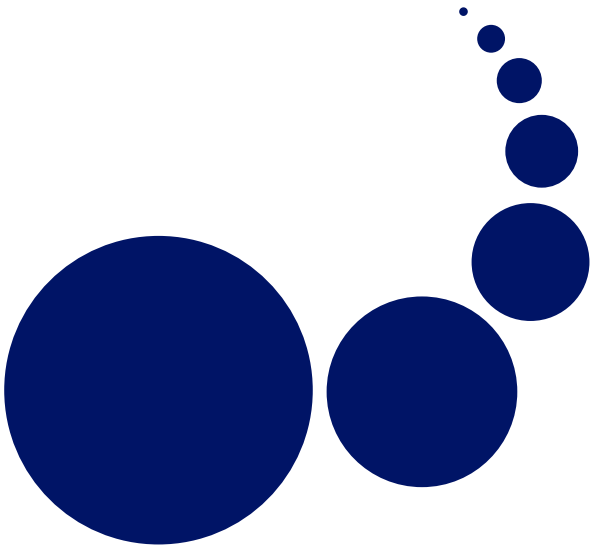
SCALABLE COMPUTING
Practice and Experience

Special Issue: Parallel, Distributed and Network-based Computing: an Application Perspective

Editors: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla



Volume 11, Number 3, September 2010

ISSN 1895-1767

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

Scalable Computing: Practice and Experience

Volume 11, Number 3, September 2010

TABLE OF CONTENTS

Scalable Computing: Practice and Experience
Volume 11, Number 3, p. i. http://www.scpe.org
ISSN 1895-1767

INTRODUCTION TO THE SPECIAL ISSUE: PARALLEL, DISTRIBUTED AND NETWORK-BASED COMPUTING: AN APPLICATION PERSPECTIVE

Parallel, distributed and network-based computing is a continuosly evolving field, driven by progress in microprocessor architecture and interconnection technology, as well as by the needs of computing- and data-intensive applications in science and engineering, and, more recently, in business. This field is currently undergoing a significant change, because of the development of multicore and manycore processors, GPUs, and FPGAs, which are the new building blocks of parallel architectures. At the other end of the parallel and distributed computing scenario, computational grids are far from being a mature infrastructure and are evolving toward cloud computing, to get a higher level of virtualization.

The availability of programming models, algorithms and software tools capable of harnessing the processing power offered by the new technologies is a key issue to make them usable by application developers. This special issue provides a view of the efforts carried out in this direction.

- Barlas introduces an optimization approach for reducing data communication and load imbalance in medical image matching on Grids.
- Binzenhöfer et al. present a distributed and scalable algorithm to monitor a p2p network.
- Cesario and Talia discuss the use of data mining models and services on Grid systems for analysis of large data repositories.
- Danelutto et al. describe a performance model for component-based applications with stream communication semantics running on Grids.
- Danese et al. describe a FPGA-based coprocessor to accelerate double precision floating point operations in high-performance applications.
- Gross and Marquardt introduce a graphical editor providing abstractions from base technology for user-friendly configuration of Ubiquitous Computing environments.

The papers collected here are selected extended versions of papers presented at PDP 2007, the Fifteenth Euromicro Conference on Parallel, Distributed and Network-based Processing, held in Naples, Italy, in February 2007. The conference was organized by the Institute for High-Performance Computing and Networking (ICAR) of the Italian National Research Council (CNR) in collaboration with the Second University of Naples, the University of Naples "Parthenope" and the University of Naples "Federico II."

We thank the editors of Scalable Computing: Practice and Experience for providing us the opportunity of publishing this issue, the authors for their contributions, and the referees for their precious help in selecting good-quality papers.

Pasqua D'Ambra
Institute for High-Performance Computing and Networking (ICAR), CNR
Naples, Italy
pasqua.dambra@cnr.it
Daniela di Serafino
Department of Mathematics, Second University of Naples
Caserta, Italy
daniela.diserafino@unina2.it
Mario Rosario Guarracino
Institute for High-Performance Computing and Networking (ICAR), CNR
Naples, Italy
mario.guarracino@cnr.it
Francesca Perla
Department of Statistics and Mathematics for Economic Research,
University of Naples "Parthenope"
Naples, Italy
francesca.perla@uniparthenope.it

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 221–237. http://www.scpe.org
ISSN 1895-1767

OPTIMIZING IMAGE CONTENT-BASED QUERY APPLICATIONS OVER HIGH LATENCY COMMUNICATION MEDIA, USING SINGLE AND MULTIPLE PORT COMMUNICATIONS

GERASSIMOS BARLAS*

Abstract. One of the earliest applications that explored the power and flexibility of the grid computing paradigm was medical image matching. A typical characteristic of such applications is the large communication overheads due to the bulk of data that have to be transferred to the compute nodes.

In this paper we study the problem of optimizing such applications under a broad model that incorporates not only communication overheads but also the existence of local data caches that could exist as a result of previous queries. We study the cases of both 1- and N-port communication setups. Our analytical approach is not only complimented by a theorem that shows how to arrange the sequence of operations in order to minimize the overall cost, but also yields closed-form solutions to the partitioning problem.

For the case where large load imbalances (due to big differences in cache sizes) prevent the calculation of a closed-form solution, we propose an algorithm for optimizing load redistribution.

The paper is concluded by a simulation study that evaluates the impact of our analytical approach. The simulation, which assumes a homogeneous parallel platform for easy interpretation of the results, compares the characteristics of the 1- and N-port setups.

Key words: parallel image registration, divisible load, high performance

1. Introduction. In the past five years there has been a big drive towards harnessing the power of parallel and distributed systems to offer improved medical services in the domain of 2D and 3D modalities. Content-based queries are at the core of these services, allowing physicians to achieve higher-accuracy diagnoses, conduct epidemiological studies or even acquire better training among other things [1].

In [2], the authors present a high-level overview of the methodologies used for medical image matching. The authors identify two broad types of approaches: *image retrieval* that utilizes similarity metrics to offer suitable candidate images and *image registration* that tries to fit the observed data onto fixed or deformable models. Finally, the authors suggest an integrated system architecture that could combine the advantages of the two approaches. A comprehensive review and classification of current medical image handling systems is published in [3].

Apart from the classification mentioned in [2], image registration techniques are also classified based on whether:

- Image features are used (control-point based) or the whole (or an area of interest) image (global registration).
- Work is done at the spatial or frequency domain.
- Global (rigid) or local (non-rigid) geometrical transformations are used.

The key problem is determining the optimum geometrical transformation. A brute-force approach entails huge computational requirements, leading researchers to either perform the search in several refinement steps [4, 5], or switch to heuristic techniques such as genetic/evolutionary algorithms and simulated annealing [6, 7]. Domain specific techniques have been also suggested [8].

A domain which has been enjoying early success is mammography [9, 1, 10]. Many projects that seek to harness the power of Grids [11] to offer advanced medical services have spawned over the last 8 years. A typical example is the MammoGrid project. Amendolia et.al present an overview of its service architecture design in [1]. On the other side of the Atlantic, the National Digital Mammography Archive Grid is a similar initiative [10]. A P2P system that seeks to address scalability issues that arise with the operation of typical client-server systems has been also proposed in [12].

While the problem of image registration is inherently 'embarrassingly' parallel, the domain has seen little work on performance optimization especially over heterogeneous platforms. In [5] the authors use wavelets to perform global registration in increasing refinement steps that allows them to reduce the search space involved. Zhou et al. also evaluate four parallelization techniques and derive their complexity in big-O notation by

*Department of Computer Science & Engineering, American University of Sharjah, P.O.B. 26666, Sharjah, UAE, gbarlas@aus.edu

implicitly assuming a homogeneous platform. However, they fail to take into account the communication overheads involved and use their analysis to optimize the load partitioning of their strategies.

Ino et al. propose a uniform inter-image 2D partitioning for performing 2D/3D registration, i. e. estimate the spatial location of a 3D volume from its projection on a 2D plane [13]. While Ino et al. discuss other possible distributions, they do not use an appropriate model that would allow for optimization. Subsequently, in [14] the authors compare very favourably a GPGPU approach with their parallel implementation on 2D/3D registration.

De Falco et al. have employed a differential evolution mechanism for estimating the parameters of an affine transformation for global registration [6]. The load distribution is performed on the population level, while at regular intervals, individuals are exchanged between neighboring nodes on the torus architecture used.

One of the early systems is the one described in [9]. Montagnat et.al use an array of high run-time cost, pixel-based, image retrieval algorithms to answer image similarity queries. As described in [15], the homogeneous system that is used to run the queries employs equal size partitioning, e.g. the M images that need to be compared against a new one, are split into k jobs of size $\frac{M}{k}$. In [15] the authors develop empirical cost models for each of the similarity metrics used to answer a content-based query. These are complemented by a study of the scheduling and data replication costs that are incurred upon submitting a job to a Grid platform.

While the models shown in [15] capture much of the inner workings of the algorithms used, they are not the most suitable for developing a strategy or criteria for optimizing the execution of content-based queries. Instead, they focus on estimating the optimum number of jobs to spawn, given the high associated cost of task/resource scheduling on Grids.

A particular problem in deriving an analytical partitioning solution is that upon performing a sequence of queries, the system is in a state where local image caches can reduce the communication cost. This is of course true as long as they refer to images of the same modality and type of content. To our knowledge, this paper is the first attempt to treat this problem in an analytical fashion that incorporates all the aforementioned system/problem parameters.

Our analytical approach belongs to the domain of Divisible Load Theory [16], which since its inception in the late 80s, has been successfully employed in a multitude of problems [17]. In [17] the problem of optimally partitioning and scheduling operations for two classes of problems identified as *query processing* and *image processing* respectively, has been studied. The problem characterizations were based on the communication characteristics and more specifically, the relation between the communication cost and the assigned load. This paper fills a gap left by that work by proposing a model and an analytical solution to *image-query processing* applications.

The contribution of our work is that for the first time a fully analytical model is employed to devise an optimizing strategy for the total execution time, given communication costs and the state (and not just the capabilities) of the parallel platform. Our simulation study shows that the benefits of the proposed framework are significant, in both a single-shot and a series of queries scenarios. Also, by isolating the specifics of the matching algorithms, our proposed solution is more adept to easy implementation and deployment, given the few system parameters that need to be known/estimated.

The organization of the paper goes as follows: in section 2 the cost model used in our analysis is introduced and explained within a broader context. Section 3.1 contains a study of the two-node scenario that cultivates to Theorem 3.1 for the optimum sequence of operations. The closed-form solutions to the partitioning problem for N nodes in 1-port configuration, is given in 3.2, while the N-port problem is solved in Section 4. An algorithm for managing the cache size of the compute nodes towards minimizing the execution time, is given in section 5. Finally, the simulation study in Section 6 highlights the benefits and drawbacks of our analytical approach and brings-up interesting facts about the different communication setups.

2. Model Formulation. The architecture targeted in this paper consists of N heterogeneous computing nodes that receive image data from a load originating node and return the results of the image matching process to it. The network architecture is a single-level tree or a bus-connected one. Because this can be a repetitive process, each node can build up a local image cache that can be reused for subsequent queries. Hence the load originating node has to communicate to the computing nodes only what they are missing, either because of the incorporation of new images or because of the departure of nodes from the computing pool.

Our treatment of the problem is based on the formulation of an affine model that describes the computation and communication overheads associated with the query data distribution, the image matching process and the

TABLE 2.1
Notations

Symbol	Description	Units
b	is the constant overhead associated with load distribution. It consists of the image to be matched in addition to any query specific data (e.g. matching thresholds).	B
d	is the constant overhead associated with result collection. Typically $d < b$.	B
e_X	is the part of the load which is resident at node X, i. e. a local image cache.	B
I	is the typical size of an image used for image matching.	B
L	the load that is has to be communicated to the computing nodes	B
l_X	is inversely proportional to the speed of the link connecting X and its load originating node.	sec/B
p_X	is inversely proportional to the speed of X.	sec/B
$part_X$	is the part of the load L assigned to X, hence $0 \leq part_X \leq 1$. The total load assigned to X is $part_X L + e_X$	NA

result collection phase. These models are closely related with the ones introduced in [17] although the semantics for some of the constants used here are different. Given a node X that is connected to a load originating node with a connection of (inverse) speed l_X, we assume that the load distribution t_{distr}, the computation t_{comp} and the result collection t_{coll} costs are given by:

$$t_{distr} = l_X \left(part_X L + b\right) \tag{2.1}$$

$$t_{comp} = p_X \left(part_X L + e_X\right) \tag{2.2}$$

$$t_{coll} = l_X d \tag{2.3}$$

The symbols used above, along with all the remaining ones to be introduced later in our analysis, are summarized in Table 2.1.

The total load to be processed by N nodes is

$$\sum_{i=0}^{N-1} \left(part_i L + e_i\right) \tag{2.4}$$

and for the communicated load parts we have:

$$\sum_{i=0}^{N-1} part_i = 1 \tag{2.5}$$

The contribution of the above components to the overall execution time of node X depends on how communication and computation overlap. We can identify two cases:

- **Block**-type computation: no overlap between communication and computation. Node X can start computing only after all data are delivered:

$$t_X = l_X \left(part_X L + b + d\right) + p_X \left(part_X L + e_X\right) \tag{2.6}$$

- **Stream**-type computation: node X can start using each local image cache immediately after receiving the query data. Computation can run concurrently with the communication of the extra data $part_X L$. There are two cases depending on the relative speed between communication and computation:
  - Communication speed is high enough to prevent X from going idle i. e.

$$p_X \left(part_X L + e_X - I\right) \geq l_X part_X L \tag{2.7}$$

where I is the size of the last image to be compared against the required one. Then:

$$t_X = l_X \left(b + d\right) + p_X \left(part_X L + e_X\right) \tag{2.8}$$

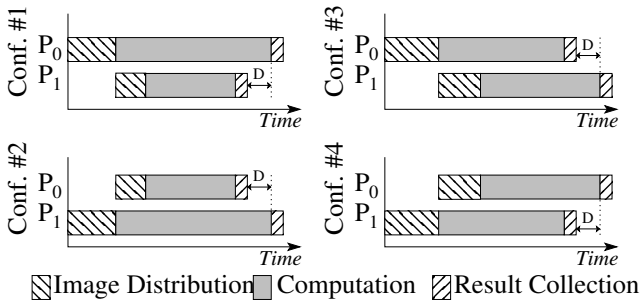


FIG. 3.1. *The four possible configurations of processing by two nodes when 1-port communications are used. Result collection is assumed to be separated by a constant delay D.*

– Node X has to wait for the delivery of data through a slow link, i. e. condition (2.7) is invalid. Then:

$$t_X = l_X\,(b+d) + l_X part_X L + p_X I \tag{2.9}$$

The additional parameter that controls the overall cost when N nodes are used, is whether single-port or N-port communications are employed, e.g. whether the load originating node can distribute L concurrently to multiple nodes.

In the remaining sections we focus on block-type tasks under both 1- and N-port communication setups. Our derivations are based on the assumptions of uniform communication media, i. e. $l_i = l\ \forall i$. A comparison between the two communication setups is performed in section 6.

It should be noted that the static model proposed in this paper, while not apparently suitable for a grid computing scenario, in which computation and communication costs change over time, it can form the basis for an adaptive scheduler that modifies load distribution over time given cost estimates. This goes beyond the scope of this paper and should be the topic of further research.

3. The 1-port Communication Case.

3.1. The two-node scenario. If we assume that there is a load originating node that distributes the load to two nodes, then if single port communications and a single installment [16] are used, the possible sequences of communication and computation operations are shown in Fig. 3.1, as imposed by the need to have no gaps between stages (otherwise, execution time is not minimized). For reasons that will become obvious in the rest of the section, we also assume that the two result collection phases are separated by a constant delay D.

The total execution time for configuration #1 is given by:

$$t_1 = l\,(part_0 L + b) + p_0\,(part_0 L + e_0) + ld \tag{3.1}$$

where

$$p_0(part_0 L + e_0) = l(part_1 L + b) + p_1(part_1 L + e_1) + l\ d + D \tag{3.2}$$

Eq. (3.2) coupled with the normalization equation $part_0 + part_1 = 1$ can provide a solution for $part_0$ and t_1. A similar procedure can produce the times for the three remaining configurations. Thus we can form the pairwise differences of running times:

$$t_3 - t_4 = \frac{l\,(e_1 p_1 - e_0 p_0) + (dl - bl + D)\,(p_1 - p_0)}{p_0 + p_1 + l} \tag{3.3}$$

$$t_3 - t_2 = \frac{l\,(e_1 p_1 - e_0 p_0 - b\,(p_1 - p_0) - dl - D)}{p_0 + p_1 + l} \tag{3.4}$$

$$t_3 - t_1 = \frac{(dl + D)(p_1 - p_0 - l)}{p_0 + p_1 + l} \tag{3.5}$$

$$t_1 - t_4 = \frac{l\,(e_1 p_1 - e_0 p_0 - b\,(p_1 - p_0) + d\;l + D)}{p_0 + p_1 + l} \tag{3.6}$$

$$t_1 - t_2 = \frac{l\,(e_1 p_1 - e_0 p_0 - (b + d)\,(p_1 - p_0))}{p_0 + p_1 + l} - \frac{D\,(p_1 - p_0)}{p_0 + p_1 + l} \tag{3.7}$$

$$t_4 - t_2 = -\frac{(d\;l + D)\,(p_1 - p_0 + l)}{p_0 + p_1 + l} \tag{3.8}$$

Clearly, the problem is too complex to have a single solution even for the simplest case of two nodes. We can however isolate a number of useful special cases that make a closed form solution to the N-node problem tractable:

- **No image caches** ($e_0 = e_1 = 0$). If we assume than $p_0 \leq p_1$ and given that $b > d$, we have:

$$t_3 - t_4 = \frac{(dl - bl + D)\,(p_1 - p_0)}{p_0 + p_1 + l} \tag{3.9}$$

$$t_3 - t_2 = \frac{l\,(-b\,(p_1 - p_0) - d\;l - D)}{p_0 + p_1 + l} \leq 0 \tag{3.10}$$

$$t_3 - t_1 = \frac{(d\;l + D)\,(p_1 - p_0 - l)}{p_0 + p_1 + l} \tag{3.11}$$

If $dl - bl + D \leq 0 \Rightarrow D \leq l\,(b - d)$, then Eq. (3.11) dictates that either configuration #3 or configuration #1 are optimum based on whether $p_1 - p_0 - l$ is negative or not. If we assume that the differences in execution speed are small relative to the communication cost l (i. e. $p_1 - p_0 \leq l$) then configuration #3 is the optimum one.
The execution time is given by

$$t_3^{(nc)} = l\left(part_0^{(nc)} L + b\right) + p_0 part_0^{(nc)} L + D + 2ld \tag{3.12}$$

where:

$$part_0^{(nc)} = \frac{p_1 L + l(L - d + b) - D}{L\,(p_0 + p_1 + l)} \tag{3.13}$$

- **Homogeneous system** ($p_0 = p_1 = p$). If we assume that $e_0 \geq e_1$ then:

$$t_3 - t_4 = \frac{pl\,(e_1 - e_0)}{2p + l} \leq 0 \tag{3.14}$$

$$t_3 - t_1 = -\frac{l\,(d\;l + D)}{2p + l} \leq 0 \tag{3.15}$$

$$t_1 - t_2 = \frac{pl\,(e_1 - e_0)}{2p + l} \leq 0 \tag{3.16}$$

which again translates to having configuration #3 as the optimum one. It should be noted that the optimum order dictates that load is sent first to the node with the biggest cache, which is a counter-intuitive result! The execution time is given by

$$t_3^{(homo)} = l\left(part_0^{(homo)} L + b\right) + p\left(part_0^{(homo)} L + e_0\right) + D + 2ld \tag{3.17}$$

where:

$$part_0^{(homo)} = \frac{p(L + e1 - e0) + l(L - d + b) - D}{L\,(2p + l)} \tag{3.18}$$

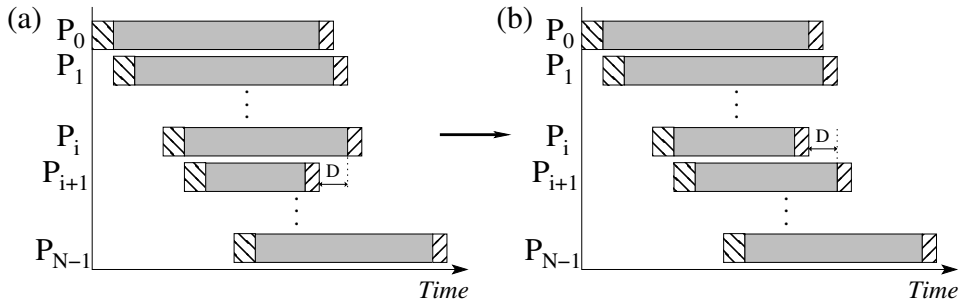


FIG. 3.2. *(a) A possible ordering of load distribution and result collection for N nodes. (b) Improving the execution time by ordering the operations of P_i and P_{i+1} in non-decreasing order of their speed (assuming $p_i \leq p_{i+1}$). Note: the phase durations are disproportionate to actual timings.*

The delay D that was introduced above allows us to extend our analytical treatment from 2 nodes to N. D is supposed to model the time taken by the result collection operations of other nodes. Hence, D is a multiple of $d \cdot l$ with a maximum value of $(N-2)\, d \cdot l$. For the case of no-caches, as long as $D \leq l\,(b-d) \Rightarrow N \leq \frac{b}{d} + 1$ and the differences in computation speed are smaller than the communication speed, configuration #3 is the optimum one as stated by the following theorem. Given the 2-3 orders of magnitude difference expected between b and d, the range of N that the theorem applies is quite broad.

THEOREM 3.1. *The optimum load distribution and result collection order for an image query operation performed by N nodes is given by:*

- ***No image caches**: distributing the load and collecting the results in non-increasing order of the nodes' speed (i. e. in non-decreasing order of the p_i parameters). The sufficient but not necessary conditions for this to be true is $N \leq \frac{b}{d} + 1$ and $|p_i - p_j| \leq l$ for any pair of nodes i, j.*
- ***Homogeneous system**: distributing the load and collecting the results in non-increasing order of the local image cache sizes.*

Proof. We will prove the above theorem for the no-caches case via contradiction. The proof for the homogeneous case is identical. Let's assume that the optimum order is similar to the one shown in Fig. 3.2(a). Without loss of generality we assume that the distribution order is $P_0, P_1, \ldots P_{N-1}$

For any two nodes P_i and P_{i+1} that do not satisfy the order proposed by Theorem 3.1, we can rearrange the distribution and collection phases so as the part of the load that is collectively assigned to them $(L\,(part_i + part_{i+1}))$ is processed in a shorter time frame (as long as $N \leq \frac{b}{d} + 1$), while occupying in an identical fashion the communication medium (see Fig. 3.2(b)). Thus, the operation of the other nodes is not influenced. At the same time the shorter execution time would allow additional load to be given to nodes P_i and P_{i+1} resulting in a shorter total execution time. The outcome is a contradiction to having the original ordering being an optimum one. The only ordering that cannot be improved upon by the procedure used in this proof, is the one proposed by Theorem 3.1. □

The above discussion settles the ordering problem, allowing us to generate a closed-form solution to the partitioning problem for N nodes.

3.2. Closed-form solution for N nodes.

3.2.1. No image caches. The following relation holds between every pair of nodes which are consecutive in the distribution and collection phases (without loss of generality we will again assume that the nodes' order is $P_0, P_1, \ldots, P_{N-1}$):

$$\begin{aligned} p_i part_i L + ld &= l\,(part_{i+1} L + b) + p_{i+1} part_{i+1} L \Rightarrow \\ part_{i+1} &= part_i \frac{p_i}{p_{i+1} + l} + \frac{l\,(d-b)}{L\,(p_{i+1} + l)} \end{aligned} \tag{3.19}$$

This can be extended to any pair of nodes P_i and P_j, where $i > j$:

$$part_i = part_j \prod_{k=j}^{i-1} \frac{p_k}{p_{k+1} + l} + \frac{l\,(d-b)}{L} \sum_{k=j+1}^{i} \left[(p_k + l)^{-1} \prod_{m=k}^{i-1} \frac{p_m}{p_{m+1} + l} \right] \tag{3.20}$$

Equipped with Eq. (3.20), we can associate each $part_i$ with $part_0$ and use the normalization equation:

$$\sum_{i=0}^{N-1} part_i = 1 \tag{3.21}$$

to compute a closed form solution for $part_0$:

$$part_0 = \frac{1 - \frac{l(d-b)}{L} \sum_{i=1}^{N-1} \sum_{k=1}^{i} \frac{\prod_{m=k}^{i-1} \frac{p_m}{p_{m+1}+l}}{(p_k+l)}}{1 + \sum_{i=1}^{N-1} \prod_{k=0}^{i-1} \frac{p_k}{p_{k+1}+l}} \tag{3.22}$$

Equations (3.22) and (3.20) solve the partitioning problem. The total execution time is:

$$t_{total}^{(nc)} = l\left(part_0 L + b\right) + p_0 part_0 L + N\ l\ d \tag{3.23}$$

The above constitute a closed form solution that can be computed in time $\frac{N^2-N}{2} + 3(N-1) + Nlg(N) = O\left(N^2\right)$, where $Nlg(N)$ is the node-sorting cost.

3.2.2. Homogeneous System. Following a similar procedure to the previous section, it can be shown that:

$$\begin{aligned} p\left(part_i L + e_i\right) + l\ d = l\left(part_{i+1} L + b\right) + p\left(part_{i+1} L + e_{i+1}\right) \Rightarrow \\ part_{i+1} = part_i \frac{p}{p+l} + \frac{l\left(d-b\right) + p\left(e_i - e_{i+1}\right)}{L\left(p+l\right)} \end{aligned} \tag{3.24}$$

This can be extended to any pair of nodes P_i and P_j, where $i > j$:

$$part_i = part_j \left(\frac{p}{p+l}\right)^{i-j} + \sum_{k=j}^{i-1} \frac{l\left(d-b\right) + p\left(e_k - e_{k+1}\right)}{L\left(p+l\right)} \left(\frac{p}{p+l}\right)^{i-k-1} \tag{3.25}$$

Again, Eq. (3.25), and the normalization equation can produce a closed form solution for $part_0$:

$$part_0 = \frac{d-b}{L} + \frac{l + \frac{l\ N(b-d)}{L}}{p + l - p\left(\frac{p}{p+l}\right)^{N-1}} - \frac{l \sum_{i=1}^{N-1} \sum_{k=0}^{i-1} \frac{(e_k - e_{k+1})}{L} \left(\frac{p}{p+l}\right)^{i-k}}{p + l - p\left(\frac{p}{p+l}\right)^{N-1}} \tag{3.26}$$

The total execution time can be then computed as:

$$t_{total}^{(homo)} = l\left(part_0 L + b\right) + p\left(part_0 L + e_0\right) + N\ l\ d \tag{3.27}$$

As with the previous case, the solution requires an $O\left(N^2\right)$ computational cost.

A special case needs to be considered if $L = 0$ as the above equations cannot be applied. The minimum execution can be achieved only if the local caches are appropriately sized to accommodate this. Similarly to Eq. (3.25) for two nodes P_i and P_j, where $i > j$ we would have:

$$\begin{aligned} p_i e_i + (j-i) l d = (j-i) l b + p_j e_j \Rightarrow \\ e_j = e_i \frac{p_i}{p_j} + \frac{(j-i) l (d-b)}{p_j} \end{aligned} \tag{3.28}$$

If the caches do not satisfy condition (3.28), the load must be reassigned/transferred between nodes. In this paper we assume that this is performed by the load originating node and not by a direct exchange between the compute nodes. Section 5 elaborates more on how we can treat this case.

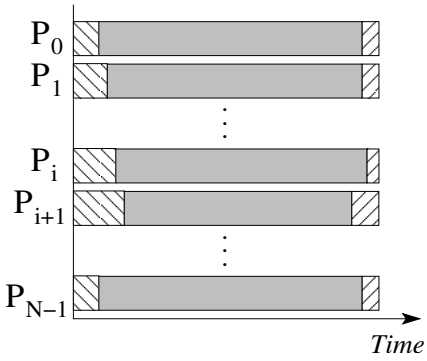


FIG. 4.1. *Optimum scheduling for a N-port communication setup.*

4. The N-port Communication Case.

4.1. Closed-form solution for N nodes. The N-port communication case is much simpler than the 1-port one since no explicit node ordering is necessary. It can be easily shown in this case that the optimum load partitioning has to produce identical running times on all the participating compute nodes, i. e. all nodes must start receiving data and finish delivering results at the same instant. Since all nodes must have the same starting and ending times as shown in Fig.4.1, for any two nodes i and j, the following has to hold:

$$\begin{aligned} l\left(part_i L + b\right) + p_i\left(part_i L + e_i\right) + ld = \\ l\left(part_j L + b\right) + p_j i\left(part_j L + e_j\right) + ld \Rightarrow \\ part_i L\left(p_i + l\right) + p_i e_i = part_j L\left(p_j + l\right) + p_j e_j \Rightarrow \\ part_i = part_j \frac{p_j + l}{p_i + l} + \frac{p_j e_j - p_i e_i}{L\left(p_i + l\right)} \end{aligned} \tag{4.1}$$

The normalization equation (3.21) can then be used to produce a closed-form solution for $part_0$ and subsequently all $part_i$:

$$\begin{aligned} \sum_{i=0}^{N-1} part_i = 1 \Rightarrow \\ part_0 \sum_{i=0}^{N-1} \frac{p_0 + l}{p_i + l} + \sum_{i=1}^{N-1} \frac{p_0 e_0 - p_i e_i}{L\left(p_i + l\right)} = 1 \Rightarrow \\ part_0 = \frac{1 + \sum_{i=1}^{N-1} \frac{p_i e_i - p_0 e_0}{L(p_i + l)}}{\sum_{i=0}^{N-1} \frac{p_0 + l}{p_i + l}} \end{aligned} \tag{4.2}$$

The total execution time is given by:

$$t_{total}^{(Nport)} = l\left(part_0 L + b\right) + p_i\left(part_0 L + e_i\right) + ld \tag{4.3}$$

4.2. Homogeneous System Solution. For a homogeneous system ($\forall p_i \equiv p$), the above equations are simplified to the following:

$$(4.1) \Rightarrow part_i = part_j + \frac{p\left(e_j - e_i\right)}{L\left(p + l\right)} \tag{4.4}$$

$$(4.2) \Rightarrow part_0 = N^{-1}\left(1 + \frac{p}{L}\sum_{i=1}^{N-1} \frac{e_i - e_0}{p + l}\right) \tag{4.5}$$

which translates to having differences in the local caches as the single cause of any imbalances in the split of the new load L. Otherwise the load should be evenly split.

5. Image Cache Management. Eq. (3.20), (3.25), (4.1) and (4.4) allow for negative values for $part_i$s. Such an event indicates that the corresponding node should not participate in the calculation, either because it is too slow or because the local cache size is too large for a node to process and keep up with the other nodes. In the latter case it is obvious that a node should use only a part of its cache. The load surplus should be transferred to other nodes. This situation can arise when following the initial distribution of load to the nodes, subsequent queries are no longer accompanied by big chunks of data, making the initial distribution a suboptimal one.

In this section we address this problem by proposing a algorithm for estimating the proper cache size that should be used, along with the corresponding load L that should be communicated to other nodes.

The algorithm presented below, is based on the assumption that the intersection of all caches is $\emptyset$. The key point of the algorithm is a re-assignment of load from the nodes with an over-full cache (identified as set S in line 8) to the nodes with little or no cache. This process reduces the total execution time as long as communication is faster than computation.

This algorithm has been also enhanced from the version presented in [18] to address the case when $L = 0$, i. e. when processing is based entirely on the nodes' local data. In that case, L can be initialized to a small value, e.g. $L = 1$ (lines 2-5), which would be subsequently subtracted once a redistribution is dimmed necessary (lines 32-36).

Set S does not change after line 8 as the subsequent increase in L due to a load shift (line 31) does not permit any other node from having a negative assignment. The loop of lines 13-46 is executed for as long as there is a negative $part_i$, or a load shift is necessary for balancing the node workload. In line 17 the size of the cache that should be used in a node with a negative assignment is estimated. Because the load is reassigned collectively in line 31, the cache size of each node in S can be under-estimated (by "bloating" the load L that should be communicated). This defeats the optimization procedure by forcing the communication of data that are already present at the nodes, and in order to guard against this possibility, lines 21-28 re-adjust any previous overestimation for nodes that subsequently got to have positive $part_j$. Lines 12 and 41-44 serve as sentinels against cases where the outer *while* loop does not converge. In that case, fixing the part assigned to the last node in the distribution sequence (smallest e) to 0, allows the convergence of the outer loop. A value for threshold $THRES$ that was found to yield good results in our experiments is 20. Threshold values that depend on the number of compute nodes did not provide any visible difference.

Lines 32-36 cancel the addition of 1 load unit that is done when $L = 0$. Finally, if L remains 0 after load redistribution is examined, cache sizes satisfy condition (3.28) for a homogeneous system and nothing more needs to be done (lines 37-40).

A key point that should be made here is that Algorithm 1 produces a sub-optimum solution when a *series* of query operations are to be scheduled. Designing an optimum algorithm for this scenario is beyond the scope of this paper.

6. Simulation Study. Single-port communication is surely not a contemporary technology limitation. It is rather a design feature whereas the load originating node dedicates its attention to a single node at a time, with the objective of minimizing the message exchange cost between itself and the corresponding node. In this section we explore the impact of the two alternative design choices with the assistance of our analytical framework. Also, we evaluate the performance achieved by the use of Algorithm 1 for managing the image caches through a battery of image queries.

We base the bulk of our simulations on the assumption of a homogeneous platform. While the requirement of a homogeneous system may seem unrealistic, it can be typical of many large scale installations in big organizations.

The key points of our simulation scenario which consists of a series of image query operations, are the following:

- The image DB[1] consists originally of 10000 images of size 1MB each. This is a small number relevant to the yearly "production" of mammograms generated at a national level. Additionally, the image size matches real data only in the order of magnitude as high resolution mammograms can be much larger (e.g. 8MB).
- Each new image that is matched against the DB is also 1MB in size, hence $b = 1MB$.

[1]We use the term DB to loosely refer to the collection of available, tagged, medical images, and not to an actual DBMS system. Storage services are offered in MammoGrid [1] by MySQL and in NDMA by IBM's DB2 [10]

Algorithm 1 Estimating the local image cache sizes that yield the minimum execution time for the next query operation

1: $load\_shift \leftarrow 0$
2: **if** $L = 0$ **then**
3: $added \leftarrow TRUE$
4: $L \leftarrow 1$
5: **end if**
6: In the case of 1-port communication and a homogeneous system, sort the nodes in descending order of their e_i parameters.
7: Calculate the load part for each node P_i via Eq. (3.26), (3.25) or (4.2), (4.1)
8: Let S be the set of nodes with $part_j < 0$
9: **if** $S \neq \emptyset$ **then**
10: Copy the cache sizes of all nodes in temporary variables $e_i^{(orig)}$
11: **end if**
12: $iter \leftarrow 0$
13: **while** $S \neq \emptyset$ OR $load\_shift \neq 0$ OR $added = TRUE$ **do**
14: $load\_shift \leftarrow 0$
15: **for** each $P_j \in S$ **do**
16: **if** $part_j < 0$ **then**
17: $aux \leftarrow part_j L + e_j$
18: $load\_shift \leftarrow load\_shift + e_j - aux$
19: $e_j \leftarrow aux$
20: **else**
21: $aux \leftarrow part_j L + e_j$
22: **if** $aux > e_i^{(orig)}$ **then**
23: $diff \leftarrow e_j^{(orig)} - e_j$
24: **else**
25: $diff \leftarrow aux - e_j$
26: **end if**
27: $load\_shift \leftarrow load\_shift - diff$
28: $e_j \leftarrow e_j + diff$
29: **end if**
30: **end for**
31: $L \leftarrow L + load\_shift$
32: **if** $added = TRUE$ **then**
33: $added \leftarrow FALSE$
34: $L \leftarrow L - 1$
35: $load\_shift \leftarrow 1$
36: **end if**
37: **if** $L = 0$ **then**
38: Set for all nodes P_j, $part_j \leftarrow 0$
39: BREAK
40: **end if**
41: $iter \leftarrow iter + 1$
42: **if** $iter > THRES$ **then**
43: Fix the $part_k$ assigned to the node with the smallest e_k to 0
44: **end if**
45: Calculate the load part for each P_i, other than the nodes fixed in step 43.
46: **end while**

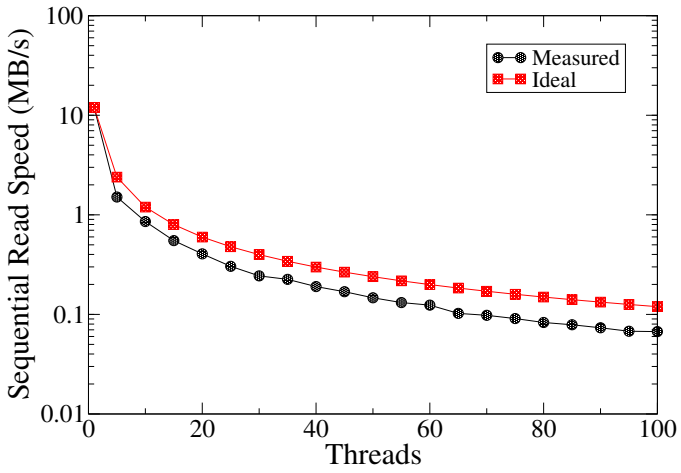


FIG. 6.1. *Average sequential disk read speed per thread. The ideal curve represents the case where the total bandwidth is evenly divided between the threads without losses.*

- Every 100 queries, 100 appropriately tagged images are incorporated in the image DB, hence the resident load increases gradually.
- The data collected from each node consist of the best 10 matches, along with the corresponding image IDs and objective function values, assumed in total to be of size $d = 10 \cdot (2 + 4) = 60B$.
- The *tiobench* utility [19] was used to estimate realistic values for the data rates between the load originating nodes and the compute nodes. A variable number of threads were used to represent simultaneous access from multiple clients. The results which were collected on a Linux laptop machine, equipped with a ATA 100 100GB hard disk spinning at 4200rpm, formatted using the ReiserFS filesystem, are shown in Fig. 6.1. The effect of the disk cache was minimized by using a 3GB file size. These speeds were used in the 1- and N-port simulations that are reported in this paper. For 1-port communications in particular, l was set equal to $0.00997 sec/Mb$, which translates to $0.0837 sec/image$.

The first question we would like to answer, is what would be the improvement of using our analytical approach over an Equal load Distribution (ED) strategy that is traditionally used in homogeneous systems [15], in a single-shot scenario, i. e. when only one query operation is performed. For this purpose, we tested both 1- and N-port approaches, where the computing speed of all nodes was set to be one of the following values $\{0.08, 0.17, 0.33, 0.67, 1.34\} sec/image$, roughly corresponding to 1x, 2x, 4x, 8x and 16x the time required to communicate a single image when 1-port communication is used. In the remainder of this section we will refer to these processing speeds as $1l$, $2l$, $4l$, $8l$ and $16l$ respectively. Such a selection of processing speeds/costs matches closely the running times reported in [15] for real-life tests and they are supposed to help us probe the effects of different computation/communication ratios and the use of different image registration algorithms.

The results for the 1-port case are shown in Figure 6.2 in the form of the improvement achieved over the ED approach. In all the comparative results reported in this section, we use the execution time provided by the 1-port non-uniform proposed distribution strategy (as given by Eq.(3.27) and denoted below as t_{SP}) as the baseline. The improvement is defined as:

$$\frac{t_{ED} - t_{SP}}{t_{SP}} \tag{6.1}$$

which is basically the percent overhead that ED (t_{ED}) is causing over the proposed analytical solution. All initial caches were set equal to 0 which is a typical initialization scenario. It should be noted that all the results reported in Fig. 6.2 and the remaining graphs of this section, correspond to cases where all available nodes can be utilized, hence the lack of data points for big values of N when p is relatively small. This qualification was imposed to avoid skewed results.

As can be observed in Fig. 6.2, the improvement is even higher when the computational cost is proportionally higher than the communication, topping around 28% for the $p = 16l$ case. In the majority of the tested cases, the gain is above 10%.

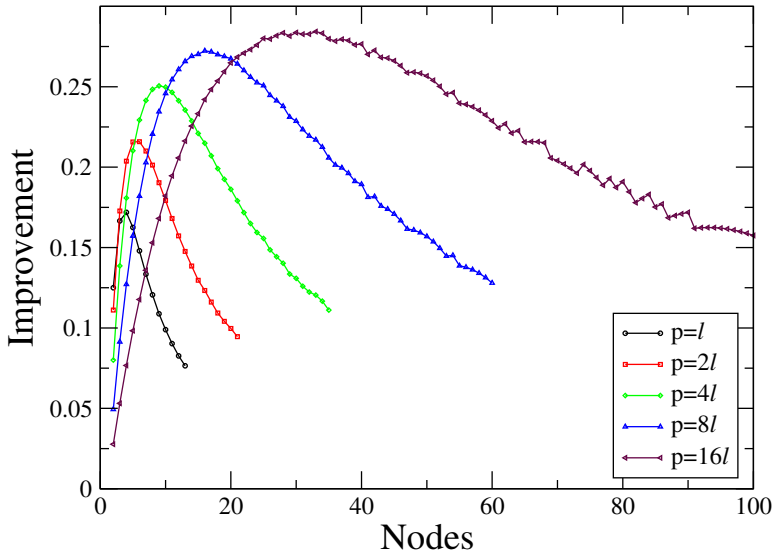


FIG. 6.2. *Execution time improvement offered by the proposed scheme over the ED strategy, for a single-shot scenario.*

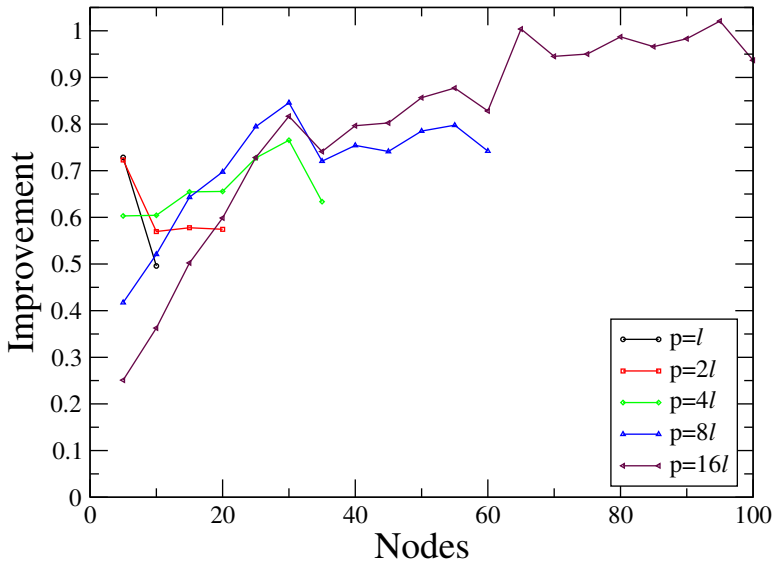


FIG. 6.3. *Execution time improvement offered by the 1-port over the N-port approach, for a single-shot scenario.*

Comparing the 1-port and N-port cases is less straightforward as there is a question of whether the N-port communication setup is accomplished by sharing the same medium - as is usually the case in non-dedicated platforms such as Networks of Workstations (NoW) -, or the load originating node is having a dedicated link for each worker. In the following paragraphs we assume that the former setup is applicable.

The improvement offered (!) by the 1-port over the N-port case is shown in Figure 6.3, where improvement is now computed by Eq. (6.1) by replacing t_{ED} with the execution time of the N-port arrangement t_{NP}. It comes as a surprise that the N-port arrangement can be such a poor performer! The reasons can be summarized as follows: (a) sharing the communication medium causes the computation phase to be overly delayed while data are being downloaded and (b) the cost of switching is taking a heavy toll on the available bandwidth, as observed in Figure 6.1 if one compares the measured against the ideal curves. In summary, the 1-port setup allows -some of- the compute nodes to start processing the load a lot sooner. Of course this result has to be seen in the *proper context*, i. e. we have block-type tasks and the nodes have no image cache. As it will be shown below, this picture is far from the truth for a sequence of query operations.

In order to test what would be the situation if a sequence of queries were performed, we simulated the successive execution of 1000 queries. The corresponding improvement for the 1-port scheme is shown in Fig. 6.4 (a). As can be observed, the ED strategy is not worst in every case due to the cost of cache redistribution that Algorithm 1 is causing. Actually for fast computation ($p = l$) and a relatively small number of nodes, ED is faster. For the majority of the other cases, the gains seems insignificant (in the order of 1%) as the constant shuffling of the caches slows down the whole process. These effects can be minimized if queries are run in batches as can be clearly seen in Fig.6.4 (b) and (c), for moderate (10 queries) and extreme batch sizes (100 queries) respectively. For batch processing the same analytical models can be applied, if we multiply the constants b, d and p by the batch size. Batching requests together does not come close to optimizing a sequence of them as performed in [20], but as it is shown in Fig.6.4, boosts performance substantially. Under such conditions the proposed strategy is consistently better than the ED one, although the actual gains depend on the ratio between computation and communication costs. If the former are dominant (e.g. as in the $p = 16l$ case), any benefits made by effectively scheduling the communication operations is marginalized.

Fig. 6.4 does not convey the complete picture though, as the gains seem insignificant. However, when the running times are as high as shown in Fig. 6.5 even small gains translate to big savings in time.

For the N-port case, batching requests produces small absolute savings as shown at the bottom of Fig. 6.5 (b), (c). While the gain barely reaches 1 hour overall, the real benefit comes from increased scalability, i. e. the ability to use bigger sets of processors for the task. For example, for $p = l$ batches of 100 queries can run on 100 nodes, while individually queries are limited to 13 nodes.

