it has more than 4000 downloads per month and many research institutes and enterprises use it to build their own cloud.

One major release of OpenNebula is launched approximately every year, with two or three minor releases to each version. The first major release was in July 2008; it supported Xen and KVM virtualization platforms to provide efficient resource management and fault tolerant design. OpenNebula 2.0 (25/10/2010) brought a significant amount of changes and new features, including: the image repository, MySQL support, authorization and authentication drivers. The next major release of OpenNebula was 3.0 (3/10/2011) and introduced many new components and features for the core and libraries, providing support to: groups users management, flexible access control list system, DB versioning and schema. Currently, the stable version of OpenNebula is 4.12.1 (8/04/2015) and presents several improvements in resource management with virtual data-centers and the exclusion of the term resource provider. Networking has been vastly improved in 4.12 (03/11/2015), with the addition of security groups, allowing administrators to define the firewall rules and apply them to the Virtual Machines.

5.1. General Architecture. A key feature of OpenNebula architecture is its highly modular design and flexibility, which facilitates integration with any virtualization platform and third-party component in the cloud ecosystem. The main components of OpenNebula architecture are: the Driver, the Core and the Tools.

The *Driver* is responsible for direct communication with the underlying operating system and for the encapsulation of the underlying infrastructure as an abstract service (e.g. virtualization hypervisor, transfer mechanisms or information services). It is designed to plug-in different virtualization, storage and monitoring technologies and cloud services into the core. These pluggable drivers are responsible for the creation, startup and shutdown of virtual machines, allocating storage for VMs and monitoring the operational status of physical machines and VMs. The roles of each service are:

- The transfer driver manages the VMs disk images on different kind of storage systems, a shared one: Network File System (NSF) or Internet Small Computer System Interface (iSCSI), or a non-shared one, such as a simple copy over Secure Shell (SSH).
- The VM driver is considered a set of hypervisor-specific drivers used to manage VMs instances on different hosts.
- The information driver is also considered a set of hypervisor-specific drivers used to monitor and retrieve the current status hosts and VMs instances through SSH.

The *Core* reflects a centralized layer that controls and monitors VM full life cycles, virtual networks (VN), storage and hosts. These components are implemented in this layer by invoking a suitable driver. The features of such component are: *(i)* VM manager allocates resources required by VMs to operate, implementing VMs deployment policies; *(ii)* VN manager interconnects VMs, and generates MAC and IP address for a VM; *(iii)* host manager manages a VM storage and allocates a VM disk; *(iv)* SQL Pool is a database (SQLite or MySQL) that stores configuration data and current status of hosts and VMs instances; and *(v)* request manager (XML-RPC) accesses the application programming interface directly.

The *Tools* contains tools distributed with OpenNebula. First, it includes Scheduler that manages the functionality provided by the core layer. Scheduler component makes VM placement decisions. These VMs are deployed on host nodes following specific user requirements and resource-aware policies, such as packing, striping, or load-aware. Secondly, Command line interface-CLI and Libvirt API [1], an open interface for VM management and communicating with users. Additionally, third party tools that can be easily created using the XML-RPC interface or the OpenNebula Client API. External users are capable of sharing these functionalities through a cloud interface which is provided by the Tools layer.

5.2. Properties.

- Live migration: is one of the advantages of OpenNebula. It is supported through shared storage, but it could demand a high-performance SAN (Storage Area Network) [18]. OpenNebula uses the libvirt TCP protocol to provide migration capabilities.
- Load balancing: is provided across NGINX [34], which is an open-source, high-performance web server. It is capable of handling large numbers of concurrent connections and represents a centralized manager to balance the workload. In order to distribute efficiently the I/O of the VMs across different disks, LUNs or several storage back-ends, OpenNebula is able to define multiple system datastores per cluster. Scheduling algorithms (used by load balancers) take into account disk requirements of a particular VM, so OpenNebula is able to pick the best execution host based on capacity and storage metrics [9].
- Fault tolerance: OpenNebula provides VM migration for those not running VMs. However, OpenNebula can only detect problems on VM level. If a service or an application corrupts unexpectedly, simply rebooting the VM on another node may break the continuity of the service and result in loss. This service is maintained by database back-end (registers VM information) to store host and VM information.
- High availability: OpenNebula delivers the availability required by most applications running in VMs. OpenNebula ensures high availability of the system by using multiple persistent databases back-end in a pools cluster of hosts that share datastores and VNs. It provides information in order to prepare for failures in the VMs or physical nodes, and recover from them. These failures are categorized depending on whether they come from the physical infrastructure (Host failures) or from the virtualized infrastructure (VM crashes). In both scenarios, OpenNebula provides a cost-effective fail-over solution to minimize downtime from server and OS failures, and supports high availability configurations [15].

- Security: OpenNebula takes many measures to ensure the security. The infrastructure administrator manages a secure and efficient Users and Groups Subsystem for pluggable authentication and authorization based on passwords, SSH and RSA key pairs, X.509 certificates or LDAP. OpenNebula also uses Firewall to configure the VMs, in order to shutdown TCP and UDP ports, filter some unwanted packets and defines a policy for ICMP connections. In addition, OpenNebula uses ACL (Access Control List) which is a collection of permit and deny conditions (ACL rules), allowing different role management with fine grain permission granting over any resource managed by OpenNebula, support for isolation at different levels. Since version 4.4 OpenNebula has special authentication mechanisms for SunStone (OpenNebula GUI) and the Cloud Services (EC2 and OCCI).
- Compatibility: OpenNebula can be deployed to existing infrastructures and integrated with various cloud services and multi-platform. OpenNebula currently includes an EC2 driver, which can submit requests to Amazon EC2 and Eucalyptus, as well as an ElasticHosts driver. OpenNebula supports different access interfaces including REST-based interfaces (e.g., EC2-Query API), OGF OCCI service interfaces, the OpenNebula Cloud API (OCA), and APIs for native drivers, for example, to connect with AWS.
- Scalability: OpenNebula has been tested in the management of medium scale infrastructures with hundreds of servers and VMs [15]. By a cloud federation, OpenNebula provides scalability, isolation, and multiple-site support to interface with external clouds. This allows complementing the local infrastructure with computing capacity from public clouds to meet peak demands. Thus, a single access point and centralized management system can be used to control multiple deployment on OpenNebula. In OpenNebula highly scalability can be achieved through database back-end with support for MySQL and SQLite, and virtualization drivers can be adjusted to achieve maximum scalability.
- Flexibility and extensibility: OpenNebula provides extension and integration to fit into any existing data center, and flexible architecture, interfaces and components, allowing its integration with any product or tool. OpenNebula offers different means to easily extend and adjust behavior of the cloud management instance to the requirements of the environment and use cases, *e.g.* new drivers can be easily written in any language for the main subsystems to easily leverage existing IT infrastructure and system management product [9].

6. Nimbus. Nimbus is an open source solution (licensed under the terms of the Apache License) for using cloud computing in the context of scientific applications. Although the focus on the scientific community, Nimbus approaches three goals targeting three different communities: (1) Enable resource owners to provide their resources as an infrastructure cloud; (2) Enable cloud users to access infrastructure cloud resources more easily, and (3) Enable scientists and developers to extend and experiment with both sets of capabilities. These goals are related with the architecture of Nimbus. Its main architecture can be divided in two components, each one responsible for attending one of the goals. The first goal is realized by the Nimbus Infrastructure and the second by the Nimbus Platform. The third goal is realized by the strong support of open source development practices via modular, extensible code and engagement with open source developers [8].

Released in 2005, to solution is kept on GitHub since 2009 and its updates are revised by an international researchers committee. Nevertheless, recently, Nimbus is missing regular updates. The last release is 2.10.1 (release date: 27/02/2013) and the last github update is the cloud client version 022 (release date: 27/10/2013). Although some new features have been introduced, like the multi-cloud VM image generator for FutureGrid users (release date: 04/06/2014) and some additions on the Phantom component (release date: 15/07/2014), the focus of the Nimbus team seems to be more on collaborating with another research groups than on developing the Nimbus.

6.1. General Architecture. This section presents further details on the components Nimbus Platform and Nimbus Infrastructure.

6.1.1. Nimbus Platform. This is an integrated set of open source tools that allows users to easily leverage Infrastructure-as-a-Service (IaaS) cloud computing systems. This includes application instantiation, configuration, monitoring, and repair [8]. The Nimbus platform is divided in three modules: the context broker, the elastic scaling tools and the deployment coordination.

The *context broker* is a service that allows clients to coordinate large virtual cluster launches automatically and repeatably. The context broker requires that each VM runs a lightweight script at boot time called the context agent. This context agent depends only on Python and on the ubiquitous curl program, that securely contacts the context broker using a secret key. To enforce security the context broker uses contextualization, *e.g.*, the key is created on the fly and seeded inside the instance. This agent gets information concerning the cluster from the context broker and then causes last minute changes inside the image to adapt to the environment [8].

