Scalable Computing: Practice and Experience

Scientific International Journal for Parallel and Distributed Computing

ISSN: 1895-1767

Universitatea de Vest
din Timişoara

Volume 17(2)

June 2016

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

Scalable Computing: Practice and Experience

Volume 17, Number 2, June 2016

TABLE OF CONTENTS

Special Issue on Distributed Computing with Applications in Bioengineering:

DOI 10.12694/scpe.v17i2.1155
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. iii–iv. http://www.scpe.org © 2016 SCPE

INTRODUCTION TO THE SPECIAL ISSUE ON DISTRIBUTED COMPUTING WITH APPLICATIONS IN BIOENGINEERING

Biomedical Engineering, usually known as Bioengineering, is among the fastest developing and one of most important interdisciplinary fields today. It connects natural and technical sciences, for all of which biological and medical phenomena, computation, and data management play important roles in science and industry. Distributed computing and parallel algorithms have proved to be effective in solving the problem with high computational complexity in a wide range of domains, including areas of computational bioengineering. This special edition collects high-quality research papers in the field of application of distributed computing systems in bioengineering applications. This issue is the achievement of the developments of modern computer and biomedical technology. Most bioengineering or bioinformatics methods need to access and analyses large amounts of data. For that it is necessary to achieve an effective way that from Big data creates knowledge and useful applications. The nature of such scientific data/information/knowledge demands the use of a powerful computing and intelligent database management systems. Areas of current and future research include advanced topics in this special issue.

The paper from Juhasz et al. present a novel, GPU-based streaming architecture that has the potential for drastically reducing execution times and at the same time providing simultaneous 2D and 3D visualization facilities. The system uses a highly-optimised and re-configurable pipeline of CPU and GPU cores that attempts to exploit the tremendous computing power whenever possible. The system can process live data arriving from an EEG device or data stored in EEG data files. The computer drives a large display wall system of four 46 monitors that provides a 4K-resolution drawing surface for visualizing raw EEG data, potential maps and various 3D views of the patient's head.

Authors from the group of M. Gusev presents the scalable balancer in Cloud computing extended to Dew Computing level. Given a successful implementation of scalable low level load balancer, implemented on the network layer. The scalability is proved with series of experiments. The experiments showed that it adds small latency of several milliseconds and thus it slightly reduces the performance when the distributed system is underutilized.

Rybicki et al. demonstrate how to realize a Software-as-a-Service solution for a variety of science software using container technologies. The presented solution utilized DARIAH-DE research infrastructure based on OpenStack and UNICORE grid to deliver an extensible solution for the digital humanities domain.

The paper from authors Forer et al. describe the extension of Cloudflow to support ApacheSpark without any adaptions to already implemented pipelines. The described performance evaluation demonstrates that Spark can bring an additional boost for analyzing next generation sequencing(NGS) data to the eld of genetics. The Cloudflow framework is open source and freely available at https://github.com/genepi/cloudflow .

The paper by Memon et al. describes an approach for executing computational jobs across HPC and HTC resources while operating on geographically dispersed data via a global federated file system. The solution is realized as a new framework that integrates UNICORE and GFFS to provide a standards-based environment to support large scale data intensive computations frequent in today's biomedical analyses.

The paper "A Parallel algorithm for the state space exploration", authored by L. Allal et al., proposes a new automatic verification technique based on model checking that determines whether a given system satisfies its specification. Such a technique suffers from the state explosion problem when traversing all possible states of systems. A new synchronized parallel algorithm (SPA) of exploration is proposed based on a fixed number of threads. Exhaustive comparative studies between the standard parallel exploration algorithm in SPIN and the new SPA show that the SPA performs slightly better regarding the execution time and memory complexity.

Smart systems in telemedicine frequently use intelligent sensor devices at large scale. Practitioners can monitor non-stop the vital parameters of hundreds of patients in real-time. The most important pillars of remote patient monitoring services are communication and data processing. Large scale data processing is done mainly using workflows. In the paper Eszter et al. give a brief overview of the different check pointing techniques and propose two new provenance based check pointing algorithms which uses the information stored in the workflow structure to dynamically change the frequency of check pointing and can be efficiently used for dynamic health care smart systems.

Papers in this Special edition are primarily in reduced form are presented on the Distributed Systems, Visualization and Biomedical Engineering Conference within MIPRO 2015 (www.mipro.hr), which was held in Opatija, Croatia. We want to thank the collaboration SCPE Editor prof. Dana Petcu on all-round assistance and all the reviewers who have contributed to improved this Special edition.