The picture is completely reversed for the N-port case when multiple queries are considered, as can be observed in Fig. 6.6. Even with the reduced bandwidth available to each compute node and the deterioration of the total available bandwidth, the N-port approach is a hands-down winner. This is especially true when the number of nodes grows beyond a limit, making this the most scalable strategy, despite the bandwidth loss identified in Fig. 6.1. Additionally, batching queries together benefits the N-port approach even more than the 1-port, non-uniform one.

7. Conclusion. In this paper we present an analytical solution to the problem of optimizing content-based image query processing over a parallel platform under communication constraints. We solve the problem analytically for both the single and N-port cases and we also prove an important theorem for the sequence of operations that minimize the execution time. Our analytical solution is accompanied by an algorithm for the cache management of the nodes of a system, either 1-port homogeneous or N-port heterogeneous. Our closed-form solution for the 1-port heterogeneous case with no image caches, can be employed when a single-shot operation is preferred.

The extensive simulations that were conducted were able to reveal the following design principles, as far as homogeneous platforms are concerned:

- If a single-shot execution is desired, a 1-port non uniform distribution as highlighted in Section 3.2.2 is the best one.
- For a sequence of operations, the N-port strategy is the best performer, especially if the computational cost is proportionally higher, or the number of nodes is high.

Future research directions could include:

- Using the proposed methodology as a part of a Grid middleware scheduler. It is possible that the high overhead of typical grid schedulers compromises the benefits shown in this paper, requiring further optimizations.
- Devising a solution for a heterogeneous system with local cache.
- Examine the case of multiple image sources instead of a single load originating node. Although current generation systems rely mostly on a single image repository, next generation ones are moving away from this paradigm [1].

REFERENCES

[1] S. R. Amendolia, F. Estrella, R. McClatchey, D. Rogulin, and T. Solomonides, "Managing pan-european mammography images and data using a service oriented architecture," in *Proc. IDEAS Workshop on Medical Information Systems: The Digital Hospital (IDEAS-DH'04)*, September 2004, pp. 99–108.

[2] M. M. Rahman, T. Wang, and B. C. Desai, "Medical image retrieval and registration: Towards computer assisted diagnostic approach," in *Proc. IDEAS Workshop on Medical Information Systems: The Digital Hospital (IDEAS-DH'04)*, September 2004, pp. 78–89.

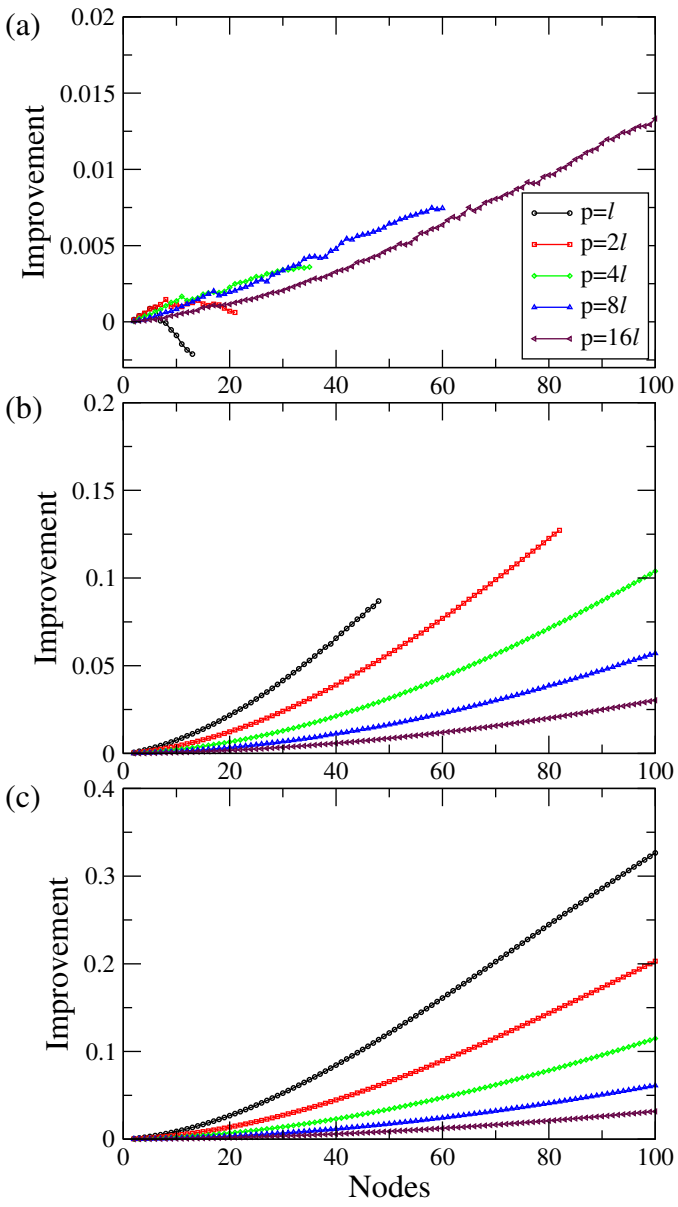


FIG. 6.4. *Execution time improvement offered by the 1-port scheme over the ED strategy, for a sequence of 1000 queries: (a) when each query is run individually, (b) when queries are run in batches of 10, and (c) when queries are run in batches of 100.*

[3] S. Sneha and A. Dulipovici, "Strategies for working with digital medical images," in *Proc. 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, January 2006, p. 100a.

[4] Y. Kawasaki, F. Ino, Y. Sato, S. Tamura, and K. Hagihara, "Parallel adaptive estimation of range of motion simulation for total hip replacement surgery," *IEICE Transactions on Information and Systems*, vol. E90-D, no. 1, pp. 30–39, 2007.

[5] H. Zhou, X. Yang, H. Liu, and Y. Tang, "First evaluation of parallel methods of automatic global image registration based on wavelets," in *2005 International Conference on Parallel Processing (ICPP'05)*, July 2005, pp. 129–136.

[6] I. D. Falco, D. Maisto, U. Scafuri, E. Tarantino, and A. D. Cioppa, "Distributed differential evolution for the registration of remotely sensed images," in *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, February 2007, pp. 358–362.

[7] S. Ait-Aoudia and R. Mahiou, "Medical image registration by simulated annealing and genetic algorithms," in *Geometric Modelling and Imaging (GMAI '07)*, July 2007, pp. 145–148.

[8] Y. Bentoutou, N. Taleb, K. Kpalma, , and J. Ronsin, "An automatic image registration for applications in remote sensing," *IEEE Trans. on Geoscience and Remote Sensing*, vol. 43, no. 9, pp. 2127–2137, September 2005.

[9] J. Montagnat, H. Duque1, J. Pierson, V. Breton, L. Brunie, and I. E. Magnin, "Medical image content-based queries using the grid," in *Proc. of HealthGrid 03*, 2003.

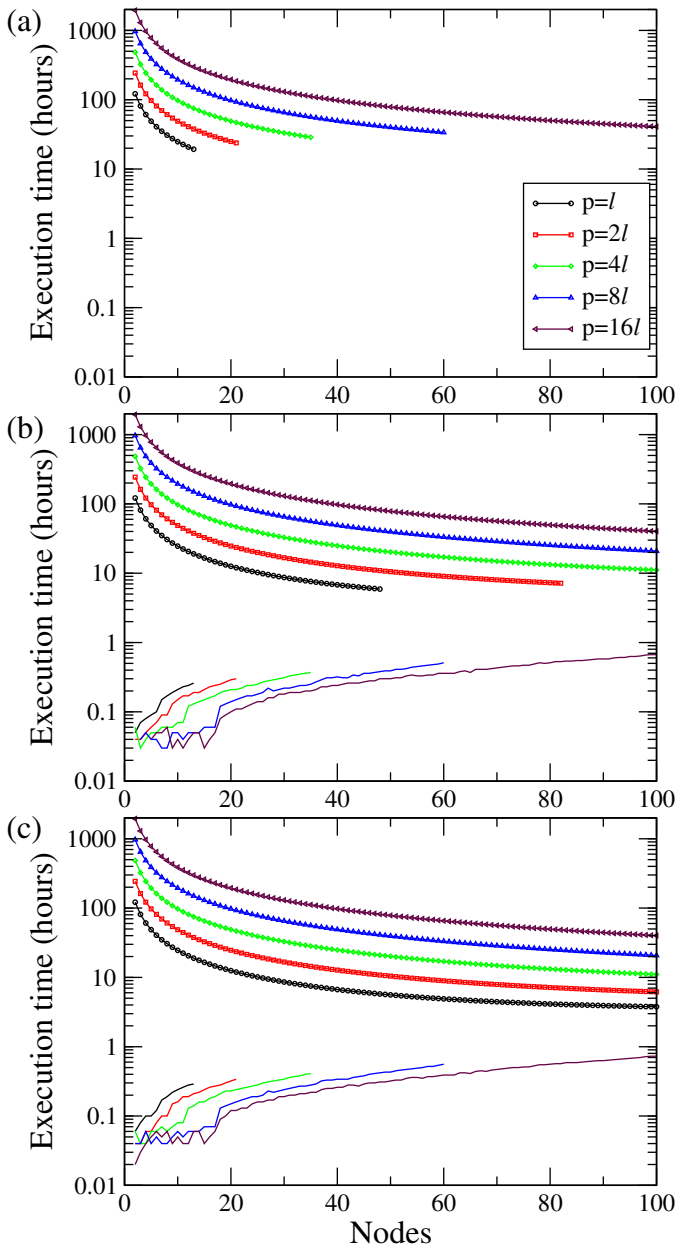


FIG. 6.5. *Total execution time offered by Algorithm 1 for a sequence of 1000 queries, under 1-port configuration: (a) when each query is run individually, (b) when queries are run in batches of 10, and (c) when queries are run in batches of 100. The plain lines at the bottom of (b) and (c) show the corresponding absolute time gain over (a).*

[10] University of Pennsylvania Consortium and National Digital Mammography Archive Grid: http://www.ibm.com/e-business/ondemand/us/innovation/univofpa.shtml [Online]. Available: http://www.ibm.com/e-business/ondemand/us/innovation/univofpa.shtml

[11] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," DRAFT document available at http://www.globus.org/research/papers/ogsa.pdf. [Online]. Available: http://www.globus.org/research/papers/ogsa.pdf

[12] I. Blanquer, V. Hernandez, and F. Mas, "A peer-to-peer environment to share medical images and diagnoses providing context-based searching," in *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, February 2005, pp. 42–48.

[13] F. Ino, Y. Kawasaki, T. Tashiro, Y. Nakajima, Y. Sato, S. Tamura, and K. Hagihara, "A parallel implementation of 2-d/3-d image registration for computer-assisted surgery," in *11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05)*, July 2005, pp. 316–320.

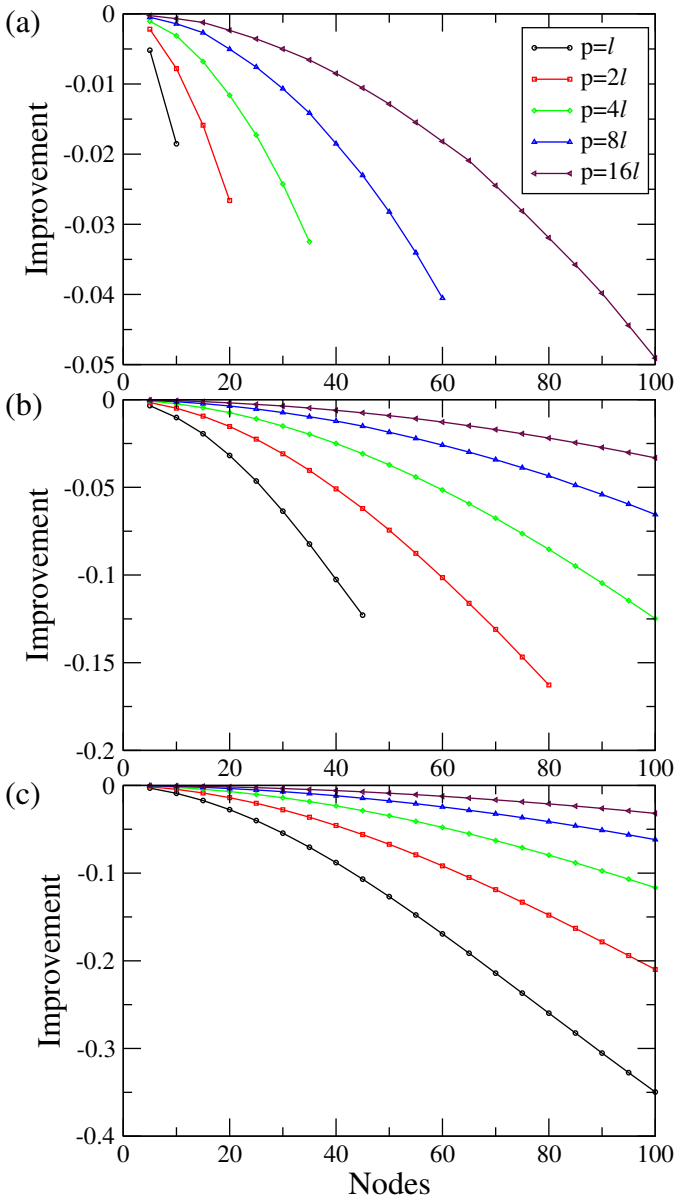


FIG. 6.6. *Execution time improvement offered by the 1-port scheme over the N-port setup, for a sequence of 1000 queries: (a) when each query is run individually, (b) when queries are run in batches of 10, and (c) when queries are run in batches of 100. Negative scores indicate a deterioration in performance, i. e. the 1-port setup is slower.*

[14] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara, "A gpgpu approach for accelerating 2-d/3-d rigid registration of medical images," in *4th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2006)*, ser. Lecture Notes in Computer Science 4330. Sorrento, Italy: Springer-Verlag, 2006, pp. 939 – 950.

[15] J. Montagnat, V. Breton, and I. Magni, "Medical image databases content-based queries partitioning on a grid," in *Proc. of HealthGrid 04, also available at http://www.i3s.unice.fr/~johan/publis/HealthGrid04.pdf*, 2004.

[16] B. Veeravalli, D. Ghose, V. Mani, , and T. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems.* Los Alamitos and California: IEEE Computer Society Press, 1996.

[17] G. D. Barlas, "Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees," *IEEE Trans. on Parallel & Distributed Systems*, vol. 9, no. 5, pp. 429–441, May 1998.

[18] G. Barlas, "Optimizing image content-based query applications over high latency communication media," in *Proc. of 15th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, PDIVM 2007*, Napoly, Italy, February 2007, pp. 341–348.

[19] Threaded I/O bench for Linux: http://tiobench.sourceforge.net/.

[20] V. Bharadwaj and G. D. Barlas, "Efficient strategies for processing multiple divisible loads on bus networks," *Journal of Parallel and Distributed Computation*, no. 62, pp. 132–151, 2002.

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 239–249. http://www.scpe.org
ISSN 1895-1767

DESIGN AND ANALYSIS OF A SCALABLE ALGORITHM TO MONITOR CHORD-BASED P2P SYSTEMS AT RUNTIME*

ANDREAS BINZENHÖFER† GERALD KUNZMANN‡ AND ROBERT HENJES†

Abstract. Peer-to-peer (p2p) systems are a highly decentralized, fault tolerant, and cost effective alternative to the classic client-server architecture. Yet companies hesitate to use p2p algorithms to build new applications. Due to the decentralized nature of such a p2p system the carrier does not know anything about the current size, performance, and stability of its application. In this paper we present an entirely distributed and scalable algorithm to monitor a running p2p network. The snapshot of the system enables a telecommunication carrier to gather information about the current performance parameters of the running system as well as to react to discovered errors.

1. Introduction. In recent years peer-to-peer (p2p) algorithms have widely been used throughout the Internet. So far, the success of the p2p paradigm was mainly driven by file sharing applications. However, despite their reputation p2p mechanisms offer the solution to many problems faced by telecommunication carriers today [8]. Compared to the classic client-server architecture they are decentralized, fault tolerant, and cost effective alternatives. Those systems are highly scalable, do not suffer from a single point of failure, and require less administration overhead than existing solutions. In fact, there are more and more successful p2p based applications like Skype [14], a distributed VoIP solution, Oceanstore [4], a global persistent data store, and even p2p-based network management [10].

One of the main reasons why telecommunication carriers are still hesitant to build p2p applications is the lack of control a provider has over the running system. At first, the system appears as a black box to its operator. The carrier does not know anything about the current size, performance, and stability of its application. The decentralized nature of such a system makes it hard to find a scalable way to gather information about the running system at a central unit. Operators, however, do not want to lose control over their systems. They want to know what their systems look like right now and where problems occur at the moment. The first problems already occur when testing and debugging a distributed application. Finding implementation errors in a highly distributed system is a very complex and time consuming process [9]. A provider also needs to know whether his newly deployed application can truly handle the task it was designed for.

The latest generation of p2p algorithms is based on distributed hash tables (DHTs). The algorithm that currently attracts the most attention is Chord, which uses a ring topology to realize the underlying DHT [12]. DHTs are theoretically understood in depth and proved to be a scalable and robust basis for distributed applications [7]. However, the problem of monitoring such a system from a central location is far from being solved. [11] gives a good overview of different approaches to monitor and debug distributed systems in general. In the field of p2p, the process of measuring and monitoring a running system was so far limited to unstructured overlays. [13], e.g., introduces a crawling-based approach to query Gnutella-like networks.

In this paper, however, we exploit the special features of structured p2p overlays and present an entirely novel and scalable approach to create a snapshot of a running Chord-based network. Using our algorithm a provider can either monitor the entire system or just survey a specific part of the system. This way, he is able to react to errors more quickly and can verify if the taken countermeasures are successful. On the basis of the gathered information it is, e.g., possible to take appropriate action to relief a hotspot or to pinpoint the cause of a loss of the overlay ring structure. The overhead involved in creating the snapshot is evenly distributed to the participating peers so that each peer only has to contribute a negligible amount of bandwidth. Most importantly, the snapshot algorithm is very easy to use for a provider. It only takes one parameter to adjust the trade off between duration of the snapshot and bandwidth needed at the central unit which collects the measurements.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of Chord with a focus on aspects relevant to this paper. The snapshot algorithm as well as some areas of application are described in Section 3. The functionality of the algorithm is verified analytically in Section 4 and by simulation in Section 5. Section 6 concludes this paper.

*Corresponding mail: binzenhoefer@informatik.uni-wuerzburg.de
†University of Würzburg, Institute of Computer Science, Germany.
‡Technical University of Munich, Institute of Communication Networks, Germany.

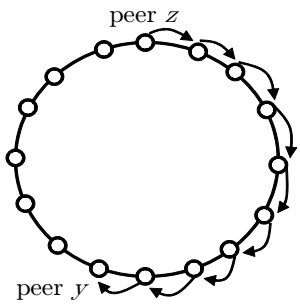


FIG. 2.1. *A simple search.*

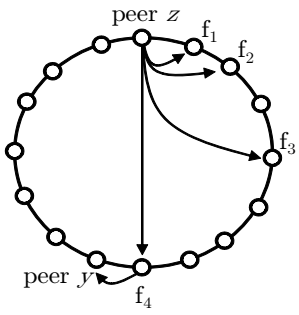


FIG. 2.2. *Search using the fingers.*

2. Chord Basics. This section gives a brief overview of Chord with a focus on aspects relevant to this paper. A more detailed description can be found in [12]. The main purpose of p2p networks is to store data in a decentralized overlay network. Participating peers will then be able to retrieve this data using some sort of search algorithm. The Chord algorithm solves this problem by arranging the participating peers on a ring topology. The position id_z of a peer z on this overlay ring is determined by an m-bit identifier generated by a hash function such as SHA-1 or MD5. In a Chord ring each peer knows at least the id of its immediate successor in a clockwise direction on the ring. This way, a peer looking up another peer or a resource is able to pass the query around the circle using its successor pointers. Figure 2.1 illustrates a simple search of peer z for another peer y using only the immediate successor. The search has to be forwarded half-way around the ring. Obviously, the average search would require $\frac{n}{2}$ overlay hops, where n is the current size of the Chord ring. To speed up searches a peer z in a Chord ring also maintains pointers to other peers, which are used as shortcuts through the ring. Those pointers are called fingers, whereby the *i-th* finger in a peer's finger table contains the identity of the first peer that succeeds z's own id by at least 2^{i-1} on the Chord ring. That is, peer z with hash value id_z has its fingers pointing to the first peers that succeed $\left(id_z + 2^{i-1}\right) \bmod 2^m$ for $i = 1$ to m, where 2^m is the size of the identifier space.

Figure 2.2 shows fingers f_1 to f_4 for peer z. Using this finger pointers, the same search does only take two overlay hops. For the first hop peer z uses its finger f_4. Peer y can then directly be reached using the successor of f_4 as indicated by the small arrow. This way, a search only requires $\frac{1}{2}\log_2(n)$ overlay hops on average. A detailed mathematical analysis of the search delay in Chord rings can be found in [3]. The snapshot algorithm presented in Section 3 makes use of the finger tables of the peers.

3. Design of the Snapshot Algorithm. In this section we introduce a scalable and distributed algorithm to create a snapshot of a running Chord system. The algorithm is based on a very simple two step approach. In step one, the overlay is recursively divided into subparts of a predefined size. In step two, the desired measurement is done for each of these subparts and sent back to a central collecting point (CP). In the following, we describe both steps in detail.

3.1. Step 1: Divide Overlay into Subparts. The algorithm *snapshot(R_s, R_e, S_{min}, CP)* divides a specific region of the overlay into subparts. This function is called at an arbitrary peer p with id_p. The peer then tries to divide the region from $R_s = id_p$ to R_e into contiguous subparts using its fingers. The exact procedure is illustrated in Figure 3.1. In this example peer p has four fingers f_1 to f_4. It sends a request to the finger closest to R_e within $[R_s; R_e]$. At first, finger f_4 is disregarded since it does not fall into the region between R_s and R_e (cf. a). This makes f_3 the closest finger to R_e in our example. If this finger does not respond to the request, as illustrated by the bolt (cf. b), it is removed from the peer's finger list and the peer tries to contact the next closest finger f_2 (cf. c). If this finger acknowledges the request, peer p recursively tries to divide the region from $R_s = id_p$ to $\widehat{R}_e = id_{f_2} - 1$ into contiguous subparts. Finger f_2 partitions the region from $\widehat{R}_s = id_{f_2}$ to R_e accordingly.

As soon as a peer does not know any more fingers in the region between the current R_s and the current R_e, the recursion is stopped. The peer changes into step two of the algorithm and starts a measurement of this specific region. In this context, the parameter S_{min} can be used to determine the minimum size of the regions, which will be measured in step two. Taking into account S_{min}, a peer will already start the measurement if it

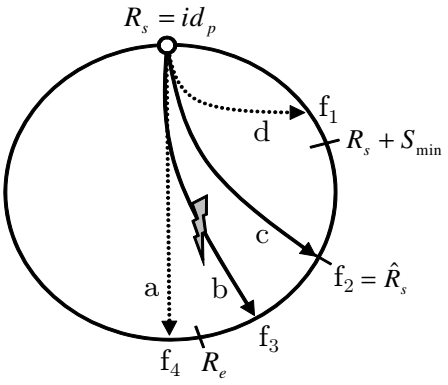


FIG. 3.1. *Visualization of the algorithm.*

does not know any more fingers in the region between the current $R_s + S_{min}$ and the current R_e. In this case finger f_1 would be disregarded, as illustrated by the dotted line (cf. d in Figure 3.1), since it points into the minimum measurement region. Parameter S_{min} is designed to adjust the trade off between the duration of the snapshot and the bandwidth needed at the collecting point. The larger the regions in step two, the longer the measurement will take. The smaller the regions, the more results are sent back to the CP.

Algorithm 2
The snapshot algorithm (first call $R_s = id_p$)

```
snapshot(R_s, R_e, S_min, CP)
send acknowledgment to the sender of the request
id_fm = max({id_f | id_f ∈ fingerlist ∧ id_f < R_e})
while id_fm > R_s + S_min do
  send snapshot(id_fm, R_e, S_min, CP) request to peer id_fm
  if acknowledgment from id_fm then
    call snapshot(id_p, id_fm − 1, S_min, CP) at local peer
    return //exit the function
  else
    remove id_fm from fingerlist
    id_fm = max({id_f | id_f ∈ fingerlist ∧ id_f < R_e})
  end if
end while
Ŝ = (R_e − R_s) / ⌈(R_e − R_s)/S_min⌉ //explanation see step two
call countingtoken(id_p, R_e, S_min, CP, ∅) at local peer
```

A detailed technical description of the procedure is given in Algorithm 2. Peer p will contact the closest finger to R_e until it does not know any more fingers in between $R_s + S_{min}$ and R_e. If so, it changes into step two and starts a measurement of this region calling the function countingtoken(id_p, R_e, S_{min}, CP, *result*) at the local peer.

3.2. Step 2: Measure a Specific Subpart. The basic idea behind the measurement of a specific subpart from R_s to R_e is very simple. The first peer creates a token, adds its local statistics, and passes the token to its immediate successor. The successor proceeds recursively until the first peer with an $id > R_e$ is reached. This peer sends the token back to the collecting point, whose IP is given in the parameter CP.

Ideally, each of the regions measured in step two would be of size S_{min} as specified by the user. The problem, however, is that the region from R_s to R_e is slightly larger than S_{min} according to step one of the algorithm. In fact, if the responsible peer did not know enough fingers, the region might even be significantly larger than S_{min}. The solution to this problem is to introduce checkpoints with a distance of S_{min} in the corresponding region. Results are sent to the CP every time the token passes a checkpoint instead of sending only one answer

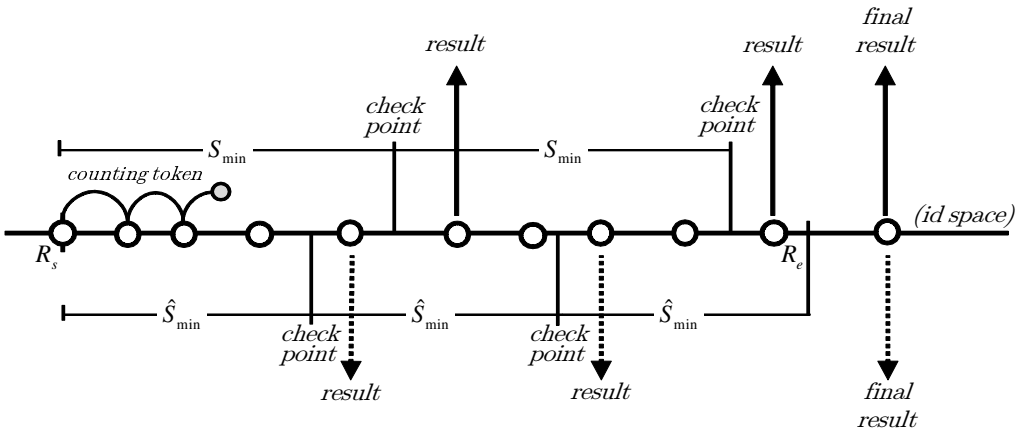


FIG. 3.2. *Results sent after each checkpoint.*

at the end of the region. This is illustrated in the upper part of Figure 3.2. The counting token is started at R_s. The first peer behind each checkpoint sends a *result* back to the CP as illustrated by the large solid arrows. The final *result* is still sent by the first peer with $id > R_e$.

A drawback of this solution is that the checkpoints might not be equally distributed in the region. In particular, the last two checkpoints might be very close to each other. We therefore recalculate the positions of the checkpoints according to the following equation:

$$\widehat{S}_{min} = \frac{R_e - R_s}{\left\lceil \frac{R_e - R_s}{S_{min}} \right\rceil}.$$

The new checkpoints can be seen in the lower part of Figure 3.2. The number of checkpoints remains the same, while their positions are moved in such a way, that the results are now sent at equal distance.

As can be seen at the end of Algorithm 2, the recalculation of S_{min} is already done in the first step, just before the counting token is started. A detailed description of the counting token mechanism is given in Algorithm 3. If a peer p receives a counting token it makes sure that its identifier is still within the measured region, i.e. $R_s \leq id_p \leq R_e$. If not, it sends a *result* back to the CP and stops the token. Otherwise it adds its local measurement to the token and tries to pass the token to its immediate successor. If it is the first peer behind one of the checkpoints, it sends an intermediate result back to the CP and resets the token.

As mentioned above the parameter S_{min} roughly determines the minimum size of the regions measured in step two. If S_{id} is the total size of the identifier space, there will be N_c counting tokens arriving at the CP, whereas:

$$2 \cdot \left\lceil \frac{S_{id}}{S_{min}} \right\rceil \geq N_c \geq \left\lceil \frac{S_{id}}{S_{min}} \right\rceil.$$

A more detailed analysis of the snapshot algorithm is given in Section 4 as well as in [1].

3.3. Collect Statistics. Generally speaking, there are two different kinds of statistics, which can be collected using the counting tokens. Either a simple mean value or a more detailed histogram. In the first case the counting token memorizes two variables, V_a for the accumulated value and V_n for the number of values. Each peer receiving the counting token adds its measured value to V_a and increases V_n by one. The sample mean can then be calculated at the CP as $\frac{\sum V_a}{\sum V_n}$. In case of a histogram, the counting token maintains a specific number of bins and their corresponding limits. Each peer simply increases the bin matching its measured value by one. If the measured value is outside the limits of the bins it simply increases the first or the last bin respectively.

There are numerous things that can be measured using the above mentioned methods. Table 3.1 summarizes some exemplary statistics and the kind of information which can be gained from them. The most obvious application is to count the number of hops for each counting token. On the one hand, this is a direct measure for the size of the overlay network. On the other hand, it also shows the distribution of the identifiers in the

Algorithm 3
The countingtoken algorithm (first call $R_s = id_p$)

countingtoken(R_s, R_e, S_{min}, CP, *result*)
send acknowledgment to the sender of the request
if $R_s \leq id_p \leq R_e$ **then**
  if $id_p > R_s + S_{min}$ **then**
    send *result* to CP
    $result = 0$
    $R_s = R_s + S_{min}$
  end if
  add local measurement to *result*
  $id_s = id$ of direct successor
  while 1 **do**
    send countingtoken(R_s, R_e, S_{min}, CP, *result*) request to direct successor id_s
    if acknowledgment **then**
      break
    else
      remove id_s from successor list
      $id_s = id$ of new direct successor
    end if
  end while
else
  send *result* to CP
end if

TABLE 3.1
Possible statistics gathered during snapshot

Statistic	Information gained
Number of hops per token	Size of the network, Distribution of the identifiers
Mean search delay	Performance of the algorithm
Sender $\stackrel{?}{=}$ predecessor	Overlay stability
Number of timeouts per token	Churn rate
Number of resources per peer	Fairness of the algorithm
Number of searches answered	User behavior
Bandwidth used per time unit	Maintenance overhead
Missing resources	Data integrity

identifier space. To gain information about the performance of the Chord algorithm, the mean search delay or a histogram for the search time distribution can be calculated and compared to expected values. Furthermore, Chord's stability can only be guaranteed as long as the successor and predecessor pointers of the individual peers match each other correspondingly. This invariant can be checked by counting the percentage of hops, where the sender of the counting token did not match the predecessor of the receiving peer. Additionally, the number of timeouts per token can be used to measure the current churn rate in the overlay network. The more churn there is, the more timeouts are going to occur due to outdated successor pointers. Similarly, the number of resources stored at each peer is a sign of the fairness of the Chord algorithm. The number of searches answered at each peer can likewise be used to get an idea of the search behavior of the end users. Finally, a peer can keep track of the number of missing resources to verify the integrity of the stored data. This can, e.g., be done counting the number of search requests which could not be answered by the peer.

All of the above statistics can be collected periodically to survey the time dependent status of the overlay. Note, that it is also possible to monitor a specific part of the overlay network by setting R_s and R_e accordingly. This can, e.g., be helpful if there are problems in a certain region of the overlay network and the operator needs to verify that his countermeasures have been successful.

4. Analysis and Optimizations. To analyze our algorithm we derive the duration of a snapshot (cf. Subsection 4.1) and the temporal distribution of the token arrival times at the CP (cf. Subsection 4.2).

4.1. Duration of a Snapshot. To calculate an estimate of the duration of a snapshot, we assume a scenario without any peers joining or leaving the network. It is quite straightforward to estimate the duration of step one, the signaling step. The last counting token which will be started is the one covering the region directly following the initiating peer. This is due to the fact, that the initiating peer will start its counting token no sooner than it divided the ring into separate regions. Before it initiates the counting token, it contacts its fingers until the first finger is closer to itself than S_{min}. The initiating peer has at most $\log_2(n)$ fingers and each of the fingers sends an acknowledgment, before the peer can go on with the algorithm. If T_O is the random variable describing one overlay hop, then the duration of step one of the algorithm is at most

$$D_{step1} = 2 \cdot \log_2(n) \cdot E[T_O]. \tag{4.1}$$

The worst case for step two would be that the initiating peer does not know any fingers and directly sends the counting token. This would take $n \cdot E[T_O]$, but is very unlikely to happen. In general, if there are n peers in the overlay, there are roughly $P_r = \frac{n \cdot S_{min}}{S_{id}}$ peers per region. Furthermore, in the worst case S_{min} is slightly larger than a power of two and the region covered by a counting token may become almost twice as large as S_{min}. Therefore a good estimate for the duration of the counting step of the algorithm is:

$$D_{step2} = 2 \cdot P_r \cdot E[T_O]. \tag{4.2}$$

This results in the following total duration of a snapshot:

$$D = \left(\log_2(n) + \frac{n \cdot S_{min}}{S_{id}} \right) \cdot 2 \cdot E[T_O]. \tag{4.3}$$

4.2. Token Arrival Time Distribution. To get a rough estimate for the distribution of the arrival times of the counting tokens at the CP, we consider the special case where the size of the overlay $n = 2^g$ is a power of two and S_{min} is such that $N_r = 2^h$ with $h < g$. Furthermore, we assume that the individual peers are located at equal distances on the ring as shown in Figure 4.1.

It can be shown, that in this case $h = \log_2(N_r)$ is the number of overlay hops it takes until the first counting token is started during a snapshot. Similarly, it takes $2 \cdot h$ hops until the last counting token is started according to our assumptions. The probability p_i that a counting token is started after exactly i hops for $i = h, h+1, ..., 2 \cdot h$ can be calculated as:

$$p_i = \frac{\binom{h}{i-h}}{\sum_{x=h}^{2 \cdot h} \binom{h}{x-h}}. \tag{4.4}$$

The above considerations are nontrivial, but can nicely be explained using the simple example shown in Figure 4.1, where $g = 4$, $h = 2$, and therefore $n = 2^4$ and $N_r = 2^2$. The solid arrows in the figure show the messages of the signaling step, the dotted arrows the corresponding acknowledgments. The numbers next to the arrows represent the number of overlay hops, which have passed since the beginning of the snapshot.

In the example, peer A starts a snapshot of the entire ring. It sends a request to B to cover the region between B and A. Peer B sends an acknowledgment back to A and a simultaneous request to C to cover the region from C to A. C has no fingers outside its minimum measurement region and starts the first counting token after $h = 2$ overlay hops. Simultaneously, it sends an acknowledgment back to B. Peer B then starts its counting token after 3 overlay hops. In the meantime A signals D to cover the region from D to B. Peer D immediately starts its counting token after a total of 3 overlay hops. Peer A waits for the final acknowledgment and starts its counting token after $4 = 2 \cdot h$ overlay hops. Summarizing the above, there are four counting tokens started after 2, 3, 3, and 4 overlay hops respectively.

According to our assumptions, each counting token needs exactly $P_r = 4$ hops to travel the corresponding region and one final hop to arrive at the CP. A rough estimate for the distribution of the arrival times of the counting tokens at the CP is therefore given by the phase diagram shown in Figure 4.3. It indicates that the signaling step takes i overlay hops with a probability p_i for $i = h, h+1, ..., 2 \cdot h$, which is followed by P_r hops of the counting token and the final hop to report the result back to the CP.

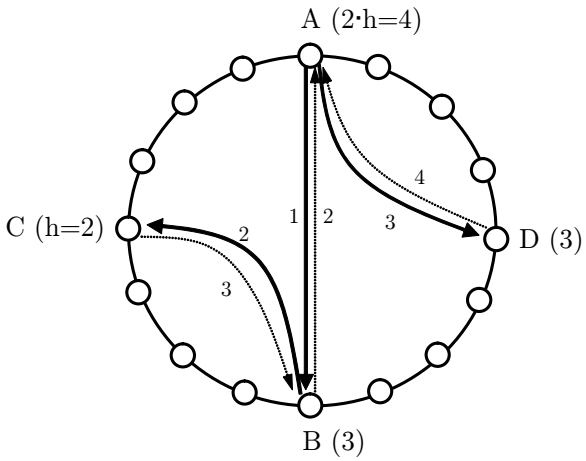


FIG. 4.1. *Starting times of the counting tokens for* $N_r = 2^2$ *and* $n = 2^4$.

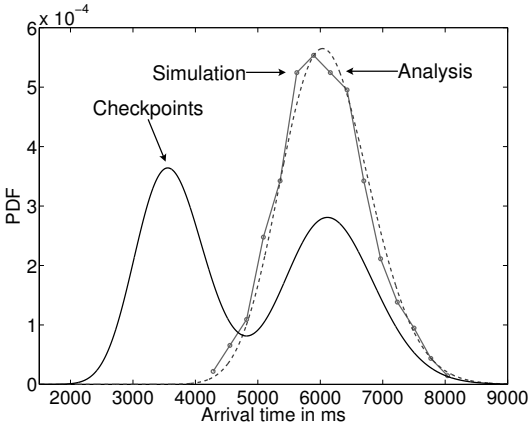


FIG. 4.2. *Probability density function of the token arrival time.*

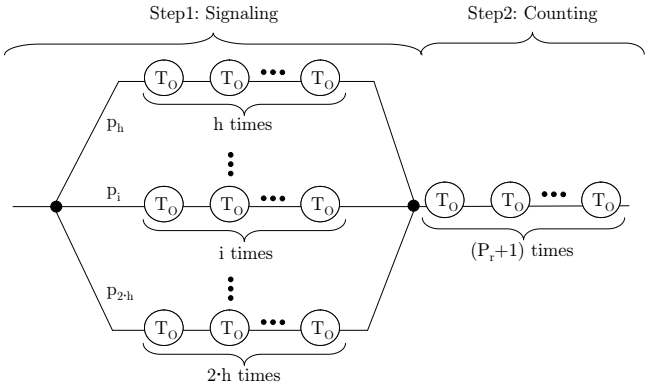


FIG. 4.3. *Phase diagram of the token arrival time distribution.*

To validate our analytical results, we simulated a Chord ring of size $n = 2^{15}$ with $S_{min} = 2^9$ according to the above assumptions. Figure 4.2 shows the probability density function of the token arrival times at the CP. Obviously, the curves match very well and the binomial distribution of the duration of step one can be recognized. So far, in our example each peer has a finger at an exact distance of S_{min}. In reality, however, the finger would sit at a slightly different position, which again would result in an additional checkpoint at the middle of the region. The curve labeled "Checkpoints" corresponds to a slightly modified phase diagram, which adds an intermediate result in the middle of the measurement region. The first rise of the probability density function (pdf) therefore represents the intermediate results sent back to the CP at the checkpoint. The second rise still represents the regular results at the end of the region. In the following section we will present simulations of more realistic scenarios including churn and timeouts.

5. Results. The results in this section were obtained using our ANSI-C simulator, which incorporates a detailed yet slightly modified Chord implementation. A good description of the general simulation model can be found in [5, 6]. In this work an overlay hop is modeled using an exponentially distributed random variable with a mean of 80ms. The results considering churn are generated using peers, which stay online and offline for an exponentially distributed period of time with a mean as indicated in the corresponding description of the figures.

The snapshot algorithm takes one single input argument S_{min} which directly translates into $N_r = \left\lceil \frac{S_{id}}{S_{min}} \right\rceil$, the number of areas the overlay will be divided into. This parameter influences the duration of the snapshot as well as the number of results arriving at the central collecting point. Figure 5.1 shows the distribution of the

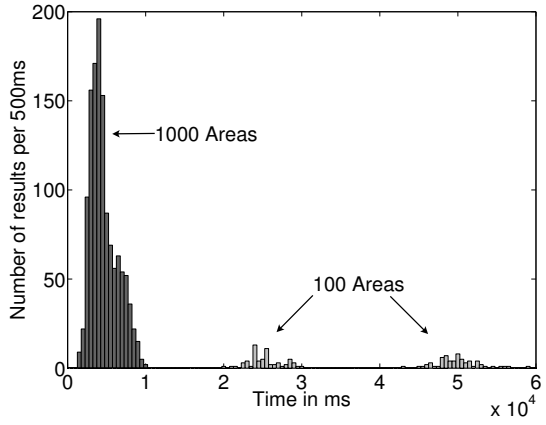


FIG. 5.1. *Arrival times of the results.*

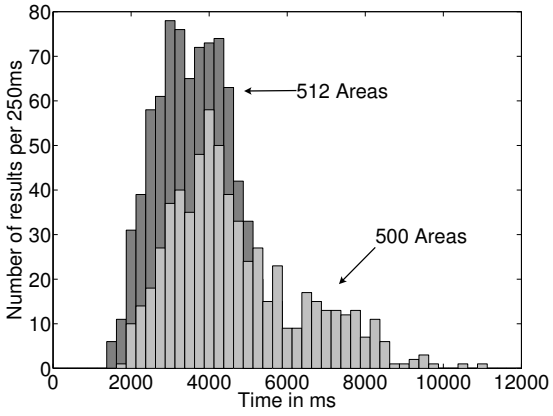


FIG. 5.2. *Influence of N_r for 20000 peers.*

arrival times of the results in an overlay of 40000 peers using $N_r = 1000$ and $N_r = 100$ areas in times of no churn. Obviously, the more areas the overlay is divided into, the faster the snapshot is completed. While the snapshot using 1000 areas was finished after about ten seconds, the snapshot with 100 areas took about one minute. In exchange the latter snapshot produces significantly smaller bandwidth spikes at the CP. The two elevations of the second histogram correspond to the intermediate results (first elevation) and the final results at the end of the measured subpart (second elevation). Note that the final results arrive about twice as late as the intermediate results. The first step of the algorithm uses the fingers to divide the ring into subparts. Since the distance between a peer and its fingers is always slightly larger than a power of two it is usually not possible to divide the ring exactly into the desired number of areas. In fact it is very likely, that a peer stops the recursion and starts its measurement once it contacted its xth finger, where $2^{x-1} < S_{min} = \frac{S_{id}}{N_r} \leq 2^x$. That is, the recursion stops at finger x with id_{f_x}, whereas the distance between the peer and this specific finger might almost be twice as large as the desired S_{min}. It is therefore advisable to choose N_r as a power of two itself in order to ensure that id_{f_x} is only slightly larger than $id_p + S_{min}$. Figure 5.2 shows the different arrival times of the results for $N_r = 512$ and $N_r = 500$ in an overlay of 20000 peers without churn. The skewed shape of the histogram in the foreground results from the fact that 500 is slightly smaller than a power of two, which in turn makes S_{min} slightly larger than a power of two. In this case it is likely that the peer has a finger just before the end of the minimum measurement region $id_p + S_{min}$. Thus, finger x sits at a distance of about twice S_{min} from the peer. The resulting counting token will therefore travel a distance of about twice S_{min} as well.

A more detailed analysis of the influence of N_r can be found in Figure 5.3, which shows the number of results received at the CP in dependence of N_r. As shown in [1], N_c, the number of counting tokens sent to the CP, is limited by $2 \cdot N_r > N_c \geq N_r$. The straight lines in the figure show the corresponding limits. The solid and dotted curves represent the results obtained for 20000 and 10000 peers, respectively. The number of results sent to the CP is within the calculated limits and independent of the overlay size. The curves roughly resemble the shape of a staircase, whereas the steps are located at powers of two. There are two main reasons for this behavior. First of all, the average counting token sends two results back to the CP, one intermediate result and the final result at the end of the measurement region. Hence, the smaller the region covered by the average counting token, the more results arrive at the CP. As explained above, the closer N_r gets to a power of two, the smaller the region covered by the average counting token. This accounts for the first part of the rise of the number of results received at the CP.

The distribution of the arrival times of the results is also influenced by the current size of the network. The larger the network, the more peers are within one region. However, the more peers are within one region, the more hops each counting token has to make, before it can send its results back to the CP. Figure 5.4 shows the token arrival time distribution for three different overlay sizes of 10000, 20000, and 40000 peers, respectively. We did not generate any churn in this scenario and set $N_r = 512$ areas. As expected, the larger the overlay network, the longer the snapshot is going to take. However, the curves are not only shifted to the right, but also differ in shape. This can again be explained by the increasing number of hops per counting token.