The *elastic scaling tools* on Nimbus are called EPU. The EPU system is used with IaaS systems to control highly available services. On these kind of services any failures are compensated with replacements. EPU is also useful with services that can be configured to be elastic; it responds to monitoring signals with adjustments of the amount of instances that composes a service. If the service cannot handle instances being added and dropped on the fly (many services have static node-number configurations), EPU still provides automatic launch, monitoring, and failure replacement capabilities [8].

6.1.2. Nimbus Infrastructure. This is a set of tools that provides the IaaS at the Nimbus cloud computing solution. The Nimbus infrastructure is divided in Workspace service and Cumulus.

The *Nimbus workspace service* is a standalone site VM manager that can be invoked by different remote protocol front-ends [8]. The workspace service is web services based and provides security with the GSI authentication and authorization. Currently, Nimbus supports two front-ends: Amazon EC2 and WSRF. The structure of the workspace service is composed by three modules: workspace control, workspace resource manager and workspace pilot.

- Workspace control: controls VM instances, manages and reconstructs images, integrates a VM to the network and assigns IP and MAC addresses. The workspace control tools operate with the Xen hypervisor and can also operate with KVM. Implemented in Python in order to be portable and easy to install [35, 8].
- Workspace resource manager: is an open source solution to manage different VMs, but can be replaced by other technologies such as OpenNebula [35, 8].
- Workspace pilot: is responsible for providing virtualization with few changes in cluster operation. This component handles signals and integrates administration tools [35, 8].

The *Nimbus Cumulus* is an open source implementation of the Amazon S3 REST API. It provides an implementation of a quota-based storage cloud. In order to boot an image on a given Nimbus cloud, that image must first be put into that same clouds Cumulus repository, although advanced use cases can by pass this restriction. In practice, it is used as the Nimbus repository solution but can also be installed standalone. Cumulus is designed for scalability and allows providers to configure multiple storage cloud implementations [8].

6.2. Properties.

- Live migration: Nimbus has no in built support to live migration. It is possible to migrate virtual machines only at hypervisor interface level [26]. The Nimbus team recently participated in a research that addresses network contention between the migration traffic and the Virtual Machine application traffic for the live migration of co-located Virtual Machines, and the live migration support is on the map of next improvements of the solution [16].
- Load balancing: At VM level, Nimbus features a system of Nagios plugins that can give information on the status and availability of the Nimbus head node and worker nodes, including changes of the virtual machines running on the worker node [24]. Specifically from the cloud provider's point of view Nimbus does the back-filling of partially used physical nodes, allowing also preemptable virtual machines. On the platform level, the cloudInit.d tool provides management to virtual machines deployed and allows compensation of stressed workloads based on policies and sensor information. Nimbus also provides tools to handle capacity allocation and capacity overflow. For example, the attribution of variable lease limits to different users - as a means of scheduling - is standard within Nimbus. In addition, the idea of allowing EC2 or another cloud the ability to pick up excess demand is heavily researched with Nimbus [28].
- Fault tolerance: Nimbus presents fault tolerance only at storage level, through the integration of Nimbus Storage Service with Globus GridFTP [10]. GridFTP - an extension of the standard File Transfer Protocol (FTP) for use with Grid computing - provides a fault tolerant implementation of FTP, to

handle network unavailability and server problems. Moreover, transfers can be automatically restarted if a problem occurs [31].

- Availability: On the platform point of view, the EPU provides high accessible services to the user. On the infrastructure point of view, Nimbus provides high-available services through the hosted service Phantom. The Nimbus Phantom leverages on-demand resources provided by infrastructure clouds and allows users to scale VMs that are running on the many clouds of FutureSystem as well as Amazon EC2. The Phantom can be extended through decision engines, components that determine the behavior of the service. Phantom is freely available on the FutureSystem infrastucture and is provided as a highly available service itself.
- Security: Nimbus provide GSI authentication and authorization - through PKI credentials, grid proxies, VOMS, Shibboleth (via GridShib) and custom PDPs. It also guarantees secure access to VMs via EC2 key generation. In addition, Nimbus allows images and image data validation.
- Compatibility: Nimbus can be integrated with various cloud services and multi-platform mainly through web services. Nimbus Infrastructure is EC2/S3-compatible, with SOAP and Query front-ends. It is possible to upload VM images to Cumulus with the Python library Boto, or with s3cmd. It is also possible to use EC2 spot instances.

7. Cloud Solution Comparisons. This section provides a comparison between the analyzed cloud solutions, aiming at three different levels: *(i)* general comparison aimed at providing high level analysis in terms of model, policy and architecture, *(ii)* functional comparison, whose goal is to compare supported functionalities, and *(iii)* property comparison, which considers cloud properties implemented in these platforms.

7.1. General Comparison. The general comparison is provided in Table 7.1, considering general aspects: licensing, commercial model, cloud model compatibility, easiness of installation, architecture and adopters.

TABLE 7.1
General comparison

Property	OpenStack	CloudStack	OpenNebula	Eucalyptus	Nimbus
Open Source License	Apache 2.0	Apache 2.0	Apache 2.0	Linux Open-Source	Apache 2.0
Commercial model	Free	Free	Free	Free, GPLv3 (only), with proprietary relicensing	Free
Compatibility with	Private, public and hybrid clouds	Private, public and hybrid clouds	Private, public and hybrid clouds	Private and hybrid clouds	Private, public and hybrid clouds
Installation Effort	Difficult (many choices, not enough automation)	Medium (Few parts to install)	Easy (process based package installers)	Difficult (different configuration possibilities)	Easy (no root account required)
Architecture	Fragmented into many pieces	Monolithic controller	Modular (third-party component)	Five part controller and AWS	Lightweight components (IaaS service and VMM node)
Large organizations adopters	Yahoo, IBM, VMWare, Rackspace, Redhat, Intel, HP, etc.	Nokia, Orange, Apple, Citrix, Huawei, TomTom, Tata, etc.	CERN, CloudWeavers, IBM, Hexafrid	UEC, NASA, Sony, HP, Cloudera, Puma, USDA, FDA	Brookhaven National Labs, Cumulus project

As it can be seen, the five frameworks are generally equal with respect to commercial model, licensing policy and cloud models. Each of them has been adopted by large organizations. Nevertheless, a big difference

is spotted from the architecture viewpoint. While OpenStack is fragmented into modules, CloudStack has a monolithic central controller, OpenNebula has three main components, Eucalyptus has five parts controllers with AWS, and Nimbus have lightweight based components. This difference is explained by the open philosophy of Open Stack and OpenNebula which tries to avoid technology lock-ins and provides high degree of flexibility, extension and availability. Nimbus is also relatively easy to install and deploy comparing to others solutions. The decentralized design and different configuration possibilities of Eucalyptus make it difficult to configure and install. Like Eucalyptus, OpenStack are also characterized by increasing complexity of installation and configuration.

7.2. Functional Comparison. The functional comparison looks at the offered functionalities or technical aspects of the five presented solutions, as described in Table 7.2. Most popular hypervisors such as: Xen and KVM are supported by all the platforms, but for example VMware is not available on Nimbus. We mention here that OpenStack and CloudStack have the largest number of supported hypervisors, even though there are ways to interface with non-supported ones. Administration feature is the interface available to interact with these platforms. All solutions present Web interfaces (Web UI) and command line interfaces (CLI). Also user management is provided in all this five platforms.

TABLE 7.2
Functional comparison between the open source Cloud solutions

Functionality	OpenStack	CloudStack	Eucalyptus	OpenNebula	Nimbus
Supported hypervisors	Xen, KVM, HyperV, VMWare, LXC, vSphere	Xen, KVM, HyperV, VMWare, LXC, vSphere	Xen, KVM, VMware	Xen, KVM, VMware, vCenter	Xen, KVM
Administration	Web UI, CLI	Web UI, CLI	Web UI, CLI	Web UI, CLI	Web UI, CLI
User management	yes	yes	yes	yes	yes

Other important functional aspect of open source cloud solutions is the activeness of its community. A solution that is always seeing new releases is constantly evolving. An active community, with well documented wiki, good bug reporting and fixing system and active users support are fundamental features for the success of an open source solution. At this topic OpenStack remains the largest and most active open source cloud computing project [21]. Nevertheless, CloudStack and Eucalyptus are growing and have an important participation in the market, while OpenNebula and Nimbus are less active. In this point, OpenStack has a bigger and more active community, followed by CloudStack, Eucalyptus, OpenNebula and finally Nimbus.

The number of releases of a platform also indicates the constant evolution of a tool. OpenStack has a strict policy of regular updates, with two big releases per year and some minor releases in each version. For Eucalyptus the last major release is Version 4.1.1 (release date: 11-05-2015) and the one before was Version 4.0.2 (release date: 20-10-2014). On CloudStack the last major release was 4.5.1 (release date 03/06/2015). For OpenNebula the last major release is old, the 4.4 Beta (release date: 07-11-2013), but 13 minor releases were announced since that time, the last one is 4.12.1 (release date: 08-04-2015). The Nimbus last major release was 2.10.1 (release date: 27/02/2013) and a minor update on cloud client (release date: 2/10/2013). From this we can conclude that OpenStack, Eucalyptus and CloudStack are the most active and evolving technologies. OpenNebula also evolves, but in a slower pace, meanwhile Nimbus gives signs of possible discontinuity.

Regarding benchmark tests, as far as we know, there is no study that have made performance tests to compare this five platforms, but there are several papers comparing two or three of them. In [39] the authors performed the performance evaluation of CloudStack and OpenStack using a mutual hypervisor and under a set of defined criteria. This study showed the effect of varying resources performances (processor and RAM, hard disk size) on deployment and deletion time. The results showed that performance of OpenStack supersedes that of CloudStack. The authors in [40] compared performances of VMs for CloudStack and Eucalyptus in terms of CPU utilization, memory bandwidth, disk I/O access speed, and network performance using different benchmarks. Mainly the results shown that CloudStack performed better than Eucalyptus in most of tests.

Other benchmarks could be found in [41] where the authors evaluated performances of Nimbus, OpenNebula and OpenStack, for High Performance Computing according to HPC Challenge (HPCC) benchmark suite. The results showed that OpenStack had the best performance for HPC.

7.3. Properties Comparison. Properties comparison gives deeper insights into the five platforms, considering some important properties that an IaaS has to provide. The comparison is given in Table 7.3.

TABLE 7.3
Properties comparison between the open source cloud solutions.

Property	OpenStack	CloudStack	Nimbus	Eucalyptus	OpenNebula
Live migration	Yes	Yes	No	No	Yes
Load balancing	Yes	Yes	Yes	Yes	Yes
Fault tolerance	VM scheduling, replication	VM scheduling, replication	Through Globus GridFTP	Through AWS Elastic Load Balancing	VM scheduling, replication
High Availability	Yes	Yes	Yes	Yes	Yes
Security	VPNs, firewall, user authentication, others	VPNs, firewall, user management, others	user authentication	group and users policies	user authentication
Compatibility	Amazon EC2, Amazon S3	Amazon EC2, Amazon S3	Amazon EC2, Amazon S3, WSRF	Amazon EC2, Amazon S3	All Amazon Interfaces

Live Migration may be approached in two ways: shared storage based live migration, and block live migration. OpenNebula and OpenStack offer both possibilities. CloudStack also offers both possibilities, but the conditions change according with the user hypervisor. Eucalyptus and Nimbus offers no integrated live migration system. This way, if the live migration is sensitive to the application, OpenNebula and OpenStack are more adequate solutions.

Load balancing can be considered on the VM level or on the host level. Load balancing at host level is implemented in OpenStack through live migration, which is the same in CloudStack; Nimbus does the backfilling of partially used physical nodes, allowing also preemptable virtual machines. All the solutions approach VM level load balancing through the establishment of a plug-in architecture. OpenNebula can be coupled with NGINX, Eucalyptus with the ELB of AWS and Nimbus with Nagios plugins. Similarly OpenStack and Cloudstack present flexible plug-in architecture on network component. In resume, at host level, live migration is used to provide load balancing. At VM level, the cloud solutions trust on plugins to provide load balancing. Also, the automatic leasing of resources seems to be a tendency. For example, allowing EC2 or another cloud the ability to pick up excess demand is heavily researched with Nimbus [28].

Fault tolerance mechanisms exists on VM or on storage/database levels. At VM level, fault tolerance is approached under the policies to schedule VM placement or services replication. OpenNebula comes with a match making scheduler - that implements the Rank Scheduling Policy - and the quote management system, that ensure that any user gets a adequate quantity of resources. OpenStack has in-built scheduling algorithms (group scheduling and rescheduling) and newer ones can be implemented by the user. Nimbus has no already implemented fault tolerance system, but it can provide through Globus GridFTP. At storage or database level, fault tolerance is achieved by using replication and synchronization to ensure that a failure occurred at one device will not break the whole system. Eucalyptus has no in-built mechanism for fault tolerance, but can provide it at storage level, through the AWS Elastic Load Balancing.

High availability is approached in all platforms by means of using redundant service instances and load balancing to distribute workloads among those instances. In the case of the replication of service instances, some synchronization technique has to be considered.

Security is provided on different levels by each cloud solution. Centralized in-built user authentication is provided on Nimbus, OpenStack and OpenNebula. The establishment of security policies for users, groups and accounts is possible on CloudStack, Eucalyptus and OpenNebula. On OpenStack is also possible to extend the security of the cloud through the addition of plugins.

Compatibility refers to the capability of the cloud solution to integrate with other tools and cloud solution. In this topic, the Amazon Web Services are a common place on the solutions, thanks to its dominance on commercial public clouds. All open source analyzed solutions, in different levels, present some integration with AWS services, being Amazon EC2 and S3 the most popular.

7.4. Resume. In general lines, Eucalyptus offers a flexible solution for users that want privacy in specific modules while keep managing their clouds with AWS. OpenNebula is for someone interested in the internal technical details of the cloud, but also seems to be a good solution for someone that wants to build up a cloud quickly using just a few machines. OpenStack and CloudStack have the largest developing community, but have opposite approaches, since OpenStack has as modularized architecture, while CloudStack has a monolithic centralized one. Nimbus is easy to deploy and is suitable for inexperienced users willing to have a first contact with cloud platforms or to scientific investigations, although the lack of recent releases and an active community.

8. Conclusions. We have presented the up-to-date architecture of five prominent open source cloud platforms, looking into details of the provided functionalities and their properties. The analyzed platforms are under continuous development. Therefore their documentation and technical reviews are often updated and have to be regularly checked. We argue that there is no best solution to any general case, but there are tools more adapted to specific audiences. The comparison was carried out from the user perspective, considering the properties that a user needs to know when choosing a IaaS cloud solution.

Acknowledgment. Work funded by the European Commission under the Erasmus Mundus Green IT project (Green IT for the benefit of civil society, 3772227-1-2012-ES-ERA MUNDUS-EMA21, Grant Agreement n 2012-2625/001-001-EMA2). And CAPES, Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior - Brasil.

REFERENCES

[1] *Libvirt home page, http:// libvirt.org.*
[2] *OpenNebula 3 Cloud Computing*, 2012.
[3] *Cloudstack live migration, http://cloudstack.apache.org/docs/en-us/apache_cloudstack/4.1.0/html/admin_guide/manual-live-migration.html*, 2014.
[4] *Cloudstack website, https://cloudstack.apache.org/docs/*, 2014.
[5] *Management server load balancing, https://cloudstack.apache.org/docs/en-us/apache_cloudstack/4.0.2/html/installation_guide/management-server-lb.html*, 2014.
[6] *Openstack history, https://www.openstack.org/*, 2014.
[7] *Eucalyptus website, https://www.eucalyptus.com/*, 2015.
[8] *Nimbus website, http://www.nimbusproject.org/*, 2015.
[9] *Opennebula home page, http://opennebula.org*, 2015.
[10] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, *Gridftp: Protocol extensions to ftp for the grid*, Global Grid ForumGFD-RP, 20 (2003).
[11] C. E. Amrani, K. B. Filali, K. B. Ahmed, A. T. Diallo, S. Telolahy, and T. El-Ghazawi, *A compartive study of cloud computing middleware*, in Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), IEEE Computer Society, 2012, pp. 690–693.
[12] A. Barkat, A. D. dos Santos, and T. T. N. Ho, *Open stack and cloud stack: Open source solutions for building public and private clouds*, in Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on, IEEE, 2014, pp. 429–436.
[13] S. A. Baset, *Open source cloud technologies*, in Proceedings of the Third ACM Symposium on Cloud Computing, (2012).
[14] C. V. Blanco, *The opennebula virtual infrastructure engine*, Distributed Systems Architecture Research Group, Universidad Complutense de Madrid, (2009).
[15] G. Chen, H. Jin, D. Zou, B. B. Zhou, W. Qiang, and G. Hu, *Shelp: Automatic self-healing for multiple application instances in a virtual machine environment.*, in CLUSTER, 2010, pp. 97–106.
[16] U. Deshpande and K. Keahey, *Traffic-sensitive live migration of virtual machines.*
[17] P. T. Endo, G. E. Gonçalves, J. Kelner, and D. Sadok, *A survey on open-source cloud computing solutions*, in Brazilian Symposium on Computer Networks and Distributed Systems, 2010.
[18] J. F., *The opennebula engine for on-demand resource provisioning*, in HEPiX Spring.