As mentioned above, the average counting token sends two results back to the CP, whereas the checkpoints are equally spaced. Thus, the final result takes twice as many hops as the intermediate result. In a network of

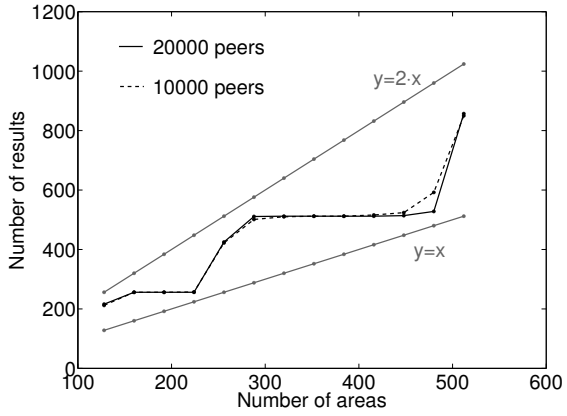


FIG. 5.3. *Number of results received at the CP.*

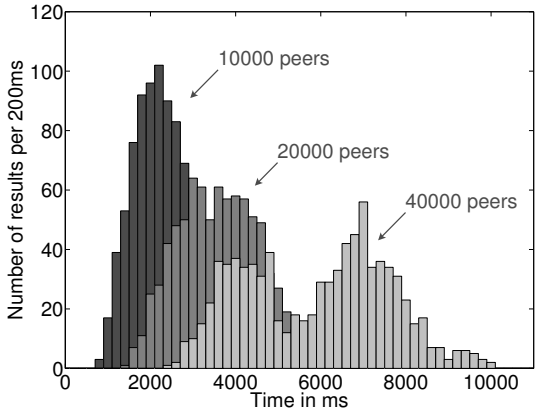


FIG. 5.4. *Arrival times of the results at the CP.*

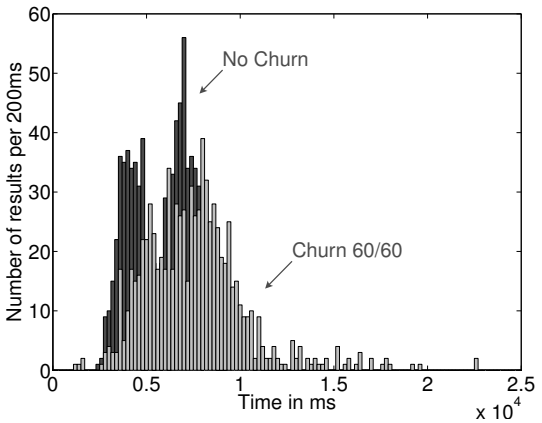


FIG. 5.5. *Influence of churn on the traffic pattern at the CP.*

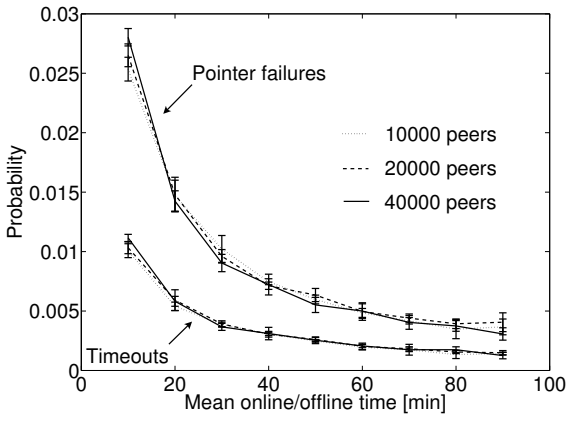


FIG. 5.6. *Relative frequency of timeouts and pointer failures.*

10000 peers there are approximately 20 peers in each of the 512 regions. The intermediate results are therefore sent after about 10 hops, the final results after about 20 hops, respectively. The two corresponding elevations in the histogram overlap in such a way, that they build a single elevation. In a network of 40000 peers, however, there are approximately 78 peers in each of the 512 regions. The intermediate results are therefore sent after about 39 hops, the final results after about 78 hops, respectively. The difference between these two numbers is large enough to account for the two elevations of the histogram in the foreground of Figure 5.4. Note, that all curves are shifted to the right as compared to the mere hop count since it takes some time for the signaling step until the counting tokens can be started. In practice the current size of the overlay can be estimated to be able to choose an appropriate value for N_r as suggested in [2].

The arrival time of the results at the CP is also affected by the online/offline behavior of the individual peers. To study the influence of churn we consider 80000 peers with an exponentially distributed online and offline time, each with a mean of 60 minutes. This way, there are 40000 peers online on average, which makes it possible to compare the results to those obtained using 40000 peers without churn. Figure 5.5 shows the corresponding histograms.

As a result of churn in the system, the two elevations of the original histogram become noticeably blurred and the snapshot is slightly delayed. This is due to the inconsistencies in the successor and finger lists of the peer as well as the timeouts which occur during the forwarding of the counting tokens. In return the spike in the diagram and thus the required bandwidth at the CP becomes smaller.

It is easy to show, that the probability to lose a token is almost negligible [1]. Therefore, a more meaningful method to measure the influence of churn is to regard the number of timeouts which occur during a snapshot. Furthermore, the influence of churn on the stability of the overlay network can be studied looking at the

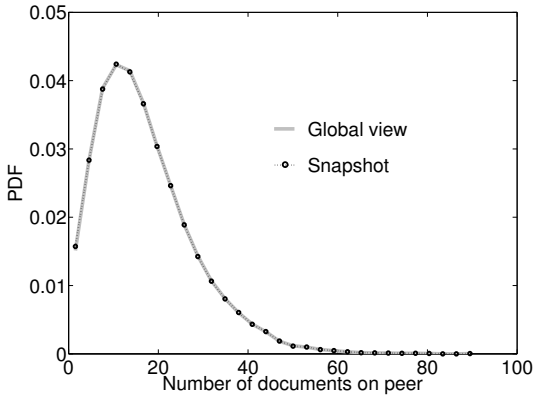


FIG. 5.7. *Results of a snapshot compared to the global view.*

frequency at which the predecessor pointer of a peer's successor does not match the peer itself. Figure 5.6 plots the relative frequency of timeouts and pointer failures against the mean online/offline time of a peer. The smaller the online/offline time of a peer, the more churn is in the system.

The results represent the mean of several simulation runs, whereas the error bars show the 95 percent confidence intervals. The relatively small percentage of both timeouts and failures is to some extent implementation specific. More interesting, however, is the exponential rise of the curves under higher churn rates. The shape of both curves is independent of the size of the overlay and only affected by the current churn rate. The curve can therefore be used to map the frequency of timeouts or failures measured in a running system to a specific churn rate.

Until now, we only regarded the traffic pattern at the central collecting point. From an operator's point of view, however, it is more important to know, whether the snapshot itself is meaningful. To validate the accuracy of the snapshot algorithm, we again simulated an overlay network with 80000 peers, each with a mean online/offline time of 60 minutes. Due to the properties of the hash function and the churn behavior in the system the number of documents on a single peer can be regarded as a random variable. The measurement we are interested in is the corresponding pdf in order to see whether the distribution of the documents among the peers is fair or not. The pdf was measured using our snapshot algorithm as explained in Section 3.3. The result of the snapshot is compared to the actual pdf obtained using the global view of our discrete event simulator (c.f. Figure 5.7). The two curves are almost indistinguishable from each other. The same is true for all the other statistics shown in Table 3.1, like the current size of the system or the average bandwidth used per time unit. That is, the snapshot provides the operator with a very accurate picture of the current state of its system. This nicely demonstrates that the results obtained by the snapshot can be used to better understand the performance of the running p2p system. The multiple possibilities to interpret the collected data are well beyond the scope of this paper.

6. Conclusion. One of the main reasons that telecommunication carriers are still hesitant to build p2p applications is the lack of control a provider has over the running system. In this paper we introduced an entirely distributed and scalable algorithm to monitor a Chord based p2p network at runtime. The load generated during the snapshot is evenly distributed among the peers of the overlay and the algorithm itself is easy to configure. It only takes one input parameter, which influences the trade-off between the duration of the snapshot and the bandwidth required at the central server which collects the results. In general it takes less than one minute to create a snapshot of a Chord ring consisting of 40000 peers. We performed a mathematical analysis of the basic mechanisms and provided a simulative study considering realistic user behavior.

The algorithm is resistant to instabilities in the overlay network (churn) and provides the operator with a very accurate picture of the current state of its system. It offers the possibility to monitor the entire overlay network or to concentrate on a specific part of the system. The latter is especially useful if a problem occurred in a specific part of the system and the operator wants to assure that his countermeasures have been successful.

REFERENCES

[1] A. Binzenhöfer, G. Kunzmann, and R. Henjes, *A Scalable Algorithm to Monitor Chord-based P2P Systems at Runtime*, Tech. Report 373, University of Würzburg, November 2005.
[2] A. Binzenhöfer, D. Staehle, and R. Henjes, *On the Fly Estimation of the Peer Population in a Chord-based P2P System*, in ITC19, Beijing, China, September 2005.
[3] A. Binzenhöfer and P. Tran-Gia, *Delay Analysis of a Chord-based Peer-to-Peer File-Sharing System*, in ATNAC 2004, Sydney, Australia, December 2004.
[4] U. B. C. S. Division, *The oceanstore project*. URL: http://oceanstore.cs.berkeley.edu/.
[5] G. Kunzmann, A. Binzenhöfer, and R. Henjes, *Analysis of the Stability of the Chord protocol under high Churn Rates*, in 6th Malaysia International Conference on Communications (MICC) icw International Conference on Networks (ICON), Kuala Lumpur, Malaysia, November 2005.
[6] G. Kunzmann, R. Nagel, and J. Eberspächer, *Increasing the reliability of structured p2p networks*, in 5th International Workshop on Design of Reliable Communication Networks, Island of Ischia, Italy, October 2005.
[7] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, *Comparing the performance of distributed hash tables under churn*, in Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04), San Diego, CA, February 2004.
[8] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu., *P2P Computing*, Tech. Report HPL-2002-57, Hewlett Packard Lab, 2002.
[9] D. L. Oppenheimer, V. Vatkovskiy, H. Weatherspoon, J. Lee, D. A. Patterson, and J. Kubiatowicz, *Monitoring, analyzing, and controlling internet-scale systems with acme*, CoRR, cs.DC/0408035 (2004).
[10] V. N. Padmanabhan, S. Ramabhadran, and J. Padhye, *Netprofiler: Profiling wide-area networks using peer cooperation*, in Fourth International Workshop on Peer-to-Peer Systems (IPTPS), Ithaca, NY, USA, February 2005.
[11] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel, *Using queries for distributed monitoring and forensics*, in 1st Eurosys, Leuven, Belgium, March 2006.
[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, in SIGCOMM 2001, San Diego, USA, August 2001.
[13] D. Stutzbach and R. Rejaie, *Capturing accurate snapshots of the gnutella network*, in INFOCOM 2005, Miami, USA, March 2005, pp. 2825–2830.
[14] S. Technologies, *Skype*. URL: http://www.skype.com.

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 251–262. http://www.scpe.org
ISSN 1895-1767

USING GRIDS FOR EXPLOITING THE ABUNDANCE OF DATA IN SCIENCE

EUGENIO CESARIO*AND DOMENICO TALIA*,†

Abstract. Digital data volumes are growing exponentially in all sciences. To handle this abundance in data availability, scientists must use data analysis techniques in their scientific practices and solving environments to get the benefits coming from knowledge that can be extracted from large data sources. When data is maintained over geographically remote sites the computational power of distributed and parallel systems can be exploited for knowledge discovery in scientific data. In this scenario the Grid can provide an effective computational support for distributed knowledge discovery on large datasets. In particular, Grid services for data integration and analysis can represent a primary component for e-science applications involving distributed massive and complex data sets. This paper describes some research activities in data-intensive Grid computing. In particular we discuss the use of data mining models and services on Grid systems for the analysis of large data repositories.

Key words: e-science, knowledge discovery, grid, parallel data mining, distributed data mining, grid-based data mining

1. Introduction. The past two decades have been dominated by the advent of increasingly powerful and less expensive ubiquitous computing, as well as the appearance of the World Wide Web and related technologies [12]. Due to such advances in information technology and high performance computing, digital data volumes are growing exponentially in many fields of human activities. This phenomenon concerns scientific disciplines, as well as industry and commerce. Such technological development has also generated a whole new set of challenges: the world is drowning in a huge quantity of data, which is still growing very rapidly both in the volume and complexity.

Jim Gray in some talks in 2006 identified four chronological steps for the methodologies employed by scientists for discoveries. The first step occurred thousand years ago, when science was empirical and it was oriented to just describe natural phenomena. The second one is temporally located around a few hundred years ago, when a theoretical branch was born, aimed at formulating some general models describing the empirical knowledge. The third step occurred in the latest few decades, when a computational branch started up and complex phenomena started to be simulated by the resources made available by the current technology. Finally, the fourth step is run today, when scientists are working to unify theories, experiments and simulations with data processing and exploration to extract knowledge hidden in it.

The abundance of digitally stored data require to consider in detail this phenomenon. In particular, there are two important trends, technological and methodological, which seem to particularly distinguish the new, information-rich science from the past:

- *Technological.* There is a lot of data collected and warehoused in various repositories distributed over the world: data can be collected and stored at high speeds in local databases, from remote sources or from the our galaxy. Some examples include data sets from the fields of medical imaging, bio-informatics, remote sensing and (as very innovative aspect) several digital sky surveys. This implies a need for reliable data storage, networking, and database-related technologies, standards and protocols.
- *Methodological.* Huge data sets are hard to understand, and in particular data constructs and patterns present in them cannot be comprehended by humans directly. This is a direct consequence of the growth in complexity of information, and mainly its multi-dimensionality. For example, a computational simulation can generate terabytes of data within a few hours, whereas human analysts may take several weeks to analyze these data sets. For such a reason, most of data will never be read by humans, rather they are to be processed and analyzed by computers.

We can summarize what we foresaid as follows: whereas some decades ago the main problem was the lack of information, the challenge now seems to be *(i) the very large volume of information* and *(ii) the associated complexity to process for extracting significant and useful parts or summaries.*

Nevertheless, the first aspect does not represent a limitation or a problem for the scientific community: current data storage, architectural solutions and communication protocols provide a reliable technological base to collect and store such abundance of data in an efficient and effective way. Moreover, the availability of high throughput scientific instrumentation and very inexpensive digital technologies facilitated this trend from both technological and economical view point. On the other hand, the computational power of computers is

*ICAR-CNR, Via P. Bucci 41C, 87036 Rende (CS), Italy (cesario@icar.cnr.it, talia@icar.cnr.it).
†DEIS-University of Calabria, Via P. Bucci 41C, 87036 Rende (CS), Italy (talia@deis.unical.it).

not growing as fast as the demand of such a data computation requires, and this represents a limit for the knowledge that potentially could be extracted. As an additional aspect, we have to consider that storage costs are currently decreasing faster than computing costs, and this trend makes things worse.

For example, the impact of foresaid issues in the biological field is well described in [20]. It points out that the emergence of genome and post-genome technology has made huge amount of data available, demanding a proportional support of analysis. Nevertheless, an important factor to be considered is that the number of available complete genomic sequences is doubling almost every 12 months, whereas according to Moore's law available compute cycles (i. e., computational power) double every 18 months. Additionally, we have to consider that analysis of genomic sequences require binary comparisons of the genes involved in it. As a direct consequence of that, the computational overhead is very very high. We can see the impact of such issues in Figure 1.1 (source: [20]), which contrasts the number of genetic sequences obtained with the number of annotations generated. The figure shows that the knowledge (annotations, models, patterns) has a sub-linear rate with respect to the the available data sequences which they are extracted from.

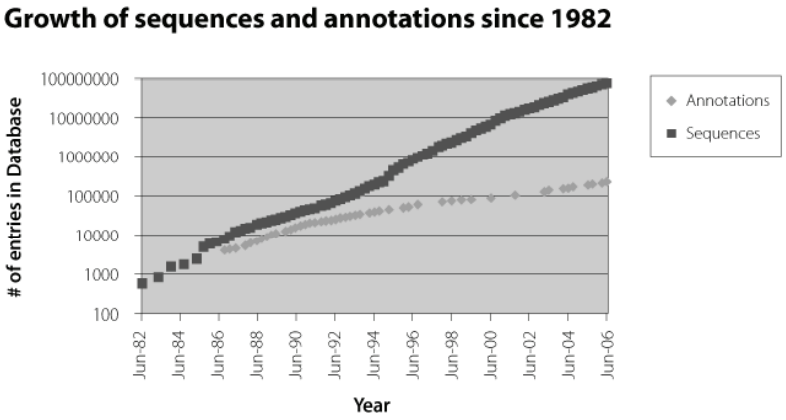


FIG. 1.1. *Growth of sequences and annotations since 1982 (Source: [20])*

To handle this abundance in data availability (whose rate of production often far outstrips our ability to analyze), applications are emerging that explore, query, analyze, visualize, and in general, process very large-scale data sets: they are named *data intensive applications*. Computational science is evolving toward data intensive applications that include data integration and analysis, information management, and knowledge discovery. In particular, knowledge discovery in large data repositories can find what is interesting in them by using data mining techniques. Data intensive applications in science help scientists in hypothesis formation and give them a support on their scientific practices and solving environments, getting the benefits coming from knowledge that can be extracted from large data sources.

When data is maintained over geographically distributed sites the computational power of distributed and parallel systems can be exploited for knowledge discovery in scientific data. Parallel and distributed data mining algorithms are suitable to such a purpose. Moreover, in this scenario the *Grid* can provide an effective computational support for data intensive application and for knowledge discovery from large and distributed datasets. Grid computing is receiving an increasing attention from the research community, watching at this new computing infrastructure as a key technology for solving complex problems and implementing distributed high-performance applications [14].

Today many organizations, companies, and scientific centers produce and manage large amounts of complex data and information. Climate, astronomic, and genomic data together with company transaction data are just some examples of massive amounts of digital data that today must be stored and analyzed to find useful knowledge in them. This data and information patrimony can be effectively exploited if it is used as a source to produce knowledge necessary to support decision making. This process is both computationally intensive, collaborative, and distributed in nature. The development of data mining software for Grids offers tools and environments to support the process of analysis, inference, and discovery over distributed data available in many scientific and business areas. The creation of frameworks on top of data and computational Grids is the

enabling condition for developing high-performance data mining tasks and knowledge discovery processes, and it meets the challenges posed by the increasing demand for power and abstractness coming from complex data mining scenarios in science and engineering. For example, some projects described in Section 2 such as NASA Information Grid, TeraGrid, and Open Science Grid use the computational and storage facilities in their Grid infrastructures to mine data in a distributed way. Sometime in these projects are used ad hoc solutions for data mining, in other cases generic middleware is used on top of basic Grid toolkits. As pointed out by William E. Johnston in [19], the use of general purpose data mining tools may effectively support the analysis of massive and distributed data sets in large scale science and engineering.

The Grid allows to federate and share heterogeneous resources and services such as software, computers, storage, data, networks in a dynamic way. Grid services can be the basic element for composing software and data elements, and executing complex applications on Grid and Web systems. Today the Grid is not just compute cycles, but it is also a distributed data management infrastructure. Integrating those two features with "smart" algorithms we can obtain a knowledge-intensive platform. The driving Grid applications are traditionally high-performance and data intensive applications, such as high-energy particle physics, and astronomy and environmental modeling, in which experimental devices create large quantities of data that require scientific analysis.

In the latest years many significant Grid-based data intensive applications and infrastructures have been implemented. In particular, the service-based approach is allowing the integration of Grid and Web for handling with data. We will briefly report some of these applications in the first of the paper; then we discuss about the use of high performance data mining techniques for science in Grid platforms.

The rest of the paper is organized as follows. Section 2 describes some Grid-based data intensive projects and applications. Section 3 gives an overview of approaches for parallel, distributed and Grid-based data mining techniques. Section 4 introduces the *Knowledge Grid*, a reference software architecture for geographically distributed knowledge discovery systems. The Section 5 gives concluding remarks.

2. Grid Technologies for dealing with Scientific data. Several scientific teams and communities are using Grid technology for dealing with intensive applications aimed at scientific data processing. As examples of this approach, in the following we shortly describe some of them.

2.1. The DataGrid Project: Grid for Physics. The European *DataGrid* [11] is a project funded by the European Union with the aim of setting up a computational and data-intensive Grid of resources for the analysis of data coming from scientific exploration. The main goal of the project is to coordinate resource sharing, collaborative processing and analysis of huge amounts of data produced and stored by many scientific laboratories belonging to several institutions. It is made effective by the development of a technological infrastructure enabling scientific collaborations where researchers and scientists will perform their activities regardless of geographical location. The project develops scalable software solutions in order to handle many PBs[1] of distributed data, tens of thousand of computing resources (processors, disks, etc.), and thousands of simultaneous users from multiple research institutions. The three real data intensive computing applications areas covered by the project are biology/medical, earth observation and particle physics. In particular, the last one is oriented to answer longstanding questions about the fundamental particles of matter and the forces acting between them. The goal is to understand why some particles are much heavier than others, and why particles have mass at all. To that end, CERN[2] has built the *Large Hadron Collider (LHC)*, the most powerful particle accelerator ever conceived, that generates huge amounts of data. It is estimated that LHC generates approximately 1 GB/sec and 10 PB/year of data. The DataGrid Project provided the solution for storing and processing this data, based on a multi-tiered, hierarchical computing model for sharing data and computing power among multiple institutions. In particular, a Tier-0 centre is located at CERN and is linked by high speed networks to approximately ten major Tier-1 data processing centres. These fan out the data to a large number of smaller ones (Tier-2).

The DataGrid project ended on March 2004, but many of the products (technologies, infrastructure, etc.) are used and extended in the *EGEE* project. The *Enabling Grids for E-sciencE (EGEE)* [13] project brings together scientists and engineers from more than 240 institutions in 45 countries world-wide to provide a seamless Grid infrastructure for e-Science that is available to scientists 24 hours/day. Expanding from originally two

[1] $PetaByte = 10^6 GigaBytes$

[2] European Organization for Nuclear Research

scientific fields, high energy physics and life sciences, EGEE now integrates applications from many other scientific fields, ranging from geology to computational chemistry. The EGEE Grid consists of over 36,000 CPUs available to users 24 hours a day, 7 days a week, in addition to about 5 PB disk of storage, and maintains 30,000 concurrent jobs on average. Having such resources available changes the way scientific research takes place. The end use depends on the users' needs: large storage capacity, the bandwidth that the infrastructure provides, or the sheer computing power available. Generally, the EGEE Grid infrastructure is ideal for any scientific research especially where the time and resources needed for running the applications are considered impractical when using traditional IT infrastructures.

2.2. The NASA Information Power Grid (IPG) Infrastructure. The *NASA's Information Power Grid (IPG)* [18] is a high-performance computing and data grid built primarily for use by NASA scientists and engineers. The IPG has been constructed by NASA between 1998 and the present making heavy use of Globus Toolkit components to provide Grid access to heterogeneous computational resources managed by several independent research laboratories. Scientists and engineers access the IPG's computational resources from any location with Grid interfaces providing security, uniformity, and control. Scientists beyond NASA can also use familiar Grid interfaces to include IPG resources in their applications (with appropriate authorization). The IPG infrastructure has been and is being used by numerous scientific and engineering efforts both within and beyond NASA. Some of its most important applications are computational fluid dynamics and meteorological data mining.

2.3. TeraGrid. *TeraGrid* [29] is an open scientific discovery infrastructure combining leadership class resources (including supercomputers, storage, and scientific visualization systems) at nine partner sites to create an integrated, persistent computational resource. It is coordinated by the Grid Infrastructure Group (GIG) at the University of Chicago. Using high-performance network connections, the TeraGrid integrates high-performance computers, data resources and tools, and high-end experimental facilities around the country. Currently, TeraGrid resources include more than 250 teraflops of computing capability and more than 30 PBs of online and archival data storage, with rapid access and retrieval over high-performance networks. Researchers can also access more than 100 discipline-specific databases. With this combination of resources, the TeraGrid is one of the world's largest and most comprehensive distributed Grid infrastructure for open scientific research.

2.4. NASA and Google. Recently NASA initiated a joint project with Google, Inc. for applying Google search technology to help scientists to process, organize, and analyze the large-scale streams of data coming from the Large Synoptic Survey Telescope (LSST), located in Chile. When completed, the LSST will generate over 30 terabytes of multiple color images of visible sky each night. Google will collaborate with LSST to develop search and data access techniques and services that can process, organize and analyze the very large amounts of data coming from the instrument's data streams in real time. The engine will create "data images" for scientists to view significant space events and extract important features from them. This joint project will show how complex data management techniques generally used in search engines can be exploited for scientific discovery.

In the general framework of this collaboration, the main NASA's goal is to make its huge stores of data collected during everything from spacecraft missions, moon landings to landings on Mars to orbits around Jupiter—available to scientists and the public. Some of the data can already be found on NASA's Web site but exploiting Google techniques with high performance facilities, this data will be accessible in an easy way.

2.5. Open Science Grid. The *Open Science Grid* [24] is a collaboration of science researchers, software developers and computing, storage and network providers. It gives access to shared resources worldwide to scientists (from universities, national laboratories and computing centers across the United States). The Open Science Grid links storage and computing resources at more than 30 sites across the United States. The OSG works actively with many partners, including Grid and network organizations and international, national, regional and campus Grids, to create a Grid infrastructure that spans the globe. Scientists from many different fields use the OSG to advance their research. Applications of OSG project are active in various areas of science, like particle and nuclear physics, astrophysics, bioinformatics, gravitational-wave science, mathematics, medical imaging and nanotechnology. OSG resources include thousands of computers and 10 of terabytes of archival data storage.

2.6. myExperiment. *myExperiment* [22] is a collaborative research environment which enables scientists to share, reuse and repurpose experiments. It is based on the idea that scientists usually prefer to share

experimental results than data. myExperiment has been influenced by social networking programs such as Wired and Flickr, and is based on the mySpace infrastructure. myExperiment enables scientists to share and use workflows and reduce time-to-experiment, share expertise and avoid reinvention. myExperiment creates an environment for scientists to adopt Grid technologies, where they can define, when they share data, with whom they share it and how much of it can be accessed. The myExperiment project mainly focuses its applications at case studies for the specific areas of astronomy, bio-informatics, chemistry and social science.

2.7. National Virtual Observatory. The *National Virtual Observatory* [23] is a new research project whose goal is to make all the astronomical data in the world quickly and easily accessible by anyone. Such a project enables a new way of doing astronomy, moving from an era of observations of small, carefully selected samples of objects in one or a few wavelength bands, to the use of multi-wavelength data for millions, or even billions of objects. Such large collection of data makes researchers able to discover subtle, but significant, patterns in statistically rich and unbiased databases, and to understand complex astrophysical systems through the comparison of data to numerical simulations. With the *National Virtual Observatory* (NVO), astronomers explore data that others have already collected, finding new uses and new discoveries in existing data. NVO enables astronomers to do a new type of research that, combined with traditional telescope observations, will lead to many new and interesting discoveries. It is worth noticing that the NVO has proposed to exploit the computational resources of the TeraGrid project (described in the Section 2.3), in order to enable astronomers in the exploration and analysis of the physical processes that drive the formation and evolution of our universe, and encouraging new ways to use supercomputing facilities for science.

2.8. Southern California Earthquake Center. The *Southern California Earthquake Center* project [26] is aimed at developing new computing capabilities, that can lead to better forecasts of when and where earthquakes are likely to occur in Southern California, and how the ground will shake as a result. The final goal is to improve mathematical models about the structure of the Earth and how the ground moves during earthquakes. The project team includes collaborating researchers from Southern California Earthquake Center (SCEC), the Information Sciences Institute (ISI) at USC, the San Diego Supercomputing Center (SDSC), the Incorporated Institutions for Seismology (IRIS), and the United States Geological Survey (USGS). The project heavily exploits Grid technologies, allowing scientists to organize and retrieve information stored throughout the country, and giving advantages of the processing power of a network of many computers.

3. Data Mining and Knowledge Discovery. After discussing significant data management issues and projects, here we focus on data mining techniques for knowledge discovery in large scientific data repositories. *Data Mining* is the semi-automatic discovery of patterns, models, associations, anomalies and (statistically significant) structures hidden in data. Traditional data analysis is assumption-driven, that is the hypothesis is formed and validated against the data. Data mining, in contrast, is discovery-driven, in the sense that the patterns (and models) are automatically extracted from data. Data mining founds its application to several scientific and engineering domains, including astrophysics, medical imaging, computational fluid dynamics, biology, structural mechanics, and ecology.

From a scientific viewpoint, data can be collected by many sources: remote sensors on a satellite, telescope scanning the sky, microarrays generating gene expression data, scientific simulations, etc. Moreover, in such infrastructures data are collected and stored at enormous speeds (GBs/hour). Both such aspects imply that scientific application have to deal with massive volume of data.

Mining large data sets requires powerful computational resources. A major issue in data mining is scalability with respect to the very large size of current-generation and next-generation databases, given the excessively long processing time taken by (sequential) data mining algorithms on realistic volumes of data. In fact, data mining algorithms working on very large data sets take a very long time on conventional computers to get results. In order to improve performances, some parallel and distributed approaches have been proposed.

Parallel computing is a viable solution for processing and analyzing data sets in reasonable time by using parallel algorithms. High performance computers and parallel data mining algorithms can offer a very efficient way to mine very large data sets [27], [28] by analyzing them in parallel. Under a data mining perspective, such a field is known as *parallel data mining* (*PDM*).

Beyond the development of knowledge discovery systems based on parallel computing platforms, a lot of work has been devoted to design systems able to handle and analyze multi-site data repositories. Mining knowledge from data captured by instruments, scientific analysis, simulation results that could be distributed over the world, questions the suitability of centralized architectures for large-scale knowledge discovery in a networked

environment. The research area named *distributed data mining* offers an alternative approach. It works by analyzing data in a distributed fashion and pays particular attention to the trade-off between centralized collection and distributed analysis of data. This technology is particularly suitable for applications that typically deal with very large amount of data (e.g., transaction data, scientific simulation and telecommunication data), which cannot be analyzed in a single site on traditional machines in acceptable times.

Grid technology integrates both distributed and parallel computing, thus it represents a critical infrastructure for high-performance distributed knowledge discovery. Grid computing was designed as a new paradigm for coordinated resource sharing and problem solving in advanced science and engineering applications. For these reasons, Grids can offer an effective support to the implementation and use of knowledge discovery systems by *Grid-based Data Mining* approaches.

In the following parallel, distributed and Grid-based data mining are discussed.

3.1. Parallel Data Mining. *Parallel Data Mining* is concerned with the study and application of data mining analysis done by parallel algorithms. The key idea underlying such a field is that parallel computing can give significant benefits in the implementation of data mining and knowledge discovery applications, by means of the exploitation of inherent parallelism of data mining algorithms. Main goals of the use of parallel computing technologies in the data mining field are: *(i)* performance improvements of existing techniques, *(ii)* implementation of new (parallel) techniques and algorithms, and *(iii)* concurrent analysis using different data mining techniques in parallel and result integration to get a better model (i. e., more accurate results).

As observed in [5], three main strategies can be identified in the exploitation of parallelism algorithms: *Independent Parallelism*, *Task Parallelism* and *Single Program Multiple Data (SPMD) Parallelism*. We point out that this is a well known classification of general strategies for developing parallel algorithms, in fact they are not necessarily related only to data mining purposes. Nevertheless, in the following we will describe the underlying idea of such strategies by contextualizing them in data mining applications. A short description of the underlying idea of such strategies follows.

Independent Parallelism. It is exploited when processes are executed in parallel in an independent way. Generally, each process has access to the whole data set and does not communicate or synchronize with other processes. Such a strategy, for example, is applied when p different instances of the same algorithm are executed on the whole data set, but each one with a different setting of input parameters. In this way, the computation finds out p different models, each one determined by a different setting of input parameters. A validation step should learn which one of the p predictive models is the most reliable for the topic under investigation. This strategy often requires commutations among the parallel activities.

Task Parallelism. It is known also as *Control Parallelism*. It supposes that each process executes different operations on (a different partition of) the data set. The application of such a strategy in decision tree learning, for example, leads to have p different processes running, each one associated to a particular subtree of the decision tree to be built. The search goes parallely on in each subtree and, as soon as all the p processes finish their executions, the whole final decision tree is composed by joining the various subtrees obtained by the processes.

SPMD Parallelism. The single program multiple data (SPMD) model [10] (also called data parallelism) is exploited when a set of processes execute in parallel the same algorithm on different partitions of a data set, and processes cooperate to exchange partial results. According to this strategy, the dataset is initially partitioned in p parts, if p is the apriori-fixed parallelism degree (i. e., the number of processes running in parallel). Then, the p processes search in parallel a predictive model for the subset associated to it. Finally, the global result is obtained by exchanging all the local models information.

These three strategies for parallelizing data mining algorithms are not necessarily alternative. In fact, they can be combined to improve both performance and accuracy of results. For completeness, we say also that in combination with strategies for parallelization, different data partition strategies may be used : *(i)* sequential partitioning (separate partitions are defined without overlapping among them), *(ii)* cover-based partitioning (some data can be replicated on different partitions) and *(iii)* range-based query partitioning (partitions are defined on the basis of some queries that select data according to attribute values).

Architectural issues are a fundamental aspect for the goodness of a parallel data mining algorithm. In fact, interconnection topology of processors, communication strategies, memory usage, I/O impact on algorithm performance, load balancing of the processors are strongly related to the efficiency and effectiveness of the parallel algorithm. For lack of space, we can just cite those. The mentioned issues (and others) must be taken into account in the parallel implementation of data mining techniques. The architectural issues are strongly

related to the parallelization strategies and there is a mutual influence between knowledge extraction strategies and architectural features. For instance, increasing the parallelism degree in some cases corresponds to an increment of the communication overhead among the processors. However, communication costs can be also balanced by the improved knowledge that a data mining algorithm can get from parallelization. At each iteration the processors share the approximated models produced by each of them. Thus each processor executes a next iteration using its own previous work and also the knowledge produced by the other processors. This approach can improve the rate at which a data mining algorithm finds a model for data (knowledge) and make up for lost time in communication. Parallel execution of different data mining algorithms and techniques can be integrated not just to get high performance but also high accuracy.

3.2. Distributed Data Mining. Traditional warehouse-based architectures for data mining suppose to have centralized data repository. Such a centralized approach is fundamentally inappropriate for most of the distributed and ubiquitous data mining applications. In fact, the long response time, lack of proper use of distributed resource, and the fundamental characteristic of centralized data mining algorithms do not work well in distributed environments. A scalable solution for distributed applications calls for distributed processing of data, controlled by the available resources and human factors. For example, let us consider an ad hoc wireless sensor network where the different sensor nodes are monitoring some time-critical events. Central collection of data from every sensor node may create traffic over the limited bandwidth wireless channels and this may also drain a lot of power from the devices.

A distributed architecture for data mining is likely aimed to reduce the communication load and also to reduce the battery power more evenly across the different nodes in the sensor network. One can easily imagine similar needs for distributed computation of data mining primitives in ad hoc wireless networks of mobile devices like PDAs, cellphones, and wearable computers [25]. The wireless domain is not the only example. In fact, most of the applications that deal with time-critical distributed data are likely to benefit by paying careful attention to the distributed resources for computation, storage, and the cost of communication. As an other example, let us consider the World Wide Web as it contains distributed data and computing resources. An increasing number of databases (e.g., weather databases, oceanographic data, etc.) and data streams (e.g., financial data, emerging disease information, etc.) are currently made on-line, and many of them change frequently. It is easy to think of many applications that require regular monitoring of these diverse and distributed sources of data.

A distributed approach to analyze this data is likely to be more scalable and practical particularly when the application involves a large number of data sites. Hence, in this case we need data mining architectures that pay careful attention to the distribution of data, computing and communication, in order to access and use them in a near optimal fashion. *Distributed data mining* (*DDM*) considers data mining in this broader context.

DDM may also be useful in environments with multiple compute nodes connected over high speed networks. Even if the data can be quickly centralized using the relatively fast network, proper balancing of computational load among a cluster of nodes may require a distributed approach. The privacy issue is playing an increasingly important role in the emerging data mining applications. For example, let us suppose a consortium of different banks collaborating for detecting frauds. If a centralized solution was adopted, all the data from every bank should be collected in a single location, to be processed by a data mining system. Nevertheless, in such a case a distributed data mining system should be the natural technological choice: it is able to learn models from distributed data without exchanging the raw data among different repositories, and it allows detection of fraud by preserving the privacy of every bank's customer transaction data.

For what concerns techniques and architecture, it is worth noticing that many several other fields influence Distributed Data Mining systems concepts. First, many DDM systems adopt the multi-agent system (MAS) architecture, which finds its root in the distributed artificial intelligence (DAI). Second, although parallel data mining often assumes the presence of high speed network connections among the computing nodes, the development of DDM has also been influenced by the PDM literature. Most DDM algorithms are designed upon the potential parallelism they can apply over the given distributed data. Typically, the same algorithm operates on each distributed data site concurrently, producing one local model per site. Subsequently, all local models are aggregated to produce the final model. In Figure 3.1 a general distributed data mining framework is presented. The success of DDM algorithms lies in the aggregation. Each local model represents locally coherent patterns, but lacks details that may be required to induce globally meaningful knowledge. For this reason, many DDM algorithms require a centralization of a subset of local data to compensate it. The ensemble approach has been applied in various domains to increase the accuracy of the predictive model to be learnt. It produces

multiple models and combines them to enhance accuracy. Typically, voting (weighted or un-weighted) schema are employed to aggregate base model for obtaining a global model. As we have discussed above, minimum data transfer is another key attribute of the successful DDM algorithm. As a final consideration, the homogeneity/heterogeneity of resources is another important aspect to be considered in the distributed data mining approaches. In this scenario, the term "resources" refers both to computational resources (computers with similar/different computational power) and data resources (datasets with horizontally/vertically partitioning among nodes). The first meaning affects only the algorithm execution time, while data heterogeneity plays a fundamental role in the algorithm design. That is, dealing with different data formats it requires algorithms designed in accordance to the different data formats.

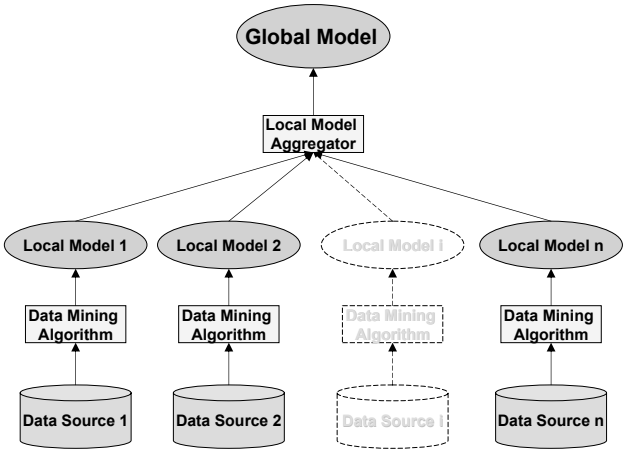


FIG. 3.1. *General Distributed Data Mining Framework.*

3.3. Grid-based Data Mining. In the last years, *Grid computing* is receiving an increasing attention both from the research community and from industry and governments, watching at this new computing infrastructure as a key technology for solving complex problems and implementing distributed high-performance applications. *Grid* technology integrates both distributed and parallel computing, thus it represents a critical infrastructure for high-performance distributed knowledge discovery. *Grid computing* differs from conventional distributed computing because it focuses on large-scale dynamic resource sharing, offers innovative applications, and, in some cases, it is geared toward high-performance systems. The *Grid* emerged as a privileged computing infrastructure to develop applications over geographically distributed sites, providing for protocols and services enabling the integrated and seamless use of remote computing power, storage, software, and data, managed and shared by different organizations.

Basic Grid protocols and services are provided by toolkits such as *Globus Toolkit* (`www.globus.org/toolkit`), *Condor* (`www.cs.wisc.edu/condor`), *Glite*, and *Unicore*. In particular, the *Globus Toolkit* is the most widely used middleware in scientific and data-intensive Grid applications, and is becoming a de facto standard for implementing Grid systems. This toolkit addresses security, information discovery, resource and data management, communication, fault-detection, and portability issues. A wide set of applications is being developed for the exploitation of Grid platforms. Since application areas range from scientific computing to industry and business, specialized services are required to meet needs in different application contexts. In particular, *data Grids* have been designed to easily store, move, and manage large data sets in distributed data-intensive applications. Besides core data management services, *knowledge-based Grids*, built on top of computational and data Grid environments, are needed to offer higher-level services for data analysis, inference, and discovery in scientific and business areas [21]. In some papers, see for example [1], [19], and [7], it is claimed that the creation of *knowledge Grids* is the enabling condition for developing high-performance knowledge discovery processes and meeting the challenges posed by the increasing demand of power and abstractness coming from complex problem solving environments.

4. The Knowledge Grid. The *Knowledge Grid* [3] is an environment providing knowledge discovery services for a wide range of high performance distributed applications. Data sets and analysis tools used in such

applications are increasingly becoming available as stand-alone packages and as remote services on the Internet. Examples include gene and DNA databases, network access and intrusion data, drug features and effects data repositories, astronomy data files, and data about web usage, content, and structure. Knowledge discovery procedures in all these applications typically require the creation and management of complex, dynamic, multi-step workflows. At each step, data from various sources can be moved, filtered, and integrated and fed into a data mining tool. Based on the output results, the developer chooses which other data sets and mining components can be integrated in the workflow, or how to iterate the process to get a knowledge model. Workflows are mapped on a Grid by assigning nodes to the Grid hosts and using interconnections for implementing communication among the workflow nodes.

For completeness of treatment, we point out some other Grid-based knowledge discovery systems and activities that have been designed in recent years. *Discovery Net* [8] is an infrastructure for effectively support scientific knowledge discovery process, in particular in the areas of life science and geo-hazard prediction. *DataSpace* [17] is a framework providing efficient data access and transfer over the Grid that implements an ad-hoc protocol for working with remote and distributed data (named DataSpace transfer protocol, DSTP). *InfoGrid* [16] is a service-based data integration middleware engine, designed to provide information access and querying services not in an 'universal' way, but by a personalized view of the resources for each particular application domain. *DataCutter* [2] is another Grid middleware framework aimed at providing specific services for the support of multi-dimensional range-querying, data aggregation and user-defined filtering over large scientific datasets in shared distributed environments. Finally, *GATES* [4] (Grid-based AdapTive Execution on Streams) is an OGSA based system that provides support for processing of data streams in a Grid environment. This system is designed to support the distributed analysis of data streams arising from distributed sources (e.g., data from large scale experiments/simulations). GATES provides automatic resource discovery and an interface for enabling self-adaptation to meet real-time constraints.

The Knowledge Grid architecture is designed according to the *Service Oriented Architecture (SOA)*, that is a model for building flexible, modular, and interoperable software applications. The key aspect of *SOA* is the concept of *service*, that is a software block capable of performing a given task or business function. Each *service* operates by adhering to a well defined interface, defining required parameters and the nature of the result. Once defined and deployed, services are like "black boxes", that is, they work independently of the state of any other service defined within the system, often cooperating with other services to achieve a common goal. The most important implementation of *SOA* is represented by *Web Services*, whose popularity is mainly due to the adoption of universally accepted technologies such as XML, SOAP, and HTTP. Also the Grid provides a framework whereby a great number of services can be dynamically located, balanced, and managed, so that applications are always guaranteed to be securely executed, according to the principles of on-demand computing.

The Grid community has adopted the *Open Grid Services Architecture (OGSA)* as an implementation of the *SOA* model within the Grid context. In *OGSA* every resource is represented as a Web Service that conforms to a set of conventions and supports standard interfaces. *OGSA* provides a well-defined set of Web Service interfaces for the development of interoperable Grid systems and applications [15]. Recently the WS-Resource Framework (WSRF) has been adopted as an evolution of early OGSA implementations [9]. WSRF defines a family of technical specifications for accessing and managing stateful resources using Web Services. The composition of a Web Service and a stateful resource is termed as WS-Resource. The possibility to define a âĂIJstateâĂİ associated to a service is the most important difference between WSRF-compliant Web Services, and pre-WSRF ones. This is a key feature in designing Grid applications, since WS-Resources provide a way to represent, advertise, and access properties related to both computational resources and applications.

The Knowledge Grid is a software for implementing knowledge discovery tasks in a wide range of high-performance distributed applications. It offers to users high-level abstractions and a set of services by which they can integrate Grid resources to support all the phases of the knowledge discovery process.

The Knowledge Grid supports such activities by providing mechanisms and higher level services for searching resources, representing, creating, and managing knowledge discovery processes, and for composing existing data services and data mining services in a structured manner, allowing designers to plan, store, document, verify, share and re-execute their workflows as well as manage their output results. The Knowledge Grid architecture is composed of a set of services divided in two layers: the *Core K-Grid layer* and the *High-level K-Grid layer*. The first interfaces the basic and generic Grid middleware services, while the second interfaces the user by offering a set of services for the design and execution of knowledge discovery applications. Both layers make

use of repositories that provide information about resource metadata, execution plans, and knowledge obtained as result of knowledge discovery applications.

In the Knowledge Grid environment, discovery processes are represented as workflows that a user may compose using both concrete and abstract Grid resources. Knowledge discovery workflows are defined using a visual interface that shows resources (data, tools, and hosts) to the user and offers mechanisms for integrating them in a workflow. Information about single resources and workflows are stored using an XML-based notation that represents a workflow (called execution plan in the Knowledge Grid terminology) as a data-flow graph of nodes, each one representing either a data mining service or a data transfer service. The XML representation allows the workflows for discovery processes to be easily validated, shared, translated in executable scripts, and stored for future executions. It is worth noticing that when the user submits a knowledge discovery application to the Knowledge Grid, she has no knowledge about all the low level details needed by the execution plan. More precisely, the client submits to the Knowledge Grid a high level description of the KDD application, named *conceptual model*, more targeted to distributed knowledge discovery aspects than to grid-related issues. The Knowledge Grid in a first step creates an execution plan on the basis of the conceptual model received from the user, and then executes it by using the resources effectively available. To realize this logic, it initially models an *abstract execution plan* (where some specified resource could remain 'abstractly' defined, i. e. they could not match with a real resource), that in a second step is resolved into a *concrete execution plan* (where a matching between each resource and someone really available on the Grid is found).

The Knowledge Grid has been used in various real scenarios, pointing out its suitability in several heterogeneous applications. For lack of space we are not able to discuss about them. For such a reason we give here just some outlines, more details can be found in the cited papers. The goal of the example described in [6] was to obtain a classifier for an intrusion detection system, performing a mining process on a (very large size) dataset containing records generated by network monitoring. The example reported in [5] was a simple meta-learning process, that exploits the Knowledge Grid to generate a number of independent classifiers by applying learning programs to a collection of distributed data sets in parallel.