[19] N. Karkare, *A survey on the live migration of virtual machines.*
[20] R. Milojicic, D. Wolski, *Eucalyptus: Delivering a private cloud*, Computer, (2011).
[21] networkworld website, URL: http://www.networkworld.com/article/2166407/cloud-computing/stack-wars–openstack-v–cloudstack-v–eucalyptus.html, July 2014.
[22] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, *The eucalyptus open-source cloud-computing system*, in Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on, IEEE, 2009, pp. 124–131.
[23] OpenStack, *Introduction to openstack high availability*, in http://docs.openstack.org/high-availability-guide/content/stateless-vs-stateful.html, OpenStack.
[24] H. A. Patel and A. D. Meniya, *A survey on commercial and open source cloud monitoring*, International Journal of Science and Modern Engineering (IJISME), ISSN, (2013), pp. 2319–6386.
[25] S. G. Rakesh Kumar, *Open source infrastructure for cloud computing platform using eucalyptus.*
[26] P. Riteau, *Building dynamic computing infrastructures over distributed clouds*, in Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on, IEEE, 2011, pp. 127–130.
[27] P. N. Sabharwal, *Integrating netscaler with cloudstack*, in Apache CloudStack Cloud Computing, Packt Publishing.
[28] P. Sempolinski and D. Thain, *A comparison and critique of eucalyptus, opennebula and nimbus*, in Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, Ieee, 2010, pp. 417–426.
[29] O. C. Software, *Chapter 1. introduction to openstack*, in http://docs.openstack.org/training-guides/content/module001-ch001-intro-text.html, OpenStack.
[30] D. Souza, R. Matos, J. Araujo, V. Alves, and P. Maciel, *Eucabomber: Experimental evaluation of availability in eucalyptus private clouds*, in Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on, IEEE, 2013, pp. 4080–4085.
[31] T. Viet-DINH, *Cloud data management*, ENS de Cachan, IFSIC, IRISA, KerData Project-Team, (2010), pp. 1–5.
[32] G. von Laszewski, J. Diaz, F. Wang, and G. C. Fox, *Qualitative comparison of multiple cloud frameworks (2012).*
[33] I. Voras, B. Mihaljevic, M. Orlic, M. Pletikosa, M. Zagar, T. Pavic, K. Zimmer, I. Cavrak, V. Paunovic, I. Bosnic, et al., *Evaluating open-source cloud computing solutions*, in MIPRO, 2011 Proceedings of the 34th International Convention, IEEE, 2011, pp. 209–214.
[34] W. Zeng, J. Zhao, and M. Liu, *Several public commercial clouds and open source cloud computing software*, in Computer Science & Education (ICCSE), 2012 7th International Conference on, IEEE, 2012, pp. 1130–1133.
[35] P. T. Endo, G. E. Gonçalves, J. Kelner, and D. Sadok, "A survey on open-source cloud computing solutions," in *Brazilian Symposium on Computer Networks and Distributed Systems*, 2010.
[36] D. Petcu and M. Rak, "Open-source cloudware support for the portability of applications using cloud infrastructure services," in *Cloud Computing*. Springer, 2013, pp. 323–341.
[37] T. Cordeiro, D. Damalio, N. Pereira, P. Endo, A. Palhares, G. Gonçalves, D. Sadok, J. Kelner, B. Melander, V. Souza *et al.*, "Open source cloud computing platforms," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*. IEEE, 2010, pp. 366–371.
[38] I. Voras, B. Mihaljević, and M. Orlić, "Criteria for evaluation of open source cloud computing solutions," in *Information Technology Interfaces (ITI), Proceedings of the ITI 2011 33rd International Conference on*. IEEE, 2011, pp. 137–142.
[39] A. Paradowski, L. Liu, and B. Yuan, "Benchmarking the performance of openstack and cloudstack," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*. IEEE, 2014, pp. 405–412.
[40] A. A. A. M. Mumtaz M.Ali AL-Mukhtar, "Performance evaluation of private clouds eucalyptus versus cloudstack."
[41] C. Li, J. Xie, and X. Zhang, "Performance evaluation based on open source cloud platforms for high performance computing," in *Intelligent Networks and Intelligent Systems (ICINIS), 2013 6th International Conference on*. IEEE, 2013, pp. 90–94.

Edited by: Dana Petcu
Received: May 14, 2015
Accepted: Jun 24, 2015

DOI 10.12694/scpe.v16i2.1090
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 205–217. http://www.scpe.org

PARALLEL WATERMARKING OF IMAGES IN THE FREQUENCY DOMAIN*

DOROTHY BOLLMAN, ALCIBIADES BUSTILLO, EINSTEIN MORALES†

Abstract. While the internet has made it possible for the consumer to easily obtain images, audio, video, etc. in digital form, it has also made it easier to illegally obtain copyrighted material. Digital watermarking is a partial solution to this problem. Embedding a watermark in a legal version of material can help the copyright owner to identify who has an illegal copy. Because of the ever increasing enormity of the flow of information, it becomes necessary to watermark files in the least amount of time possible. For this reason it is natural to turn to parallel computing. In this work we compare the performance of three different implementations on a cluster of SMPs, in OpenMP, MPI, and CUDA, of a simple algorithm for watermarking digital images. Our experiments show that CUDA with one gpu is almost 300 times faster than the sequential version and many times faster than OpenMP and MPI using 1 up to 8 nodes.

Key words: parallel computing, digital watermarking, discrete cosine transform, frequency domain, OpenMP, MPI, CUDA.

AMS subject classifications. 68W10, 94A60, 68P25

1. Introduction. Watermarking is the process of embedding data into multimedia, including text, still images, video, or audio, that is typically used in order to show ownership. A watermark can be either perceptionally visible or invisible to the human eye. Visible watermarks are used to protect copyright or to simply identify a source of material such as a library or organization. A visible watermark identifies the owner of material, but does not necessarily prevent other uses. On the other hand, invisible watermarks are usually used in order to detect fraudulent use of material. For example, a seller might want to identify a person who has used his/her material without paying royalties or the government might want to detect the identity of a person who released classified material.

In order for a watermark to be useful, it must be either detectable or extractable by the owner. It must also be "robust" or resistant to attacks, either intentional or non-intentional. That is, the watermark must remain intact after attacks.

Although many different techniques for embedding watermarks in digital images have appeared in the literature for at least the last twenty years, only a few have considered the possibility of applying parallel computing and those that do consider only the use of GPUs. In this paper we give an embarrassingly parallel algorithm for a certain family of watermarking algorithms in the frequency domain and we compare performance of sequential, OpenMP, MPI, and CUDA implementations of a simple representative of this family, with particular emphasis on OpenMP and MPI.

In the following section, we briefly review several digital image watermarking algorithms that have been considered in the literature. In Sect. 3. we define the discrete cosine transform ("DCT") and describe the symmetries that we take advantage of in our implementations. In Sect. 4. we describe and compare our implementations of a watermarking procedure in OpenMP, MPI, and CUDA. In Sect. 5. we discuss experimental results of the three implementations. In Sect. 6. we make some concluding remarks.

2. Digital Image Watermarking. Digital image watermarking can be done either in the spatial domain or the frequency domain (or perhaps, as in [5], in both). Spatial domain techniques involve direct manipulation of the pixel values. For example, colors of certain pixels could be changed. Another very simple spatial domain technique is the "least significant bit" method in which a given number of least significant bits of the host image are replaced by the most significant bits of the watermark image. A frequency domain method consists of embedding the watermark in the "frequency domain", i.e., in the functional image of a discrete transform such as a Fourier, cosine, or wavelet transform.

Here we are interested in frequency domain methods and we briefly mention only a representative sampling of some of the work that has been done in this area. Although there are several works in which grid (GPU)

*This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

†Department of Mathematical Sciences, University of Puerto Rico at Mayagüez (dorothy.bollman, alcibiades.bustillo, einstein.morales)@upr.edu

computing has been applied to image watermarking, we know of no previous work involving the application of OpenMP and MPI.

A frequency domain method typically consists of three stages: (1) convert the host image I from the spatial domain to the frequency domain, i.e., compute the discrete transform $\mathcal{T}$, such as the Fourier, cosine, or wavelet transform, of I; (2) apply an embedding algorithm E to $\mathcal{T}(I)$ and $\mathcal{S}(W)$, where $\mathcal{S}$ is some function defined on the watermark W, to obtain a new image array $E = E(\mathcal{T}(I), \mathcal{S}(W))$; (3) Convert E back to the spatial domain to obtain the watermarked image E'. An extraction process consists of a procedure X which takes the watermarked image E' and possibly the original image I and produces the original watermark $W = X(E', I)$.

The DCT and IDCT (inverse DCT) of images used in frequency domain methods are applied either to blocks of the image or to the complete image. Shieh *et al* [10] develop a method in which a genetic algorithm is used to insert 8×8 blocks of the binary watermark into corresponding blocks of the DCT of the host image. The watermarked image then consists of the concatenation of the IDCT of each resulting block. Using this algorithm, the watermark can be extracted without the need for the original host image.

Garcia-Cano *et al* [2] implement the Shieh algorithm on a GPU and compare performance results with those of a sequential version. Other works in which GPUs have been used in watermarking are, for example, those of Lin, Zhoa, and Yang [6] and Vihari and Mishra [16]. Lin *et al* extract features from the low and middle frequency domain of the DCT and embed them in the high frequency domain. Vihari and Mishra use Huffman coding to encode copyright data that is then embedded using the "Modified Auxiliary Carry Watermarking" method.

Cox *et al* [1] develop an invisible watermark consisting of real numbers $x_1, x_2, \cdots, x_n$ that are chosen according to a normal distribution with mean 0 and variance 1. The DCT of the host image, as a single block, is computed and the most perceptually significant components, determined by the largest DCT coefficients, are replaced by $v_i(1 + \alpha x_i)$ where v_i is a frequency component of the host image and α is a scalar factor. The watermarked image then consists of the inverse DCT applied to this result.

A typical frequency domain procedure uses the DCT and partitions the image I and a logo watermark image W into 8×8 blocks of pixels The DCT of each block I_{ij} of the host image I is replaced $\alpha_{ij} I_{ij} + \beta_{ij} W_{ij}$ where $W_{ij}(n)$ is the DCT of the corresponding block of the watermark image and where α_{ij} and β_{ij} are values that are chosen in accordance with properties of block I_{ij}. The watermarked image $I(W)$ then results from applying the inverse DCT to each of the resulting blocks and concatenating the resulting blocks. In symbols

$$I(W) = \bigcup_{ij} IDCT(\alpha_{ij} DCT(I_{ij}) + \beta_{ij} DCT(W_{ij}))$$

Kankanhali *et al* [4] choose α_{ij} and β_{ij} according to perceptual methods developed by Tao and Dickinson [15]. Mohanty *et al* [8] claim to improve this latter procedure by taking texture into consideration. We call this type of watermarking procedure, an "$\alpha - \beta$" method. In an $\alpha - \beta$ method each pair of blocks (I_{ij}, W_{ij}) can be processed independently, i.e., such a method is "embarrassingly parallel".

Frequency domain watermarking is generally regarded as being more robust than spatial domain methods, mainly because of its resistance to lossy compression attacks. Indeed, the steps involved in frequency domain watermarking are very similar to those of image compression such as JPEG compression. Typically, these steps consist of the following: (1) partition the image into 8×8 blocks of pixels; (2) apply the two dimensional DCT to each such block; (3) apply "quantization" to each block of resulting DCT coefficients; (4) apply the inverse DCT to each of the blocks; (5) apply entropy coding to the quantized data. Quantization is a process of reducing the number of possible values of a quantity and entropy coding is a method for representing the quantized data in a compact form.

3. The Discrete Cosine Transform. The one-dimensional discrete cosine transform is a linear function $F : \mathbb{R}^N \to \mathbb{R}^N$ defined by

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos\left(\frac{\pi(2x+1)u}{2N}\right), u = 0, \ldots, N-1 \tag{3.1}$$

The inverse transform exists and is defined by

$$f(x) = \sum_{x=0}^{N-1} \alpha(u)C(u)\cos\left(\frac{\pi(2x+1)u}{2N}\right), x = 0, \ldots, N-1 \tag{3.2}$$

where

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & x = 0 \\ \sqrt{\frac{2}{N}} & x \neq 0 \end{cases} \tag{3.3}$$

The one-dimensional DCT is a linear function and can thus be represented as a matrix, i.e.,

$$[C_{ij}] = \left[\alpha(i)\cos\left(\frac{(2j+1)\pi i}{2N}\right)\right] \qquad i, j = 0, 1, \cdots, N-1$$

and similarly for the DCT inverse. The two-dimensional DCT is a function $F : \mathbb{R}^{N^2} \to \mathbb{R}^{N^2}$ which when applied to a matrix can be computed by first computing the one-dimensional DCT of the rows and then using the result to compute the one-dimensional DCT of the columns.

Since an $\alpha - \beta$ method applies the DCT to 8×8 blocks of images multiple times, it is of interest to minimize the number of operations in its computation. For this we use an idea of Obukhov and Kharlamov [9] which is described in the following.

The matrix form of the one-dimensional DCT exhibits various symmetries that can be taken advantage of. For $N = 8$ the representation is as follows:

$$F = \frac{1}{\sqrt{8}} \left[\begin{array}{cccc:cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ a & c & d & f & -f & -d & -c & -a \\ b & e & -e & -b & -b & -e & e & b \\ c & -f & -a & -a & d & a & f & -c \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ d & -a & f & c & -c & -f & a & -d \\ e & -b & b & -e & -e & b & -b & e \\ f & -d & c & -a & a & -c & d & -f \end{array}\right]$$

$$a = \sqrt{2}\cos\left(\frac{\pi}{16}\right)$$

$$b = \sqrt{2}\cos\left(\frac{\pi}{8}\right)$$

$$c = \sqrt{2}\cos\left(\frac{3\pi}{16}\right)$$

$$d = \sqrt{2}\cos\left(\frac{5\pi}{16}\right)$$

$$e = \sqrt{2}\cos\left(\frac{3\pi}{8}\right)$$

$$f = \sqrt{2}\cos\left(\frac{7\pi}{16}\right)$$

Separating even and odd numbered rows we have

$$\begin{bmatrix} Y(0) \\ Y(2) \\ Y(4) \\ Y(6) \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ b & e & -e & -b \\ 1 & -1 & -1 & 1 \\ e & -b & b & -e \end{bmatrix} \begin{bmatrix} X(0)+X(7) \\ X(1)+X(6) \\ X(2)+X(5) \\ X(3)+X(4) \end{bmatrix}$$

FIG. 4.1. *(a) Image (Lena), (b) Watermark (Barbara), (c) Watermarked image*

$$\begin{bmatrix} Y(1) \\ Y(3) \\ Y(5) \\ Y(7) \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} a & -c & d & -f \\ c & f & -a & d \\ d & a & f & -c \\ f & d & c & a \end{bmatrix} \begin{bmatrix} X(0)-X(7) \\ X(2)-X(1) \\ X(4)-X(5) \\ X(5)-X(3) \end{bmatrix}$$

Thus, the one-dimensional DCT applied to a vector of length 8 can be computed using only 28 multiplications and 56 additions. The usual product of an 8×8 matrix times a vector of length 8 requires 64 multiplications and 56 additions.

4. Parallel Implementations of an $\alpha - \beta$ Frequency Domain Watermarking Algorithm. Given grayscale images of a host I and a watermark W, each of the same size, which we assume to be a power of 2, an $\alpha - \beta$ algorithm embeds W in I by (1) partitioning both I and W into blocks of 8×8 pixels and computing the DCT of each of the corresponding blocks, I_{ij} and W_{ij}; (2) computing $\alpha_{ij} = \alpha(I_{ij}, W_{ij})$ and $\beta_{ij} = \beta(I_{ij}, W_{ij})$; (3) replacing each value $DCT(I_{ij})$ in the host image by $\alpha_{ij}DCT(I_{ij}) + \beta_{ij}DCT(W_{ij})$; (4) computing the inverse DCT of each of the blocks $\alpha_{ij}DCT(I_{ij}) + \beta_{ij}DCT(W_{ij})$ and concatenating the results.

Following is an example where $\alpha = \alpha_{ij} = 0.9$ and $\beta = \beta_{ij} = 0.14$ for all i, j. The degree of visibility of the watermark is determined by the values of α and β.

Let us begin with a sequential version of this algorithm for an image of size $dim \times dim$ (Algorithm 1).

Algorithm 1 Sequential

```
1: procedure WATERMARKING(I, W, dim, α, β)
2:     m = dim/8
3:     for i = 0 : m − 1 do
4:         for j = 0 : m − 1 do
5:             X_ij = DCT(I_ij)
6:             Y_ij = DCT(W_ij)
7:             Compute α = α(I_ij, W_ij) and β = β(I_ij, W_ij)
8:             Z_ij = αX_ij + βY_ij
9:             Z_ij = IDCT(Z_ij)
       return Z
```

Each pair (I_{ij}, W_{ij}) of image blocks in the above algorithm can be processed independently of the others and hence in parallel and an efficient implementation consists of determining how the available resources can be used to most effectively achieve this. We shall see how this can be done in OpenMP, MPI, and CUDA.

4.1. OpenMP. OpenMP ("Open Multi-Processing") is a shared memory programming model in which the programmer can insert compiler directives or "pragmas" into ordinary sequential C, C++, or FORTRAN programs in order to partition tasks into parallel threads, the smallest unit of processing that can be scheduled by an operating system. The most common and easiest way to parallelize code in OpenMP is by parallelizing **for** loops.

In the above procedure WATERMARKING, each of the (i, j) iterations is independent of the others and so ideally, it would be convenient to use just one parallel **for** loop. However, because of the double indexing, we must use two nested **for** loops and in OpenMP it is possible to parallelize only the exterior **for** loop. However, recent versions of OpenMP allow one to parallelize multiple loops in a nest without introducing nested parallelism by making use of the collapse pragma. Thus, we have Algorithm 2.