As a scientific application scenario, let us consider the collection of sky observations and the analysis of their characteristics. Let us suppose to have distinct image data obtained by observations and simulations, from which we want to extract significant metrics. Generally, a significative set of astronomy data is very large size ($\approx 20 - 30$ terabytes). In addition, such kind of observation are very high-dimensional, because each point is usually described by $\approx 10^3$ attributes (including morphological parameters, flux ratios, etc.). Finally, they usually are full of missing values and noise. Then, the main issue here is to analyze a distribution of $\approx 20 - 30$ terabytes of points in a parameter space of $\approx 10^3$ dimensions. Let us suppose that our effort is devoted to identify how many distinct types of objects are there (i. e., stars, galaxies, quasars, black holes, etc.), and grouping them with respect to their type. This can be obtained by a clustering analysis, however it is a non-trivial task if we consider the large size data and their high dimensionality. To such a purpose, a distributed framework can be suitable to get results in a reasonable time. Initially we have a data repository where all such an observed sky data is collected (for example, an astronomic observatory). Then, such a data is processed by a distributed clustering algorithm. In order to do that, they are partitioned on many nodes and processed on those nodes in parallel. The results of every clustering algorithm are collected and combined to obtain a global clustering model. In addition, each outlier can represent a possible (rare) new object. For such a reason, and in order to get more knowledge from them, all the detected outliers are transferred to another node for a further classification, i. e. by a decision tree.

Figure 4.1 shows such a distributed meta-learning scenario, in which a global clustering model classifier CM is obtained on $Node_C$ starting from the original data set DS stored on $Node_A$ (i.e, where the observatory is located). Moreover, all the outliers detected are collected in an outlier set OS and are processed by a classifier Cl on a $Node_B$. This process can be described through the following steps:

1. On $Node_A$, data sets $DS_1, \ldots, DS_n$ are extracted from DS by the partitioner P. Then $DS_1, \ldots, DS_n$, are respectively moved from $Node_A$ to $Node_1, \ldots, Node_n$.
2. On each $Node_i (i = 1, \ldots, n)$ the clusterer C_i applies a clustering algorithms on each dataset DS_i. Then, each local result is moved from $Node_i$ to $Node_C$.
3. On $Node_C$, local models received from $Node_1, \ldots, Node_n$ are combined by the combiner C to produce the global clustering model CM. Moreover, outliers detected are collected in an outlier set OS, and moved to the $Node_B$ for further analysis.

4. On $Node_B$, the classifier Cl processes the OS outlier data set and extracts a suitable classification model (i. e., a decision tree) from it.

Being the Knowledge Grid a service oriented architecture, the Knowledge Grid user interacts with some services to design and execute such an application.

As an additional consideration, we notice that a client application, that wants to submit a knowledge discovery computation to the Knowledge Grid, has to interact not with all of these services, but just with some of them; there are, in fact, two layers of services: *high-level* services (*DAS*, *TAAS*, *EPMS* and *RPS*) and *core-level* services (*KDS* and *RAEMS*). The design idea is that user level applications directly interact with high-level services that, in order to perform a client request, invoke suitable operations exported by the core-level services. In turn, core-level services perform their operations by invoking basic services provided by available grid environments running on the specific host, as well as by interacting with other core-level services. In other words, operations exported by high-level services are designed to be invoked by user-level applications, whereas operations provided by core-level services are thought to be invoked both by high-level and core-level services. More in detail, the user can interacts with the *DAS* (*Data Access Service*) and *TAAS* (*Tools and Algorithms Access Service*) services to find data and mining software and with the *EPMS* (*Execution Plan Management Service*) service to compose a workflow (execution plan) describing at a high level the needed activities involved in the overall data mining computation. Through the execution plan, computing, software and data resources are specified along with a set of requirements on them. The execution plan is then processed by the *RAEMS* (*Resource Allocation and Execution Management Service*), which takes care of its allocation. In particular, it first finds appropriate resources matching user requirements (i. e., a set of concrete hosts $Node_1, \ldots, Node_n$, offering the software $C_1, \ldots, C_n$, and a node $Node_W$ providing the C combiner software and a node $Node_Z$ exporting the classifier Cl), then manages the execution of overall application, enforcing dependencies among data extraction, transfer, and mining steps. Finally, the *RAEMS* manages results retrieving, and visualize them by the *RPS* (*Results Presentation Service*) service (that offers facilities for presenting and visualizing the extracted knowledge models).

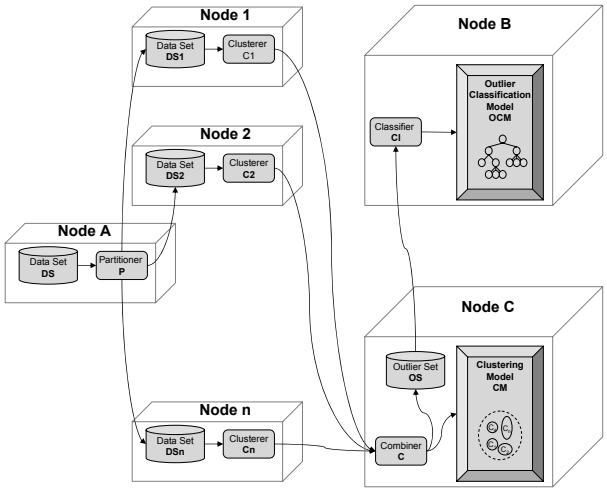


FIG. 4.1. *A distributed meta-learning scenario.*

5. Conclusion. In this paper we have pointed out that digital data volumes are growing exponentially in science and engineering. Often digital repositories and sources increase their size much faster than the computational power offered by the current technology. To handle this abundance in data availability, scientists must embody knowledge discovery tools to find what is interesting in them.

When data is maintained over geographically distributed sites, Grid computing can be used as a distributed infrastructure for service-based intensive applications. Various scientific applications based on Grid infrastructures, described in the paper, concretely show how it can be exploited for scientific purposes. Moreover, the computational power of distributed and parallel systems can be exploited for knowledge discovery in scientific data. Parallel and distributed data mining suites and computational Grid technology are two critical elements of future high-performance computing environments for e-science. In such a direction, the *Knowledge Grid*

is a reference software architecture for geographically distributed knowledge discovery systems that allows to implement complex data analysis applications as a collection of distributed services.

REFERENCES

[1] F. Berman, *From TeraGrid to Knowledge Grid*, Communications of the ACM, 44(11) (2001), pp. 27–28.
[2] M. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, *Distributed Processing of Very Large Datasets with DataCutter*, Parallel Computing, 27(11) (2001), pp. 1457–1478.
[3] M. Cannataro and D. Talia, *The Knowledge Grid*, Communications of the ACM, 46(1) (2003), pp. 89–93.
[4] L. Chen, K. Reddy, and G. Agrawal, *GATES: A Grid-Based Middleware for Processing Distributed Data Streams*, Proc. of the 13th IEEE Int. Symposium on High Performance Distributed Computing (HPDC), (2004), pp. 192–201.
[5] A. Congiusta, D. Talia, and P. Trunfio, *Parallel and Grid-Based Data Mining*, in Data Mining and Knowledge Discovery Handbook, Springer, 2005, pp. 1017–1041.
[6] A. Congiusta, D. Talia, and P. Trunfio, *Using Grids for Distributed Knowledge Discovery*, in Mathematical Methods for Knowledge Discovery and Data Mining, IGI Global Publisher, 2007, pp. 248–298.
[7] A. Congiusta, D. Talia, and P. Trunfio, *Distributed Data Mining Services Leveraging WSRF*, Future Generation Computer Systems, 23(1) (2007), pp. 34–41.
[8] V. Curcin, M. Ghanem, Y. Guo, M. Kohler, A. Rowe, J. Syed J., and P. Wendel, *Discovery Net: Towards a Grid of Knowledge Discovery*, Proc. of the 8th Int. Conference on Knowledge Discovery and Data Mining (KDD), (2002).
[9] K. Czajkowski et al., *The WS-Resource Framework Version 1.0*, http://www.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf, 2004.
[10] F. Darema, *SPMD model: past, present and future*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting, Springer, 2001, p. 1.
[11] *DataGrid Project*, http://web.datagrid.cnr.it, 2001.
[12] S. G. Djorgovski, *Virtual Astronomy, Information Technology, and the New Scientific Methodology*, Proc. of the 7th Int. Workshop on Computer Architectures for Machine Perception (CAMP), (2005), pp. 125–132.
[13] *EGEE Project*, http://www.eu-egee.org/, 2005.
[14] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, Globus Project, www.globus.org/alliance/publications/papers/ogsa.pdf, 2002.
[15] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, *The Physiology of the Grid*, in Grid Computing: Making the Global Infrastructure a Reality, F. Berman, G. Fox, and A. Hey, eds., Wiley, 2003, pp. 217–249.
[16] N. Giannadakis, A. Rowe, M. Ghanem, and Y. Guo, *A Web Infrastructure for the Exploratory Analysis and Mining of Data*, Information Sciences, 2007, pp. 199–226.
[17] R. Grossman, and M. Mazzucco, *DataSpace: A Data Web for the Exploratory Analysis and Mining of Data*, IEEE Computing in Science and Engineering, 4(4), 2002, pp. 44–51.
[18] *IPG Project*, http://www.gloriad.org/gloriad/projects/ project000053.html, 1998.
[19] W. E. Johnston, *Computational and Data Grids in Large Scale Science and Engineering*, Future Generation Computer Systems, 18(8) (2002), pp. 1085–1100.
[20] F. Meyer, *Genome Sequencing vs. Moore's Law: Cyber Challenges for the Next Decade*, CTWatch Quarterly, 2(3) (2006), http://www.ctwatch.org/quarterly/articles/2006/08/genome-sequencing-vs-moores-law/.
[21] R. Moore, *Knowledge-based Grids*, Proc. of the 18th IEEE Symposium on Mass Storage Systems and 9th Goddard Conference on Mass Storage Systems and Technologies, (2001).
[22] *myExperiment Project*, http://www.eu-egee.org/, 2006.
[23] *National Virtual Observatory Project*, http://www.us-vo.org/, http://www.virtualobservatory.org/, 2001.
[24] *Open Science Grid Project*, http://www.opensciencegrid.org/, 2004.
[25] B. Park, and H. Kargupta, *Distributed Data Mining: Algorithms, Systems, and Applications*, in Data Mining Handbook, IEA Publisher, 2002, pp. 341–358.
[26] *Southern California Earthquake Center Project*, http://epicenter.usc.edu/cmeportal/index.html, 2001.
[27] D. Skillicorn, *Strategies for Parallel Data Mining*, IEEE Concurrency, 7(4) (1999), pp. 26–35.
[28] D. Talia, *Parallelism in Knowledge Discovery Techniques*, Proc. of the 6th Int. Conf. on Applied Parallel Computing, (2002), pp. 127–136.
[29] *TeraGrid Project*, http://www.teragrid.org/, 2005.

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 263–275. http://www.scpe.org
ISSN 1895-1767

MODELING STREAM COMMUNICATIONS IN COMPONENT-BASED APPLICATIONS *

M. DANELUTTO† D. LAFORENZA‡ N. TONELLOTTO‡, M. VANNESCHI†, AND C. ZOCCOLO†

Abstract. Component technology is a promising approach to develop Grid applications, allowing to design very complex applications by hierarchical composition of basic components. Nevertheless, component applications on Grids have complex deployment models. Performance-sensitive decisions should be taken by automatic tools, matching developer knowledge about component performance with QoS requirements on the applications, in order to find deployment plans that satisfy a Service Level Agreement (SLA).

This paper presents a steady-state performance model for component-based applications with stream communication semantics. The model strictly adheres to the hierarchical nature of component-based applications, and is of practical use in launch-time decisions.

Key words: grid computing; heterogeneous environments; stream computations; performance model; mapping.

1. Introduction. Grid computing is an emerging technology that enables the aggregation of heterogeneous, distributed resources to solve computational problems of ever increasing size and complexity. The applications that best perform on Grid platforms are the ones requiring large computational power, or the treatment of large data sets, i. e. a subclass of High-Performance Applications [17].

Such applications (e.g. data-mining [12], query processing [3], image processing and visualization [2] and multimedia streaming [38]) can be conveniently expressed using a formalism based on two fundamental notions: streams of data flowing between components, and components (either sequential or parallel) processing them. Several programming languages are built on these concepts. Skeleton-based languages (e.g. SkIE [4] and SBASCO [14]) and skeleton libraries (e.g. eSkel [11] and Kuchen's C++ skeleton library [21]) exploit the notion of streams for task-parallel skeletons (e.g. pipe and farm). More general languages like ASSIST [33] and Datacutter [15] introduce modules and streams as primitive concepts to structure parallel applications.

Grid programming frameworks (e.g. GrADS [9], ASSIST [13]) are in charge of the complete automation of application execution management, efficiently exploiting Grid resources. Moreover, they should be able to execute the application with user-required QoS, adapting the execution to the dynamic changes of Grid resources.

The traditional component mapping strategy, in which components are *statically* deployed in a distributed environment by their developers, does not fit well in such scenario. A broader deployment model is required, featuring

(i) manual mapping, in which the components are already paired with their resources (on which they are deployed),

(ii) resources discovery and selection at launch time, to guarantee the initial desired performance,

(iii) adaptive components management, that at run-time adjust the set of computing resources exploited [31, 1], in order to adapt to different performance requirements (on-demand computing) or to changing resources availability.

According to this model, the deployment framework must automatically manage the operations needed to enforce the application desired QoS. This can be obtained with the specification of a *performance contract* [34].

Our approach intends to automatise the tasks needed to start the execution of HPC applications. Our final goal is to allow an as large as possible user community to gain full benefits from the Grid, and at the same time to give the maximum generality, applicability and easy of use.

The main contributions of this paper are as follows:

(i) We propose an analytical model of the dynamic behavior of sequential/parallel components, hierarchical components and component applications, communicating through typed streams of data. It is suited to be used in simulation environments, to synthetically generate components and applications to test mapping/scheduling solutions in a repeatable and controlled setting. Eventually, the proposed dynamic model can be exploited in the implementation of dynamic reconfiguration policies [1].

*This work has been supported by: the Italian MIUR FIRB Grid.it project, No. RBNE01KNFP, on High-performance Grid platforms and tools, and the European CoreGRID NoE (European Research Network on Foundations, Software Infrastructures and Applications for Large Scale, Distributed, GRID and Peer-to-Peer Technologies, contract no. IST-2002-004265).

†Department of Computer Science, University of Pisa, Pisa, Italy

‡Information Science and Technologies Institute, National Research Council, Pisa, Italy

(ii) Starting from the dynamic model we identify the set of variables that can be used to describe the performance behavior of an application, and we derive the set of relations among them which hold at steady-state (performance model). In this way we abstract from particular runtime platforms and we capture all possible steady-state behaviors of an application. Moreover, their formulation by means of linear algebra allows us to hierarchically compose the performance models of several components to derive the steady-state model of new components or applications.

(iii) We introduce a definition of *performance model* for stream applications, which is exploited in launch-time mapping and runtime reconfiguration decisions.

After a survey of related work (Sect. 2), this paper presents a dynamic model of stream-based computations (Sect. 3), and in Sect. 4 such model is exploited to derive a steady-state performance model for stream-based applications. In Sect. 5, such model is applied to a case study, to predict the program behavior at run-time, and to devise a correct initial mapping for specified QoS levels. Section 6 concludes the paper, discussing the presented approach and future work.

2. Related Work. Performance specification of components and their interactions is a basic problem that must be solved to enable software engineers to assemble efficient applications [27]. Moreover, performance modeling is one of the key aspects that needs to be addressed to face scheduling/mapping problems in heterogeneous platforms. It arises in automatic component placement and reconfiguration. Several recent works focus on performance modeling techniques to analyze the behavior of component-based parallel applications on distributed, heterogenous, dynamic platforms.

Analytic performance models in software engineering make extensive use of UML formalism to describe software component behavioral models [35] and to derive models based on Queuing Networks [19] or Layered Queueing Networks [36] to be exploited in design phase of the lifecyle of software. The same holds for Stochastic Petri Nets [20] and Stochastic Process Algebras [18]. Such models typically translate a parallel application into an analytic representation of its execution behavior and the target runtime system (according to the Software Performance Engineering methodology [28]). A detailed survey of such models is in [5]. Such translation is usually not straightforward. It may require approximations to obtain mathematical models [29] for which a closed-form solution is known. Stochastic models usually require the solution of the underlying Markov chain which can easily lead to numerical problems due to the space state explosion [5]. More complex models can be solved by means of simulation, at the cost of a larger computation time.

Symbolic performance modeling [32] is a methodology that enables a rapid development of low complexity and parametric performance models. Symbolic performance models can be derived from simulation models, trading off result accuracy for model evaluation cost. In [32] a symbolic performance model for the PAMELA modeling language is introduced. It derives lower bounds for steady-state performances of applications starting from a model of the program and of the shared resources, combining deterministic Direct Acyclic Graphs (DAGs) modeling with mutual exclusion. One of the strengths of the PAMELA approach is that it is fast and easy to transform a regularly structured application into a performance model. The main limitation of such approach is that it computes lower bounds of the performance of a program. Symbolic performance models share several properties with the model we propose: both can be extracted from the structure of programs, are parametric, and can be efficiently evaluated. The main difference is that the presented model does not compute a lower bound, but the asymptotic steady-state performance of an application, that is in general a better approximation of the real performance.

The asymptotic steady-state analysis has been pioneered by Bertsimas and Gamarnik [10]. This approach has been recently applied to mapping and scheduling problems of parallel applications on heterogeneous platforms [23, 7, 6], in which the analysis is applied to particular classes of parallel applications (divisible load [23], master/slave [6], pipelined and scatter operations [7]), in the hypothesis that the set of resources is known in advance. The existing steady-state approaches apply only to a restricted class of structured parallel applications, assuming to know the runtime environment in such a way to derive optimal scheduling of the application components. In a dynamic environment like a Grid an optimal initial placement of the components may become useless very soon, because the conditions of the execution platform may vary dynamically. The presented steady-state analysis can be applied to a broader class of structured parallel applications and tries to solve a different problem, i. e. to build a concrete model of components/applications to be exploited in their mapping on previously-unknown target platforms.

Structural performance models [25] are the first effort to develop compositional performance models for component applications. Most scientific and Grid component models rely on the concept of algorithmic skeleton. Skeletons are common, reusable and efficient structured parallelism exploitation patterns. One advantage of the skeletal approach is that parametric cost models can be devised for the evaluation of runtime performance of skeleton compositions. In [14, 8] different cost models are associated to each skeleton of an application to enhance its runtime performance through parallelism/replication degree adjustments and initial mapping selection, respectively. The authors of [14] propose parametric cost models for PIPE, FARM and MULTIBLOCK skeletons, that can be arbitrarily composed and nested. In [8], analytic cost models for applications composed by PIPEs and DEALs are derived within a stochastic process algebra formulation. Structural performance models are extended by the presented model by proposing a methodology well-suited for generic composition of skeletons, and by taking into account the synchronization problems introduced by using streamed communications.

Trace-based performance models [34, 26] are currently exploited in parallel/Grid environments to model the performance of sets of kernel applications. Recording and analyzing execution traces on reference architectures of such application it is possible, with a certain degree of precision, to forecast the performance of the same or similar applications on different resources. Trace information is exploited in the presented model, but in different way with respect to the existing approaches. Instead of profiling a whole application on a set of representative resources, the application model is kept independent from resources. When the application will be mapped on actual resources, historical information will be used to model the runtime behavior of single components, and then such information will be coupled with the component interactions information to obtain a prediction of the performance of the whole application.

The problem of deriving a performance model for components has been addressed also in the context of component frameworks such as EJB [37], COM+/.NET [16] and CCA [24]. Such works apply analytical performance model (LQN) or trace-based performance model to derive a model for components. In [30], trace-based models are exploited to select the most suitable components, when multiple choices are available, to build an optimal application, from the point of view of performance.

3. Dynamic Behavior. An application can be structured as a hypergraph whose nodes represent primitive components and whose (hyper)edges represent communications or synchronizations between components. Nodes interact with input (server) interfaces and output (client) interfaces. Edges are directed and can connect two or more nodes through their interfaces. Two nodes may be linked by more than a single edge.

3.1. Communications. Communications between components are implemented through input/output interfaces bindings. In this work data-flow stream communications are studied. Every component receives data through one or more input interfaces, performs some computations, and generates new data to be sent through one or more output interfaces.

In this context, a *stream* represents a typed, unidirectional communication channel between a non-empty, finite set of components (producers) and a non-empty, finite set of components (consumers). The atomic piece of information transferred through a stream is called *item*. A producer is connected to a stream through an output interface, while a consumer is connected to a stream through an input interface. Every node can be producer or consumer of several streams, and it is possible to specify cyclic structures (i. e. the communication structure is not restricted to be a DAG).

Components can be connected by streams according to three different patterns:

(i) **unicast**: one-to-one connection. Every item sent on the output stream interface is received in order by the input stream interface.

(ii) **merge**: many-to-one connection. Every item sent on the output stream interfaces is received by the input stream interface. The temporal ordering of the items coming from each input interface is preserved, but the interleaving between the different sources is non-deterministic.

(iii) **broadcast**: one-to-many connection. Every item sent on the output stream interface is received in order by the input stream interfaces. The receptions happening on different input interfaces are not synchronized.

3.2. Computations. Components implement sequential as well as parallel computations. A sequential component executes a single function in a single active thread, processing items as they are received. For a parallel component, two scenarios are possible:

(i) **data parallel**: a single function is executed in parallel on different portions of the same data;

(ii) **task parallel**: several functions (or activations of the same function) are executed in parallel on independent data.

A primitive component, either sequential or parallel, at runtime repeatedly receives items from its input streams, performs some computations and delivers result items to its output streams.

A component can have several input streams. The set of input streams is partitioned between the computations associated with the components. Each input stream is associated to only one computation; nevertheless, spontaneous computations may exist, that do not need input items to activate, but follow own activation policies (e.g. periodically).

A computation can be activated if the following conditions hold:

(i) the component can execute a new function (this means that it is idle, or it is parallel and threads are available to execute it),

(ii) the associated input items have been received, or no item is necessary.

A sequential component can activate a new function only when it is idle. A parallel component can have at most one active data-parallel computation at any given time (composed by a fixed number of threads), or several task-parallel computations running in parallel (up to the maximum number of threads in the component).

A component can have several output streams. One or more computations of the component can dispatch data on each output stream.

3.3. Node Behavior. In order to describe the behavior of a computation at runtime, consider Fig. 3.1.

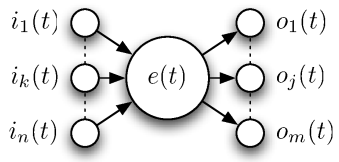


FIG. 3.1. *Sequential component at runtime*

Without loss of generality, a sequential component is considered; the displayed quantities represent:

(i) $i_k(t)$: total number of received items at time t from the k^{th} input interface;

(ii) $e(t)$: total number of computations carried out at time t;

(iii) $o_j(t)$: total number of sent items at time t through the j^{th} output interface.

Continuous quantities are used to model partial evolution, e.g. $e(t) = 3.5$ means that the node reached the half way point in the fourth computation.

The activation of a computation can happen only when the number of items completely received on each associated stream is greater than the number of partially computed items:

$$\forall k = 1, \ldots, n \qquad \lfloor i_k(t) \rfloor - e(t) > 0 \tag{3.1}$$

The node implementation will exploit finite buffers to store received items for each input interface, therefore for each input interface and associated computation the following must hold:

$$\forall k = 1, \ldots, n \qquad i_k(t) - \lfloor e(t) \rfloor \leq \tau_{1k} \tag{3.2}$$

where τ_{1k} represents the maximum number of elements that can be received on the k^{th} input interface before the stream blocks. Then the maximum admissible value for $i_k(t)$ at time t is:

$$i_k^{max}(t) = \tau_{1k} + \lfloor e(t) \rfloor \tag{3.3}$$

Assuming that no sensible delays are present between the end of computations and the beginning of the transmission of the produced items, the total number of transmitted items is related to the progress of the computations of the node. In the general case of a node with s functions, the following equation holds for each output interface:

$$\forall j = 1, \ldots, m \qquad o_j(t) = f_j\big(e_1(t), \ldots, e_s(t)\big) \tag{3.4}$$

where $e_i(t)$ represents the number of activations carried out at time t for the $i-th$ function. The *transfer function* f_j relates the number of data outputs $o_j(t)$ to the number of performed computations $e_1(t), \ldots, e_s(t)$.

3.4. Edge Behavior. In order to describe the behavior of a data transmission on a stream, consider a unicast stream. The involved variables are $o(t)$, total number of items sent at time t from source interface, and $i(t)$, total number of items received at time t by the destination interface. A new transmission begins only after a full item is produced:

$$i(t) \leq \lfloor o(t) \rfloor \tag{3.5}$$

The edge implementation will exploit finite communication buffers and the network layer transfers chunks of data. Let q^{-1} be the minimum fraction of item transferred atomically. Then

$$o(t) - \frac{\lfloor q \cdot i(t) \rfloor}{q} \leq \tau_2 \tag{3.6}$$

where τ_2 represents the maximum number of items that can be buffered. Therefore the maximum admissible value for $o(t)$ at time t is:

$$o^{max}(t) = \tau_2 + \frac{\lfloor q \cdot i(t) \rfloor}{q} \tag{3.7}$$

Whenever an edge buffer is full, a producer will block as soon as it tries and sends a new item. From (3.4) we obtain:

$$o^{max}(t) - f\big(e_1(t), \ldots, e_m(t)\big) \leq 0 \tag{3.8}$$

For *merge* streams with k source interfaces and *broadcast* streams with k destination interfaces, the general constraints (Eqs. (3.5) and (3.6) for the unicast stream) become:

$$\text{merge: } \begin{cases} i(t) \leq \sum_k o_k(t) \\ \sum_k o_k(t) - i(t) \leq \tau_{2k} \end{cases} \tag{3.9}$$

$$\text{broadcast: } \begin{cases} \forall k \quad i_k(t) \leq o(t) \\ \forall k \quad o(t) - i_k(t) \leq \tau_{2k} \end{cases} \tag{3.10}$$

For simplicity, in the previous equations the network quantization constant q has been suppressed.

3.5. Runtime Behavior. At runtime, a component can be seen as a dynamic system. The system state at time t is described by a set of state variables: $i_{1,\ldots,n_i}(t)$, $e_{1,\ldots,n_e}(t)$, $o_{1,\ldots,n_o}(t)$. Thus, the state space $\mathbb{P}$ is a $n = n_i + n_e + n_o$ dimension Euclidean space. The dynamic behavior of a component can be modeled by a trajectory $p(t)$ in such state space.

The runtime behavior of a component is fully specified when it is coupled with hosting resources. A computing resource is modeled by $w(t)$, the available computing power at time t (measured in MFlop/s) and a communication link is modeled by $b(t)$, the instantaneous bandwidth at time t (measured in MByte/s). Moreover, a characterization of the items is required. It is assumed that an item processed by a component requires l units of computing work to be processed (measured in MFlop) and s units of communication work to be transmitted (measured in bytes).

Introducing the *step function* $u(x)$, the number of performed (partial) computations per time unit is:

$$\begin{aligned} \frac{de}{dt} = u\Big(\min\Big(\lfloor i_1(t) \rfloor, \ldots, \lfloor i_n(t) \rfloor\Big) - e(t)\Big) \cdot \\ \cdot u\Big(o^{max}(t) - f\big(e_1(t), \ldots, e_m(t)\big)\Big) \cdot \frac{w(t)}{L} \end{aligned} \tag{3.11}$$

while the equations governing the number of packets flowing in the unicast, merge and broadcast streams per time unit are, respectively:

$$\frac{di}{dt} = u\Big(\lfloor o(t) \rfloor - i(t)\Big) \cdot u\Big(i^{max}(t) - i(t)\Big) \cdot \frac{b(t)}{s} \tag{3.12a}$$

$$\frac{di}{dt} = u\Big(\sum_k \lfloor o_k(t) \rfloor - i(t)\Big) \cdot u\Big(i^{max}(t) - i(t)\Big) \cdot \frac{b(t)}{s} \tag{3.12b}$$

$$\frac{di_k}{dt} = u\Big(\lfloor o(t) \rfloor - i_k(t)\Big) \cdot u\Big(i^{max}(t) - i_k(t)\Big) \cdot \frac{b(t)}{s} \tag{3.12c}$$

Note that an important assumption has been made. The work required to perform a computation is supposed to be *independent* from the values of the incoming items; their values are used just to perform computations. This is a common assumption in parallel data-flow programming, but there are applications (e.g. query processing and data mining) that do not respect this assumption.

The dynamic equations provided by the model can be written in the general form:

$$\dot{p}(t) = U(p(t))\,\alpha(t) \tag{3.13}$$

We denote with $U : \mathbb{P} \to \mathbf{M}_{n,n}$ the function that, for every point in the state space, provides the control part of the differential equations (the ones involving the step functions), and with $\alpha(t)$ the resources part (involving $w(t)$ and $b(t)$).

We observe that the control matrix is piece-wise constant over non-infinitesimal time intervals: it descends from quantization in the general equations for the nodes (3.11), and in the equations for the streams (3.12). Then, the Cauchy problem can be solved constructively. Starting with $t_0 = 0, p_0(t_0) = 0, U_0 = U(0)$, we inductively define

$$\begin{aligned} p_i(t) &= \int_{t_i}^{t} U_i \alpha(\tau) d\tau \\ t_{i+1} &= \sup\{t > t_i | U(p_i(t)) = U_i\} \\ U_{i+1} &= \lim_{t \to t_i^+} U(p_i(t)) \end{aligned}$$

In this way, $p(t)$ is defined as the concatenation of the pieces $p_i|_{[t_i, t_{i+1})}$: it is a continuous function ($p_i(t_i) = p_{i+1}(t_i)$) and piece-wise differentiable.

4. STEADY STATE BEHAVIOR. The steady-state behavior of the system can be analysed by studying mean values $\bar{p}$ for the rate of change of the state variables:

$$\bar{p} = \mathbb{E}[\dot{p}|_{[t_0,\infty)}] = \int_{t_0}^{\infty} \dot{p}(t) dt = \lim_{t \to \infty} \frac{p(t) - p(t_0)}{t - t_0} \tag{4.1}$$

The choice of t_0 is arbitrary, in fact the weight of the transient phase fades away considering infinite executions. However, to ease the reasoning about these quantities, we can interpret t_0 as the end of the transient phase, e.g. when the last stage consumes the first data item in a pipeline.

The essential aspect to point out is that for the steady-state model the focus is on relations among the steady-state variables, rather than in their values. In this way it is possible to abstract from particular target platforms, and capture the class of all possible steady-state behaviors of an application.

The steady-state behavior of a node can be modelled associating to each computation $e_k(t)$ its activation rate

$$\bar{e}_k = \lim_{t \to \infty} \frac{e_k(t) - e_k(t_0)}{t - t_0} \tag{4.2}$$

Spontaneous computations are free variables in the steady-state model. Computations that are activated by data reception, instead, are subject to the following condition.

PROPOSITION 4.1. *The steady-state execution rate of a computation is bound to be equal to the input rates on the input interfaces that activate the computation.*

Proof. Let $k \in A_i$, we will prove that $\bar{e}_i - \bar{\imath}_k = 0$

$$\begin{aligned} \bar{e}_i - \bar{\imath}_k &= \lim_{t \to \infty} \frac{e_i(t) - e_i(t_0)}{t - t_0} - \lim_{t \to \infty} \frac{i_k(t) - i_k(t_0)}{t - t_0} \\ &= \lim_{t \to \infty} \frac{e_i(t) - e_i(t_0) - i_k(t) + i_k(t_0)}{t - t_0} \\ &= \lim_{t \to \infty} \frac{e_i(t) - i_k(t)}{t - t_0} - \frac{e_i(t_0) - i_k(t_0)}{t - t_0} \end{aligned}$$

The numerator of the first addend is limited by constants: (3.1) gives

$$e_i(t) - i_k(t) \leq 0$$

and (3.2) (noting that $e(t) \geq \lfloor e(t) \rfloor$) gives

$$e_i(t) - i_k(t) \geq -\tau_{1k}$$

while the numerator of the second addend is constant, so the limit tends to zero when the denominator tends to infinity. □

The data transmission rate $\bar{o}_k$ of an output stream will depend on the activation rates of one or more computations of the node. In the previous section, the number of data outputs has been related to the number of performed computations by means of a *transfer function* f_k (Eqn. (3.4)).

PROPOSITION 4.2. *If the transfer function is (asymptotically) linear*

$$o_k = f_k(e_1, \ldots, e_m) = \alpha_k^1 e_1 + \ldots \alpha_k^m e_m + c_k(e_1, \ldots, e_m)$$

with

$$\lim_{\|e\| \to \infty} \frac{\|c_k(e)\|}{\|e\|} = 0$$

then a steady-state is eventually reached, in which the output rate is a linear combination of the computation rates:

$$\bar{o}_k = \sum_{i=1}^{m} \alpha_{ki} \bar{e}_i \tag{4.3}$$

Proof.

$$\bar{o}_k = \lim_{t \to \infty} \frac{f_k(e(t)) - f_k(e(t_0))}{t - t_0} = \lim_{t \to \infty} \frac{\alpha_k \cdot (e(t) - e(t_0)) + c(e(t)) - c(e(t_0))}{t - t_0} =$$
$$\alpha_k \cdot \lim_{t \to \infty} \frac{e(t) - e(t_0)}{t - t_0} + \lim_{t \to \infty} \frac{c(e(t)) - c(e(t_0))}{t - t_0} = \alpha_k \cdot \bar{e} + 0 = \sum_{i=1}^{m} \alpha_k^m \bar{e}_i$$

□

The steady-state behavior of streams can be modelled by associating to each endpoint its data transmission rate. Balance equations relating input and output endpoints are derived.

PROPOSITION 4.3. *The steady-state transmission rate at the endpoints of a stream are characterised by the following balance equations:*

$$\textit{unicast: } \bar{o}_A = \bar{\imath}_B \tag{4.4a}$$
$$\textit{merge: } \bar{o}_A + \bar{o}_B = \bar{\imath}_C \tag{4.4b}$$
$$\textit{broadcast: } \bar{o}_A = \bar{\imath}_B = \bar{\imath}_C \tag{4.4c}$$

These equations are easily extended in the case of more endpoints.

Proof. The proof is similar to the one of Prop. 4.1, exploiting:

(i) (3.5) and (3.6) for unicast,
(ii) (3.9) for merge,
(iii) (3.10) for broadcast.

□

The *execution rate* for each computation, and the *data transfer rate* for each input/output interface completely specify the application state from the point of view of its performance, therefore we will call them the **performance features** of our application.

Proposition 4.2 allows us to express output rates as linear combinations of execution rates, provided that we know the related coefficients. These coefficients must be provided by developers of programs/components

by means of some **performance annotations**, in order to build a performance model. Proposition 4.1 allows us to eliminate execution rates associated to data-dependent computations. Proposition 4.3 allows us to relate output rates to input rates of linked modules.

The **performance model** is therefore defined as an homogeneous system of simultaneous linear equations, that describe the relations that hold in the steady-state among the **performance features**. The set of solutions of the system is a vector subspace of $\mathbb{R}^n$ (where n is the total number of variables, either input rates, output rates or execution rates); we call the dimension of the solution space the number of **degrees of freedom** of the application. If this dimension is 1, then the system is completely determined as soon as a single value for any variable is imposed. The degenerate case of a space with dimension 0 implies that the only solution to the system is the null vector (i. e. every variable must be zero): this means that the predicted steady-state is a deadlock state, in which no computation or communication can proceed. The number of degrees of freedom of the system will impact on how many constraints must be provided in order to derive the expected values for every variable.

Clearly, only positive values of the rates are meaningful, so we can conclude that every assignment of positive values for the vector $[\mathbf{i}\,\mathbf{e}\,\mathbf{o}]^T \in \mathbb{R}^n$ that is a solution of the system is a possible "operation point" for the modeled application.

The outlined approach is efficient, in fact the simplification of the simultaneous equations can be achieved using well known techniques.

5. Application of the Model. We show how the presented model can be applied to a real application (see Fig. 5.1), a rendering pipeline. The first stage requests the rendering of a sequence of scenes while the second renders each scene (exploiting the PovRay rendering engine), interpreting a script describing the 3D model of objects, their positions and motion. The third stage collects images rendered by the second one, and builds Groups Of Pictures (GOP), that are sent to the fourth stage, performing DivX compression. The last stage collects DivX compressed pieces and stores them in an AVI output file.

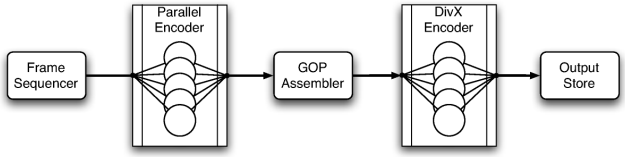


FIG. 5.1. *Graph of the render-encode application*

For GOPs of 12 pictures, the performance model for our test application is (we eliminated execution rates for data-dependent computations):

$$C_{1e} = C_{1o} = C_{2i} = C_{2o} = C_{3i} = 12 \cdot C_{3o} = \\ = 12 \cdot C_{4i} = 12 \cdot C_{4o} = 12 \cdot C_{5i}$$

and has one degree of freedom.

5.1. Convergence to Steady State. We start showing that the application behavior actually tends to steady-state.

Figure 5.2 shows performance features taken from a real execution of the test application on a Blade cluster consisting of 32 computing elements, each equipped with an Intel Pentium III Mobile CPU at 800MHz and 1GB of RAM, interconnected by a switched Fast Ethernet dedicated network. The application was configured to exploit 20 machines in the render computation, and one machine for each remaining node.

Performance features are measured as in (4.2), i. e. averaging the number of performed computations on the duration of the execution. The top diagram shows the performance of the Render and the GOP Assembler nodes, which operate on frames, while the bottom diagram shows the Encoder and Collector nodes, which operate on GOPs. The similarity of the curves in the left and the right diagrams shows empirically that Prop. 4.2 is satisfied not only at the steady-state, but also during the finite computation, as soon as buffers are filled (curves in the same diagram are related by a factor of 1, while between the two diagrams there is a scaling factor of 12).

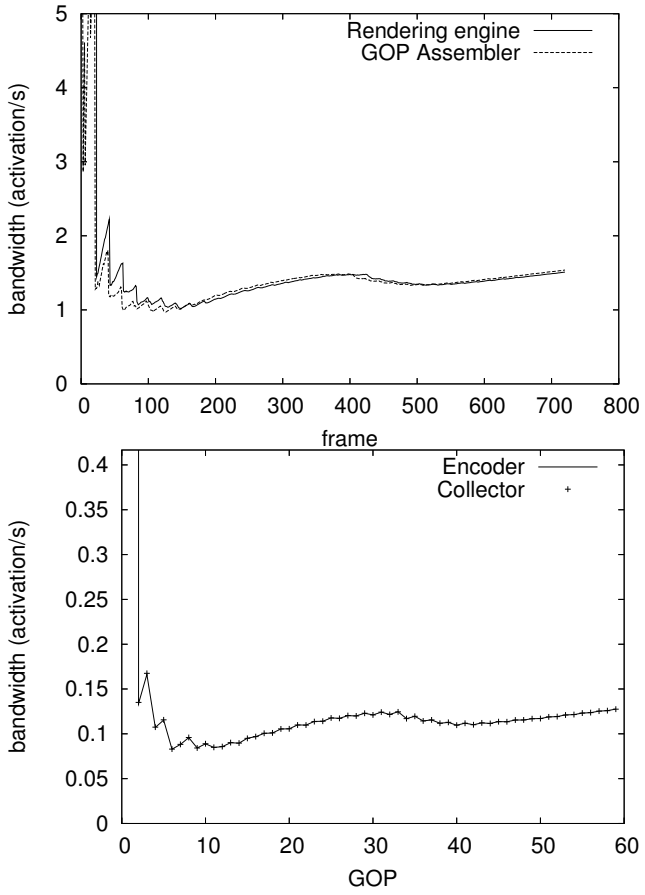


FIG. 5.2. *Convergence to steady-state of averaged performance features*

Moreover, Fig. 5.2 shows that the averaged computation rates stabilize during the computation, allowing us to adopt a steady-state model to approximate the actual application run.

5.2. From Desired Performance to Resource Requirements. Typically, if someone is facing a problem by means of HPC tools, he has clear in mind some sort of performance requirement for his application. This can be expressed in different forms, e.g. completion time, computation rate, response time, etc. In our framework we express requirements as bounds on computation rates. That is the most natural way dealing with stream parallelism. This means that, if the problem is expressed in different terms, some sort of preliminary transformation should be applied (e.g. study the initial transient length to relate completion time to computation rate, or use the Little's Law to translate response time requirements in computation rate ones).

Suppose that we require 1 frame/s (the constraint is expressed by $C_{5i} \geq \frac{1}{12}$, because each input for C_5 is composed by 12 frames). Applying the performance model we derive required computation and transfer rates for each computation and communication.

These values, paired with program annotations (see Tab. 5.1) on the weight of computation or communication (e.g. MFLOP per task/MB transferred to/from memory and message size, respectively) can be used to derive requirements that the resources must fulfill in order to meet the performance requirements on the application.

For instance, we can show the requirement for stream $S_2 = C_{2o}$. Since it is required to carry 1.19MB messages with at least rate 1/s, a link of 9.5 Mbit/s is sufficient. Likewise, the test application will never scale above 10 frames/s with a 100 Mbit/s network, and needs to be redesigned, if we want to reach higher performances.

Computational requirements are handled in the same way. The performance model solution gives, for each computation, the minimum required execution rate. Then we need an invertible performance model for

TABLE 5.1
Deployment annotations for the example application.

Component	C_1	C_2	C_3	C_4	C_5
Processor	i686	i686	i686	i686	i686
Memory (MB)		64	256	64	
CPU Work		3307		52	
Mem. Work		302		104	

Connector	S_1	S_2	S_3	S_4
data type	param	pic	GOP	zip
data size	54B	1.19MB	14.24MB	2MB

each atomic computation that, given the required execution rate, produces the resource requirements. This is essential in an execution environment in which resources are not known in advance.

The model presented in [22] suits our needs. We can associate to each computation a weight, represented by a pair of values $w = (w_{MFLOP}, w_{MB})$, specifying the number of floating point operations (expressed in MFLOP) and the data transferred to/from main memory (expressed in MB) per activation. Resource power is described by the pair $p = (p_{MFLOP/s}, p_{MB/s})$, and execution time is therefore estimated as $t(p, w) = \frac{w_{MFLOP}}{p_{MFLOP/s}} + \frac{w_{MB}}{p_{MB/s}}$.

This model can be employed also to find appropriate parallelism degree for parallel computation nodes. We, in fact, can relate $t(p, w)$ for an aggregate resource $p = [p_1, \ldots, p_k]$ to the performance of the code on single resources $t(p_i, w)$.

Assuming perfect speedup, we obtain:

$$t(p, w) = \Big(\sum_i t(p_i, w)^{-1} \Big)^{-1}$$

In this way we can derive, for each computation node, matching resource requirements. These will concern single resources for sequential nodes, and aggregate ones for parallel nodes.

Results commented. In Fig. 5.3, two mappings (top on an homogeneous cluster, bottom with heterogeneous resources) for the same constraint are displayed. The first thing to note is that, even if the heterogeneous run has more variance in achieved bandwidth, the average bandwidth is comparable with the homogeneous one. This provides evidence that the employed performance model correctly handles heterogeneous sets of resources, determining the correct parallelism degree. The good performance in heterogeneous run (its completion time is even shorter than the one for homogeneous run) is explained by the fact that the model can match computation requirements with suitable resources, i. e. schedule memory bound computations (e.g. encoding) on machines with faster memory, and FPU bound ones (e.g. rendering) on machines with faster FPU.

The obtained results are as expected: the mapping computed using the performance model fulfills the constraint, at the beginning and most of the time of the application run. This occurs because, in order to build our model, we sampled the achieved performance on the first frames of the movie, but the application workload slightly changes with the evolution of the movie. This is evidenced by the smoothed bandwidth curve, that has the same course in the two experimental settings: the workload is heavier around 100s and 300s, while it is lighter in the middle and at the end.

6. Conclusions and Future Work. In this work we described an analytical approach to map a class of applications on a Grid. These applications interact through streams of data, processed by several autonomous software components, either sequential or parallel. We presented a steady state performance model for these applications and we applied it to a case study, a rendering pipeline of sequential and parallel components. The model was exploited to predict a program behavior at run-time. Then we showed a general methodology to devise a correct initial mapping for the application, driven by specified QoS levels. At last, we showed the results of our mapping methodology with the presented application, and we discussed the results of the mapping and the execution on homogeneous and heterogeneous sets of resources. We obtained good results in both cases. The application was correctly mapped and the QoS requirement respected with a small error.

Analytical [35, 19, 36, 20, 18, 29] and structural performance models [25, 14, 8] discussed in Sect. 2 need the full knowledge of the target platform to derive performance measures. Therefore, to compare results of different mappings, they must be evaluated multiple times. Our approach decouples the modeling

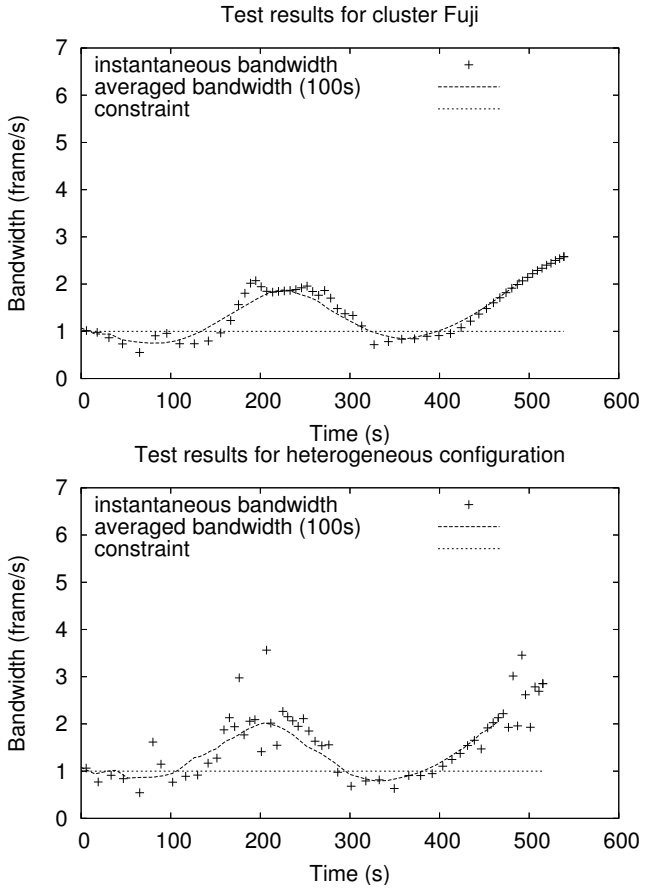


FIG. 5.3. *Two executions of the test application: top) homogeneous clusters of Athlons XP 2600+, down) set of heterogeneous resources (9 P4@2GHz, 1 Athlon XP 2800+, 1 P4@2.8GHz).*

of the application performance from the target platform, allowing us to evaluate the model once to derive enough information to drive the mapping process. Trace-based approaches [34, 26] are used to overcome the limitations of previously discussed approaches, but they are not compositional. Therefore they must be applied from scratch to every new application, even if it is built from the same set of components.