The number of threads used is specified at run time.

Algorithm 2 OpenMP-Version

procedure WATERMARKING(I, W, dim, α, β)
  $m = \frac{dim}{8}$
  #**pragma omp parallel for collapse**(2)
  for $i = 0 : m - 1$ **do**
    for $j = 0 : m - 1$ **do**
      $X_{ij} = DCT(I_{ij})$
      $Y_{ij} = DCT(W_{ij})$
      Compute $\alpha = \alpha(I_{ij}, W_{ij})$ and $\beta = \beta(I_{ij}, W_{ij})$
      $Z_{ij} = \alpha X_{ij} + \beta Y_{ij}$
      $Z_{ij} = IDCT(Z_{ij})$
  return Z

4.2. MPI. MPI ("Message Passing Interface"), originally designed for distributed memory architectures, is a library of functions that can be used in conjunction with C, C++, or FORTRAN, as well as other languages, in order to effect communications between processors. In the MPI implementation of our watermarking algorithm we partition the grid of 8×8 blocks of both the host image as well as the watermark into n equal size strips of rows of blocks, where n is the number of processors. The processor, say p_0, that initially contains both images, I and W, sends to each of the n processors a pair of strips (one from the host image and one from the watermark). Each processor then processes its share of the two images as in the sequential version and then sends its result to p_0 which assembles the result into the complete watermarked image (Algorithm 3).

4.3. CUDA. CUDA ("Compute Unified Device Architecture"), created by Nvidia, is a parallel computing platform and programming model in which GPUs (Graphics Programming Units) can be accessed by programmers for general purpose computing through the use of CUDA-accelerated libraries, compiler directives, and extensions of C, C++, and FORTRAN, as well as other languages. A CUDA application can involve hundreds of cores and thousands of parallel threads. Parallel portions of an application can be defined as functions that are executed on the GPU. Such functions are called kernels. Threads are grouped into blocks of up to 512 threads, which in turn are organized into grids. All threads in a grid execute the same kernel. A grid can be one-, two-, or three-dimensional.

With CUDA we achieve our original goal of assigning a thread to each iteration of the nested **for** loop of the sequential version of our algorithm presented in Sect. 4. For this we make use of dct8×8_kernel2 of the Nvidia code [9] for computing the DCT and IDCT in 8×8 blocks, in which the grid constructed is 3-dimensional with 64 threads per block (Algorithm 4).

Algorithm 3 MPI-Version

procedure WATERMARKING(I, W, dim, α, β)
  n = number of processors
  $m1 = \frac{dim}{8}$
  $m2 = \frac{m1}{n}$
  send a section $I^{(p)}$ of $m2$ rows of 8×8 blocks of I to each processor p
  send a section $W^{(p)}$ of $m2$ rows of 8×8 blocks of W to each processor p
  for $i = 0 : m2 - 1$ **do**
    for $j = 0 : m1 - 1$ **do**
      $X_{ij}^{(p)} = DCT(I_{ij}^{(p)})$
      $Y_{ij}^{(p)} = DCT(W_{ij}^{(p)})$
      Compute $\alpha = \alpha(I_{ij}, W_{ij})$ and $\beta = \beta(I_{ij}, W_{ij})$
      $Z_{ij}^{(p)} = \alpha X_{ij}^{(p)} + \beta Y_{ij}^{(p)}$
      $Z_{ij}^{(p)} = IDCT(Z_{ij}^{(p)})$
  receive each $Z^{(p)}$ in Z

Algorithm 4 CUDA-Version

procedure WATERMARKING(I, W)
  for each thread-block in *block-grid* **do** in parallel
    $X = kernel\ DCT(I_{thread})$
    $Y = kernelDCT(W_{thread})$
    $\alpha = kernel\ \alpha(I_{ij}, W_{ij})$
    $\beta = kernel\ \beta(I_{ij}, W_{ij})$
    $Z_{thread} = kernel\ linearcomb(\alpha, X, \beta, Y)$
    $Z_{thread} = kernel\ IDCT(Z_{thread})$
  return Z

5. Experimental Results. For our actual implementations we chose a simple representative from the $\alpha - \beta$ family of algorithms in which the values of α and β are the same for all blocks (as for example in cf. Fig. 4.1), thus replacing the computation step for α and β in the above algorithms by inputs. Experiments were conducted on the Stampede supercomputer [11], located at the Texas Advanced Computer Center (TACC) and sponsored by the Extreme Science and Engineering Environment (XSEDE) with funding by the National Science Foundation. The Stampede system is a 10 PFLOPS (PF) Dell Linux Cluster consisting of 6,400 + Dell PowerEdge server nodes, each one of which has 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor (MICZ Architecture). For the OpenMP and MPI programs the authors used the 16 compute node "development queue", each node of which consists of 16 cores with 32 GB of shared memory. For the CUDA experiments, the authors used the "gpudev queue" consisting of 4 compute nodes each one of which is equipped with a NVIDIA K20GPU with 8GB of GDDR5 memory.

We tested our OpenMP, MPI and CUDA programs with greyscale images of 512, 1024, 2048, 4096, and 8192 square pixels. For OpenMP and MPI we used a maximum of all 16 cores per node. For CUDA we used one compute node with a GPU.

It will be noted that the behavior for 16 cores can sometimes be erratic, especially for small images. This is because some of a node's resources are necessarily dedicated to other instrinsic processes of the system.

5.1. OpenMP Results. The times for OpenMP are depicted in Fig. 5.1. Speedups for OpenMP are given in cf. Fig. 5.2.

As can be seen from Table 5.1 the OpenMP implementation has almost linear speedup for 2 up to 12 threads.

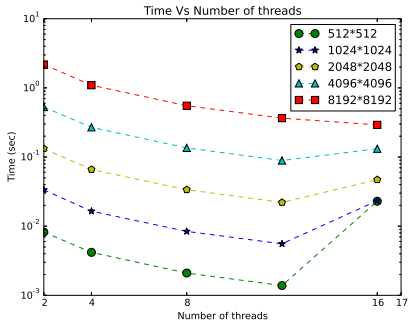


FIG. 5.1. *OpenMP times*

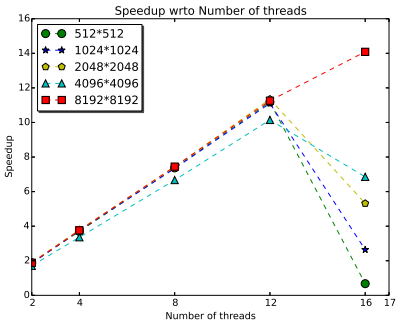


FIG. 5.2. *OpenMP Speedup*

TABLE 5.1
Speedup

threads	Speedup				
	512×512	1024×1024	2048×2048	4096×4096	8192×8192
2	1.884852	1.843737	1.893246	1.708771	1.885878
4	3.699033	3.747434	3.790393	3.357762	3.7578
8	7.372234	7.338334	7.427529	6.668273	7.44652
12	11.18421	11.10579	11.34707	10.14434	11.25846
16	0.674747	2.642157	5.317085	6.853785	14.08695

5.2. MPI Results. Times and speedups with respect to MPI tasks on 1, 2, 4, and 8 nodes are given in cf. Fig. 5.3 through 5.10.

Using all 16 cores on each node, speedups with respect to the number of nodes are given in cf. Fig. 5.11 and 5.12.

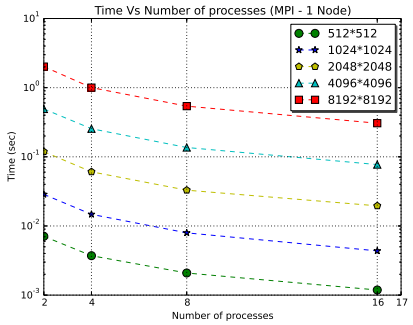


FIG. 5.3. *MPI 1 Node*

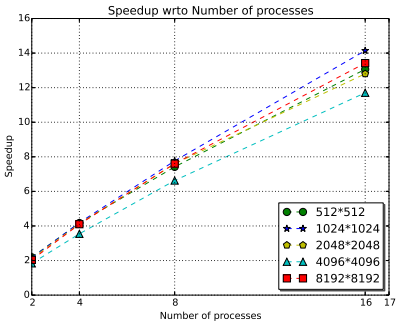


FIG. 5.4. *Speedup 1 Node*

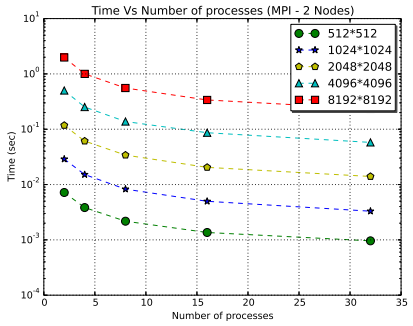


FIG. 5.5. *MPI 2 Nodes*

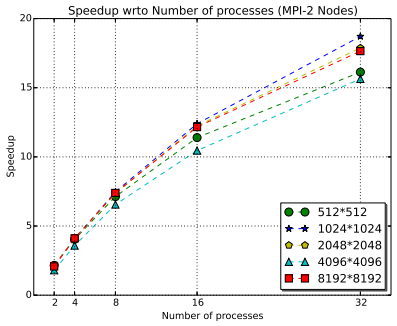


FIG. 5.6. *Speedup 2 Nodes*

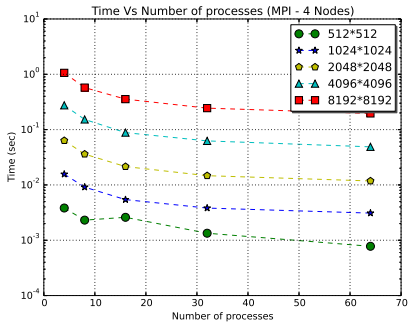


FIG. 5.7. *MPI 4 Nodes*

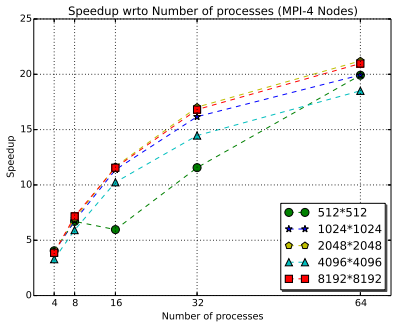


FIG. 5.8. *Speedup 4 Nodes*

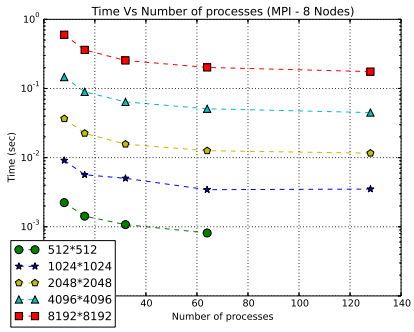


FIG. 5.9. *MPI 8 Nodes*

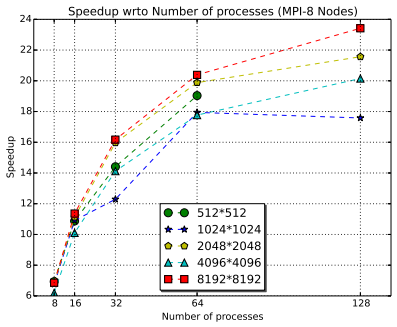


FIG. 5.10. *Speedup 8 Nodes*

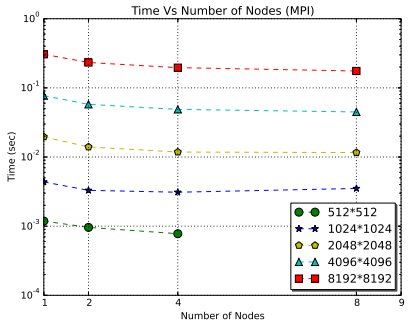


FIG. 5.11. *MPI times*

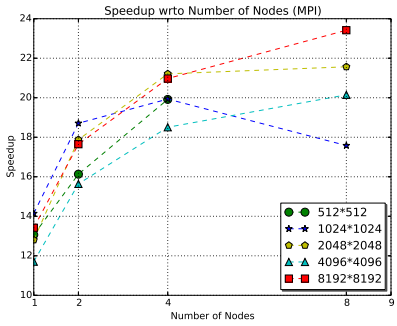


FIG. 5.12. *MPI Speedups*

As can be seen, the MPI implementation exhibits poor scalability with respect to the number of nodes (i.e., sequential time divided by the time for n nodes using all 16 cores), even when using all 16 cores. However, it is not true that the best time for a given number of nodes used corresponds to 16 processes (cores). But even using the best times, speedups with respect to the number of nodes is almost flat, as shown in Table 5.2 for an image of size 1024×1024.

TABLE 5.2
Speedup (1024×1024)

	Speedup (1024×1024)
1 node/ 16 processes	14.1429884
2 nodes/ 32 processes	18.7097302
4 nodes/ 64 processes	19.9331964
8 nodes/ 128 processes	17.5811589

On the other hand, when we compute speedups with respect to the number of processes (i.e., for a fixed number of nodes, the sequential time divided by the time for the number of processes), the best scalability occurs when we use just one node. In fact in this case as shown in the Table 5.3 we have superlinear speedup for 2 and 4 processes, except for size 4096×4096.

TABLE 5.3
Speedup (1 Node)

	Speedup				
processes	512×512	1024×1024	2048×2048	4096×4096	8192×8192
2	2.18684251	2.155103	2.110129	1.835898	2.037115
4	4.16952721	4.189678	4.1226	3.545565	4.100949
8	7.41785612	7.7622	7.600967	6.62734	7.605353
16	13.0625843	14.14299	12.80332	11.70284	13.42

5.3. OpenMP vs MPI. In cf. Fig. 5.13 through 5.17 we compare the performance of OpenMP for 2, 4, 8 and 16 threads with MPI on one node for 2, 4, 8, and 16 tasks, respectively and we see that the performance of the two are nearly the same up to 8 threads/tasks. For 16 threads/tasks, MPI wins for sizes up to 4096×4096, but the gap narrows for increasing image size and OpenMP wins by a very narrow margin for image size 8192×8192.

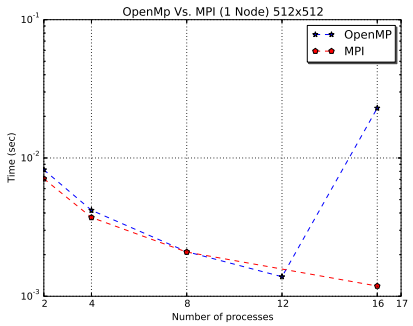


FIG. 5.13. *MPI vs. OpenMP (512×512)*

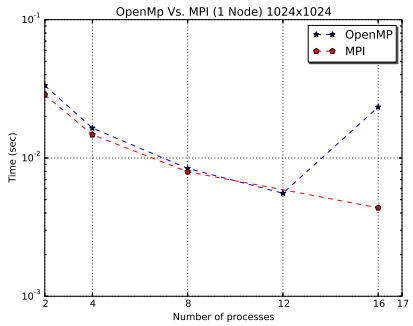


FIG. 5.14. *MPI vs. OpenMP (1024×1024)*

5.4. Hybrid. We developed a hybrid OpenMP MPI program in the standard way, by starting with the MPI version and distributing strips of 8×8 blocks of pixels among the nodes just as we did in the MPI version except

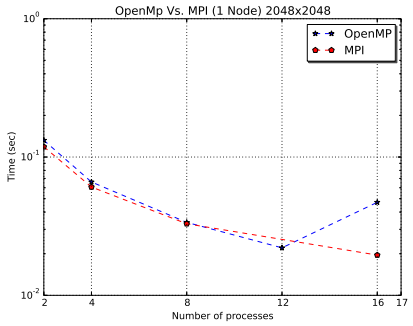


FIG. 5.15. *MPI vs. OpenMP (2048×2048)*

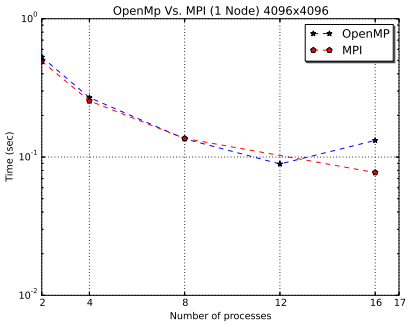


FIG. 5.16. *MPI vs. OpenMP (4096×4096)*

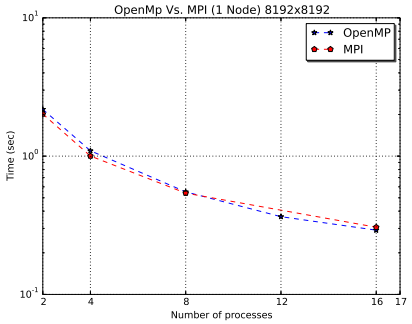


FIG. 5.17. *MPI vs. OpenMP (8192×8192)*

.

that now each node does its work in parallel using OpenMP as in the OpenMP-only version. Unfortunately, this version under performed both the OpenMP and MPI versions. This is really not surprising. Indeed, there are well known cases (see *e.g.*, [3]) where hybrid is slower than OpenMP and MPI. Hybrid improves performance by reducing communications between nodes and increasing opportunites for parallelism and neither of these opportunities exist in the MPI version of our algorithm. Furthermore, it turns out that the scatter and gather operations are many times slower in the hybrid version than in the MPI version when applied to the same sets of data.