All those models and the presented one share an assumption on the behavior of the applications: computation executions must be independent from the actual values of the input set. Otherwise, two executions of the same application would be not comparable (this is called ergodicity for stochastic models). For applications that do not meet this requirements, the best solution is to resort to runtime adaptation.

The presented approach is not perfect. The initial mapping can be considered a good "hint" to start the execution of an application on a Grid. The dynamic changes in resources during the execution can not be easily included in launch-time strategies. Our approach must be coupled with rescheduling strategies at runtime to solve such problems. Our future work is going in this direction. The presented steady state model can be exploited at run-time to adapt the behavior of components to changes in resource performances. In this way, it should be possible to fulfill the QoS requirements during the whole execution of the application.

REFERENCES

[1] M. ALDINUCCI, A. PETROCELLI, E. PISTOLETTI, M. TORQUATI, M. VANNESCHI, L. VERALDI, AND C. ZOCCOLO, *Dynamic reconfiguration of grid-aware applications in ASSIST*, in Proc. 11th Euro-Par Conference, Lisboa, Portugal, Aug. 2005.

[2] P. AMMIRATI, A. CLEMATIS, D. D'AGOSTINO, AND V. GIANUZZI, *Using a structured programming environment for parallel remote visualization.*, in Proc. 10th Euro-Par Conference, Pisa, Italy, Sept. 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and issues in data stream systems*, in Proc. 21st ACM-SIGMOD-SIGACT-SIGART Symposium on Principles of database systems (PODS'02), Madison, USA, 2002, pp. 1–16.

[4] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, *SkIE: a heterogeneous environment for HPC applications*, Par. Comp., 25 (1999), pp. 1827–1852.

[5] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, *Model-Based Performance Prediction in Software Development: A Survey*, IEEE Trans. on Software Engineering, 30 (2004), pp. 295–310.

[6] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, *Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processors platforms*, IEEE Trans. on Parallel and Distributed Systems, 15 (2004), pp. 319–330.

[7] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, *Steady-State Scheduling on Heterogeneous Clusters: Why and How?*, in Proc. of 18^{th} International Parallel and Distributed Processing Symposium (IPDPS 04) (IPDPS'04), April 2004.

[8] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, *Scheduling Skeleton-Based Grid Applications Using PEPA and NWS*, The Computer Journal, 48 (2005), pp. 369–378.

[9] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, *The GrADS Project: Software Support for High-Level Grid Application Development*, Int. J. of High Performance Computing Applications, 15 (2001), pp. 327–344.

[10] D. Bertsimas and D. Gamarnik, *Asymptotically optimal algorithm for job shop scheduling and packet routing*, Journal of Algorithms, 33 (1999), pp. 296–318.

[11] M. Cole, *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming*, Par. Comp., 30 (2004), pp. 389–406.

[12] M. Coppola and M. Vanneschi, *High-Performance Data Mining with Skeleton-based Structured Parallel Programming*, Par. Comp., Sp. Iss. on Parallel Data Intensive Computing, 28 (2002), pp. 793–813.

[13] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi, *HPC Application execution on Grids*, in FGG: Future Generation Grid, CoreGRID, Springer, 2006.

[14] M. Dìaz, B. Rubio, E. Soler, and J. M. Troya, *SBASCO: Skeleton-based Scientific Components*, in Proc. of 12^{th} Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'04), A Coruña, Spain, February 2004.

[15] W. Du and G. Agrawal, *Language and compiler support for adaptive applications*, in Proc. 2004 ACM/IEEE Conference on Supercomputing (SC'04), Pittsburgh, USA, Nov. 2004.

[16] N. Dumitrascu, S. Murphy, and L. Murphy, *A Methodology for Predicting the Performance of Component-Based Applications*, in Proc. of 8^{th} International Workshop on Component-Oriented Programming (WCOP 03), Darmstadt, Germany, July 2003.

[17] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Pub., July 1998.

[18] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo, *Software performance modelling using PEPA nets*, in Proc. of 4^{th} International Workshop on Software and Performance (WOSP 04), New York, NY, USA, 2004, ACM Press, pp. 13–23.

[19] K. Kant, *Introduction to Computer System Performance Evaluation*, McGraw-Hill, 1992.

[20] P. J. B. King and R. Pooley, *Derivation of Petri Net Performance Models from UML Specifications of Communications Software*, in Proc. of 11^{th} International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 00), London, UK, 2000, Springer-Verlag, pp. 262–276.

[21] H. Kuchen, *A Skeleton Library*, in Proc. 8th Euro-Par Conference, London, UK, Aug. 2002.

[22] A. Litke, A. Panagakis, A. D. Doulamis, N. D. Doulamis, T. A. Varvarigou, and E. A. Varvarigos, *An advanced architecture for a commercial grid infrastructure.*, in European Across Grids Conference, M. D. Dikaiakos, ed., vol. 3165 of Lecture Notes in Computer Science, Springer, 2004, pp. 32–41.

[23] L. Marchal, Y. Yang, H. Casanova, and Y. Robert, *A realistic network/application model for scheduling divisible loads on large-scale platforms*, in Proc. of 19^{th} International Parallel and Distributed Processing Symposium (IPDPS 05) (IPDPS'05), April 2005.

[24] J. Ray, N. Trebon, R. C. Armstrong, S. Shende, and A. D. Malony, *Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study*, in Proc. of 18^{th} International Parallel and Distributed Processing Symposium (IPDPS 04), Santa Fé, USA, April 2004.

[25] J. Schopf, *Structural prediction models for high-performance distributed applications*, in Proc. of the Cluster Computing Conference (CCC'97), Atlanta, USA, March 1997.

[26] L. J. Senger, M. J. Santana, and R. H. C. Santana, *Using Runtime Measurements and Historical Traces for Acquiring Knowledge in Parallel Applications*, in Proc. of the 2004 International Conference on Computational Science (ICCS 04), M. Bubak, G. D. van Albada, P. M. Sloot, and J. J. Dongarra, eds., vol. 3036 of Lecture Notes in Computer Science, Kraków, Poland, June 2004, Springer Verlag, pp. 661–665.

[27] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, *Performance specification of software components*, in Proc. of the 2001 Symposium on Software Reusability (SSR 01), Toronto, Ontario, Canada, 2001, ACM Press, pp. 3–10.

[28] C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[29] B. Spitznagel and D. Garlan, *Architecture-Based Performance Analysis*, in Proc. of 10^{th} International Conference on Software Engineering and Knowledge Engineering (SEKE 98), Y. Deng and M. Gerken, eds., 1998, pp. 146–151.

[30] N. Trebon, A. Morris, J. Ray, S. Shende, and A. Malony, *Performance Modeling of Component Assemblies with TAU*, in Proc.of CompFrame 2005, Atlanta, USA, June 2005.

[31] S. Vadhiyar and J. Dongarra, *Self Adaptability in Grid Computing*, Concurrrency and Computation: Practice and Experience, 17 (2005), pp. 235–257.

[32] A. J. C. van Gemund, *Symbolic Performance Modeling of Parallel Systems*, IEEE Trans. on Parallel and Distributed Systems, 14 (2003), pp. 154–165.

[33] M. Vanneschi, *The programming model of* ASSIST*, an environment for parallel and distributed portable applications*, Par. Comp., 28 (2002), pp. 1709–1732.

[34] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed, *Performance Contracts: Predicting and Monitoring Grid Application Behavior*, in Proc. of 2^{nd} International Workshop on Grid Computing (GRID 01), London, UK, 2001, Springer-Verlag, pp. 154–165.

[35] L. G. Williams and C. U. Smith, *PASA(SM): An Architectural Approach to Fixing Software Performance Problems*, in Proc. of 28^{th} International Computer Measurement Group Conference, Reno, Nevada, USA, 2002, pp. 307–320.

[36] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, *The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software*, IEEE Trans. on Computer, 44 (1995), pp. 20–34.

[37] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, *Performance modeling and prediction of enterprise javabeans with layered queuing network templates*, in Proc. of the 2005 Conference on Specification and Verification of Component-based Systems (SAVCBS 05), New York, NY, USA, 2005, ACM Press.

[38] A. Zhang, Y. Song, and M. Mielke, *NetMedia: Streaming Multimedia Presentations in Distributed Environments*, IEEE MultiMedia, 9 (2002), pp. 56–73.

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 277–288. http://www.scpe.org
ISSN 1895-1767

HIGH PERFORMANCE COMPUTING THROUGH SOC COPROCESSORS

GIANNI DANESE, FRANCESCO LEPORATI, MARCO BERA, MAURO GIACHERO, NELSON NAZZICARI, AND ALVARO SPELGATTI*

Abstract. In this paper we describe DPFPA (Double Precision Floating Point Accelerator), a FPGA-based coprocessor interfaced to the CPU through standard bus connections; it is conceived to accelerate double precision floating point operations, featuring two double precision floating point units, a pipelined adder and a pipelined multiplier with a suitable number of stages. We tested its performance by implementing a Montecarlo-Metropolis simulation of a dipolar system, using a proper software development environment designed and realized in our laboratory. DPFPA can provide a speed-up equal to 4, with respect last generation PC, showing also a good scalability in terms of clock frequency, memory capability and number of computing units.

Key words: FPGA; hardware accelerator; high performance embedded system; parallel processing.

1. Introduction. Scientific research owes a lot to computer systems which allowed the achievement of results otherwise unthinkable [Marsh, 2005][Boghosian et al., 2005]. A powerful computing system permits the study of several phenomena through the employment of simulations like statistical ones into which the system under analysis is made to evolve from a certain initial condition, by modifying a few of its characteristic parameters and by evaluating the feasibility on the basis of a proper merit function. These operations are iterated thousands of times to bring the system in a new stable state.

Several of these simulations perform double precision floating point operations since they provide the accuracy required to appreciate even the smallest fluctuations in the typical variables of the simulated phenomena. On the other hand, this could represent a hard task even for the most powerful processors which take a lot of clock cycles to execute a single floating point operation.

The lack of computing power is generally overcome by resorting to supercomputers or clusters [Dongarra et al., 2005] but in the last years the use of accelerators, i. e. dedicated hardware systems, is gradually establishing as a valid alternative, due to the feature of these devices which allow to perform those operations in less time than traditional processors [Buell et al., 2007][Herbordt et al., 2007]. Several researchers worked in these years not only in this sense but also to improve "methodology, tools and practices supporting the integration of hardware and software components during system design and development" [Hankel et al., 2003][Wolf, 2003].

At present a similar project concerning a Double Precision Floating Point Accelerator (DPFPA) to process complex functions has been carried out in the Microcomputer laboratory at the University of Pavia (Italy). This activity suites well with the mission of the laboratory which aims to design and develop special purpose architecture for computationally intensive applications. The designed accelerator is implemented onto a FPGA device lodged on a board interconnected with a Personal Computer and is able to execute floating point operations faster than a traditional processor [Danese et al, 2007]. Moreover, a proper specific programming language and a suitable software development environment were realised allowing the user to write, translate and load proper instructions sequences written in a specific language.

This paper describes the implementation, onto the accelerator, of a Montecarlo-Metropolis simulation of a dipolar system, a typical computational challenge for supercomputers.

The Montecarlo Metropolis algorithm is an excellent benchmark to test performance of a special purpose calculation system, since its computational core consists of few floating point operations (double precision) repeated over and over: this represents the ideal condition to exploit an application specific architecture devoted to the acceleration of only particular instructions.

Moreover, the algorithm features a SIMD fashion so it is suitable for a distributed implementation which can exploit more calculation units so increasing the overall achieved speed up.

Finally, typical Montecarlo simulations involve hundreds thousands particle systems and can run for weeks or months on the most performing computers with a single CPU: the availability of powerful accelerating units, in case connected into a cluster configuration, makes possible simulations currently unfeasible or simulations with more particles than now, achieving a better comprehension of the physical phenomena under analysis.

* Department of Informatica and Sistemistica, Pavia University, Italy, E-mail: francesco.leporati@unipv.it, Phone: +39 0382 985678

In the past other research groups proposed accelerators based on FPGA for Montecarlo simulations:

- one of the first proposal is presented in [Postula et al., 1996] where is described a metallurgical sintering simulation implemented on a FPGA device with a two orders of magnitude speed-up with respect to a mid 90's workstation;
- in the same years, other authors conceived a FPGA implementation of a particular Montecarlo technique (Swendsen-Wang clustering) with a considerable acceleration with respect to a 15 MHz DSP or making use of cellular automata [Cowen et al, 1994][Monaghan et al, 1992];
- more recently, a reconfigurable computer was designed devoted to heat transfer simulations, working on single precision floating point data and achieving an order of magnitude speed-up relative to a 3 GHz P4 processor [Gokhale et al, 2003]; the peculiarity of this contribute is the idea of using widely available floating point libraries for implementing a calculation function onto FPGA, thus shortening design time;
- finally, in [Zhang et al, 2005] it is presented a simulation of a financial model implemented on a FPGA device to accelerate double precision floating point calculations. The achieved speed-up is 26 relative to a 1.5 GHz P4 processor;
- with regard to FPGA based architectures specifically devoted to physics simulations, the recent literature proposed the works of Cruz and Belletti [Cruz et al, 2001][Belletti et al, 2006]; the first one provides interesting architectural issues although using Altera Flex 10K30 components limits the working frequency to 48 MHz; the second is a project subsequent to our one, employing Altera Stratix family components and aims to build a cluster of accelerators based on the most recent FPGA devices.

For what concerns a more general use of SoC for computing intensive applications there is a wide literature to which the reader could refer. The most part of the October 2007 issue of IEEE Computer was devoted to that topic [Wolf, 2007].

In the next section the architectural features of the accelerator, of the specific language designed and of its software development environment will be described. Then, the basic physical principles of the simulation and its needed modifications for optimizing the use of the accelerator will be highlighted. Finally, we will see the implementation of the algorithm on the accelerator, taking advantage from the use of a 'dedicated stage' pipeline and the comparison with a few commercial and popular processors showing a clear speed-up. Some remarks explaining the evolution of the project will conclude the paper.

2. The Accelerator. We realized a FPGA-based accelerator connected to a host PC to accelerate the hardest part of a calculus. Our idea refers to a board with a FPGA device (Altera Stratix family) and a Flash memory storing the configuration code; a JTAG port is used to send the program to the Flash memory from the PC. Recently, Altera has made available some boards with these features. These boards can communicate with PC through the network requiring a proper network manager. In this case, both the accelerators and the network processor can be loaded on the same FPGA. The board we bought is equipped with a Stratix 1S40 FPGA component on which a 32 bit RISC CPU, called Nios, is implemented; this processor can be programmed using C language and is supplied with basic libraries to easily handle the on board devices: 2 MB Ram, 8 MB Flash Memory, 16 MB Compact Flash Memory, 100 Mb/s Ethernet Interface, 2 Serial ports.

We designed an accelerating unit that is able to implement different functions (also complex like sin, cos, log, ..., through Taylor series). Thus, it can be used for several applications, also very diversified. Moreover, the instruction set is fully re-programmable according to the particular calculation to be performed.

The designed unit (DPFPA) can exploit the parallelism present in the operations since double precision Floating Point MAC operations can be executed at the same time in the sum and multiply pipelines present onto it. The main part of DPFPA is DPFPP (double precision floating point processor), whose architecture consists of (fig. 2.1):

- 2 accelerating units, independently working;
- a Cache Memory (4 banks), which can store input data and results for the two accelerating units;

A suitable bus devoted to communication between Accelerator and Nios processor ("sub bus") has been also implemented. The Math Unit functional core is a double-precision floating-point ALU, which integrates both an adder and a multiplier operating in a parallel fashion. Both devices are pipelined (9 stages for the adder and 15 for the multiplier) so that high clock rates are achievable. Note that, in the expected applications, accurate coding can minimize the negative effects of such latency.

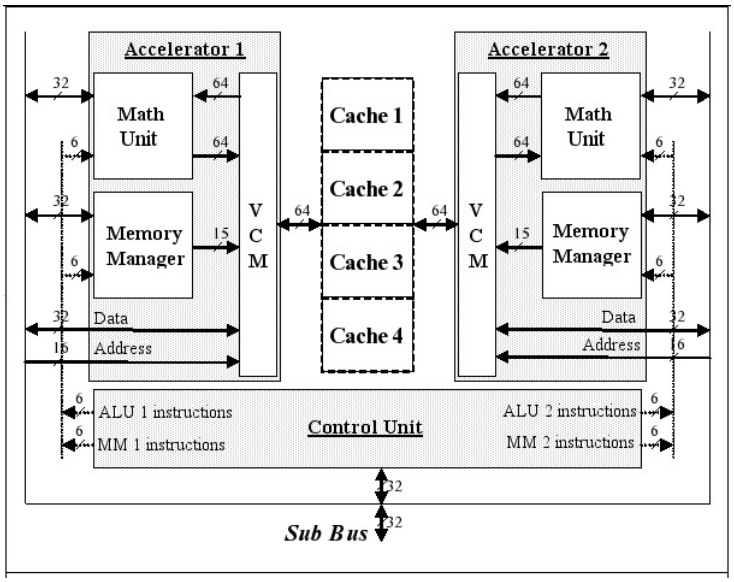


FIG. 2.1. *Architecture of the computational unit implemented onto the FPGA device.*

Together with the adder and the multiplier, the ALU also contains 3 register banks, each able to store 4 double-precision floating point numbers. The banks are each tied to a particular purpose (one is for input data, one for adder results and one for multiplier results).

Like in many similar applications, to make computing elements and storage space independent, a FIFO memory for both inputs and outputs is implemented (there are two FIFO queues on the output since arithmetic results are separated from logical ones).

The ALU operations are encoded in 37-bit words, able to simultaneously trigger either a sum or a comparison, a multiplication, a data fetch, 3 write operations to the internal register banks and the output of a result.

To achieve better performance with our specific task, the operands of the adder can optionally be multiplied by $[-2, -1, 2]$ for the first operand, and $[-1, -0.5, 0.5]$ for the second one. In a similar way, the multiplier result can be doubled, halved or negated without extra clock cycles.

Since feeding the op-codes would require a large and mostly wasted bandwidth (the code is essentially cyclic, so that the same op-codes are executed over and over again) the code sequences are stored in a Microcode Sequencer. This device stores the program sequences in an internal RAM and associates to them a 6-bits op-code (this is much like having a CPU with a micro-programmed control unit whose code can be changed by the application to define a custom instruction set).

The Math Unit itself has no addressing capabilities toward either input or output channels, so every memory I/O operation must be managed by an external device. A Memory Manager was deemed to that task and conceived for a specific application class: those where most computations are performed on data logically organized in three-dimensional matrices. Decoupling the allocation issues from the computing algorithm, the Memory Manager computes the memory addresses from semantic-level inputs, such as addresses in the matrix domain ($X - Y - Z$ coordinates) or offsets between elements (the matrix is supposed to be cyclic, so that e.g. the leftmost element in a row is adjacent to the rightmost element in the same row). This is of extreme importance, since otherwise the same code would require at least a recompilation to be executed on matrices with different sizes.

The internal Control Unit (CU) decodes instructions coming from the host computer and drives the control signals implementing the requested function. It mainly consists of 3 units:

- **Instruction Decode**: selects between data and instructions from host to the DPFPA. Only in the last case it generates proper control signals;
- **Jump Unit**: sets the RAM address to the starting point of the next instruction sequence to be executed;
- **RAM**: stores sequences corresponding to the instruction set for the particular function to implement.

Instructions are 64 bit wide exploiting part of the redundancy present in the IEEE 754 standard of floating point representation, to distinguish them from double precision numbers. Two kinds of instructions have been implemented:

- *Programming instructions* to store in the CU RAM executive sequences.
- *Executive instructions* to perform specific calculations, recalling sequences already loaded.

Programming instructions to store in the CU RAM executive sequences. Executive instructions to perform specific calculations, recalling sequences already loaded.

A great advantage of our approach is that the sequences of an executive instruction are performed in an iterative manner until a new executive instruction will be received by the CU. So, during the execution of the calculus, CU has to decode only few instructions and can save a great amount of time.

3. Programming DPFPP. As previously stated, DPFPP can handle two types of instructions: programming instructions and executive instructions. The former are used to store microcode sequences into the CU RAM, making microcode words to be loaded at the correct address into the RAM of CU. The word of microcode, allows the assertion of needed control signals for each clock cycle.

Each executive instruction allows, on the other hand, the recalling of sequences already stored.

We realised soon, that the sequence development using binary microcode was a very hard and inefficient work. Thus, we chose to design and develop a pseudo-assembly dedicated language that simplifies the sequence writing. The instructions of the language are mapped directly on the hardware and reflect the operation that DPFPP can execute. Table 3.1 shows the list of the instructions and their syntax.

TABLE 3.1
List and syntax of the language instructions.

Instruction Syntax
MOV reg;
SUM c1 op c2 op ; SUM c1 op ; SUM c1 op op SUM op c2 op; SUM op op
MUL c op op; MUL op op; MUL c op;
OUT xx;
INT;

A proper translator was also developed, using standard Unix tools such as Lex and Yacc.

Furthermore, we developed an allocator for an easy generation of the file with the programming instructions that must be sent to the DPFPP. Finally, we designed a simulator, reproducing exactly the DPFPP working and enabling pipeline and register inspection. The simulator also allows the visualisation of the clock cycles needed by a specific sequence or by a set of sequences. Thanks to this tool, we can execute microcode sequences without loading them into the DPFPP; thus, we can simplify the sequence debug, verify the results' correctness and check the performance.

All these tools are integrated in a unique development environment, realised in the Microcomputer laboratory to ease the sequence development. There are four main steps: first, we write and compile source code using an internal editor, then we test the code using the simulator. Finally, we produce the programming file that has to be sent to the DPFPP by using the allocator. More details on the hardware and software for DPFPA are in [Danese et al., 2003].

4. The Considered Problem. Liquid crystals and colloidal suspensions are two examples of systems for which the orientation order has been widely studied through simulations. In both cases interactions among particles play a dominant role. In previous works, we realized a cubic lattice model describing the interactions effects in a dipolar system in presence of an external lattice field [Bellini er al., 2001]: simulations made with this model identified the presence of two phase transitions and the obtained results could in part explain the phenomenon known as "anomalous bi-refringence" as analyzed in [O' Konski et al., 1950][Radeva et al., 1996].

On the other hand, simulations take unacceptably long times even on the most recent and powerful computing systems ranging from a few days up to some weeks depending on the size of the simulated system. The core of the computation is, in fact, the evaluation of the energy since, according to the implemented algorithm (Montecarlo-Metropolis), equilibrium in a system with N particles is reached through a sequence of moves, carried out by randomly selecting a spin, changing its orientation through a random angular displacement and

evaluating the corresponding change in energy. Each move can be accepted or rejected depending on the variation of the energy associated with it [Metropolis et al., 1953]. We simulated lattice systems with particles ranging from a few hundreds up to 100.000 considering only first neighbor interactions, i. e. the interaction between each spin and the six closest ones in the $X+$, $X-$, $Y+$, $Y-$, $Z+$, $Z-$ directions. Periodic boundary conditions were applied [Frenkel et al., 1996]. The associated energy of each dipole due to the presence of an external field oriented toward z axis is:

$$E_{dip} = mom_z(dip) \tag{1}$$

The terms due to the interactions between the considered dipole and each of its first neighbours are:

$$\begin{aligned}
E_{X+} &= 2 * mom_x(dip) * mom_x(X+) + \\
&\quad -mom_y(dip) * mom_y(X+) - mom_z(dip) * mom_z(X+) && (2)\\
E_{X-} &= 2 * mom_x(dip) * mom_x(X-) + \\
&\quad -mom_y(dip) * mom_y(X-) - mom_z(dip) * mom_z(X-) && (3)\\
E_{Y+} &= -mom_x(dip) * mom_x(Y+) + \\
&\quad +2 * mom_y(dip) * mom_y(Y+) - mom_z(dip) * mom_z(Y+) && (4)\\
E_{Y-} &= -mom_x(dip) * mom_x(Y-) + \\
&\quad +2 * mom_y(dip) * mom_y(Y-) - mom_z(dip) * mom_z(Y-) && (5)\\
E_{Z+} &= -mom_x(dip) * mom_x(Z+) + \\
&\quad -mom_y(dip) * mom_y(Z+) + 2 * mom_z(dip) * mom_z(Z+) && (6)\\
E_{Z-} &= -mom_x(dip) * mom_x(Z-) + \\
&\quad -mom_y(dip) * mom_y(Z-) + 2 * mom_z(dip) * mom_z(Z-) && (7)
\end{aligned}$$

where the components of the moments for each dipole are:

$$mom_x(dip) = cos(\theta) * sin(\theta) * cos(\varphi) \tag{8}$$

$$mom_y(dip) = cos(\theta) * sin(\theta) * sin(\varphi) \tag{9}$$

$$mom_z(dip) = cos'(\theta) \tag{10}$$

and θ, φ are the angular co-ordinates of a generic dipole. The overall energy of the dipole is the sum of all these contributes:

$$E_{TOT}[dip] = -0,5 * [E_{dip} - k * (E_{X+} + E_{X-} + E_{Y+} + E_{Y-} + E_{Z+} + E_{Z-})] \tag{11}$$

The global energy in the system is the sum extended on the whole dipolar set.

The simulated system is characterised by an initial random particle distribution not corresponding to that achievable at the equilibrium. This means that the change in the orientation of a dipole will modify the moments and the energy in the others, mainly in the neighbours. These ones, in turn, will influence their respective neighbours and so on, propagating those variations in the moments throughout the lattice. This reflects in energy fluctuations that disappear only after a sufficient number of cycles into which ETOT for each dipole is calculated (equilibration). Only at this point, the Metropolis test on energy variation can be applied. This loop series corresponds to nearly the 85% of the calculation but it consists of only few instructions, so justifying the idea of an accelerator specialized in processing only those operations. To do this, we employed the FPGA technology, which is cheaper and simpler than ASIC in terms of design and test.

However, during the design phase, we considered convenient to realise a more general chip able to accelerate those double precision floating point instructions which can be often found in scientific simulations. This extends the applicability of the DPFPA both to models different to that used (i. e. hexagonal lattices instead of cubic ones) or to completely different fields where high performance computing is mandatory.

5. Energy Evaluation and Implementation. To simplify the readability of the energy calculation on the DPFPA, as it will be described in the following, let's rewrite the expressions reported in section 3. The interaction energy of each dipole can be written as $-\frac{(CT*CT)}{2}$ and the sum on all the dipoles will return the

global energy in the system. CT is the local field generated by the neighbors of the considered dipole and can be expressed as:

$$CT = CTX * SC * k + CTY * SS * k + CTZ * C * k + C \tag{12}$$

where k is a constant depending on the system density and $SC = sin(\theta)cos(\varphi)$, $SS = sin(\theta)sin(\varphi)$, $C = cos(\theta)$, with θ, φ angular co-ordinates of the dipole. CTX, CTY e CTZ are the local components of the field generated by the neighbour dipoles. They are respectively equal to:

$$CTX = (MXX + MX * X + MYX + MY * X + MZX + MZ * X) \tag{13}$$

$$CTY = (MXY + MX * Y + MYY + MY * Y + MZY + MZ * Y) \tag{14}$$

$$CTZ = (MXZ + MX * Z + MYZ + MY * Z + MZZ + MZ * Z) \tag{15}$$

We identify with MXX, MXY, MXZ the local field components generated by the first neighbor dipole in the direction $X-$, and with $MX * X$, $MX * Y$, $MX * Z$ the local field components generated by the first neighbor dipole in the direction $X+$. The other terms due to the effect of dipoles in directions $Y+/Y-$ and $Z+/Z-$ are defined accordingly to the same notation. Moreover the local field, due to the neighbors, changes the components of the dipolar moment. These should be evaluated each time according to the following expressions:

$$mom_x(dip) = CT * SC, \quad mom_y(dip) = CT * SS, \quad mom_z(dip) = CT * C \tag{16}$$

While the SC, SS and C terms are evaluated at each movement, the other terms should be re-calculated for the number of cycles necessary to equilibrate the energy in the system. All these operations, finally, are repeated $M * N$ times with M =cycle number (i. e. 10.000) and N = number of dipoles in the system. This accounts for the high computational weight of the elaboration.

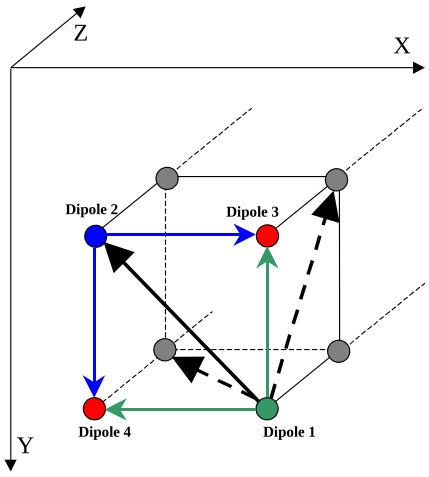


FIG. 5.1. *Diagonal scanning.*

With '*scanning*' we mean the order through which the dipoles are processed during the simulation. The identification of a suitable order can significantly affect the algorithm efficiency in terms of memory access and reuse of data. If we would not use any particular scanning order but if we only would consider dipoles in the same order of memorization (1^{st}, 2^{nd}, 3^{rd}, ...), their elaboration would need 21 input data (SC, SS and C of the moved dipole plus the moments of its six first neighbors), returning the 3 new components of the moment of the considered dipole.

However, if the selection order considers dipoles close to each other toward a diagonal direction, these last ones share two first neighbors whose parameters are no more needed for the elaboration of the new dipole. Fig. 5.1 shows an example of this, since passing from dipole 1 to 2, dipoles 3 and 4 are preserved as first neighbors. This reduces to 15 the number of input data needed, and a correspondent saving in transfer time per each dipole is obtained. Another advantage yielded by the diagonal scanning consists in avoiding calculations. Considering

again fig. 5.1 we note that dipoles 3 and 4 give the following contributes to each component of the local field in dipole 1:

$$MX * X + MY * Y = 2 * mom_x(4) - mom_x(3) \tag{17}$$
$$MX * Y + MY * Y = -mom_y(4) + 2 * mom_y(3) \tag{18}$$
$$MX * Z + MY * Z = -mom_z(4) - mom_z(3) \tag{19}$$

If we now consider the contribute of the same dipoles to dipole 2, the next reached by the diagonal scanning, we find:

$$MXX + MYX = 2 * mom_x(3) - mom_x(4) \tag{20}$$
$$MXY + MYX = -mom_y(3) + 2 * mom_y(4) \tag{21}$$
$$MXZ + MYZ = -mom_z(3) - mom_z(4) \tag{22}$$

The values on the right are obtained by substituting at the terms on left, those values reported in equations in section 3.

Equations 19 and 22 are equal and can be calculated only once. The same considerations are applicable in case of movements toward YZ or XZ direction with a consistent sparing of operations.

Finally, the moment components involved in equations 17–19 for the dipole 1 are also present (with different coefficients) in equations 20–22 and, again, they can be calculated only once (i. e. for dipole 1, storing them in registers from which they can be retrieved later for the next dipole) with a further saving of time.

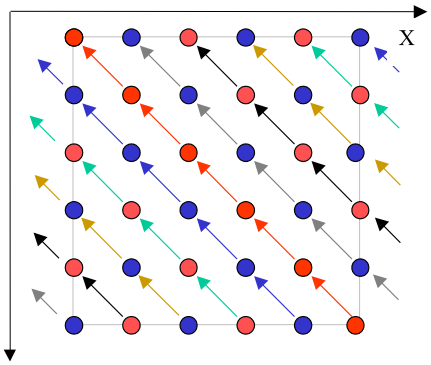


Fig. 5.2. *Diagonals for scanning in XY face.*

The diagonal scanning basically consists of XY movements as shown in fig. 5.2.

The cubic lattice is considered as made by 'slices' and when the last dipole is reached on an XY face, a little movement toward the YZ or XZ direction allows to skip to the next XY slice. In each slice, different starting points can be chosen depending on the odd/even number of dipoles present on the edge of the lattice, but for sake of simplicity we don't want to excessively detail these simulation aspects.

6. Implementation on DPFPA. As previously said, a sequence consists of a microinstruction set and could be identified as a Setup or a Loop sequence. The first problem to deal with is the definition of those operations more frequently executed which should be inserted into the Loop sequence. In the diagonal scanning, the most frequent operation regards the interaction between dipoles located on diagonals belonging to the XY side: thus, the Loop sequence should implement the energy calculation of these dipoles, while the Setup should execute the movements in the XZ or YZ faces of the lattice, through which the algorithm considers the first dipole of the next XY 'slice' and another Loop sequence begins.

According to what said in the previous section, the number of the needed sums is 14 for evaluating CTX, CTY e CTZ (one add is shared with the previous dipole), 3 for CT and 1 more to add these values to the partial total energy obtained from the previous dipoles considered. Thus the adder pipeline is used as its best, if 18 clock cycles are taken. For what concerns multiplications, instead, 6 are needed to calculate CT, 3 for the new moment components of the considered dipole and 1 more for its global energy. Thus, 10 multiplications are required. Let's see how these operations could be efficiently implemented.

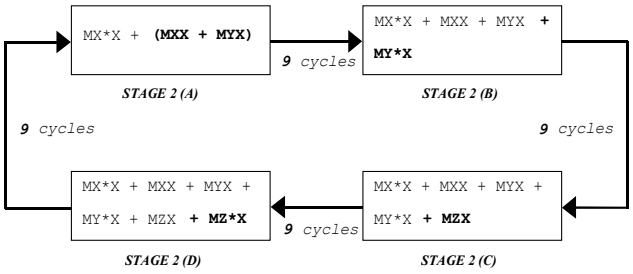


FIG. 6.1. *Stage 2 in the adder pipeline during the Loop phase*

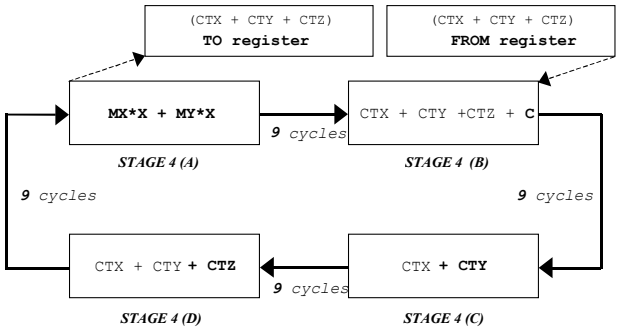


FIG. 6.2. *Stage 4 in the adder pipeline during the Loop phase*

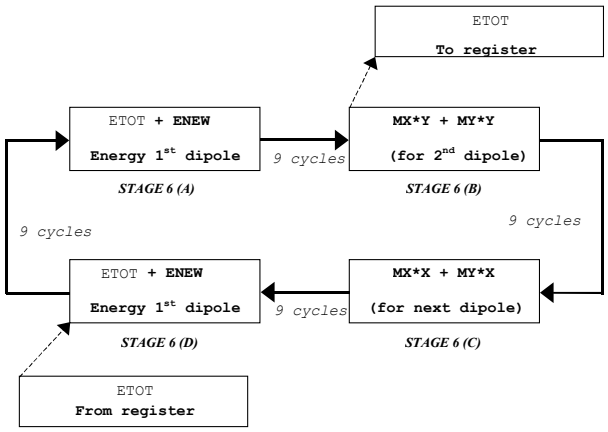


FIG. 6.3. *Stage 6 in the adder pipeline during the Loop phase*

6.1. Adder Unit. Each stage is considered as an independent register containing the partial result which can be stored every L clock cycles (L is the pipeline length). The Loop sequence evaluates the energy of dipoles considered in the XY direction: 4 stages of adder pipeline were devoted to calculate CTX, CTY, CTZ and CT. In fig. 6.1, the second pipeline stage devoted to the calculation of CTX is shown, with the particular calculation highlighted in bold in each of the four sums needed. In the first step, the term in parentheses is 'shared' with the previous dipole considered and does not need to be re-calculated (see previous section). Each partial result is available only when it has run across the whole pipeline i. e. after 9 clock cycles and the complete value of CTX is available after 36 clock cycles. Then the stage proceeds to evaluate the CTX for the next dipole. The same considerations can be made for CTY and CTZ. The calculation of CT is implemented in the stage 4, which works again for 36 clock cycles. The CTX, CTY, CTZ values used in this case are those coming from the multiplier where they have been multiplied by SC, SS and C. Since the calculation of CT takes less than 36 cycles, the first stage is used to calculate that value shared with the next dipole:

Stage	*Term*	Cycles 1 - 9	Cycles 10 - 18	Cycles 19 - 27	Cycles 28 - 36
9	CT2	**CTX2+CTY2 (n-3)**	CTX2+CTY2+ **CTZ2 (n-3)**	**mx*y + my*y**	CTX2+CTY2+CTZ2 + **C2 (n-3)**
8	CTZ2	mx*z+my*z + **(mxz + myz) (n-1)**	Mxz+myz+mx*z+ my*z + **mzz (n-1)**	mxz+myz+mx*z+ my*z+mzz+ **mz*z (n-1)**	**mx*z + my*z (n+1)**
7	CTY2	mxy+myy+mx*y+ my*y + **mzy (n-1)**	mxy+myy+mx*+ my*y+mzy+ **mz*y (n-1)**	**(mxy+myy)+ mx*y (n-1)**	mx*y+myy+mxy + **my*y (n+1)**
6	ETOT	Etot + **Enew1 (n-6)**	**mx*y + my*y**	**mx*x + my*x**	Etot+ **EneW2 (n- 5)**
5	CTX2	mxx+myx+mx*x+ my*x + **mzx (n-1)**	mxx+myx+mx*x+ my*x+mzx + **mz*x (n- 1)**	**(mx*x+mxx)+ myx (n+1)**	mx*x+myx+mxx+ **my*x (n+1)**
4	CT1	**mx*x + my*x**	CTX1+CTY1+CT Z1 + **C1 (n- 4)**	**CTX1 + CTY1 (n-2)**	CTX+CTY+**CTZ (n-2)**
3	CTZ1	mx*z+my*z+mxz+ myz+mz*z + **mzz (n-2)**	**mx*z + my*z (n)**	mx*z+my*z+ **(mxz + myz) (n)**	mx*z+my*z+mxz+ myz + **mz*z (n)**
2	CTX1	**mx*x + (mxx+myx) (n)**	mx*x+mxx+myx + **my*x (n)**	mx*x+my*x+mxx+ myx + **mzx (n)**	mx*x+my*x+mxx+ myx+mzx+ **mz*x (n)**
1	CTY1	**(mxy+myy) + mzy (n)**	mxy+myy+mzy + **mx*y (n)**	mxy+myy+mx*y+ my*y + **mzy (n)**	mxy+myy+mx*y+ my*y+mzy+ **mz*y (n)**

FIG. 6.4. *Stage 6 in the adder pipeline during the Loop phase.*

Therefore the partial value of CT is saved in a register from which it will be retrieved during stage 4B (fig. 6.2). To optimise the use of the pipeline the remaining stages are devoted to implement the same calculations for a second dipole, so as to process 2 dipoles in 36 clock cycles. This corresponds, as previously seen, to an optimal use of the adder. Finally, stage 6 is devoted to add to the global energy value E_{TOT}, those two energy contributes (E_{NEW}) calculated in the other stages of the pipeline up to this moment (fig. 6.3). Basically it works in the same way as stage 4, including two sums shared with the successive elaborated dipoles (again to optimise the pipeline use). Even though, during the 36 clock cycles all the sums needed for the energy of two dipoles have been performed, the dipoles involved in the elaboration are more than 2. In fact, while the adder is evaluating CTX, CTY and CTZ for the two dipoles, it is not possible to determine at the same time the correspondent CT terms, since the previous calculations (CTX, CTY and CYZ) should be completed and they should also be multiplied by $SC1 * k$, $SS1 * k$ and $C1 * k$ (k is a suitable constant depending on the system density). Therefore the CT term really computed refers to the previous Loop sequence. This means that while CTX, CTY and CTZ for dipoles (n) and $(n+1)$ are evaluated, the CT terms refer to $(n-3)$ and $(n-4)$ dipoles and the E_{NEW} corresponds to the couple $(n-5)$ and $(n-6)$ previously started. Moreover, also the couple $(n-1, n-2)$ is subjected to a partial elaboration making the pipeline always working.

This configuration brings a consistent level of parallelisation in the execution of the algorithm. Fig. 6.4 shows the complete set of operations calculated during the 36 clock cycles of each Loop sequence. Per each stage and clock cycle, the effective sum performed is reported in bold.

6.2. Multiplier Unit. This unit executes the multiplications needed in the terms that must be added, i. e. 10 per each of the two dipoles of the adder unit (globally 20) and in a sequential way. To synchronise the operations in the multiplier pipeline with those of the adder, the length of the pipeline (15 stages) is extended to 18 by adding three NOP (no operation) cycles: this means that in 36 clock cycles the multiplier works effectively for 30 cycles, a time sufficient to execute the required 20 products, without loosing the synchronisation with the correspondent terms in the adder unit. Fig. 6.5 describes the operations performed together with the output from the pipeline at that instant, per each clock cycle. In parenthesis the order number is reported of the dipole to which the calculation refers: n is the dipole for which the calculation of the energy is initiated in the current sequence. At the end of each Loop sequence the pipeline outputs new moments and energy of the dipole couple which started the evaluation 3 sequences before. Fictitious products have been inserted when needed to force the pipeline going one step beyond.

7. Results. The whole system has been tested by executing Montecarlo simulations of different size lattices ($4 < ND < 100$, where ND is the number of dipoles on each side of the cubic lattice).

Output	Cc	OPERATION		Output	Cc	OPERATION
SC1*K (n-2)	1	Fictitious product		---	19	SC1*K (n)
C1*K (n-2)	2	Fictitious product		---	20	C1*K (n)
SS1*K (n-2)	3	CTX1*(SC1*K) (n-2)		CTX1*SC1*K (n-2)	21	SS1*K (n)
-1/2*CT1^2 (n-5)	4	CTY1* (SS1*K) (n-2)		CTY1*SS1*K (n-2)	22	-1/2*CT1^2 (n-3)
CTX2*SC2*K (n-3)	5	SC2*K (n-1)		SC2*K (n-1)	23	CTX2*(SC2*K) (n-1)
C1*1 (n-2)	6	NOP		C1*1 (n-2)	24	NOP
C1*1 (n-2)	7	C1*1 (n-2)		C1*1 (n-2)	25	C1*1 (n)
C1*1 (n-4)	8	Fictitious product		---	26	C1*1 (n-2)
CTY2 * SS2 * K (n-3)	9	SS2*K (n-1)		SS2*K (n-1)	27	CTY2*(SS2*K) (n-1)
CT1*C1 (n-6)	10	CT2*SC2 (n-5)		CT2*SC2 (n-5)	28	CT1*C1 (n-4)
CT1*SC1 (n-6)	11	CT2*SS2 (n-5)		CT2*SS2 (n-5)	29	CT1*SC1 (n-4)
CT1*SS1 (n-6)	12	NOP		CTZ1*C1*K (n-2)	30	NOP
CT1*SS1 (n-6)	13	CTZ1*(C1*K) (n-2)		CTZ1*C1*K (n-2)	31	CT1*SS1 (n-4)
C2*1 (n-5)	14	CT2*C2 (n-5)		CT2*C2 (n-5)	32	C2*1 (n-3)
---	15	-1/2*CT2^2 (n-5)		-1/2*CT2^2 (n-5)	33	Fictitious product
C2*1 (n-3)	16	C2*1 (n-1)		C2*1 (n-1)	34	C2*1 (n-1)
CTZ2*C2*K (n-3)	17	C2*K (n-1)		C2*K (n-1)	35	CTZ2*(C2*K) (n-1)
---	18	NOP		SC1*K (n)	36	NOP

FIG. 6.5. *Operations performed in the multiplication pipeline during 36 clock cycles.*

Performance has been evaluated as speed-up respect to the execution of the same simulation on an Intel P4 processor with 1GB Ram memory; also FPGA occupation was used as a performance parameter. Simulation code was written in C language and optimized using Microsoft Visual C++ environment. The Accelerator elaboration times were measured by means of the clock counters implemented in the interface between Nios and the coprocessor previously described.

In fig. 7.1 we show the performance as speed-up factor respect to two Intel P4 processors with 3 GHz and 1.7 GHz frequency respectively, calculating the dipolar energy of the simulated system. That computational core is repeatedly executed $k*N*10000$ times where k is the coefficient responsible for the interaction settlement (equilibration) and N is the dipole number: this gives reason of the high computational load which can lead (for big particle systems, e. g. 100000 dipoles) to wait a lot for results, if the simulation should be performed on a PC. The speed-up factor is increasing for the 1.7 GHz processor due to cache effect, while for the most performing Intel processor (3 GHz) sets around 2.

Considering the size of the FPGA we used, other 2 accelerating units could be implemented, we can reasonably state that a speed-up factor equal to 4 can be achieved in case of a "full" implementation on the FPGA component we chose (Stratix EP1S40). Further speed-up could be obtained if other components of the Altera's family (Stratix2 or Stratix3 now available) should be employed.

The cost of each board we bought was nearly $1200: this represents an important indication when predicting trade-off between a cluster of workstations versus a cluster of FPGA based accelerators. In practice, our work indicates that each FPGA unit gives a computational power 4 times greater, only doubling costs with respect

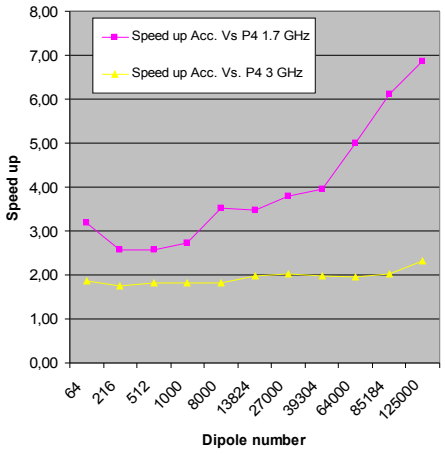


FIG. 7.1. *Speed-up of the FPGA based accelerator with respect the P4 Intel processors.*

to a computational unit in a PC cluster, providing the scientist with a COTS desktop computing system on which he/she can run simulations.

8. Conclusions. Simulations allow the analysis of a physical system, even complex, without experimental measures or, sometimes, to confirm what was experimentally observed. In certain situations such as microscopic systems, simulations represent the simplest if not the only way to quickly foresee the behaviour of a particle system in different environmental conditions. The high number of variables involved together with complex interaction laws often make simulation times unacceptably long. Finally, several of the requested calculations ask for double precision floating point arithmetic, further increasing the computational power needed.

In this paper, we have shown how an application specific architecture (DPFPA) specifically designed for this kind of problems and based on FPGA technology could represent a good compromise between processing capabilities and low costs. DPFPA can be programmed with a dedicated language to execute complex floating point functions and it is equipped with a suitable software development environment. We executed the dipole energy calculation through the simulator, achieving, thanks also to the new scanning algorithm purposely designed and here described, a performance twice as that of a last generation Personal Computer but can be easily "extended" to 4.

A further improvement could be achieved by a full custom ASIC implementation of the Accelerator which is not justified at a prototyping level while it allows a large scale manufacturing with reduced costs. This would make available several computing units connected in cluster fashion by means of a point to point network, providing the user with a great computing power.

REFERENCES

[BELLETTI F., et al.,] "An adaptive FPGA computer", IEEE Computing in Science & Engineering, vol. 8(1), January-February 2006, pp. 41-49.

[BOGHOSIAN B., et al.] "Scientific applications of grid computing", IEEE Comp. in Science & Engin., vol. 7(5), Sept.-Oct. 2005, pp. 10-13.

[BUELL D., et al] "High Performance Reconfigurable Computing", IEEE Computer. March 2007, pp. 23–26.

[COWEN C. P. et al.] "A reconfigurable Montecarlo clustering processor (MCCP)", FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on 10–13 April 1994, pp. 59 – 65.

[CRUZ A., et al.] "A Special Purpose Computer for spin glass models", Computer Physics Communications, vol. 133, n° 2-3, 2001, pp. 165–176

[DANESE G., et al.] "A development and simulation environment for a floating point operations FPGA based accelerator", Proc. of DSD '03 – 3rd Euromicro Symposium on Digital System Design, Belek (Turkey), September 2003, pp. 173-179.

[DANESE G., et al.] "An application specific processor for Montecarlo simulations", IEEE conference on Parallel and Distributed Processing (PDP07), Naples, February 2007, pp. 262-269.

[DANESE G., et al.] "Field induced anti-nematic ordering in assemblies of anisotropically polarizable spins", Europhysics Letters 55(3), pp. 362-368, 2001.

[DONGARRA J., et al.] "High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions", IEEE Comp. in Science & Engin., vol. 7(2), Mar–Apr 05, pp. 51–59.

[FRENKEL D., et al.] "Understanding computer simulations", Acad. Press New York, pp. 28–30, 1996.

[GOKHALE M., et al.] "Monte Carlo radiative Heat Transfer Simulation on a reconfigurable Computer", Proc. of FPL 2004, LNCS 3203, pp. 95–104, Springer-Verlag ed.
[HANKEL J., et al.] "Taking on the embedded system design challenge", IEEE Computer, April 2003, 35–37.
[HERBORDT M. C.] "Achieving High Performance with FPGA-Based Computing", IEEE Computer. March 2007, pp. 50-57.
[MARSH P.] "High performance horizons", Computing & Control Engineering Journal, vol. 15(6), December–January 2004/2005, pp. 42–48.
[METROPOLIS N., et al.] "Equation of State Calculations by Fast Computing Machines", Journal of Chem. Physics, 21, (1953), pp. 1087–1092.
[MONAGHAN S., et al.] "Reconfigurable special purpose hardware for scientific computation and simulation", Computing & Control Engineering Journal, Vol. 3, Issue 5, Sept. 1992, pp. 225–234.
[O' KONSKI C. T., et al.] "New method for studying electrical orientation and relaxation effects in aqueous colloids: preliminary results with tobacco mosaic virus", Science, 111, pp. 113–116 (1950).
[POSTULA A., et al.] "The design of a specialized processor for the simulation of sintering", EUROMICRO 96. 'Beyond 2000: Hardware and Software Design Strategies', Proc. of the 22nd EUROMICRO Conference, 2–5 September 1996, pp. 501 – 508.
[RADEVA T., et al.] "Electric Light Scattering from Polytetrafluorethylene Suspensions", Coll. And Surf. 119, 1 (1996).
[WOLF W.] "A decade of hardware/software codesign", IEEE Computer, April 2003, 38–42
[WOLF W.] "The embedded systems landscape", IEEE Computer, October 2007, 29–33.
[ZHANG G. L., et al.] "Reconfigurable Acceleration for Montecarlo based financial simulation", Proc. of FPT05, IEEE Conference on field programmable technology, Singapore Dec. 11–14 2005.

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 289–303. http://www.scpe.org
ISSN 1895-1767

CREATING, EDITING, AND SHARING COMPLEX UBIQUITOUS COMPUTING ENVIRONMENT CONFIGURATIONS WITH COLLABORATIONBUS

TOM GROSS* AND NICOLAI MARQUARDT†

Abstract. Early sensor-based infrastructures were often developed by experts with a thorough knowledge of base technology for sensing information, for processing the captured data, and for adapting the system's behaviour accordingly. In this paper we argue that also end-users should be able to configure Ubiquitous Computing environments. We introduce the CollaborationBus application: a graphical editor that provides abstractions from base technology and thereby allows multifarious users to configure Ubiquitous Computing environments. By composing pipelines users can easily specify the information flow from selected sensors via optional filters for processing the sensor data to actuators changing the system behaviour according to their wishes. Users can compose pipelines for both home and work environments. An integrated sharing mechanism allows them to share their own compositions, and to reuse and build upon others' compositions. Real-time visualisations help them understand how the information flows through their pipelines. In this paper we present the concept, implementation, and user interface of the CollaborationBus application.

Key words: ubiquitous computing; editor; configuration

1. Introduction. The development of early sensor-based infrastructures often required expert programmers with a thorough knowledge of base technology for sensing information, for processing the captured data, and for adapting the system's behaviour accordingly [10] [23] [24] [26]. In this paper we argue that also end-users should be able to configure Ubiquitous Computing environments. There are some research projects providing easy-to-use configuration interfaces for non-expert users to create sensor-based Ubiquitous Computing applications, yet mostly only for the private home [4] [8] [15] [18] [25]. Furthermore, most systems lack integrated facilities for the collaborative exchange of users' configurations. Only some systems—typically complex configuration tools [2] [3] [5] [16]—provide enhanced visualisations of the data flow and sensor-network data to support users while creating or configuring applications.

In this paper we introduce CollaborationBus: a graphical editor that provides adequate abstractions from base technology and thereby allows multifarious users—ranging from novice to experts—to easily configure complex Ubiquitous Computing environments.

By composing pipelines users can easily specify the information flows from selected sensors via optional filters for processing the sensor data to actuators changing the system behaviour according to their wishes. Whenever the sensors capture values that are in the range indicated by the users, the actuators perform the specified actions. All pipeline compositions are stored in the respective user's personal repository. A central interface allows users to control their respective repository—they can create new pipeline compositions, or edit, activate or deactivate existing ones.

An integrated sharing mechanism allows users to share their own pipeline compositions with others users. In an analogous manner they can add others' compositions to their own repository, and build new compositions based on these compositions. Real-time visualisations display relations between incoming and outgoing events of the pipeline, and let the user interactively adjust and keep track of the information flow through their pipelines. They help the users understand the information flow through their compositions, which can become quite complex consisting of sets of sensors, filters, and actuators.

In this paper we present the concept, implementation, and user interface of the CollaborationBus application. First, we develop scenarios of configurations for Ubiquitous Computing environments and derive requirements. Then we describe the concept and implementation of CollaborationBus, and present its user interface. We continue with a discussion of related work. Finally, we draw conclusions and report on future work.

2. Requirements. In this section we develop scenarios of configurations for Ubiquitous Computing environments and derive requirements for the CollaborationBus editor.

2.1. Application Scenarios. Users should be able to configure environments in their private homes as well as in their workplaces.

*Faculty of Media, Bauhaus-University Weimar, Germany (e-mail: tom.gross(at)medien.uni-weimar.de).
†Department of Computer Science, University of Calgary. Calgary, AB, CANADA, T2N 1N4. (e-mail: nicolai.marquardt(at)ucalgary.ca).

Smart Telephone. In a first scenario users wish to control the sound volume of their music players and start their calendar application in dependence of their office telephones' state. A simple binary sensor attached to the telephone is the first input source of this pipeline. The second input source checks whether the user is currently logged in at the office computer. The condition modules check the telephone sensor state as well as the login information. Finally, the user specifies the desired information flow: if the attached sensor detects that the phone is used, a script is started (e.g., AppleScript on a Mac OS X computer, or a shell script on Windows) and mutes the volume of the computer (e.g., Mac, or PC), an infrared control (e.g., on a sensor board) mutes the sound system, and another script starts the user's calendar application (e.g., iCal, or Outlook), so that the user can input new appointments during the phone call. When the phone call ends, the application fades the music back in again after a few seconds.

Personal Notification Selection. In a second scenario, users want to get information about the current activities of their remote co-workers and friends. Users can add a state sensor to the instant messaging application as well as movement and noise sensors as sources of their pipeline. Then users can specify queries with keyword filters that analyse the sensor data of the instant messaging sensor and check if they match the names of their remote co-workers or project descriptions. As actuators the users might wish to specify that all events are collected and sent as a daily email summary once a day. Additionally, if the number of messages containing the keywords reaches a specified occurrence threshold, the system additionally sends the users an immediate summary message to their mobile phones via an SMS gateway (a short message service sending a message to the mobile phone).

Informal Group Awareness. In a final scenario, the users of two remote labs want an information channel of the lab activities as RSS feed that can be integrated into tickertape displays or screensavers. They wish to receive information on the activities at the other site. They create a pipeline composition and add the following information sources as input sources: the current lab members logged in on the server and in the instant messaging system, the current CVS submissions of the developers, the average values of the movement and noise sensors and the current temperature of the two labs and the coffee lounge. As actuator component for the output they add an RSS feed generator and publish the RSS file to a server. Now, the lab members can access this RSS feed and add it to their favourite notification display (e.g., a Web browser, or a screensaver). This summary of group events and activities can help users to find out more about the whole development team, and can facilitate the informal and spontaneous communication between the colleagues.

2.2. Functional Requirements. The following functional requirements were derived from various application scenarios (we described three of them in the preceding sub-section), and from a detailed study of related work (we present some examples of related work in Section 6 below).

- *Provide adequate abstraction for various applications domains:* Configuration editors should allow users to integrate a variety of software and hardware sensors capturing information, and software and hardware actuators adapting the behaviour of the environment accordingly. The integration of existing and new sensors and actuators should be easy. Various configurations should be possible—ranging from configurations for home environments as well as for work environments.
- *Support diverse users with heterogeneous knowledge, ranging from novice to experts:* Configuration editors should facilitate the immediate utilisation. For this purpose, they should provide a pre-defined library of common configurations and configuration assistants that allow the users—especially novice users—to use the editor immediately and to incrementally explore its functionality. Additionally, configuration editors should offer guided compositions. Therefore, the user interface and the functionality provided should be restricted to significant and needed functions; functions that are not adequate or not needed should be disabled (e.g., if a sensor captures data in the form of text strings, calculations such as average should be disabled). Finally, configuration editors should provide details on demand. For this purpose, especially more experienced users should be able move from more abstract to more fine-grained layers, and to see and manipulate details.
- *Support the exchange of configurations among users:* Configuration editors should allow the sharing of configurations among users. The sharing of configurations is useful for workgroups and friends, because it allows users to build on the results of other users, and gives less experienced users the chance to benefit of the knowledge of more experienced users. Subsequently we present the concept and implementation of CollaborationBus addressing these functional requirements.

TABLE 3.1
Applications with sensors/actuators in home and work environments.

	Home actuators	**Office actuators**
Home sensors	*a) Smart Home Applications:* often connections between hardware sensors and actuators (e.g., to control electrical devices (power plugs) in dependence of observed sensor values, or to control multimedia home devices)	*b) Home Awareness Applications:* mixed use of software and hardware sensors and actuators (e.g., to observe the private home from the work office, or to display state of family members at home)
Office sensors	*c) Office Awareness Applications:* mixed use of software and hardware sensors (e.g., to summarise information of projects and to inform people at home about the working activity)	*d) Collaborative Work Applications:* often applications based on software sensors and actuators (e.g., to observe computer logins, instant messenger presence, and other activities)

3. Concept. In this section we describe CollaborationBus' key concepts for a generic approach, for pipelines, for a diverse user experience, and for collaborative sharing.

3.1. Generic Approach. The approach of CollaborationBus is generic—it works across multiple applications domains, temporal patterns, and complexity patterns.

3.1.1. Spanning Application Domains. Sensor- and actuator-based applications in the private home differ from those in the cooperative work domain. While we try to integrate a common, universal user interface and metaphors for users of both domains, these domains can vary in their use of hardware and software sensors as illustrated in Table 3.1.

Smart Home Applications (cf. *a* in Table 3.1) are mainly built with hardware sensors and actuators, where the developed sensor-based applications adapt the home environment automatically to the requirements of the private users. While computer applications provide appropriate functionality for the configuration and creation of these applications, the computer and its applications should disappear during the everyday execution of the sensor-based applications. In order to support the development of appropriate applications, the CollaborationBus editor supports a variety of hardware sensors and actuators, and the editor is only needed for composing the setting.

In contrast to these mainly hardware-based applications, most *Collaborative Work Applications* are based on both hardware and software sensors and actuators (cf. *d* in Table 3.1). Since computers are in general part of the workplace, software sensors and their events (e.g., appointments, emails, tasks, project activity) and software actuators (e.g., for sending emails, displaying messages on the computer screen) can be used to create sensor-based applications for awareness and information-flow of workgroups. At the same time, the integration of hardware—both sensors and actuators—and their physical user interfaces can facilitate the interaction with these applications. This results in tangible user interfaces for applications at the workplace (e.g., physical sliders so set the presence in an instant messaging systems; LCD displays for displaying important email messages; audio signals to inform about the current project's state). CollaborationBus supports the creation and configuration of all these free combinations of physical user interfaces with software events as a main feature and allows users to create their envisioned interfaces themselves.

In between these two domains are applications that bridge the gap between the private home and the business work (cf. *b* and *c* in Table 3.1). *Home Awareness Applications* (cf. *b* in Table 3.1) support connections to family members and friends at the workplace. For instance, ambient displays let the users perceive the information in multi-sensory ways. This includes that users can configure their sensor-based applications at home as well as at their office; thus a universal application interface is required.

Office Awareness Applications bridge the gap between the home and the work environment (cf. *c* in Table 3.1) by informing users about events from the office while they are at home. Users define their own information channels that connect home environment with their work environment (e.g., project report summaries that are generated and delivered to the private home, important email or instant messages that are forwarded to the private home). Here, the configuration editor requires in most cases a variety of software sensors in the work environment that are connected to physical actuators in the private home.

On a whole both environments—home and work—have become increasingly intertwined in the recent years (e.g., telework). Therefore, utilities need to allow the building of universal sensor-based applications spanning both ambiences and the integration of software sensors and actuators as well as hardware sensors are needed.

3.1.2. Spanning Temporal Patterns. In any application domain various patterns with regard to capturing ongoing data and starting actuators can be identified:

- Recurrent, permanent (e.g., applications with ongoing collection of data)
- Recurrent, occasionally (e.g., applications depending on day-time, during the holidays, at night)
- One-time (e.g., applications with call-back if the required person is reachable)

The software needs adequate methods to support any of these temporal patterns, and should provide a structured overview of the current configurations of a user. Another important aspect is to enable the easy re-use of created configurations in the past: a copy method and templates can speed up the creation process. Systems supporting all these temporal patterns are needed.

3.1.3. Spanning Complexity Patterns. Each setting can have a specific complexity pattern ranging from simple sensor-actuator tuples to networks of sensors and actuators:

- One sensor, one actuator (e.g., one binary sensor controls one actuator)
- Sensor, filters, actuator (e.g., only react to certain temperature values of a temperature sensor)
- Multiple parallel sensors, filters and actuators (e.g., create summaries of various sensor sources, control a set of actuators)
- Complex network of components (e.g., determine the current activity or even mood of a person)

The CollaborationBus editor supports any application domain, and any temporal pattern described above. It supports any complexity pattern, except for complex networks. Complex networks are typically not configured with a graphical editor, but rather developed with programming languages; therefore, here a graphical editor would not be used anyways.

3.2. Pipelines. In CollaborationBus all relations between sensors and actuators are handled with a pipeline metaphor.

Pipelines are compositions that include several components: at least one sensor and one actuator component, and additionally further filter components for processing sensor values (e.g., to delimit the forwarded values, or to convert data formats). All components inside of a composition are connected via pipelines that forward events between them. Pipelines can be nested in various ways: several parallel sub-pipelines can be added (this represents the OR condition); sequences of sensor sources can be created (AND condition); or negations can be specified (NOT condition).

Sensors are the sources of any initial event in a pipeline. They can either be hardware sensors (e.g., sensors for temperature, movement, light intensity) or software sensors (e.g., sensors for unread emails, mouse activity, shared workspace events, open applications).

Actuators are at the sink-side of the pipeline composition. Hardware actuators affect the real environment of the users (e.g., activate light sources or devices), while software actuators only influence the computer system (e.g., display screen messages, start applications).

Filters for processing the captured data are between sensors at the one side and actuators at the other. The filter components can process all incoming events of a sensor source. Each filter component represents a single condition or transformation based on the incoming event value. Filters typically generate data of particular formats (e.g., integer values, Boolean values, strings). There are universal filter types that can be applied to any type of sensor data and specific filter types that can only be applied to particular types of sensor data. The respective filter types can do the following processing:

- Universal (e.g., count the event occurrence, create event summary reports)
- Numerical processing (e.g., numeric threshold, interpolation, average)
- String processing (e.g., search for specified keywords)
- Binary processing (e.g., negation, conjunction)
- Transformations (e.g., numeric value to string message, binary value to numeric)

Filters can be assembled in many different combinations. This includes, for example, an adaptive behaviour to changed conditions of the sensor sources (e.g., modified upper or lower limit of a temperature sensor, or a changed scale of values) by transmitting these changed conditions to all pipeline components. Each component can decide if a modification of its settings is necessary, and eventually display a confirmation dialog. The

components also include a variety of transformation methods (e.g., for generating a short message to the mobile phone (SMS) a string message can be entered, and the values of the respective sensors can be attached).

With CollaborationBus users can easily connect local sensors and actuators or sensors and actuators from remote locations and build new configurations in a few seconds by visual programming though point-and-click. Each pipeline composition includes all these components—sensors, actuators, and filters—and defines a complete information flow through them. Experts can program new pipeline components by deriving new classes from the *PipelineComponent* class (cf. next section for details). All compositions of a user are stored in a personal repository. This repository includes all data to dynamically instantiate the included pipeline compositions.

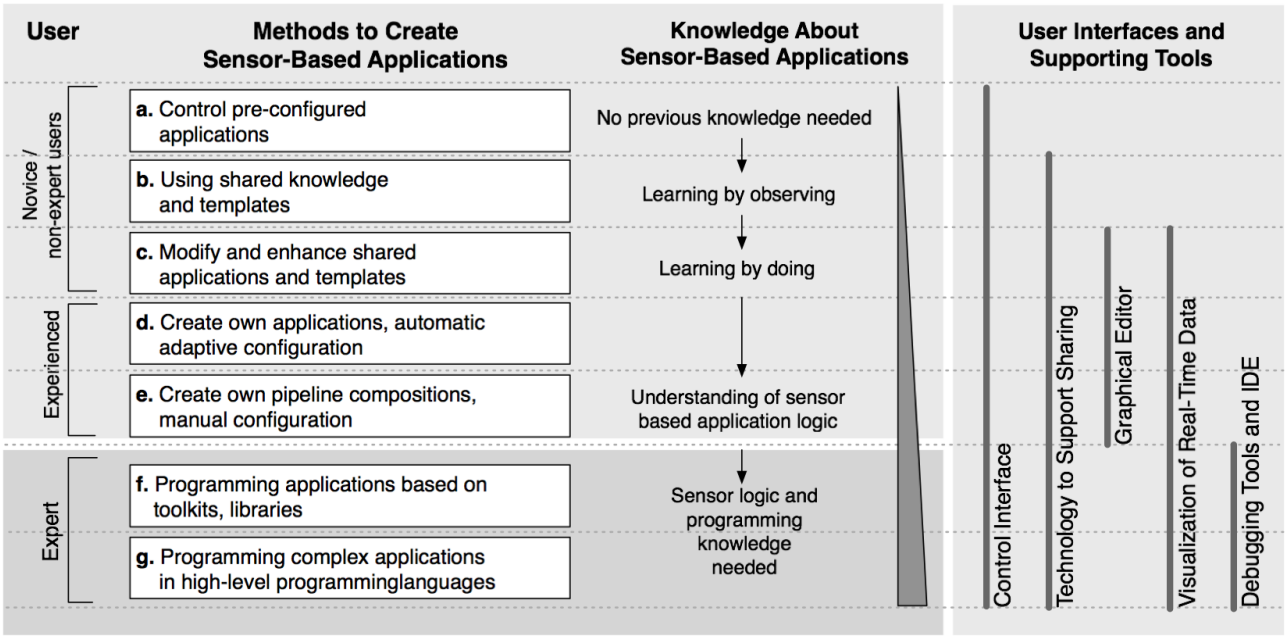


Fig. 3.1. *User experience levels, and adequate tools to support users.*

3.3. Experience for Diverse Users. The CollaborationBus editor can be used by users with diverse levels of technical background. Users' knowledge can range from no experience at all to very thorough technical knowledge. Figure 3.1 shows various user types ranging from novice users with no experience to more experienced users and to experts. It shows the methods that are available and can be used in dependence of the existing knowledge. It also shows the user interfaces and support tools that are offered for the respective user types (the user interfaces and supporting tools are described below).

Novice users with no prior technical knowledge can start using CollaborationBus by loading and adapting pre-configured application configurations that are part of the CollaborationBus distribution. As they progress, they can use the integrated sharing tool to load other users' configurations and to use them as templates for their own configurations. They can, furthermore, modify and enhance the application configurations and templates.

More experienced users can create their own application configurations, and execute them in order to learn more about intra-pipeline event forwarding.

Expert users can create the envisioned system-behaviour by developing the required software in a high-level programming language. Typically, for these activities they use toolkits, platforms, libraries, and development and debugging environments to facilitate and speed up the development process.

Taking these diverse user types into consideration is a core concept of CollaborationBus and its user interface (the latter is described below).

3.4. Collaborative Sharing. Users can build their own personal pipeline compositions from scratch, or build on shared compositions from colleagues and friends. Three types of sharing are possible:

- *Sensor and event sharing:* users either share the events of their own sensors, or the processed events of their sensors.
- *Actuator sharing:* users share the control of a personal actuator with other users, so that other users can send commands to the actuator and control the system behaviour.
- *Pipeline sharing:* users share complete more or less complex pipeline compositions with others.

The first sharing method lets users create their own configuration in dependence of remote located sensors of other users. The second sharing method lets users control the actuators of other users (leading to new challenges of potentially concurrent access to actuators). And the third sharing method lets users exchange and re-instantiate complete pipeline compositions, requiring a unified description format and exchange protocol for pipeline compositions. In the latter case the recipients of the compositions can change this released pipeline composition to fit to their requirements. Because each user creates a new instance of this pipeline composition, the changes of other users are not affecting the original composition.

CollaborationBus supports security and privacy protection thought adequate levels of abstraction and control over access privileges of the own information sources are needed. In order to restrict the shared information, users can choose the sharing of abstract templates. In these shared pipeline compositions, only the skeleton of a pipeline is shared, and the original sensors and actuators of a user are not included in the sharing entry. Thus, the abstract template of a composition contains mainly the configuration of all filter components between the sensors and actuators. Using this abstract template, other users can insert their own sensors in the placeholders at the beginning of the pipeline composition, and their own actuators at the end. This let them use the knowledge of the processing filter components of the composition, while at the same the user who shares his pipeline composition does not share his own sensors and actuators.

These integrated collaborative sharing methods provide a powerful and easy-to-use method of knowledge exchange between different users of the system. As prior described in Figure 3.1, a novice and inexperienced user can use pre-configured pipeline compositions of another user (if this user shares the complete pipeline), or the user can load an abstract pipeline template and fill in his own sensors and actuators. At every time it is very easy for the users to share their new pipeline compositions again, and store them in the shared repository.

The following example illustrates a situation where these abstract templates are appropriate. A user has created an ambient notification display of important incoming email messages: all messages are scanned for adequate keywords or sender addresses, and if the scan was successful, a message will be displayed on an ambient external LC display. The user decides to share this configuration, while at the same time it stands to reason that the user do not want to share his personal email-sensor, or the exact configuration of the keyword filter. By using the abstract template, the user can share the basic concatenation of incoming sensors, filters, and the actuator display, without sharing his personal sensors.

On the other hand, a user who has created a SMS notification service for the average temperature of a series of temperature sensors may wish to share this complete configuration, and therefore shares the pipeline compositions with all the associated sensors.

4. Implementation. In this section we describe the implementation of CollaborationBus: software architecture and class diagram.

4.1. CollaborationBus Software Architecture. Figure 4.1 provides an overview of the software architecture of CollaborationBus. All sensor and actuator components are connected to the *SensBase* infrastructure, which provides adapters for the connection of sensors and actuators, a central registry of all connected components and a database for persistent storage of sensor event data. *SensBase* was implemented with the *Sens-ation* platform [13]. *SensBase* provides inference engines that can transform, interpret, and aggregate sensor values. A variety of gateways (e.g., Web Service, XML-RPC, Sockets) provide interfaces for the retrieval of sensor descriptions, event data, actuators, and so forth.

The *CBServer* uses these gateways to register for the sensor values needed for the users' pipeline compositions. Each time when changes occur at one of the connected sensors, the *SensBase* server forwards a change event to the *CBServer*. These events are forwarded to the adequate components inside of each pipeline composition. The compositions are inside of the Personal Repository of each user and include the complete description of all assembled components (in serialised XML format, for platform independency and easy exchange of pipeline composition descriptions). The *CBServer* can serialise and de-serialise these XML descriptions, and validate and process these descriptions. If a XML description of a pipeline composition is de-serialized, the *CBServer* creates instances of proxy objects for each of the pipeline components (sensors, filter, actuators).

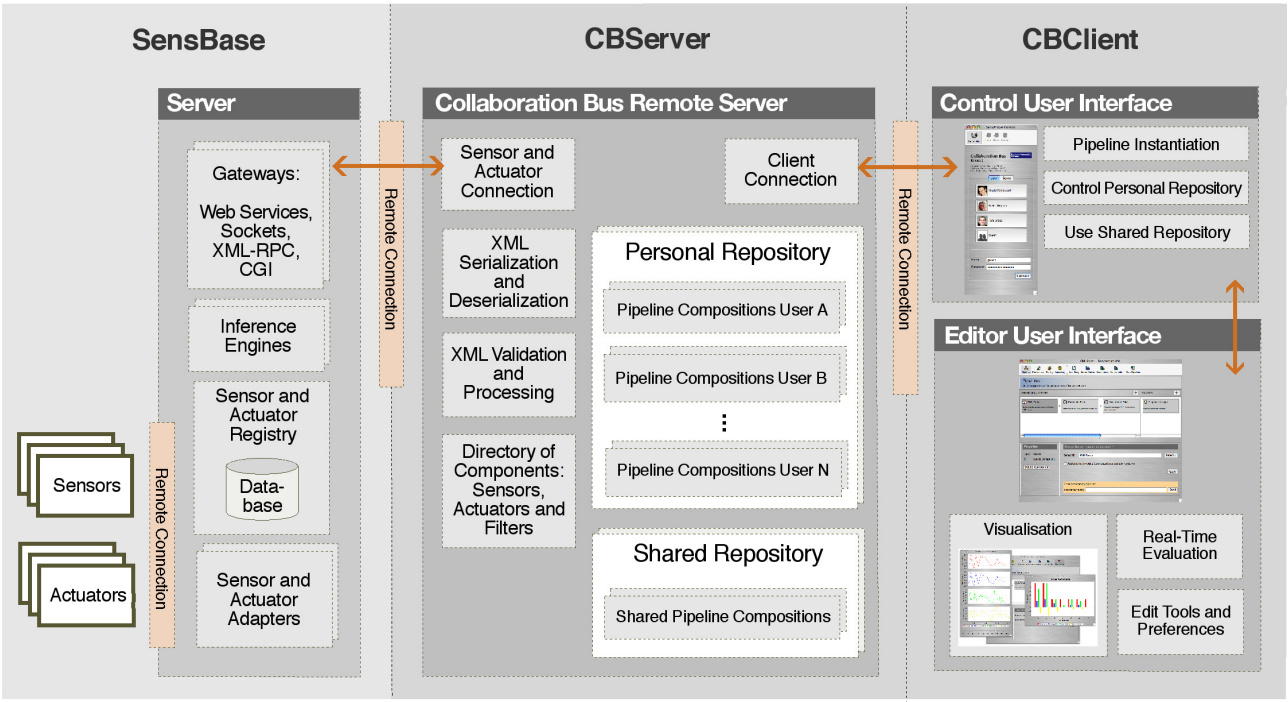


Fig. 4.1. CollaborationBus *software architecture.*

Inside of these components we have multiple threads running to ensure rapid processing of data as well as rapid forwarding of events to the subsequent component. While the sensor and actuator components inside of these compositions act as a proxy for the existing (and connected) devices, the filter objects represent the processing and transformation part inside of these compositions. A pipeline composition can include multiple processing pipelines simultaneously, and the users can run and stop as many of these compositions as they like (by using the Control User Interface).

In the Shared Repository the published pipeline compositions are stored. They are saved in the XML format as well, and XML processing is used to operate based on these descriptions (e.g., to modify existing entries, or to create an abstract pipeline composition template). Furthermore, the *CBServer* manages a directory of all the various sensor and actuator types, as well as filter components, and submits them to the client application. The dynamic directory can be extended with new components at any time, and this ensures the easy extendibility of CollaborationBus. If users want to integrate different actuators or sensors, they need to implement a new adapter driver at the *SensBase* level; this is independent from the CollaborationBus architecture. However, if new filter components are needed for a different data processing, then a new class (by deriving from an abstract base class with the core functionality of each filter component) is needed to represent this processing step. While this can be done with minor effort by any software developer, it is not easy to add a new filter for non-programmers.

The *CBClient* implements the GUIs described above. For creating, controlling and editing pipeline compositions it is necessary to support all the XML operations of the server, and the methods for instantiating pipeline compositions as well (for the editor and testing tools).

4.2. CollaborationBus Class Diagram. The class structure of the repositories and pipeline compositions is illustrated in an UML class diagram in Figure 4.2. The *PersonalRepository* class provides methods to add, remove, modify, and get *PipelineComposition* objects. The *SharedRepository* contains a collection of *SharedRepositoryEntries*, which wraps one *PipelineComposition* and specify the sharing attributes of this *PipelineComposition* (e.g., abstract or complete template).

The *PipelineComposition* object is a composite object for a series of *PipelineComponents*. It encapsulates methods for controlling pipeline compositions (e.g., start and stop), and for adding and removing pipeline components. *PipelineComponent* is the abstract base class for the *Sensor*, *Filter*, and *Actuator* base classes.

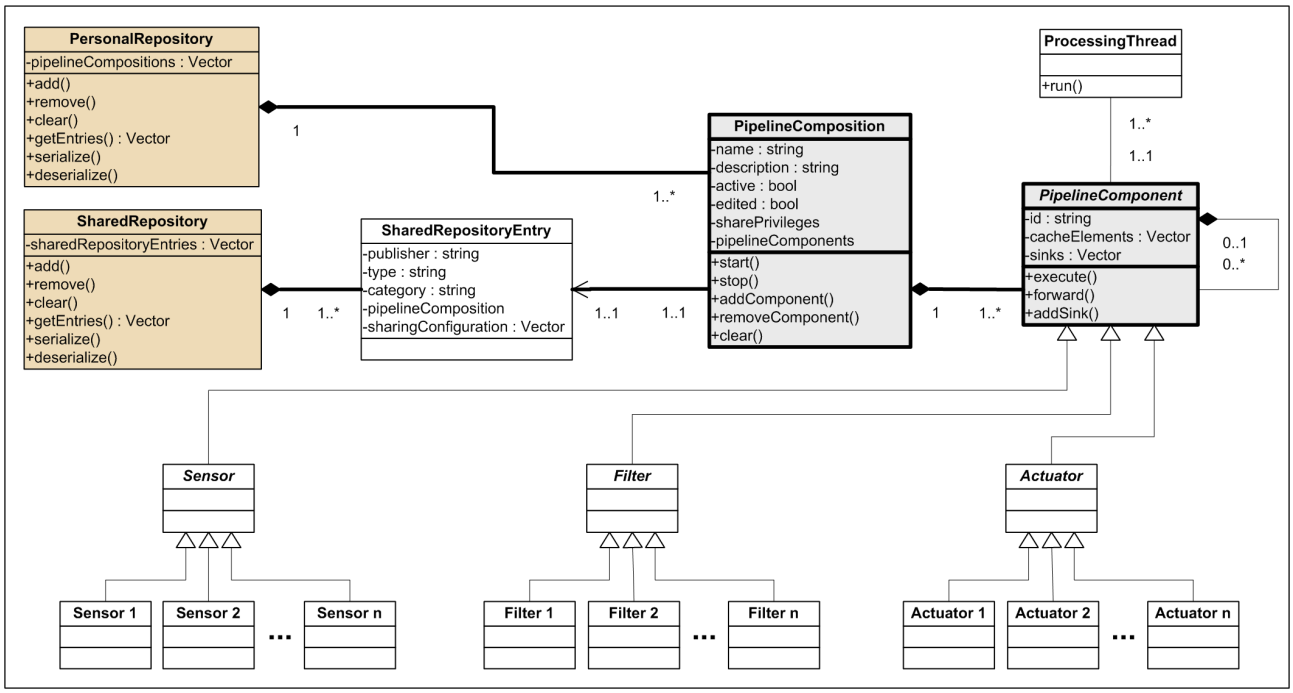


Fig. 4.2. CollaborationBus *repository and pipeline UML class diagram.*

It provides common methods for each pipeline component (like processing, forwarding and caching of events). Inside the *PipelineComponents* multiple threads (*ProcessingThread*) are running to ensure rapid processing of data as well as rapid forwarding of events to the subsequent component. *Sensor*, *Filter*, and *Actuator* are abstract base classes for the concrete pipeline components allowing to respectively: retrieve sensor values from any number of sensors from the SensBase infrastructure and push them into the pipeline process (e.g., sensor values from the Embedded Sensor Board or Phidgets hardware devices [11]); process incoming values (e.g., keywords, average, or threshold filter); and control the actuator elements (e.g., generate an RSS feed, show a message on a text display, or drive other applications via AppleScript). CollaborationBus is implemented in Java with Swing libraries for the GUIs. Several libraries are used for XML [30] processing (e.g., for the serialisation of pipeline compositions [27], for parsing sensor descriptions, for creating XPath expressions [29]); and for remote connections (e.g., XML-RPC [28], and SOAP [1]).

5. User Interface. The CollaborationBus editor provides four major graphical user interface (GUI) components: the Login and Control GUI; the Editor GUI; the Shared Repository GUI; and the Real-Time Visualisation GUI.

5.1. Login and Control GUI. The Control GUI is the central access point for all users to their personal repository of configurations. In order to get to their Control GUI, users have to login first. Figure 5.1 shows the Login and the Control GUIs.

After login, users can see the Control GUI with the listing of their pipeline compositions, including an indicator of the current state of each pipeline composition (rectangle to the right of the pipeline name): *Off* (grey), *Running* (green), or *In Edit Mode* (orange).

All functions for modifying the repository and its compositions are available from within this interface: Add, Remove, Rename, and Clone pipeline compositions (via the Commands button). Users can Start and Stop the threaded execution of each composition (via the Start/Stop button). And, they can use the Share method to upload the selected composition directly to the shared repository (via the Commands button).

5.2. Editor GUI. While the basic functions for the personal repository are available in the Control GUI, the underlying filter composition of each of the pipelines is only available in the Editor GUI that can be opened for each of the pipeline compositions. Figure 5.2 shows the Editor GUI. In the top area the user can choose several buttons for loading the Pipelines (via the Pipelines button), change the Preferences (via the Preferences

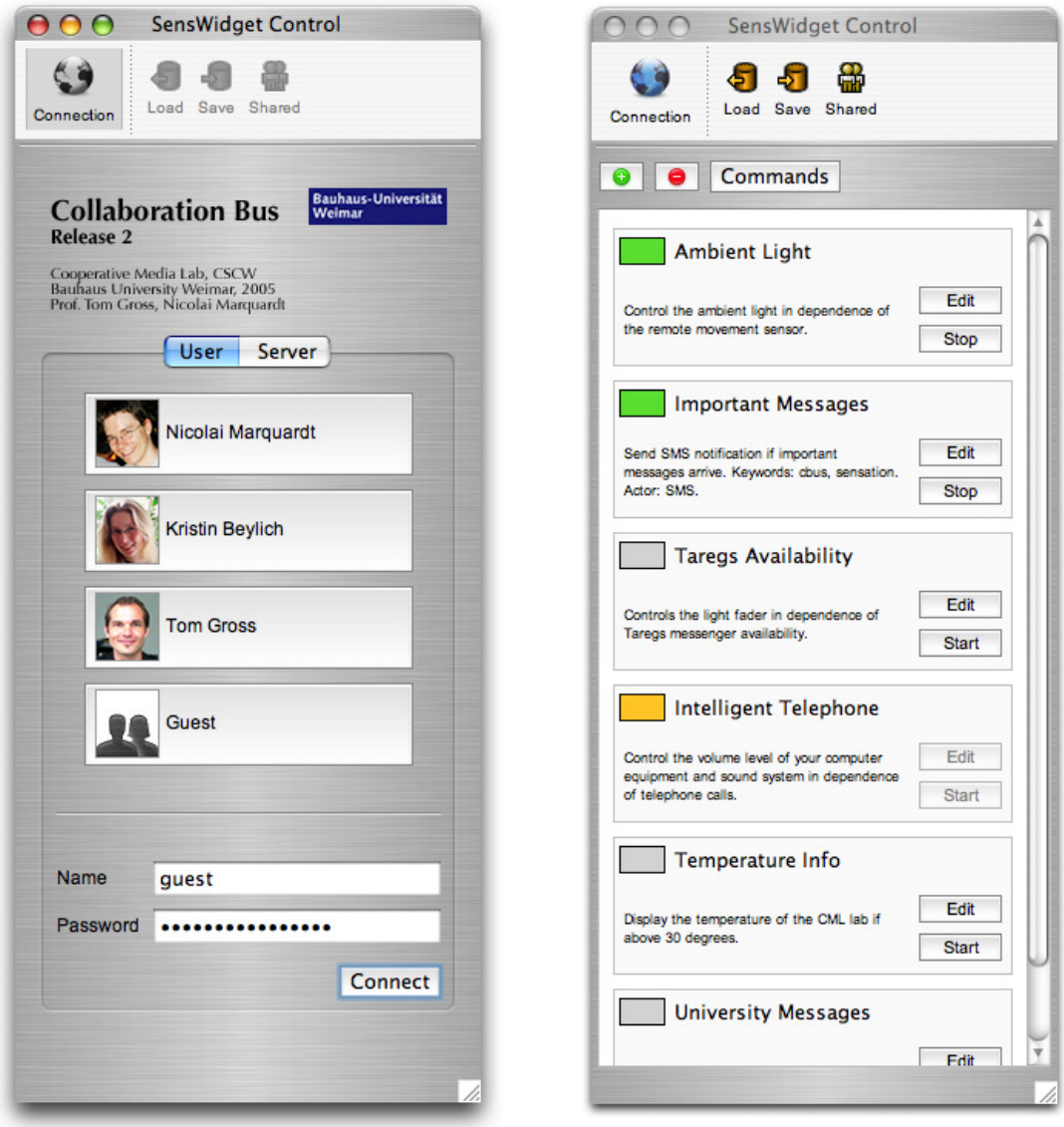


FIG. 5.1. *Login GUI and Control GUI.*

button), etc. In the middle area the respective pipeline with its sensors, conditions, and actuators is shown (each individual item is represented as a rectangular box). In the bottom area the properties of the currently selected pipeline part (rectangular box) are shown and can be altered.

In order to create a new pipeline composition, users can first discover the available sensor sources (e.g., movement sensor, temperature, sensor telephone sensor, instant messenger status sensor) of the infrastructure in a graphical sensor browser (the browser can be started by pressing the +-sign to the right of *Sensors and Conditions*), and add the sensors they need to the pipeline. Then they can specify rules and conditions (these can also be viewed by pressing the +-sign to the right of *Sensors and Conditions*) for the sensor values by adding sets of filters and operators. For each sensor types with the according sensor value type, specific filters and operators can be selected (e.g., an event value threshold, a counter for number of occurrences). Finally, the actuators can be specified by selecting them in the graphical actuator browser (the browser can be started by pressing the +-sign to the right of *Actuators*). Here, the editor provides the option to specify the mapping between the pipeline output and the actuator commands (e.g., if the pipeline output is a message, it can be displayed; if the pipeline output is a simple temperature value, the corresponding sound volume can be set).

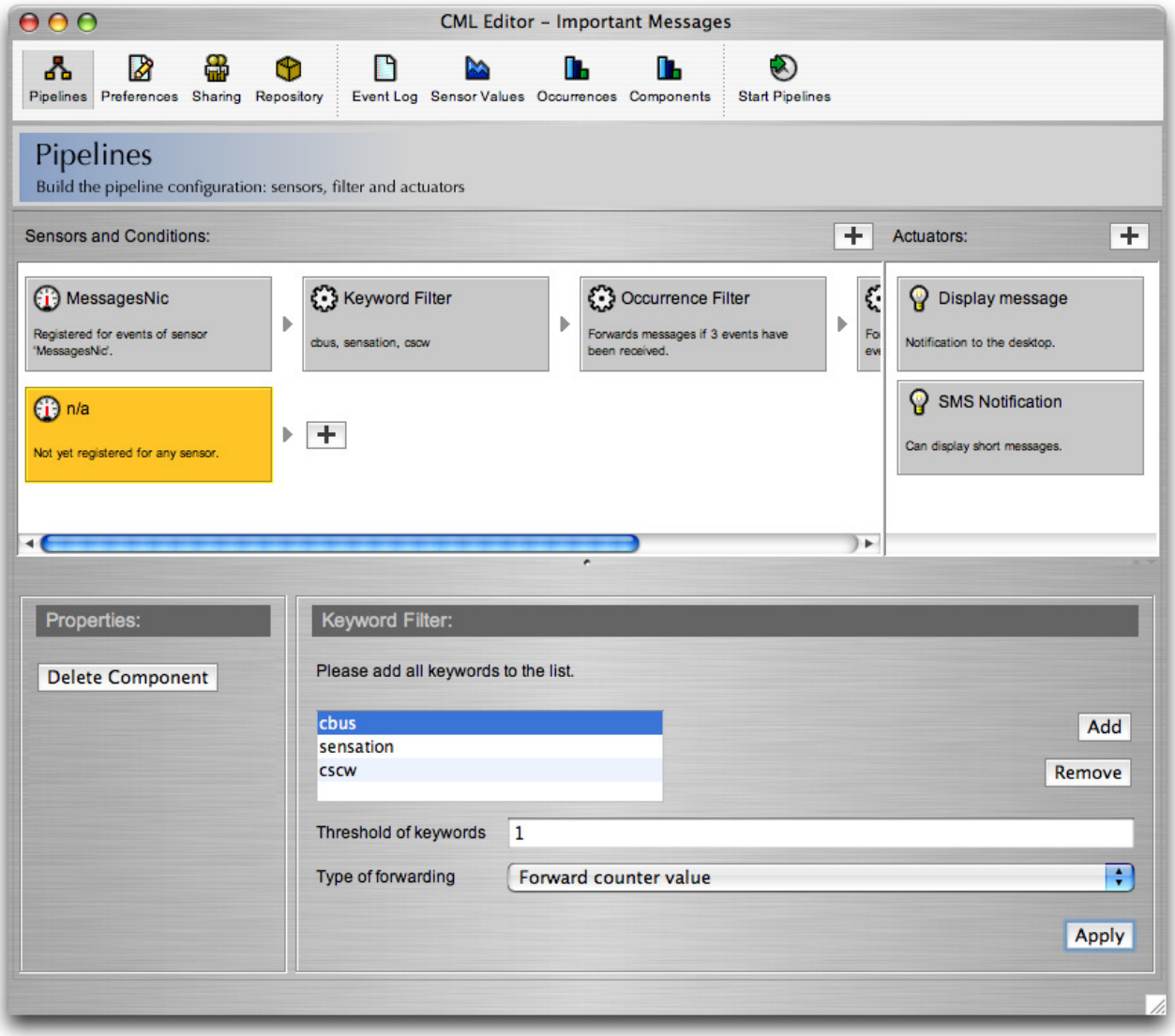


Fig. 5.2. *Editor GUI.*

5.3. Shared Repository GUI. The collaborative sharing mechanism described above is integrated in the Control GUI and in the Editor GUI. In order to make a pipeline composition available for others, users have two options. They can either select the Share method in the Control GUI (via the Shared button; cf. Figure 5.1). Here the default settings for sharing are used and no additional parameters are needed. Or they can choose the Sharing command in the Editor GUI (via the Sharing button; cf. the top area in Figure 5.2) to specify further settings for the shared composition. Further settings include description, category, and type of sharing (cf. three types of sharing above). Finally the users can upload the pipeline composition.