5.5. CUDA Results. For the CUDA code, we used the optimized code of kernel2 described in [9] for computing the 8×8 DCT on a NVIDIA GeForce 6800. We added kernels for embedding the watermark. Further experiments are needed in order to determine if performance on the NVDIA K20 can be improved even further by manipulating block sizes. Comparisons between the four implementations are depicted in cf. Fig. 5.18. Our CUDA implementation on just one GPU was many times faster than both OpenMP and MPI on any number of nodes up to 8.

Table 5.4 gives the times for 4096×4096 images for the four different versions on just one node, where OpenMP uses 16 threads and MPI uses 16 processes.

Thus, on just one node, CUDA is 287 times faster than the sequential version, 42 times faster than OpenMP, and 24 times faster than MPI. The fastest time, .0448525 sec, for MPI occurred on 8 nodes using 128 processes. Thus CUDA on just one node is 14 times faster than MPI on 8 nodes.

5.6. Watermark Quality. There are several metrics that are commonly used to measure the the quality of watermark algorithms. The Peak Signal to Noise Ratio (PSNR) evaluates image degradation or reconstruction fidelity. It is defined for two images I and Z of size $M \times N$ as:

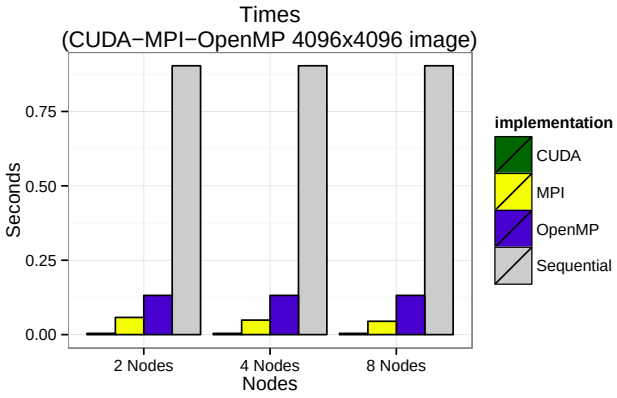


FIG. 5.18. *Cuda-MPI-OpenMP 4096×4096 image*

TABLE 5.4
Times for 4096×4096 images for the four different versions on one node (OpenMP uses 16 threads, MPI uses 16 processes)

Version	**Time (Seconds)**
Sequential	0.903651
OpenMP	0.131847
MPI	0.0772164
CUDA	0.0031519

$$PSNR(I,Z) = 20\log_{10}\frac{\max(I)}{\sqrt{MSE(I,Z)}} \tag{5.1}$$

where I is the original image and Z is the reconstructed image, $\max(I)$ is the maximum pixel value in I, and MSE is the mean square error between I and Z.

$$MSE(I,Z) = \frac{1}{M}\frac{1}{N}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\|I(i,j) - Z(i,j)\|^2 \tag{5.2}$$

In image reconstruction the $PSNR$ values vary between $[30, 50]$. A $PSNR$ value of 50 or more indicates that the images are almost identical.

Robustness represents the resistance of a watermark against attacks, such as compression, scaling, cropping, rotation, smoothing, etc. The Normalized Correlation (NC) measures the correlation between the original watermark and the extracted watermark after attack. It is defined by

$$NC = \frac{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}[W(i,j)W'(i,j)]}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}[W(i,j)]^2} \tag{5.3}$$

A value of NC equal to 1 indicates that the original and extracted watermark are exactly the same.

The quality of an $\alpha - \beta$ algorithm depends on the method for computing α and β. Our interest in this work has been focused on the computational aspects of $\alpha - \beta$ algorithms and we chose for our experiments the simplest representative, in which α and β are constant over all DCT blocks. Although we would not expect the quality for this representative to compare the most favorably with other members of the $\alpha - \beta$ family, it is

nevertheless of interest to determine values of PSNR and NC for this case. To this end, we used the software StirMark to determine the robustness for our algorithm using the values $\alpha = 0.9$ and $\beta = 0.2$.

Table 5.5 shows the results of applying the software StirMark [12] [13] to the watermarked image in Fig. 4.1 (c) of size 1024×1024 produced by our algorithm and subjected to the indicated attacks.

TABLE 5.5
The results of applying the software StirMark to the watermarked image

Image	**PSNR**	**NC**
Watermarked image	36.72	0.97
Compressed image	34.20	1.45
Smooth image	33.86	1.44
Rotated+scaling image	33.53	1.04
Randomly distorted image	30.21	1.09

The values of PSNR compare well with other watermarked image (e.g., [14]), in spite of the simplicity of our algorithm. On the other hand, we would expect values of NC closer to one for other more specialized $\alpha - \beta$ algorithms

5.7. Conclusions. We have shown how a commonly used family of watermarking algorithms in the frequency domain can be implemented in parallel and we compared the OpenMP, MPI, CUDA implementations of a simple representative of the $\alpha - \beta$ family of algorithms. We found that the CUDA implementation on just one GPU runs many times faster than the OpenMP version and the MPI version even when executed on 8 nodes. Thus the fastest implementation runs on just one node equipped with one GPU. For one node without a GPU, there are only slight differences between OpenMP and MPI, with OpenMP winning only for large images and MPI exhibiting a more regular speedup.

We observed that in spite of its simplicity, the chosen representative is resistant to attacks such as compression, distortion, rotation plus scaling, and smoothing.

The same method developed here for implementing this simple case could be applied to any $\alpha - \beta$ algorithm. Of course, the running times of all implementations will vary depending on the complexity of the calculations of the α_{ij} and β_{ij}.

REFERENCES

[1] I.J. Cox *et al* , *A secure robust watermarking for multimedia*, Proc. of First International Workshop on Information Hiding, Lecture Notes in Comp. Sc., Springer-Verlag, 1174 (1996), pp. 185–206.
[2] C. Garcia-Cano, B. Rabil, R. Sabourin, *A parallel watermarking application on a GPU*, Congreso Internacional de Investigacion en Nuevas Tecnologias Informáticas-CIINTI 2012, http://ciinti.info.memorias.
[3] Y. He, C. Ding, *MPI and OpenMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition*, SCPE, 5 (2002), No. 2.
[4] M.S. Kankanhalli, *Adaptive visible watermarking of images*, Proc. ICMCS99, Florence, Italy, June (1999).
[5] Lenarczyk, Piotr and Piotrowski, Zbigniew, *Parallel blind digital image watermarking in spatial and frequency domains*, Springer US, Telecommunication Systems, 54 (2013), pp. 287-303, doi :10.1007/s11235-013-9734-x
[6] C. Lin, L. Zhao, and J. Yang, *A high performance image authentiction algorithm on GPU with CUDA*, I.J. Intelligent Systems and Applications, 2011, 2, pp. 55-59.
[7] S. Mohanty, *Digital watermarking: a tutorial review*, http://informatika.stei.itb.ac.id/ rinaldi.munir/Kriptografi/WMSurvey1999Mohanty.pdf.
[8] S. Mohanty, K. Ramakrishnan, M. Kankanhali, *A DCT domain visible watermarking technique for images*, International Conference on Multimedia Computing and Systems/International Conference on Multimedia and Expo-ICME(I Multimedia and Expo 2000, ICME(CMCS), pp. 1029–1032.
[9] A. Obukhov and A. Kharlamov, *Discrete cosine transform for 8×8 blocks with CUDA*, https://svn.inf.ufsc.br/luis.custodio/TCC-Dantas/CUDA/.../dct8x8.pdf
[10] C. Shieh, H. Huang, F. Wand, J. Pan, *Genetic Watermarking based on Transform-domain Techniques*, Pattern Recognition 37 (2004) pp. 555–565.
[11] Stampede User Guide, *https://portal.tacc.utexas.edu/user-guides/stampede*

[12] Petitcolas, FabienA.P. and Anderson, RossJ. and Kuhn, MarkusG. , *Attacks on Copyright Marking Systems*, Springer Berlin Heidelberg, Information Hiding, Lecture Notes in Computer Science, 1525 (1998), pp. 218-238
[13] Petitcolas, FabienA.P., *Watermarking schemes evaluation*, Signal Processing Magazine, IEEE, 17 Sep (2000), pp. 58-64
[14] Singh, A.K.; Dave, M.; Mohan, A., *A novel technique for digital image watermarking in frequency domain*, Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on, vol., no., pp.424,429, 6-8 Dec. 2012 doi: 10.1109/PDGC.2012.6449858
[15] B. Tao and B. Dickinson, *Adaptive watermarking in DCT domain*, Proc IEEE International Conf. on Acoustics, Speech and Signal Processing, ICASSP-97, 4 (1997), pp. 2985–2988.
[16] P. Vihari and M. Mishra, *Image authentication algorithm on GPU*, 2012 International Conference on Communication Systems and Network Technologies, pp. 874-878.

Edited by: Dana Petcu
Received: Apr 10, 2015
Accepted: Jun 24, 2015

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2$_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.