In order to use one of the shared pipeline compositions, the user can access the Shared Repository GUI from within the Control GUI. Figure 5.3 shows the Shared Repository GUI. By selecting one of the available compositions in the list at the left side, the information for this entry is displayed at the right side of the dialogue (description, owner, category, type of sharing, used sensor sources and actuators). Users can then download the respective composition.

5.4. Real-Time Visualisation GUI. In the assembly of pipeline compositions with a variety of components it can be difficult to keep track of the intra-pipeline communication between the components and the processing of the forwarded pipeline events. The Real-Time Visualisation GUI of the CollaborationBus

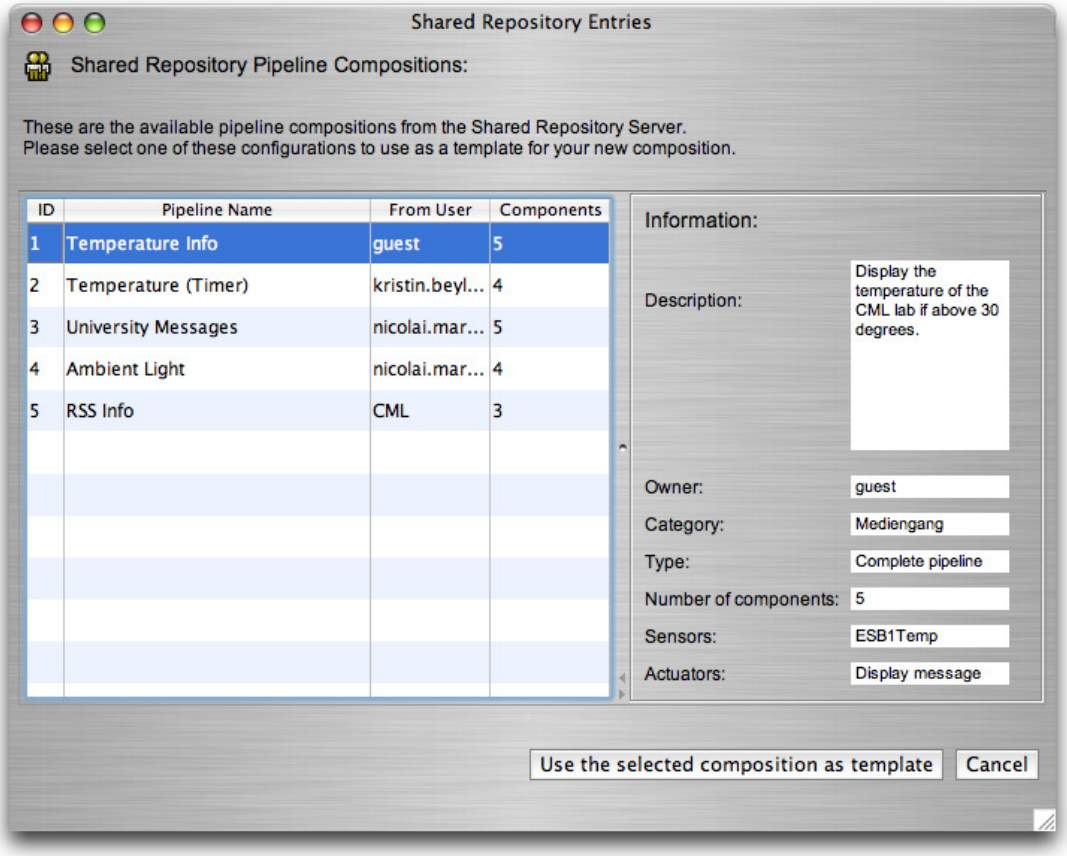


FIG. 5.3. *Shared Repository GUI.*

provides a variety of graph visualisations that can either display the forwarded values of each component of the pipeline (e.g., useful for interpolation and threshold filters) or the quantity of forwarded values (e.g., useful for gate filters, counters or timers).

Figure 5.4 shows the Real-Time Visualisation GUI with a time plot visualisation on the left (showing the absolute values of 4 temperature sensors), and an overview of the pipeline events on the right (showing the number of occurrences of events in a specific pipeline).

With these visualisations, the user obtains an inside view of the pipeline processing. The command Start Pipelines (via the Start Pipelines button) activates all components of the respective pipeline(s) and registers for the respective sensor events, starts the processing of threads, prepares the actuator modules, and generates and dynamically updates the visualisations. When any of the components of a pipeline is changed (e.g., a threshold, or an interpolation settings), the implication to the processing can be recognised immediately. Thus the adjustment and fine-tuning of component parameters becomes easier. In order to enable the testing of pipeline composition, we have, furthermore, integrated an input interface for simulated sensor events. It allows the users to manually insert sensor values to test and verify the pipeline composition without having to wait for real sensor values from the sensors. So, the processing of the data though the whole pipeline can be simulated.

6. Related Work. This chapter gives an overview of research related to the composition of sensor- and actuator-based applications. We introduce examples of programming tools for Ubiquitous Computing applications, software for controlling sensor networks, and collaborative sharing between users.

6.1. Programming Ubiquitous Computing Applications. Several research projects address the challenge to allow end-users to create and configure intelligent applications for in-home environments. With *iCAP*, Sohn and Dey introduce an application that allows end-users to rapidly prototype Ubiquitous Computing applications [25]. Similar to CollaborationBus, it uses rule-based conditions; especially the disjunction and

FIG. 5.4. *Real-Time Visualisation GUI (with time plot visualization, and overview of the pipeline events).*

junction of rules in their *sheets* is similar to our parallel and sequential pipelines (yet we think that workflow-adapted pipelines stimulate a better understanding of rule compositions than free arrangements). *iCAP* does not support sharing, or real-time visualisations.

Irene Mavrommati et al. have introduced an editing tool for creating device associations in an in-home environment [18]. Their editor connects various components called *e-Gadgets* to realise Ubiquitous Computing scenarios at home (similar to our connected processing components). Yet, it does not support workplace environments. The jigsaw editor of Jan Humble et al. [15] [22] demonstrates another application for getting control over the technological home environment. The metaphor of specifying the applications' behaviour by assembling pieces of a jigsaw puzzle sounds intuitive. Yet, we would like to give the users more control over their application than the encapsulated jigsaw pieces allow.

Some systems are based on mobile devices to control configurations from every location at every time. This includes systems for PDAs [18], mobile phones [4], and TabletPCs [15]. These mobile systems often provide only limited access to complex configuration methods. We have not created a version for mobile devices yet, but a lightweight mobile version of CollaborationBus would certainly be highly complementary to the existing version.

Another approach for configuring Ubiquitous Computing environments is programming by demonstration. This method requires an extended period of observation of relevant sensor values. In a later definition and learning phase, the users specify relevant sensor events in the event timeline, so that algorithms from artificial intelligence can detect patterns in the observed sensor values and automatically execute desired actuators [8]. Programming by demonstration tools hide most specific details of the underlying mechanisms from the users. On the one hand this reduces the barrier for non-technical users to configure Ubiquitous Computing environments, but on the other hand restricts the influence and control methods for users.

The related work applications mentioned so far address the development of complete sensor-based applications in a rather abstract way. In the *eBlocks* project [6] [7] a user interface for building sensor-based applications and configuring Boolean condition tables is introduced. As the authors show in their evaluation, users still need support in building these Boolean tables (e.g., support by different colours or written text [6]). Therefore, we introduced pipelines to allow the easy combination of Boolean AND, OR, and NOT conditions, simply by adding components to a pipeline processing stream or by adding a new parallel pipeline.

The *Phidgets* toolkit created at the GroupLab by Greenberg and Fitchett [11] facilitates the development of physical user interfaces. It provides a range of sensor and actuator elements as building blocks for letting developers rapidly prototype sensor-based applications. The included developers' toolkit allows easy access to these hardware components from within the software. This approach was further extended to distributed architectures by Marquardt and Greenberg [17]. In summary, the use of Phidgets requires few hardware skills, but considerable programming knowledge and is therefore not suitable for end-users.

6.2. Sensor Network Composition Software. A variety of applications for the compositions of sensor-based networks is available [3] [21]. For instance, the VisualSense modelling and simulation framework as part of the PTOLEMY II project [2] [3] is a toolkit for the control over fine granular sensor network communication and processing. The GUI includes functionality for processing component assembly, and for graph visualisations to display the processed values of components.

Since the evaluation of the communication in sensor networks can be difficult for newly created applications, several special complex development environments have been presented (e.g., SensorSim [21], EmTOS [9], TinyDB [16], and J-Sim [24]). These tools provide adequate development environments for expert users (because they include programming languages, operator sets, mathematical processing libraries, visualisation tools, etc.). The integration of visualisations for the event flow inside of sensor-network arrangements is interesting for our purpose [5]. However, users with a non-technical background probably have difficulties in using these applications. Furthermore, these latter environment do not support the sharing of development configurations.

6.3. Collaborative Sharing. While in Computer-Supported Cooperative Work (CSCW) collaborative sharing of location information, files, workspaces, software and patterns is wide-spread [12], an approach to sharing sensor- and actuator-based applications among users is still missing. In [12] design issues of CSCW applications that use data sharing are examined. This includes proposals for access control, adding meta-information, version history, and methods for handling updates and concurrency difficulties. Further common classifications of sharing between users are described in [19] [20]. They have found common groups with similar sharing preferences, and patterns in the sharing behaviour of users. Integrating support for these clustered groups could facilitate the usage of sharing mechanisms.

Hilbert and Trevor describe the importance of personalisation as well as shared devices for Ubiquitous Computing applications [14]. With the modification of applications to the personal needs, the use of these applications becomes easier for users.

7. Conclusion. In this paper we have introduced the CollaborationBus editor allowing any users to create sensor-actuator relations.

7.1. Summary. Even novice users can easily specify complex Ubiquitous Computing environments with the CollaborationBus editor, without having to deal with complex configuration settings or programming details. The CollaborationBus editor provides novel abstractions by encapsulating and hiding the details of the underlying base technology (e.g., the sensor infrastructure, the sensor and actuator registration, the sensor event registration). At the same time, more experienced users can control the pipeline composition configuration in any technical detail they need and get details on demand.

Furthermore, users can share their pipeline compositions with colleagues and friends via a shared repository. Users can also decide how accurate they want to share (e.g., complete compositions, abstract template, only the processed event value). With a minimum effort, each user can browse the shared repository and download shared pipeline compositions and adapt the used shared repository template to fit to their needs (by specifying their own personal properties of the pipeline). This way the CollaborationBus features an incrementally growing library of ready-to-use pipeline compositions that form a diverse network of collaborative sensor-actuator-relations.

7.2. Evaluation. While the evaluation of the CollaborationBus GUI and functionality as well as the produced pipeline compositions is of vital interest to us, a formal user evaluation is still missing. Nevertheless, we have collected several user opinions at the public demonstration of CollaborationBus to many visitors at the Cooperative Media Lab Open House 2005 from 14 to 17 July 2005, where the visitors had the chance to try out the CollaborationBus software in detail (with a huge set of connected sensors and actuators).

Most of the visitors quickly started to create their own compositions, and to select desired sensors, actuators and filters. At the same time, they hesitated to change the configuration of the filter components, and were somehow not completely confident about whether they change the right parameters. A helpful support in this

case was the Real-Time Visualisation GUI; in particular, the activation of the graph views of all pipeline events. It supported users in understanding the effect of parameter changes.

The most popular function of the tool was the integrated sharing mechanism. Users enjoyed browsing the large set of ready-to-use pipeline compositions in the shared repository. Often they used one of the shared compositions as template, modified parameters in the compositions or built a new configuration on the basis of this composition and sometimes shared this composition again. They also liked the idea of sharing their own compositions with others.

A typical barrier of users when creating sensor-based applications with CollaborationBus was that they worried about privacy issues. Many of the visitors said that it is an important criterion influencing their decision to use such as systems was to exactly know all outgoing or shared personal data and to be able to quickly and easily change the settings.

7.3. Future Work. Currently all components of the CollaborationBus system presented in this paper have been implemented. In the future we would like to evaluate the created pipeline compositions of users (especially those in the shared repository), and identify common patterns in the created compositions. From that we would like to develop assistive functions that provide users suggestions for reasonable compositions. The configuration interface of the filter components in the Editor GUI can also be improved to become more intuitive for the user. A graphical mapping could allow users to drag and drop the desired input and output commands and the component configuration.

A final important aspect related to security and privacy is the introduction of a system-wide authorisation and authentication system in order to further secure the access to the sensor values and pipeline compositions. For this purpose the CollaborationBus repository storage and the sensor value access could be integrated in the security system of the Sens-ation platform.

8. Acknowledgement. We would like to thank all members of the Cooperative Media Lab—especially Tareg Egla, and Christoph Oemig—for inspiring discussions on CollaborationBus, and for providing the PRIMI and Sens-ation platforms. Thanks to the anonymous reviewers for valuable comments.

REFERENCES

[1] Apache Software Foundation, *WebServices - Axis Architecture Guide. http://ws.apache.org/axis/java/architecture-guide.html*, 2005. (Accessed 5/5/2010).

[2] P. Baldwin, S. Kohli, E. Lee, X. Liu, and Y. Zhao, *Modelling of sensor nets in ptolemy ii*, in Proceedings of the Third International Symposium on Information Processing in Sensor Networks - IPSN 2004 (Apr. 27-27, Berkeley, CA), New York, NY, USA, 2004, ACM Press, pp. 359–368.

[3] ———, *Visualsense - Visual Modeling for Wireless and Sensor Network Systems*, tech. report, Report Number: UCB ERL Memorandum UCB/ERL M04/8, Ptolemy Project, 2004.

[4] L. Barkhuus and A. Vallgarda, *Smart Home in Your Pocket*, in Interactive Poster: Presented at The Fifth International Conference on Ubiquitous Computing - UbiComp 2003 (Oct. 12-15, Seattle, WA)., 2003.

[5] C. Buschmann, D. Pfisterer, S. Fischer, S. Fekete, and A. Kroeller, *SpyGlass: A Wireless Sensor Network Visualiser*, SIGBED Review 2, 1, (2005), pp. 1–6.

[6] S. Cotterell and F. Vahid, *A Logic Block Enabling Logic Configuration by Non-experts in Sensor Networks*, in Extended Abstracts of the 23th ACM Conference on Human Factors in Computing Systems - CHI 2005 (Portland, Oregon, USA), New York, NY, USA, 2005, ACM Press, pp. 1925–1928.

[7] S. Cotterell, F. Vahid, W. Najjar, and H. Hsieh, *First Results with eBlocks: Embedded Systems Building Blocks*, in Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES+ISSS 2003 (Oct. 1-3, Newport Beach, CA), IEEE Computer Society, 2003, pp. 168–175.

[8] A. Dey, R. Hamid, C. Beckmann, and D. Hsu, *A CAPpella: Programming by Demonstration of Context-Aware Applications*, in Proceedings of the Conference on Human Factors in Computing Systems - CHI 2004 (Apr. 24-29, Vienna, Austria), ACM Press, 2004, pp. 33–40.

[9] H. Gellersen, G. Kortuem, A. Schmidt, and M. Beigl, *Physical Prototyping with Smart-Its*, IEEE Pervasive Computing, 3 (2004), pp. 74–82.

[10] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer, *A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks*, in In Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems - SenSys 2004 (Nov. 3-5, Baltimore, MD), ACM Press, 2004, pp. 201–213.

[11] S. Greenberg and C. Fitchett, *Phidgets: Easy Development of Physical Interfaces Through Physical Widgets*, in Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology - UIST 2001 (Orlando, Florida, USA), New York, NY, USA, 2001, ACM Press, pp. 209–218.

[12] I. Greif and S. Sarin, *Data sharing in group work.*, ACM Transactions on Offce Information Systems 5, 2, (1987), pp. 187–211.

[13] T. Gross, T. Egla, and N. Marquardt, *Sens-ation: A Service-Oriented Platform for Developing Sensor-Based Infrastructures*, International Journal of Internet Protocol Technology (IJIPT), 1 (2006), pp. 159–167.
[14] D. Hilbert and J. Trevor, *Personalizing shared ubiquitous devices*, ACM interactions 11, 3, (2004), pp. 34–43.
[15] J. Humble, A. Crabtree, T. Hemmings, K.-P. Åkesson, B. Koleva, T. Rodden, and P. Hansson, *"Playing with the Bits" - User-Configuration of Ubiquitous Domestic Environments*, in Proceedings of the Fifth International Conference on Ubiquitous Computing - UbiComp 2003 (Seattle, Washington, USA), Seattle, WA, USA, 2003, pp. 256–263.
[16] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, *TinyDB: An Acquisitional Query Processing System for Sensor Networks*, ACM Transactions on Database Systems 30, 1, (2005), pp. 122–173.
[17] N. Marquardt and S. Greenberg, *Distributed Physical Interfaces with Shared Phidgets*, in Proceedings of the 1st International Conference on Tangible and Embedded Interaction - TEI 2007 (Baton Rouge, LA, USA), New York, NY, USA, 2007, ACM Press, pp. 13–20.
[18] I. Mavrommati, A. Kameas, and P. Markopoulos, *An Editing Tool that Manages Device Associations In an in-home Environment*, Personal and Ubiquitous Computing, 8 (2004), pp. 255–263.
[19] J. Olson, J. Grudin, and E. Horvitz, *Towards Understanding Preferences for Sharing and Privacy*, tech. report, Report Number: MSR-TR-2004-138, Microsoft Research, 2004.
[20] ———, *A Study of Preference for Sharing and Privacy*, in Extended Abstracts of the Conference on Human Factors in Computing Systems - CHI 2005 (Apr. 2-7, Portland, OR), ACM, 2005, pp. 1985–1988.
[21] S. Park, A. Savvides, and M. Srivastava, *SensorSim: A Simulation Framework for Sensor Networks*, in Proceedings of the 3rd ACM International Workshop on Modelling, Analysis, and Simulation of Wireless and Mobile Systems - MSWiM 2000 (Aug. 11, Boston, MA), ACM, 2000, pp. 104–111.
[22] T. Rodden, A. Crabtree, T. Hemmings, B. Koleva, J. Humble, K.-P. Akesson, and P. Hansson, *Between the Dazzle of a New Building and its Eventual Corpse: Assembling the Ubiquitous Home*, in Proceedings of the 5th ACM Conference on Designing Interactive Systems - DIS 2004 (Cambridge, Massachusetts, USA), New York, NY, USA, 2004, ACM Press, pp. 71–80.
[23] D. Salber, A. K. Dey, and G. D. Abowd, *The Context Toolkit: Aiding the Development of Context-Enabled Applications*, in Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI 1999 (Pittsburgh, Pennsylvania, USA), New York, NY, USA, 1999, ACM Press, pp. 434–441.
[24] A. Sobeih, W.-P. Chen, J. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and K. Zhang, *J-Sim: A Simulation Environment for Wireless Sensor Networks*, in Proceedings of the 38th Annual Symposium on Simulation (Apr. 2-8, San Diego, CA), IEEE Computer Society, 2005, pp. 175–187.
[25] T. Sohn and A. Dey, *iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications*, in Extended Abstracts of the 21st ACM Conference on Human Factors in Computing Systems - CHI 2003 (Fort Lauderdale, Florida, USA), New York, NY, USA, 2003, ACM Press, pp. 974–975.
[26] S. Talja, *Information Sharing in Academic Communities*, New Review of Information Behaviour Research 3, (2002), pp. 143–159.
[27] J. Walnes, *XStream - Architecture Overview. http://xstream.codehaus.org/architecture.html*, 2010. (Accessed 5/5/2010).
[28] D. Winer, *XML-RPC Specification. http://www.xmlrpc.com/spec*, 1999. (Accessed 5/5/2010).
[29] World Wide Web Consortium (W3C), *XML Path Language (XPath). http://www.w3.org/tr/xpath*. W3C Recommendation, November 1999. (Accessed 5/5/2010).
[30] ———, *Extensible Markup Language (XML) 1.0. http://www.w3.org/xml/*. W3C Recommendation, August 2006. (Accessed 5/5/2010).

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 305–325. http://www.scpe.org
ISSN 1895-1767

WIDE AREA DISTRIBUTED FILE SYSTEMS—A SCALABILITY AND PERFORMANCE SURVEY

KOVENDHAN PONNAVAIKKO* AND JANAKIRAM DHARANIPRAGADA*

Abstract. Recent decades have witnessed an explosive growth in the amounts of digital data in various fields of arts, science and engineering. Such data is generally of interest to a large number of people spread over wide geographical areas. Over the years, several Distributed File Systems (DFS) have, to varying degrees, addressed this requirement of sharing large amounts of data, stored in the form of files, among several users and applications. Scalability and performance are two important measures that determine the suitability of a file system for the applications executing over them. We perform a detailed comparative analysis of popular distributed file systems in terms of these measures in our survey.

1. Introduction. In recent decades, we have been witnessing increasingly large rates of data generation and growing numbers of widely spread collaborative applications. For example, data requirements of *High Performance Computing* (HPC) applications have been continuously growing over the past few years and are expected to grow even more rapidly in the years to come [23]. Experimental setups, deployments of sensors, simulators, agents, etc. generate large amounts of data which researchers world over can have use for. Other examples include WikipediaFS [10], and large scale telemedicine [24].

Organizing and sharing raw and processed data files owned by different users and groups calls for the need of large scale *Distributed File Systems* (DFS) [46] [7] [8].

Any file system that allows files to be placed across the network and yet make accesses appear local is a distributed file system. Certain systems are *Client-Server* based (*Asymmetric*) in that dedicated servers exist to provide file services. In *Peer-to-Peer* (P2P) or *Symmetric* file systems, data/metadata management load is distributed among all the nodes. *Clustered* file systems are those in which the data/metadata server is replaced by a cluster of servers to better distribute load and handle failures. A *Parallel* file system enables concurrent reads and writes of the same file and parallel I/O [22]. Some parallel file systems support the striping of a file across multiple storage devices.

There exist several large scale distributed file systems. For our survey, we consider a set of popular production systems and research prototypes (table 1.1)[1]. This set has been chosen so as to cover the major architectural variations of existing systems.

These systems vary in terms of their typical application workloads and the geographical spread of their typical usage. For example, some of them are designed for desktop workloads and some for scientific applications. Some of the analyzed systems are not designed to be wide area file systems, i. e., clients and servers are not designed to be geographically spread across *Wide Area Networks* (WAN). However, other features such as high scalability have prompted researchers to adapt even such systems for use across WANs. Some examples include the usage of Lustre file system in [42] and Parallel Virtual File System 2 in [5].

Keeping in mind the common nature of new generation applications, we analyze the architectures of these systems with respect to the following application requirements. The first requirement is that of scalability with respect to the number of nodes and files. In other words, increasing the number of nodes and/or files must not adversely affect query/access times. The other major requirement is that of maintaining high application performance. For HPC applications, performance can be measured in terms of makespan, computation or I/O throughput, etc. In file systems maintained for home directories and such, performance can be measured in terms of query response latencies, file access/update times, and so on.

Using a few system parameters, we attempt to characterize the effects of increasing query and I/O loads on individual file system servers. We also study the support provided by the different systems for sophisticated data placement and migration strategies, which are critical for high application performance. In section 2, we discuss some of the design considerations in the context of large scale DFSs. Section 3 summarizes the system architectures of the various DFSs analyzed in this survey. The comparative analysis is presented in section 4.

*Distributed and Object Systems Lab, Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

[1] An extensive list of computer file systems can be found at [3]. Comparisons of general and technical features of a large number of file systems can be found at [2].

TABLE 1.1
Set of Analyzed File Systems

Andrew File System
Ceph
Common Internet File System
Edge Node File System
Farsite
Google File System
Ivy
Lustre File System
OceanStore
Panasas Parallel File System
Pangaea
Parallel Virtual File System 2
WheelFS

TABLE 2.1
Classification of the Analyzed File Systems

Category	**Name**	**Systems**
I	Traditional Distributed File Systems	Andrew File System, Common Internet File System
II	Asymmetric Cluster File Systems	Ceph, Google File System, Lustre File System, Panasas Parallel File System, Parallel Virtual File System 2, WheelFS
III	Self-Organizing P2P File Systems	Edge Node File System, Farsite, Ivy, OceanStore, Pangaea

2. Design Considerations. Traditionally, distributed file system designers have adopted a client-server model. In these asymmetric systems, dedicated servers exist to provide file services and clients only consume the services. Typically, the server exports hierarchical namespaces and clients mount the exported hierarchies in their local namespaces.

A client-server approach has several advantages such as ease of maintenance, efficient management of concurrent reads and writes of the same file, and centralized security control. However, the presence of a centralized server presents significant scalability constraints. File system performance degrades with increasing file sizes, and increasing numbers of files and users.

One of the early approaches to improve file system performance is client side caching. While caching helps in reducing network traffic, it also introduces consistency issues. Cached content can become stale and write collisions can occur, especially in file systems with stateless servers.

In later distributed file system designs, a multitude of strategies have been employed to address issues related to scalability. Individual servers have been replaced by clusters of servers. Analogous to *Sharding* in databases, in such file systems, namespaces are partitioned and distributed among the different servers in the cluster. This helps in the distribution of load and hence better performance.

Another effective strategy is to decouple data management from metadata management. While data refers to the actual content of files, metadata in the context of file systems refers to the data about file contents. Unlike data operations, metadata operations are usually small, random and non-sequential.

Decoupling is achieved by using different sets of servers for data and metadata management. In a typical file system, a large proportion of queries are related to file metadata. On the other hand, responses to data access queries are much more voluminous. Using different sets of servers for managing data and metadata therefore helps improve system performance. Clustering and decoupling data and metadata have enabled other scalability and performance optimizing strategies such as replication and striping a file's content across multiple storage devices.

DFS features such as concurrent access, file striping and replication complicate the task of presenting a consistent view of the file system to all users. Concurrent accesses can be controlled by associating data and

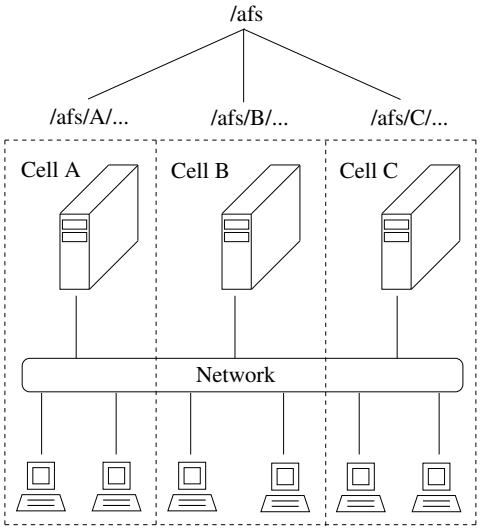


FIG. 3.1. *AFS System Architecture*

metadata with different kinds of locks. In UNIX, the two common locking mechanisms, *fcntl* and *flock*, allow *Exclusive* and *Shared* locks to be applied to files/blocks. All exclusive locks must have been released before shared locks can be obtained by clients and all kinds of locks (shared and exclusive) must be released before an exclusive lock can be obtained.

While pessimistic approaches such as locking allow file systems to support *Strict Consistency Semantics*[2], they also affect application performance by increasing messaging overheads and wait times. Certain file systems support weaker consistency semantics by allowing concurrent accesses in conflicting modes. In such systems, applications either ensure that colliding accesses do not occur, or have appropriate conflict resolution mechanisms in place.

High availability of data and metadata is usually a crucial requirement of distributed file systems. Several approaches exist to improve a file system's availability, each associated with certain overheads. Some of the approaches are replication, caching, versioning, logging, and anticipatory reads. Different systems employ different combinations of these techniques to achieve the required levels of availability.

Though clustered file systems are more scalable than traditional client-server systems, their scalability is limited because of the manually maintained set of server clusters. A central augmentable set of servers has other drawbacks too. Clusters are expensive to set up and maintain. Storage of entire file systems in a limited number of sites makes access from distant locations inefficient as a result of high network latencies. Moreover, such setups create single points of failure, and are prone to physical vulnerabilities.

Increasing rates of data generation and number of collaborations among geographically distributed groups of users have created the need for *Global* and P2P file systems. P2P systems involve minimal or no central coordination. In P2P or symmetric file systems, data and metadata management load is distributed among all the nodes in the system. These systems are generally designed to be self-organizing due to the impracticality of manually administrating huge numbers of storage/compute resources.

Based on the different evolutionary stages of DFS design, we classify the analyzed systems into the categories of *Traditional Distributed File Systems*, *Asymmetric Cluster File Systems* and *Self-Organizing P2P File Systems* (table 2.1).

3. System Architectures. In this section, we present brief independent reviews of the system architectures of the considered file systems.

3.1. Traditional Distributed File Systems. Though *Network File System* (NFS) [39] (up to version 3) is one of the most commonly used distributed file system protocols, it is usually used in a local area network or within a single administrative domain. We have therefore not included NFS in this survey. Influenced by

[2]A read returns the most recently written value.

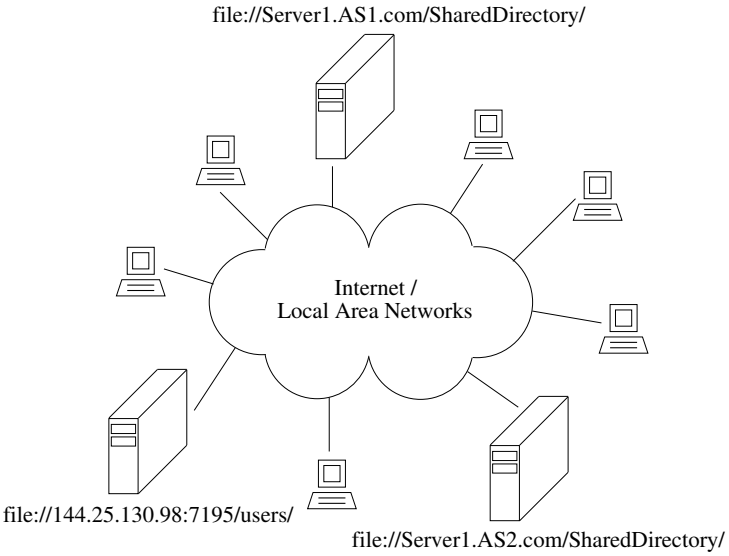


FIG. 3.2. *CIFS System Architecture*

Andrew File System [21] and Common Internet File System [28], version 4 of NFS [43] supports stateful servers and locks, includes other performance improvements and can be used in wide area networks.

3.1.1. Andrew File System (AFS). Started at the Carnegie Mellon University, AFS [21] uses a set of trusted servers for sharing a common directory structure among several thousand client machines. AFS relies on data caching to address the issue of scalability. While earlier versions of AFS required clients to fetch whole files, versions since AFS 3 support the transfer of smaller blocks of files.

Servers maintain state about clients which have files open. *Callbacks* are used to maintain the consistency of cache contents. Whenever file contents are altered, servers send invalidation messages to the corresponding clients. A client, on the other hand, informs the server about the changes that it has made only at the time of closing. As a result, AFS only supports *Session Semantics*[3] and not *One-Copy Update Semantics*[4], which is supported by UNIX.

The AFS model (figure 3.1) comprises of a set of cells, each cell usually being a set of hosts with the same Internet domain name. Each cell has servers executing the *Vice* process and clients executing the *Venus* process. AFS provides location independence by performing the mapping between filenames and locations at the servers. The hierarchical directory structure is partitioned into *Volumes*, which act as containers for related files and directories. Volumes can be transparently migrated between servers. Read-only cloned copies of volumes must be created by administrators to enable recovery in the case of failures. The Kerberos [44] protocol is used for the mutual authentication of clients and servers.

3.1.2. Common Internet File System (CIFS). CIFS [28] is Microsoft's version of the *Server Message Block* (SMB) protocol along with certain other protocols. CIFS provides remote file access over the Internet (figure 3.2) with features such as global naming, caching, volume replication, remote sharing and locking. SMB uses flat namespaces to address files and CIFS makes use of the Internet naming system, *Domain Name Service* (DNS). While changes in file addresses are difficult to propagate in SMB, CIFS uses the scalable notification system of DNS to handles such changes. Unlike several other wide area file systems, *Unicode* filenames are supported.

Parallelism is supported at the directory level only and individual files cannot be split among multiple servers. Since each file/directory must be associated with particular servers and servers are manually administered, scalability with respect to installations and query/data transfer loads in CIFS is limited.

[3] Changes made to a file are visible to the other clients only after the writing client closes the file.

[4] In one-copy update semantics, every read sees the effect of all previous writes and a write is immediately visible to clients who have the file open for reading.

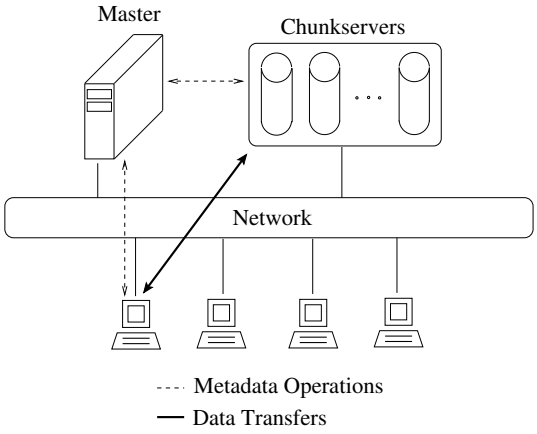


FIG. 3.3. *GoogleFS System Architecture*

3.2. Asymmetric Cluster File Systems. There are different kinds of storage architectures that distributed file systems use. Traditional distributed file systems discussed in section 3.1 such as NFS, AFS and CIFS adopt a *Network-Attached Storage* (NAS) architecture. Servers in these systems provide file-based access to their dedicated storage devices, to clients across networks.

In the *Storage Area Network* (SAN) architecture, large storage devices such as arrays of disks are shared by a cluster of nodes. Unlike NAS, data access is block-based (finer granularity), which results in increased flexibility in storing huge files. SAN based file systems translate file-level operations to block-level operations at the client. Metadata management is either handled by a central server or distributed among the cluster nodes.

IBM's *General Parallel File System* (GPFS) [18] is an example for a clustered file system that adopts the SAN architecture. GPFS uses a distributed token management system to handle concurrent file accesses among cluster nodes. It also supports data sharing among multiple GPFS clusters.

Another storage architecture employed by several clustered file systems such as Lustre [40], Panasas [50] and Ceph [48], uses *Object-based Storage Devices* (OSD). OSDs are evolved disk drives that can directly handle the storage and serving of objects as against normal disk drives which work at the level of bits, tracks, and sectors. In other words, an OSD handles lower level functionalities related to object management within the device and exposes object access interfaces to applications.

In block-based file systems, file metadata, which includes block locations, is managed by the file system. As a result, performance is effected for large files since metadata sizes are also large. On the other hand, OSD based file systems manage objects only. The lower level details about content striping are handled by the storage devices themselves. This results in improved performance and throughput.

Several client applications benefit from moving computation to where the data is, instead of getting the content transferred to the clients [36] [47]. For such applications, performance depends on the intelligence of OSDs [17], in terms of their ability to execute user specified computations, as well as on their processing power.

3.2.1. Google File System (GoogleFS). GoogleFS [19] is a DFS for data intensive applications, custom-built for the application workload and technical environment at Google. A GoogleFS cluster comprises of a single *Master* and several *Chunkservers*, as shown in figure 3.3.

The master manages the metadata and the chunkservers store the data. The master uses *Heartbeat* messages to periodically monitor the chunkservers. A *Shadow* master is maintained in order to handle the failure of the primary master. Files are split into fixed size chunks. A certain number of replicas (three is the default number) of the chunks are stored in the chunkservers. Chunk replicas are spread across racks to maximize availability.

The master maintains information about the location of each chunk and access control information. The master performs periodic re-balancing of data to ensure that the chunkservers are uniformly loaded at all times. Clients obtain file metadata from the master and perform all data related operations at the chunkservers.

The datasets that applications at Google work with are usually huge in size and the workload primarily involves append operations. Hence, GoogleFS supports record append operations only and not random write operations. Servers are stateless and clients do not cache data in GoogleFS. That is because applications at

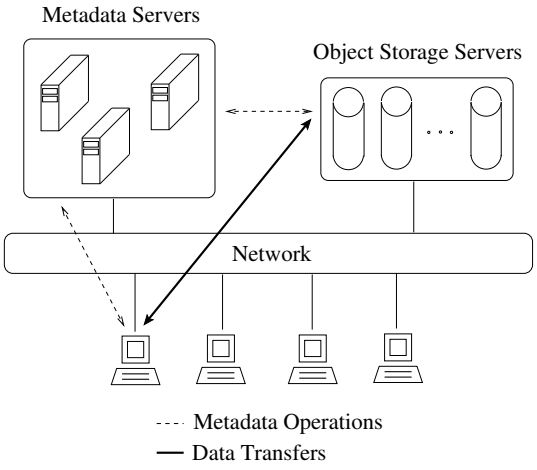


FIG. 3.4. *Lustre System Architecture*

Google usually require certain operations to be performed on file contents and only the result to be returned to them. In fact, the predominant class of application is *MapReduce* [16].

The architecture of GoogleFS makes it suitable for a specialized set of workloads only. Also, its centralized master can become a performance bottleneck, especially for metadata intensive workloads. *Hadoop Distributed File System* (HDFS) [13] is an open source Java product with almost the same architecture as that of GoogleFS.

3.2.2. Lustre File System. Lustre [40] is an object based DFS used primarily for large scale cluster computing. It is a production system used in several HPC clusters. The system architecture of the Lustre file system is shown in figure 3.4. The system comprises of three main components, namely, file system clients, *Object Storage Servers* (OSS) which provide file I/O services, and Metadata servers (MDS).

Typically, the above three components are on independent nodes which communicate over the network. Using an intermediate network abstraction layer, Lustre supports multiple network types such as *Ethernet* and *Infiniband*. Redundancy, in the form of an active/passive pair of MDSs and active/active pairs of OSSs, helps Lustre maintain high availability.

Lustre enforces strict consistency semantics, using locks to enforce serialization. It also uses the *Journaling File System Technology*[5] to prevent data/metadata corruption due to system failures and to enable persistent state recovery.

Since metadata servers as well as object storage servers need to be manually administered, Lustre does not scale transparently.

3.2.3. Panasas Parallel File System. Panasas [50] uses parallel and redundant access to OSDs to provide a high performance DFS. At a high level, the system model of Panasas is similar to that of the Lustre (figure 3.4).

The Panasas object storage nodes have a *Blade* architecture, each blade comprising of disks, a processor, memory, and a network interface. Thus, adding storage capacity includes the addition of the required computing power to efficiently manage the new disks. The storage blades use a specialized file system which implement the object storage primitives. A per-file *RAID* system [32] is used to provide for data integrity and scalable performance.

The storage blades are managed by a set of *Quorum-based* cluster managers. The set of managers maintains the replicated system state using a quorum-based voting protocol. Managers stripe file contents across the OSDs. They also handle multi-user access, consistent metadata management, client cache coherence, and recovery from client and OSD failures. *Transaction Log Replication* protocol is used to tolerate metadata server crashes.

3.2.4. Parallel Virtual File System, Version 2 (PVFS2). PVFS2 [4] is an open source DFS that provides high performance and scalable file system services for large node clusters. Each cluster node can be a

[5]Maintains logs of impending changes before committing them to the file system.

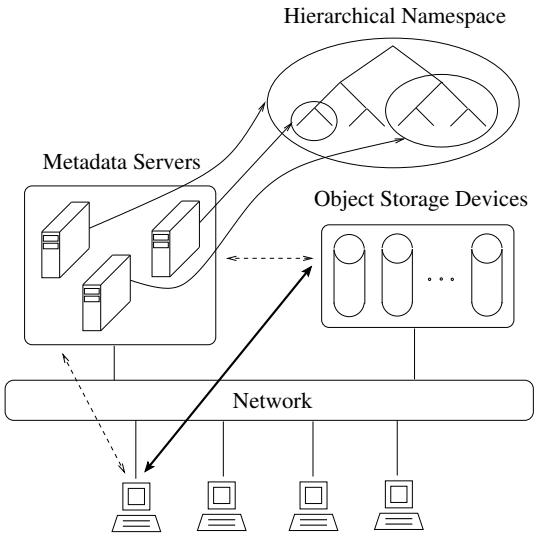


FIG. 3.5. *Ceph System Architecture*

server, a client, or both. Like several other clustered file systems, PVFS2 also supports the striping of a file's data across several storage nodes. PVFS2 allows for a subset of the servers to be configured as metadata servers.

PVFS2 servers are stateless and as a result, locks are not supported. Client failures thereby do not affect the system in anyway. While this lets the system scale to a large number of clients, it results in little support for different kinds of access semantics. While PVFS2 provides atomicity guarantees for updates to non-overlapping portions of a file, simultaneous writes to overlapping regions can result in inconsistent file states.

New file/directory creation is performed by first creating the data object and the corresponding metadata object, and then making the metadata object point to the data object, and finally creating a directory entry pointing to the metadata object. This way, the file system remains in a consistent state always. This mechanism can result in significant amounts of clean up load in case of collisions, i. e., in case of simultaneous updates to the same portions of the namespace.

PVFS2 specializes in supporting flexible data distribution as well as flexible data access patterns. For example, it supports access to non-contiguous portions of a file in a single operation. In that sense, PVFS2 implements *MPI-IO Semantics* closely.

Like Lustre, PVFS2 uses an intermediate layered interface to support multiple network types. Traditional solutions for high availability, such as those used by Lustre, can be used in PVFS2. An experimental comparison of PVFS2 and Lustre for large scale data processing is presented in [41].

3.2.5. Ceph. Ceph [48] is an object-based distributed file system designed to provide high performance, reliability and scalability. *Dynamic Subtree Partitioning* and the distribution of objects using a pseudo random function, are a couple of its unique features. The system (figure 3.5) comprises of clients, OSDs and a metadata servers cluster.

Ceph completely does away with allocation lists and *inode* tables. Instead, a pseudo random function called *CRUSH* [49] is used for the distribution of objects among the OSDs. Clients can therefore calculate the location of file objects instead of performing a look-up.

Some file systems use static subtree partitioning to delegate authority for different subtrees of a hierarchical namespace to different metadata servers. Another approach uses hash functions to distribute metadata. While the first approach cannot handle dynamic loads efficiently, the later approach does away with metadata locality. Ceph uses a dynamic subtree partitioning strategy, in which responsibilities for different subtrees of the namespace are dynamically distributed among the MDSs. The distribution ensures that server loads are kept balanced with changing access patterns. Popular portions of the namespace are also replicated on multiple servers.

Ceph replicates data using a variant of the *Primary-Copy Replication*[6] technique to maintain high availability. The usage of CRUSH rules out the possibility of considering specific node characteristics while making

[6]One of the replicas, which is made the primary copy, serializes transactions and sends updates to the secondary replicas.

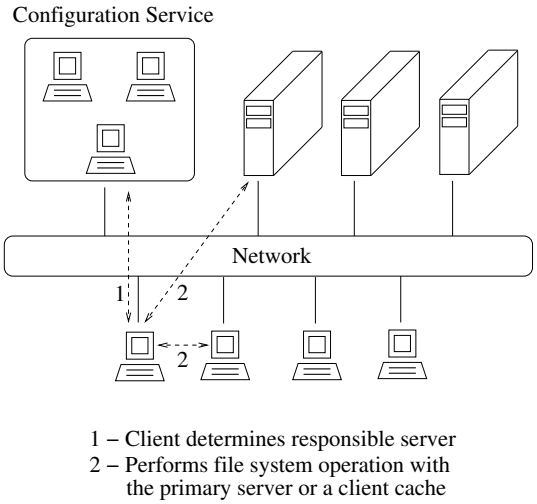


FIG. 3.6. *WheelFS System Architecture*

object placement decisions. In wide area installations, the average network latency between clients and Ceph's metadata servers can be high, affecting the performance of applications involving large proportions of metadata operations.

3.2.6. WheelFS. WheelFS [46] provides applications control over replica placement, consistency and failure handling mechanisms using *Semantic Cues*. The system allows applications to manage the trade-off between the immediacy of update visibility and the independence of client sites to operate on the data. A set of WheelFS servers (figure 3.6) store file and directory objects. Each file/directory has a primary server which holds its latest content. Clients also maintain local caches of the files accessed. By default, WheelFS uses strict *Close-to-Open Consistency Semantics*[7], with the primary server being responsible for serializing operations.

Semantic cues can be used to specify application policies with respect to placement, durability, consistency and large reads. To reduce the effects of network latency, data can be placed close to clients that are likely to use the data. Files can be clustered together to optimize the performance of operations that access multiple files, and replication levels can be specified.

The system can be adjusted to wait for only a specified number of replicas to be created or updated before acknowledging a client's new file or file update request respectively. This helps in achieving quicker response times even in the presence of slow servers. Consistency related cues allow clients to specify time-out periods for remote communications corresponding to file system operations. Applications can also use the *Eventual Consistency Semantics*[8] to improve availability.

Also, a client can prefer to read stale copies of files when the primary servers are hard to reach. While reading large files, clients can choose to prefetch entire files into its local cache. Cues also enable clients to obtain file contents from multiple cached sources in parallel to reduce the load on the primary server.

A *Configuration Service*, maintained as a replicated state machine at multiple sites, is used by clients to learn about the servers responsible for the different objects. Based on the first S bits of the object identifier, the identifier space is split into 2^S slices. The configuration service maintains a mapping between slices and the primary and replica servers responsible for the slices.

While resource location aware data placement is supported, WheelFS does not provide resource characteristics aware data placement. The configuration service, maintained as a replicated state machine, can be a bottleneck for large system sizes and heavy query loads.

3.3. Self-Organizing P2P File Systems. In P2P systems, every node is both a supplier and consumer of resources. Some of the benefits of such an architecture are distribution of load among all the peers, increased robustness, and lack of a single point of failure. On the other hand, high system dynamics is one of its major

[7] When A opens a file after B has modified and closed it, A is guaranteed to see B's updates.
[8] If no new updates are made, the latest updates will propagate through the system eventually and make all the replicas consistent.

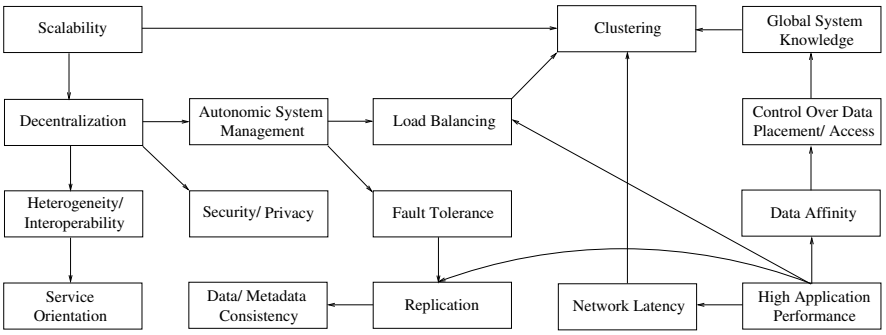


FIG. 3.7. *System Requirements of P2P File Systems ($A \rightarrow B$ represents B supports A)*

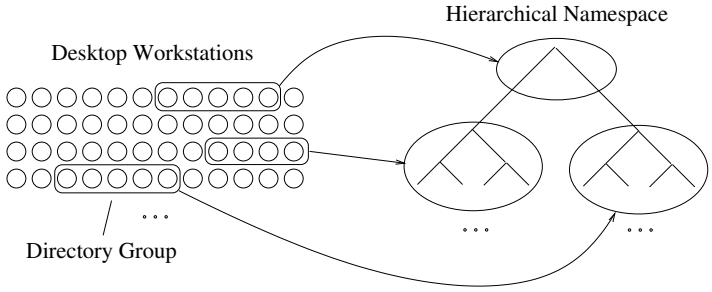


FIG. 3.8. *Farsite System Architecture*

drawbacks. In P2P file systems, peers share the load of file storage and metadata management. Figure 3.7 shows some of the system requirements of P2P file systems. As discussed earlier, scalability and high application performance are the two primary requirements under consideration.

It is well known that decentralization of control and autonomous system management are central to the design of scalable distributed systems. In such systems, load balancing and resource discovery are complex tasks because of the lack of any central entity with knowledge about the entire system.

However, awareness of resource characteristics and locations while placing file replicas is critical for achieving high application performance. That is because network bandwidth and latency concerns dictate that data and metadata be placed in proximity to where they are consumed. Achieving a trade-off between these conflicting requirements of decentralization and system awareness is an important design consideration, especially in the case of P2P file systems. One of the approaches to achieve the trade-off is to design the system as a federation of manageable clusters.

3.3.1. Farsite. Farsite (Federated, Available, and Reliable Storage for an Incompletely Trusted Environment) [6] [12] is a DFS from Microsoft Research built over a network of unstructured desktop workstations. Farsite provides high file availability and security utilizing the unused storage space and processing power of a large number of nodes. Issues of security and trust are addressed using *Public-Key Cryptographic Certificates* such as namespace, user and machine certificates. Users and directory groups authenticate each other before performing file system operations.

File contents are encrypted and replicated and the corresponding metadata are managed by *Byzantine-Replicated* finite state machines [33]. Farsite provides hierarchical directory namespaces, each namespace having its own root. Roots are maintained by a designated group of nodes. Directory groups can split to distribute metadata management load. Splitting can happen by randomly selecting a group of nodes and designating a portion of the namespace to them (figure 3.8).

Content hashes of files are stored in the corresponding directory groups to maintain file integrity. Different kinds of leases are issued on files to clients. Caching is used for improving access times and reducing network load. Updates made to files are not immediately propagated to all the replicas. Instead, a lazy propagation mechanism is employed in order to improve performance.

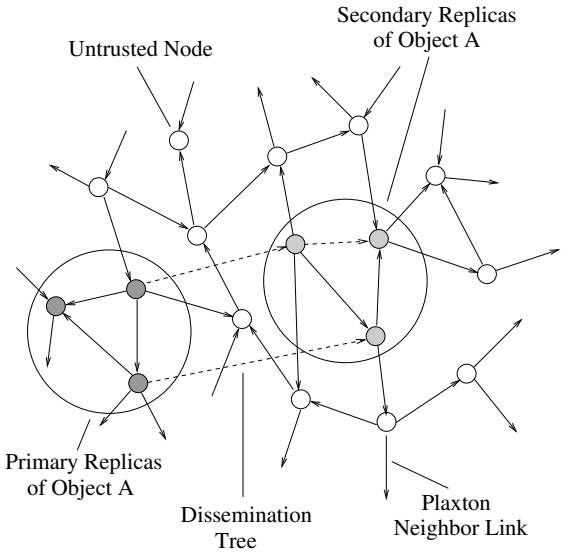


FIG. 3.9. *OceanStore System Architecture*

As with other hierarchy traversal systems, locating the directory group for a file deep in the hierarchy may require several hops, thus making metadata access expensive. In systems with high churn rates, group membership can keep changing, resulting in high group management overheads.

3.3.2. OceanStore. OceanStore [26] is a global scale data storage utility that uses untrusted infrastructure. The primary objective is to provide continuous access to persistent information.

Each object in OceanStore is assigned a unique global identifier and is replicated and stored in a set of servers. A few of the servers in the high connectivity and high bandwidth regions are made primary replicas and the rest are made secondary replicas (figure 3.9). Updates made to the objects are ordered by the primary replicas using a *Byzantine Fault Tolerant* algorithm [14]. Secondary replicas communicate with the primary replicas and among themselves to propagate updates in an epidemic manner. Every update results in the creation of a new version which is archived in the system, making the system inefficient for large sized files.

Each object is associated with a root node in the system which holds information about the object's replica locations. A variation of *Plaxton*'s randomized hierarchical distributed data structure [34] is used by nodes to reach the root of any object in $O(logN)$ hops, where N is the number of nodes in the system. A probabilistic algorithm using attenuated *Bloom Filters* [11] is also used to rapidly locate objects if they are in the local vicinity.

The policy of *Promiscuous Caching* which allows files to be replicated in any node in the system makes OceanStore highly scalable. However, the overheads involved in the maintenance of two tiers of nodes and a dissemination tree for each data object can be high. High churn rates among the primary tier nodes can also result in expensive maintenance overheads. Maintenance of Bloom filters and the Plaxton data structure at each node can result in high network usage.

3.3.3. Ivy. Ivy [31] is a P2P read/write file system based on logs. Each participant maintains a log with information about all the changes made to the files in the system by the participant. The logs of all the participants need to be parsed to be able to get the current state of a file. Updating a file's contents however requires an append to the participant's log only. Ivy uses *DHash* [1] as the *Distributed Hash Table* (DHT) [45] for storing all its logs and, as a result, all its data. The set of all logs in the file system is referred to as *View* (figure 3.10).

A participant's log is a linked list of log records. The log-head points to the most recent entry. Content hashes are used as keys for storing log records in DHash. The public key of a participant is the key for a log-head. The log-head is digitally signed by the participant's private key. The digital signatures and content hashes help ensure the integrity of logs in Ivy.

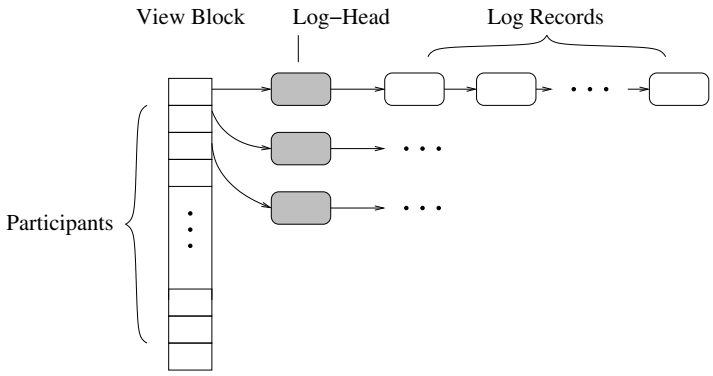


FIG. 3.10. *Ivy: File System View*

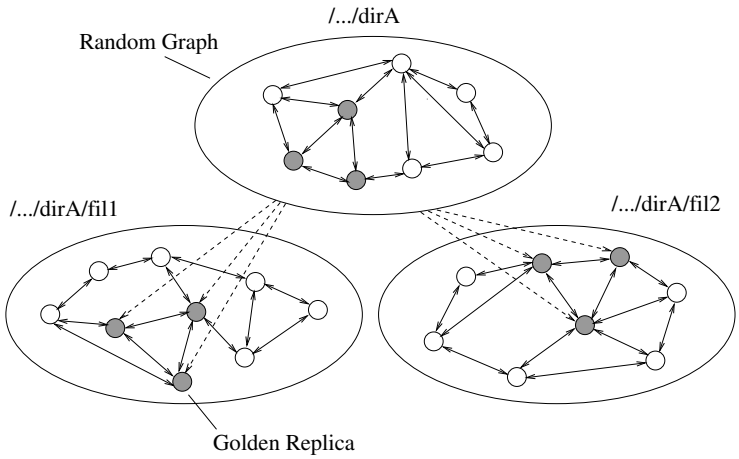


FIG. 3.11. *Pangaea System Architecture*

A private snapshot of the system is maintained by the participants in order not to have to scan all the logs for every read. Only the most recent log records need to be scanned. Since Ivy avoids using shared mutable data structures, locking is not necessary. Ivy logs contain version vectors and timestamps. These can help applications in detecting and resolving conflicts that may arise due to concurrent updates.

This strategy of maintaining per-participant logs makes Ivy suitable only for a small number of cooperating users. Moreover, high possibilities of conflicting concurrent updates result in Ivy providing weak consistency semantics.

3.3.4. Pangaea. The objective of Pangaea [38] is to build a planetary-scale P2P file system used by groups of collaborating users all over the world. The system attempts to achieve low access latency and high availability using *Pervasive Replication* techniques. Whenever and wherever a file is accessed, a replica is created. Popular files therefore get heavily replicated and personal files reside only on the nodes used by the owners.

A random graph of all the replicas is maintained for propagating updates and ensuring availability (figure 3.11). The random graph is created by making each replica maintain links to k other replicas chosen randomly. A few of the replicas are designated as *Golden* replicas. The golden replicas maintain links with each other and ensure that their set always maintains specified membership levels. Replicas perform random walks starting from one of the golden replicas to create random links. This way the graph stays connected.

Links to the golden replicas are recorded in the data object's parent directory (which is also maintained as a file). To access and replicate a file, its parent directory must be accessed and hence replicated. The recursive operation can proceed all the way to the file system's root.

By default, update propagation happens lazily. A strategy involving *Harbinger* messages is used to build a spanning tree which is used for quick update propagation. Strict consistency semantics are also supported by

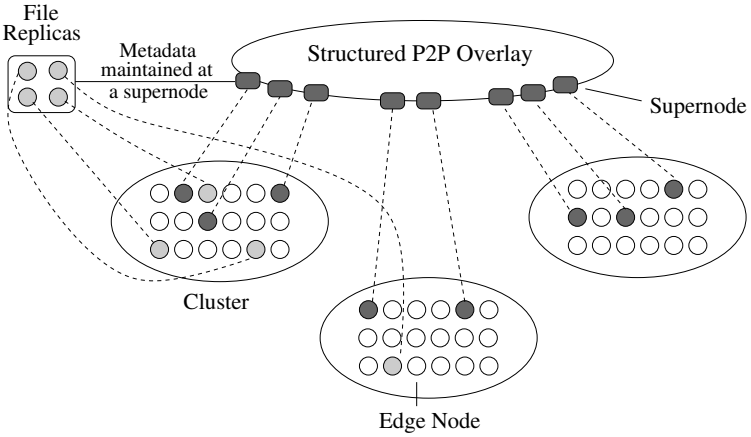


Fig. 3.12. *ENFS System Architecture*

making the updating client wait for acknowledgments from the replicas. A version vector based algorithm [37] is used for resolving conflicting updates.

3.3.5. Edge Node File System (ENFS). ENFS [25] exploits the resources of Internet edge nodes to provide scalable DFS services. Undedicated Internet edge nodes are enabled to function as both data and metadata servers. The presence of a large number of edge nodes results in scalable metadata access and high I/O throughputs.

ENFS uses proximity-based clustering of edge nodes (figure 3.12) for the efficient management of resources, balancing of load (storage, computational, query), and handling latency issues. A few reliable and capable edge nodes from each cluster are chosen to be the metadata servers (*Supernodes*) for that cluster. These supernodes are chosen based on capabilities such as network bandwidth, processor speed, storage space, and memory capacity. Each supernode is associated with a replica set consisting of a fixed number of other supernodes from the same cluster. The replica sets ensure high system availability.

Supernodes from all the clusters form a single system-wide structured P2P overlay network for use as a distributed hash table. By connecting up all the clusters in the system, the overlay enables nodes of a cluster to discover supernodes (of other clusters) which are responsible for specific portions of the file namespace. The structured overlay also helps in the efficient discovery of resources with specific characteristics in the entire system.

Since the sets of data and metadata servers change autonomously and dynamically to suit prevalent workloads, ENFS scales transparently. The architecture of the system allows data placement/access decisions to be based on applications' requirements of resource characteristics and locations. The metadata of each file has a single point of access (one of the cluster supernodes). This allows ENFS to support a large spectrum of access semantics.

4. Comparative Analysis. In this section, we analyze the above reviewed systems with respect to their scalability and the support they provide for high application performance only. We do not address other aspects of distributed file systems such as user/group management, security and trust, etc. In [30], the authors provide a survey of decentralized access control mechanisms in large scale distributed file systems. An overview of I/O systems (including file systems) dealing with massive data is presented in [22].

The manner in which the load on different file system servers vary with increasing numbers of users, and therefore user files, primarily determines the scalability of a distributed file system. Increase in the number of files results in an increase in the number of queries and in the amount of data I/O.

The system parameters used in the analysis are shown in table 4.1. For the sake of simplicity, we assume uniform server capabilities and that the file system metadata and data are equally distributed among the servers. We also assume that the metadata queries and I/O requests are generated in an independent and completely random manner.

We study the dependence of metadata and data server loads on the query and I/O rates in tables 4.2 and 4.3 respectively. The overheads of overlay network management also add to server loads, especially in the P2P file systems. The overheads are presented in table 4.4.

TABLE 4.1
System Parameters and Metrics

Parameter	Details
N	Number of nodes (servers/clients/peers) in the system
N_M	Number of metadata servers in the system
N_D	Number of storage nodes (data servers) in the system
F	Number of data items (files and directories) in the system
R	Average number of replicas per data item
Q	Number of metadata queries generated per unit of time in the system
D	Data transfer demand to and from the data servers in the system per unit of time
l_C	Network latency between nodes within a cluster/LAN (Intranet)
l_W	Network latency between nodes in different clusters (Internet)
$P(n)$	Cost of achieving consensus (Paxos [27], Byzantine fault tolerant algorithm, quorum-based voting) among n nodes in terms of time and number of messages
L_{MS}	Average query handling load on a metadata server
L_{DS}	Average I/O load on a data server
L_{OM}	Message, time and space overheads of maintaining the different overlays

In GoogleFS, Lustre, Panasas, PVFS2, Ceph, OceanStore and ENFS, support for file striping and parallel I/O helps in distributing data server load at a finer granularity. From table 4.3, we can see that, L_{DS}, the data server load, can be represented as $f(D/N_D)$ for category I and category II file systems and as $f(D/N)$ for category III file systems.

The components that get overloaded in the first category of file systems are clearly the servers. In these systems, the N_M metadata servers are usually the data servers also. The load on each server therefore is $L_{MS} + L_{DS}$. Both increasing query rates and I/O demands affect the same set of servers.

In the second category of file systems, decoupling of data and metadata helps in splitting the load among different sets of servers (L_{MS} for metadata servers and L_{DS} for data servers). However, due to rigid server configurations which require manual administration, the values of N_M and N_D are more or less fixed. This results in these systems supporting only constrained levels of metadata and I/O demands. Additionally, in WheelFS, the configuration service can potentially become a bottleneck with increasing query rates.

Since Farsite, OceanStore, Ivy, Pangaea and ENFS are P2P file systems (category III), the load on each node is $L_{MS} + L_{DS} + L_{OM}$. The number of nodes, N, is however virtually unlimited. Therefore, the loads are well distributed.

However, Ivy is a log-based file system and so performance falls significantly with increasing numbers of participants. Network usage is excessively high in OceanStore and Pangaea due to overlay management messages, pervasive replication and update propagations. Since a considerable number of peers in a wide area installation may possess low bandwidth connections, system performance can be affected by increasing load levels in these two systems.

The performance of applications executing over file systems depends mainly on the speed of metadata access and data I/O throughput. Metadata query and update times experienced by applications depend on several factors such as metadata server load, query routing mechanism, network latency, and consistency management strategy. Table 4.5 analyzes these factors in the various systems.

Data I/O throughput depends on server load and network latency/bandwidth. Server loads are discussed in table 4.3. The support provided by the file systems to reduce the effects of network latency and bandwidth on data transfer/processing speed, and hence on application performance, is discussed in table 4.6.

TABLE 4.2
Metadata Server Load as a Function of Query Rate

System	**Load/Server (L_{MS})**	Comments
AFS	$f(Q/N_M)$	The load is distributed among the N_M servers. Since the number of servers is fixed and can be extended only through administrator intervention, server load keeps increasing with Q.
CIFS	$f(Q/N_M)$	The load is distributed among the N_M servers that are sharing content. Typically, the number of servers in CIFS installations are much larger than in AFS installations. Query loads are therefore better distributed.
GoogleFS	$f(Q)$	The master server handles all the queries. As a result, such an architecture's scalability is limited.
Lustre/ Panasas/ PVFS2	$f(Q/N_M)$	The query load is distributed among the N_M metadata servers. Since the number of MDSs is fixed and can be extended only by manual intervention, load on an MDS keeps increasing with Q.
Ceph	$f(\alpha \cdot Q/N_M)$	The metadata query load is distributed among the servers in the MDS cluster. The dynamic subtree partitioning scheme employed by Ceph distributes the query load among the servers uniformly. Moreover, since clients can calculate object locations themselves, metadata server loads are significantly reduced (represented by α).
WheelFS	$f(Q/N_M)$	The query load is distributed among the N_M WheelFS primary servers.
	$f(Q)$	Clients get information about the primary servers responsible for files from the configuration service. The load on the configuration service therefore increases along with Q.
Farsite	$f(Q/(\kappa \cdot N))$	When query rates increase, directory groups split and distribute the load among more nodes. Since any peer can be a part of a directory group, query loads are shared by a significant fraction (κ) of all the nodes in the system.
OceanStore	$f(Q/N)$	Information about files in OceanStore are obtained using pure P2P algorithms. The metadata query load is therefore distributed among all the peers.
Ivy	$f(Q/N)$	Metadata queries result in getting the recent log records of all participants and scanning the records locally at the querying peer. Thus, the query load is distributed among all the peers.
Pangaea	$f(Q/N)$	Metadata accesses happen using P2P routing protocols and result in replicas getting created at the querying peers. Thus the query load is shared by all the peers.
ENFS	$f(Q/(\kappa \cdot N))$	The number of supernodes increases with increasing query loads (Q). Since any node in the system can be made a supernode, the load is shared by a significant fraction (κ) of N, as in Farsite.

TABLE 4.3
Data Server Load as a Function of the I/O Demand

System	Comments
AFS	Callback promises and invalidations, and whole file caching help in reducing the load on the AFS servers. This is one of the main reasons for AFS scaling better than NFS.
CIFS	Stateful servers, elaborate locking mechanisms, caching, and read-aheads, help in reducing the load on the servers. A large number of servers sharing files helps distribute the load better than in AFS.
GoogleFS	The data load is distributed among the N_D chunkservers in the GoogleFS cluster. GoogleFS does not support client side caching, especially because the applications usually require computations to be performed at the chunkservers itself.
Lustre	The load is shared among the N_D object storage servers. Server based distributed file locking protocols and client side caching in Lustre help reduce data server loads.
Panasas	The data serving load is shared among the N_D OSDs. File locking services and consistent client caching is supported in Panasas.
PVFS2	PVFS2 does not cache data on the clients and so the entire load is distributed among the N_D I/O servers.
Ceph	Client side caching absorbs some load off the N_D OSDs.
WheelFS	All clients maintain caches of files read. Semantic cues help in satisfying a client's data needs with nearby caches as much as possible. Such *Cooperative Caching* mechanisms help in reducing the loads on WheelFS servers significantly.
Farsite	All the nodes in the system are capable of storing data. As data loads increase, more replicas can be created among the peers. Thus, data transfer loads are shared by a large number of nodes ($O(N)$).
OceanStore	Promiscuous caching and P2P data location algorithms enable data serving loads to be distributed among the peers in the system.
Ivy	All the data objects in Ivy are stored in the DHash DHT, which comprises of all the nodes in the system. Thus data transfer load is shared by the entire set of nodes.
Pangaea	Pervasive caching results in files and directories getting replicated in a large number of peers in the system. I/O load is therefore distributed widely.
ENFS	Supernodes ensure that file contents in ENFS are distributed uniformly across all the storage nodes in the system. Data transfer loads are therefore shared by a large number of nodes ($O(N)$).

Apart from data server loads, application performance largely depends on the network distance between servers and clients. In most file systems of category I and II, server locations are fixed and so in wide area installations, data access usually happens across long distances. Data caching helps in reducing the distance to some extent, especially in AFS and WheelFS.

File systems belonging to category III, however, do not have fixed servers. The peer-to-peer nature of these systems support the creation of new file replicas closer to their users. ENFS goes a step further and pro-actively creates file replicas on nodes which are likely to process the contents, based on user specification or application type.

4.1. Observations. In summary, our analysis of these systems has led to the following observations:

- *Decentralization* Most of the production file systems today use central servers (or clusters of servers). While such an infrastructure can support a large number of users and files, their scalability is limited. Since the digital data generation capabilities of the masses has increased tremendously, the next few years are expected to witness huge rates of data creation. Decentralization is therefore essential to manage the accompanying data management demands. Decentralization also has other benefits such as not having to completely trust one central entity, lack of a single point of failure, robustness, and lack of the need for expensive servers.

TABLE 4.4
Overlay Maintenance Overheads

System	Overhead (L_{OM})	Comments
WheelFS	$f(C_{RSM})$	The configuration service is implemented as a replicated state machine with a certain number of nodes. Maintaining the state machine involves operations such as handling membership changes, and electing a new leader. C_{RSM} represents the corresponding message and time overheads for the configuration service nodes.
Farsite	$f(C_{BFT})$	All the nodes in Farsite which are part of a directory group incur the overheads of maintaining a Byzantine fault tolerant group. The overhead associated with Byzantine fault tolerance is represented by C_{BFT}.
OceanStore	$f(logN, C_{BF})$	Every node in OceanStore maintains a routing table associated with the Plaxton scheme for global data location. The size of the table is $O(logN)$. Moreover, changing object contents in a node and its local vicinity, results in changes to its attenuated Bloom filter. The network and computational (multiple hashing) overheads of maintaining the filters is also significant and is represented by C_{BF}.
Ivy	$f(logN)$	Nodes in Ivy are part of the DHash DHT and so maintain routing tables with $O(logN)$ entries.
Pangaea	$f((F \cdot R \cdot k)/N)$	Every replica of a data item must maintain at least k links to other replicas. This results in significant message, time and space overheads.
ENFS	$f(logN_M)$	Supernodes from all the clusters form a structured overlay in ENFS. Each supernode maintains a routing table of size $O(logN_M)$.

- *Autonomic System Management* Since decentralized systems usually exploit the resources of unreliable nodes, mechanisms must be in place to provide notions of reliability and availability to the users/applications. It is impractical for large distributed systems to be manually administered. Essential tasks such as handling node failures, and load balancing must be autonomically managed for better resource utilization and application performance.
- *Pervasive Replication* High levels of replication, especially of read-only files, increases availability and brings data closer to the users, thereby improving application performance. Replication has the added benefit of enabling parallel access to files. Parallel access enables computations on different parts of a file to be performed simultaneously. In a well designed system, the benefits of replication must over-weigh the overheads of additional data transfer and consistency management.
- *Flexible Consistency Semantics* Often, the stronger the consistency semantics supported by a system, the poorer the application performance. The consistency requirements of different applications vary widely. Thus, file systems must be capable of flexing their consistency semantics in accordance to application requirements. This way, users/applications can themselves adjust the required levels of consistency/performance trade-off.
- *Data Affinity* Data affinity refers to the concept of ensuring that files are stored close to the nodes which are most suited and likely to process their contents. For example, in HPC applications, due to large data set sizes, schedulers attempt to schedule computations on resources which contain the required data [36] [47], thus reducing the amount of data movement. Therefore, file systems which support resource characteristics aware data placement are highly useful. Data migration with changing access patterns is also beneficial.
- *Proximity-based Node Clustering* A large system which cannot be managed by a central controller is best managed by being partitioned into proximity-based node clusters of manageable sizes. In dis-

TABLE 4.5
Factors affecting Metadata Query Response Times

System	Comments
AFS	$f(L_{MS} + L_{DS}, l_C(or)l_W)$ Servers are usually distributed across wide areas. Servers in every cell possess information about the servers hosting different data volumes across the entire system. Therefore, there are no query routing overheads. The effect of network latency depends on whether queries are made for files served locally or by a server in a different cell. Data volumes are placed close to users/groups owning the corresponding data items and so latency effects are generally low.
CIFS	$f(L_{MS} + L_{DS}, l_C(or)l_W)$ CIFS servers are usually distributed across wide areas. Clients either possess information about servers hosting different data items or can use browsing protocols to search for servers. When a client queries a distant CIFS server, high network latency is likely to affect the response time.
GoogleFS	$f(L_{MS}, l_C, P(2))$ Since GoogleFS installations are usually cluster based, network latency is l_C. All metadata queries are handled by the master server. Metadata updates must be serialized in the master server and its shadow.
Lustre	$f(L_{MS}, l_W, P(2))$ The set of metadata servers are clustered in a single location and so most client queries have to travel across the network in a wide area installation. Metadata updates must be serialized in the active and passive metadata servers associated with a data item.
Panasas	$f(L_{MS}, l_W, P(N_M))$ Panasas uses a quorum-based voting protocol to commit metadata operations in its metadata servers. As in Lustre, network latency is usually l_W since the servers are clustered in one location.
PVFS2	$f(L_{MS}, l_W)$ PVFS2 avoids serialization of independent metadata operations using an explicit state machine, threads (to provide non-blocking access), and a component that monitors completion of operations across devices. Avoiding serialization makes metadata access faster.
Ceph	$f(L_{MS}, l_W, P(k))$ Since the metadata servers are clustered, far-off clients experience high network latencies. Metadata updates must be synchronously journaled to a cluster (of size k) of OSDs for safety.
WheelFS	$f(L_{MS}, l_W)$ Accessing the configuration service to determine the primary may involve a query to a far-off node. Clients can specify location preferences for the primary servers for their files and directories based on expected access patterns and so latency overheads of accessing the primary servers are optimized.
Farsite	$f(L_{MS} + L_{OM}, d \cdot l_W, P(k))$ Metadata access may require traversal from the root to the directory of interest. Each directory may be managed by a different group. d represents the average number of hops between directory groups required to reach a data item. Metadata updates require Byzantine fault tolerant agreement among the k directory group members.
OceanStore	$f(L_{MS} + L_{DS} + L_{OM}, l_W \cdot logN, C_{ARC})$ Locating the root of an object in OceanStore can require $O(logN)$ hops across a wide area network. Some files, especially popular ones, can however be located in the local vicinity of the client. Every update (or group of updates) involves storing the object in an archival form. C_{ARC} represents the corresponding costs of encoding the file using erasure coding and distributing it across hundreds of machines.
Ivy	$f(L_{MS} + L_{OM}, p \cdot (logN) \cdot l_W)$ Accessing the metadata requires the gathering of the most recent log records of all the participants (p). Metadata updates are performed in the local log alone.
Pangaea	$f(L_{MS} + L_{OM}, l_C, C_{ST})$ The pervasive replication strategy results in most data items being available in close proximity. Propagation of updates happens in two phases along the spanning tree for that data item rooted at the source. The corresponding message and time costs are represented by C_{ST}.
ENFS	$f(L_{MS} + L_{OM}, l_C(or)l_W, P(k))$ Metadata of user files are managed by supernodes in the same cluster as that of the user. However, accessing the metadata of files in other clusters requires across network querying. Metadata servers responsible for individual files/directories are identified using index files stored in the system wide DHT and actively cached in the local cluster's supernodes. Discovery can therefore usually happen within a couple of hops. Metadata updates are serialized in the responsible supernode and its replica set. k represents the supernode replica set size.

TABLE 4.6
Support for Application Performance

System	Support
AFS/ CIFS	Servers in these systems only perform file I/O. Any other operation to be performed on the data must be performed at the client site. Client side caching is supported to varying degrees. AFS, especially, improves application performance using whole file caching. However, the benefits of caching come at the expense of consistency management. AFS provides weak consistency semantics. CIFS uses elaborate locking mechanisms to provide strong consistency semantics. I/O throughputs are largely dependent on client-server network distance.
GoogleFS	GoogleFS is optimized for the MapReduce class of applications. GoogleFS's support for appending records to existing datasets in a quick, atomic and race-free manner is critical for MapReduce applications. GoogleFS stores replicas of data chunks on different machines. This increases the chances of MapReduce scheduling mappers on nodes with the data or on nodes close to the data. GoogleFS supports relaxed consistency semantics, which helps speed up data appends.
Lustre/ Panasas/ Ceph	Since object-based storage devices support the storage and serving of objects directly at the hardware level, better I/O throughputs can be achieved compared to normal disc I/O. Application specific processing/computations however cannot be performed at the servers. These systems provide strong consistency semantics. I/O throughputs are largely dependent on client-server network distance.
PVFS2	Client side caching is not supported. Client server distance can therefore be detrimental to application performance. PVFS2 implements *Non-Conflicting Write* semantics, thus allowing clients to update non-conflicting portions of the namespace simultaneously without locks.
WheelFS	Placement semantic cues such as *.Site*, *.KeepTogether* and *.RepSites* allow owners to place their data close to the users most likely to use the data. This helps optimize data throughputs. Cues can also be used to fetch file contents from the cache of other clients in parallel.
Farsite	Farsite does not attempt to reduce latency. It is designed to support typical user home directory I/O instead of the high performance I/O of scientific applications. Byzantine fault tolerant agreement protocols and leases help in providing strong consistency guarantees in Farsite.
OceanStore	Users choose primary and secondary tier storage nodes on which to store their files. Moreover, popular files get widely cached. These measures help in improving data throughputs. Based on application requirements, OceanStore can provide a variety of consistency semantics.
Ivy	Nodes maintain a private snapshot of all the logs and so file reads only require the most recent records to be obtained from the DHash DHT. Ivy provides weak consistency semantics with application assisted conflict resolutions.
Pangaea	In Pangaea, replica locations are determined by user activities. Files can therefore usually be located close to the clients. By default, Pangaea implements weak consistency semantics. However, stronger guarantees can be provided by trading off performance.
ENFS	ENFS focuses on the principle that awareness of the capabilities of storage nodes is critical for a file system to be useful for applications. Cluster supernodes can inexpensively discover resources with specific characteristics across the entire system. File/Replica placement decisions are based on the requirements of the applications expected to operate on the files. This helps applications achieve high performance. Home-based consistency protocols allow a wide variety of access semantics to be supported.

tributed systems, clustering supports the scalable and efficient discovery of data and resources with specific characteristics from the entire system [35]. Clustering also provides for efficient communication mechanisms among proximal and far-off nodes in the system. Co-location of servers and their associated clients, which helps in optimizing network latency, also becomes simpler when the system is partitioned into clusters. A lot of work, done on network distance measurement [15], topology discovery [20] [9] and proximity-based node clustering [51] [29] [35], can be used for autonomous cluster formation and management.

- *Capability-based Role Assignment* Farsite and ENFS are examples of P2P file systems in which peers are assigned different roles based on their current capability levels (CPU load, memory, network). Nodes with relatively high levels of capability are made responsible for file metadata services. This helps in reducing the effects of system dynamics on file availability and access. OceanStore and Pangaea do not perform capability-based role assignment. These systems therefore use up a lot of network bandwidth and space in maintaining per-file overlays.

5. Conclusions. This survey analyzes popular wide area distributed file systems for their scalability and the support they provide for high application performance. Several design decisions affect the way file systems scale and applications perform.

We categorize the systems as *Traditional Distributed File Systems*, *Asymmetric Cluster File Systems* and *Self-Organizing P2P File Systems*, based on the extent of data/metadata distribution across the system.

We perform scalability analysis by characterizing the loads on file system servers as functions of query rates and data I/O demands. Application performance is studied by characterizing query response times as functions of the appropriate system parameters. Data I/O throughputs and support for data affinity are also analyzed. The summarized observations are presented in section 4.1.

It is not possible to design a wide area distributed file system that performs ideally for all kinds of applications. Often, providing support for one feature affects another negatively. For wider acceptance, distributed file systems must allow client applications to conveniently control the different trade-offs amongst file system features.

REFERENCES

[1] *The Chord/DHash project. An implementation of the Chord DHT in C++.* `http://pdos.csail.mit.edu/chord/`.

[2] *Comparison of file systems.* `http://en.wikipedia.org/wiki/Comparison_of_file_systems`.

[3] *List of file systems.* `http://en.wikipedia.org/wiki/List_of_file_systems#Distributed_file_systems`.

[4] *Parallel Virtual File System, version 2.* `http://www.pvfs.org/`.

[5] *Framework for loosely coupled wide area file system access.* `http://www.diceprogram.org/reports/docs/FrameworkforLooselyCoupledWideAreaFileSystemAccess.pdf`, 2008.

[6] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, *Farsite: Federated, available, and reliable storage for an incompletely trusted environment*, SIGOPS Operating Systems Review, 36 (2002), pp. 1–14.

[7] T. Anderson and A. Vahdat, *GENI distributed services.* `http://www.geni.net/GDD/GDD-06-24.pdf`, 2006.

[8] P. Andrews, P. Kovatch, and C. Jordan, *Massive high-performance global file systems for Grid computing*, in SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Washington, DC, USA, 2005, IEEE Computer Society, p. 53.

[9] Y. Bejerano, Y. Breitbart, M. N. Garofalakis, and R. Rastogi, *Physical topology discovery for large multisubnet networks*, in Proceedings IEEE INFOCOM âĂŹ03, 2003, pp. 342–352.

[10] M. Blondel, *WikipediaFS.* `http://wikipediafs.sourceforge.net/`, 2007.

[11] B. H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, 13 (1970), pp. 422–426.

[12] W. J. Bolosky, J. R. Douceur, and J. Howell, *The Farsite project: a retrospective*, SIGOPS Operating Systems Review, 41 (2007), pp. 17–26.

[13] D. Borthakur, *The Hadoop Distributed File System: Architecture and design.* `http://hadoop.apache.org/core/docs/current/hdfs_design.html`, 2008.

[14] M. Castro and B. Liskov, *Practical Byzantine Fault Tolerance*, in Operating Systems Design and Implementation, Feb. 1999, pp. 173–186.

[15] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, *Vivaldi: A decentralized network coordinate system*, in Proceedings of the ACM SIGCOMM'04 Conference, 2004, pp. 15–26.

[16] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, Communications of the ACM, 51 (2008), pp. 107–113.

[17] A. Devulapalli, I. T. Murugandi, D. Xu, and P. Wyckoff, *Design of an intelligent object-based storage device.* `http://www.osc.edu/research/network_file/projects/object/papers/istor-tr.pdf`.

[18] S. Fadden, *An introduction to GPFS Version 3.3*, IBM White Paper, (2010).

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google File System*, SIGOPS Operating Systems Review, 37 (2003), pp. 29–43.
[20] R. Govindan and H. Tangmunarunkit, *Heuristics for Internet map discovery*, in Proceedings IEEE INFOCOM âĂŹ00, 2000, pp. 1371–1380.
[21] J. H. Howard, *An overview of the Andrew File System*, in USENIX Winter, 1988, pp. 23–26.
[22] R. Hubovsky and F. Kunz, *Dealing with massive data: From parallel I/O to Grid I/O*, Master's thesis, University of Vienna, 2004.
[23] T. Kosar and M. Livny, *Stork: Making data placement a first class citizen in the Grid*, in Proceedings of the International Conference on Distributed Computing Systems '04, 2004, pp. 342–349.
[24] P. Kovendhan and D. Janakiram, *Arogyashree: A distributed file system for large scale Internet-based telemedicine*, in Proceedings of MobiHealthInf '09, The First International Workshop on Mobilizing Health Information to Support Healthcare-related Knowledge Work, 2009, pp. 105–114.
[25] P. Kovendhan and D. Janakiram, *The Edge Node File System: A distributed file system for high performance computing*, Scalable Computing: Practice and Experience, 10 (2009), pp. 115–130.
[26] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, *Oceanstore: An architecture for global-scale persistent storage*, SIGARCH Computer Architecture News, 28 (2000), pp. 190–201.
[27] L. Lamport, *The part-time parliament*, Transactions on Computer Systems, 16 (1998), pp. 133–169.
[28] P. Leach and D. Naik, *A Common Internet File System Protocol (CIFS)*. Internet draft, Internet Engineering Task Force (IETF), 1997.
[29] E. K. Lua, J. Crowcroft, and M. Pias, *Highways: Proximity clustering for scalable Peer-to-Peer networks*, in Proceedings 4th International Conference on P2P Computing, 2004, pp. 266–267.
[30] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis, *Decentralized access control in distributed file systems*, ACM Computing Surveys, 40 (2008), pp. 1–30.
[31] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, *Ivy: A read/write Peer-to-Peer file system*, SIGOPS Operating Systems Review, 36 (2002), pp. 31–44.
[32] D. A. Patterson, G. A. Gibson, and R. H. Katz, *A case for Redundant Arrays of Inexpensive Disks (RAID)*, in SIGMOD Conference, 1988, pp. 109–116.
[33] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, Journal of the ACM, 27 (1980), pp. 228–234.
[34] G. Plaxton, R. Rajaraman, and A. W. Richa, *Accessing nearby copies of replicated objects in a distributed environment*, in SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, ACM, 1997, pp. 311–320.
[35] L. Ramaswamy, B. Gedik, and L. Liu, *A distributed approach to node clustering in decentralized Peer-to-Peer networks*, IEEE Transactions on Parallel and Distributed Systems, 16 (2005), pp. 814–829.
[36] K. Ranganathan and I. Foster, *Simulation studies of computation and data scheduling algorithms for data Grids*, Journal of Grid Computing, 1 (2003), pp. 53–62.
[37] D. H. Ratner, *Roam: A Scalable Replication System for Mobile and Distributed Computing*, PhD thesis, University of California Los Angeles, 1998.
[38] Y. Saito and C. Karamanolis, *Pangaea: a symbiotic wide-area file system*, in EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop, New York, NY, USA, 2002, ACM, pp. 231–234.
[39] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, *Design and implementation of the Sun Network Filesystem*, in Proceedings Summer 1985 USENIX Conference, 1985, pp. 119–130.
[40] P. Schwan, *Lustre: Building a file system for 1000-node clusters*, in Proceedings of Linux Symposium, 2003, pp. 380–386.
[41] Z. Sebepou, K. Magoutis, M. Marazakis, and A. Bilas, *A comparative experimental study of parallel file systems for large-scale data processing*, in LASCO'08: First USENIX Workshop on Large-Scale Computing, Berkeley, CA, USA, 2008, USENIX Association, pp. 1–10.
[42] S. C. Simms, G. G. Pike, and D. Balog, *Wide area filesystem performance using Lustre on the TeraGrid*, in in Proceedings of the TeraGrid 2007 Conference, 2007.
[43] B. P. Spencer, D. Noveck, D. Robinson, and R. Thurlow, *The NFS version 4 protocol*, in Proceedings of International System Administration and Networking (SANE) Conference '00, 2000, pp. 94–113.
[44] J. Steiner and J. I. Schiller, *An authentication service for open network systems*, in USENIX Conference Proceedings, 1988, pp. 191–202.
[45] I. Stoica, R. Morris, D. Karger, F. M. Kaashoek, and H. Balakrishnan, *Chord: A scalable Peer-to-Peer lookup service for Internet applications*, in Proceedings of ACM SIGCOMM '01, vol. 31, October 2001, pp. 149–160.
[46] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, *Flexible, wide-area storage for distributed systems with WheelFS*, in Networked Systems Design and Implementation, Boston, MA, April 2009, pp. 43–58.
[47] A. Takefusa, O. Tatebe, S. Matsuoka, and Y. Morita, *Performance analysis of scheduling and replication algorithms on Grid Datafarm architecture for high-energy physics applications*, in Proceedings of 12th IEEE High Performance Distributed Computing, 2003, pp. 34–43.
[48] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, *Ceph: A scalable, high-performance distributed file system*, in Operating Systems Design and Implementation, 2006, pp. 307–320.
[49] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, *CRUSH: Controlled, scalable, decentralized placement of replicated data*, in Proceedings of Supercomputing '06, 2006, pp. 31–31.
[50] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, *Scalable performance of the Panasas parallel file system*, in Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2008, pp. 1–17.

[51] Q. Xu and J. Subhlok, *Automatic clustering of Grid nodes*, in Proceedings of 6th IEEE/ACM International Workshop on Grid Computing, IEEE Computer Society, 2005, pp. 227–233.

Edited by: Thomas Ludwig
Received: June 6-th, 2010
Accepted: September 22-nd, 2010

Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 327–327. http://www.scpe.org
ISSN 1895-1767

BOOK REVIEWS

EDITED BY JIE CHENG

Programming Massively Parallel Processors. A Hands-on Approach
David Kirk and Wen-mei Hwu
ISBN: 978-0-12-381472-2

Introduction. This book is designed for graduate/undergraduate students and practitioners from any science and engineering discipline who use computational power to further their field of research. This comprehensive test/reference provides a foundation for the understanding and implementation of parallel programming skills which are needed to achieve breakthrough results by developing parallel applications that perform well on certain classes of Graphic Processor Units (GPUs). The book guides the reader to experience programming by using an extension to C language, in CUDA which is a parallel programming environment supported on NVIDIA GPUs, and emulated on less parallel CPUs. Given the fact that parallel programming on any High Performance Computer is complex and requires knowledge about the underlying hardware in order to write an efficient program, it becomes an advantage of this book over others to be specific toward a particular hardware. The book takes the readers through a series of techniques for writing and optimizing parallel programming for several real-world applications. Such experience opens the door for the reader to learn parallel programming in depth.

Outline of the Book. Kirk and Hwu effectively organize and link a wide spectrum of parallel programming concepts by focusing on the practical applications in contrast to most general parallel programming texts that are mostly conceptual and theoretical. The authors are both affiliated with NVIDIA; Kirk is an NVIDIA Fellow and Hwu is principle investigator for the first NVIDIA CUDA Center of Excellence at the University of Illinois at Urbana-Champaign. Their coverage in the book can be divided into four sections. The first part (Chapters 1–3) starts by defining GPUs and their modern architectures and later providing a history of Graphics Pipelines and GPU computing. It also covers data parallelism, the basics of CUDA memory/threading models, the CUDA extensions to the C language, and the basic programming/debugging tools. The second part (Chapters 4–7) enhances student programming skills by explaining the CUDA memory model and its types, strategies for reducing global memory traffic, the CUDA threading model and granularity which include thread scheduling and basic latency hiding techniques, GPU hardware performance features, techniques to hide latency in memory accesses, floating point arithmetic, modern computer system architecture, and the common data-parallel programming patterns needed to develop a high-performance parallel application. The third part (Chapters 8–11) provides a broad range of parallel execution models and parallel programming principles, in addition to a brief introduction to OpenCL. They also include a wide range of application case studies, such as advanced MRI reconstruction, molecular visualization and analysis. The last chapter (Chapter 12) discusses the great potential for future architectures of GPUs. It provides commentary on the evolution of memory architecture, Kernel Execution Control Evolution, and programming environments.

Summary. In general, this book is well-written and well-organized. A lot of difficult concepts related to parallel computing areas are easily explained, from which beginners or even advanced parallel programmers will benefit greatly. It provides a good starting point for beginning parallel programmers who can access a Tesla GPU. The book targets specific hardware and evaluates performance based on this specific hardware. As mentioned in this book, approximately 200 million CUDA-capable GPUs have been actively in use. Therefore, the chances are that a lot of beginning parallel programmers can have access to Telsa GPU. Also, this book gives clear descriptions of Tesla GPU architecture, which lays a solid foundation for both beginning parallel programmers and experienced parallel programmers. The book can also serve as a good reference book for advanced parallel computing courses.

Jie Cheng,
University of Hawaii Hilo

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_ε 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.