Prof. dr. Karolj Skala, Ruer Boşković Institute, Zagreb, Croatia
Prof. dr. Roman Trobec, Jozef Stefan Institute, Ljubljana, Slovenia
Dr. Enis Afgan, Johns Hopkins University, Baltimore, USA and Ruer Boşković Institute, Zagreb, Croatia

DOI 10.12694/scpe.v17i2.1156
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 61–78. http://www.scpe.org © 2016 SCPE

A GPU-BASED SOFT REAL-TIME SYSTEM FOR SIMULTANEOUS EEG PROCESSING AND VISUALIZATION

ZOLTAN JUHASZ AND GYORGY KOZMANN [*]

Abstract. EEG processing is generally acknowledged as a computationally very intensive task. The execution of pre-processing steps, frequency domain operations and source localisation algorithms result in long execution times, which prohibit the use of high-resolution EEG brain imaging techniques outside research laboratory settings. We present a novel GPU-based streaming architecture, which has the potential to drastically reduce execution times and, at the same time, provide simultaneous 2D and 3D visualization facilities. The system uses a highly-optimised and re-configurable pipeline of CPU and GPU cores that attempts to exploit the tremendous computing power whenever possible. The system can process live data arriving from an EEG device or data stored in EEG data files. The computer drives a large display wall system consisting of four 46-inch monitors, which provides a 4K-resolution drawing surface for visualising raw EEG data, potential maps and various 3D views of the patient's head. Two example brain imaging algorithms, the surface Laplacian and the spherical forward solution are used as an illustration for the effective use of the massively parallel GPU hardware in speeding up computations. The paper describes the architecture of the system, the key design decisions, and the performance optimization steps that were required to achieve sub-millisecond per-sample execution times. The control flow of the system is expressed in a very modular fashion in Java but the performance-critical algorithms are programmed in CUDA and run on the GPU. Relying on the CUDA-OpenGL interoperability bridge, the computing subsystem feeds visualisation results directly into the OpenGL pipeline, eliminating unnecessary GPU-Host data transfers. The system demonstrates that up to three orders of magnitude speedups are achievable compared to MATLAB implementations, and this processing speed can be maintained during simultaneous interactive 3D visualisation of the results.

Key words: Parallel Computing, GPU, EEG Processing, Brain Activity, Source Localisation

AMS subject classifications. 68Q10, 68W10, 92C55

1. Introduction.

High-resolution EEG imaging is an increasingly important tool in neuroscience [1]. Unlike CT or MRI that only provide structural information (head anatomy), EEG can measure functional brain activity. Its temporal resolution is typically in the millisecond range, which is vastly superior to alternative functional imaging methods, such as PET or fMRI. Consequently, EEG is indispensable in areas where brain activity mapping with high temporal resolution is required, e.g. in epilepsy diagnosis in the clinical setting [2, 3] or in cognitive experiments based on evoked or event-related potentials [4].

The neuroscience community relies on commercial and open source software products for EEG processing. Commercial programs are more common in the clinical area, whereas open source packages, such as the EEGLAB [5], Fieldtrip [6] or CSD [7] MATLAB toolboxes, dominate the research labs. While the open source approach reduces development time, facilitates sharing of methods and data, as well as serves as a verification infrastructure, by being de-facto standards they may halt innovation in the area of EEG software architecture and algorithm design. We argue that innovation is very much needed in these areas, since the processing speed achieved during routine evaluations with existing methods is intolerably low.

State-of-the-art EEG systems can use up to 256 electrodes and typically operate at a sampling frequency of 1-2 kHz . As a result, even a few-second measurement can generate hundreds of MByte's of data. Consequently, the evaluation of these experiments results in long execution times (varying from minutes to hours depending on the methods, data size and control parameters used in the experiments). This can be somewhat tolerated in single-shot experiments but it clearly presents a serious obstacle in large multi-patient studies, or if EEG is ever to be considered as a diagnostic tool in the daily clinical routine. Another complication is related to the visualisation of the results. Large amounts of intermediate results need to be stored in order to avoid repeated executions of long-running algorithms. Visualisation of temporal changes present special challenges too; the typical approach being to generate images of consecutive time samples and from these create a time-lapse video.

This paper presents a novel system developed in our department that employs a GPU-based massively parallel computational environment for simultaneous EEG processing and visualisation. The system demonstrates that near real-time processing speed is achievable in key brain imaging algorithms, such as the spherical surface

[*] Department of Electrical Engineering and Information Systems, University of Pannonia, Egyetem u. 10, Veszprem, Hungary, 8200 (juhasz,kozmann@virt.uni-pannon.hu).

Laplacian [8] or the forward solution for source localization [9], and shows that visualisation can be coupled to computation removing the need for storing intermediate data and/or generating time-lapse videos. While much work has been done in porting specific EEG algorithms to GPUs [10, 11, 12, 13], we are not aware of any system that are similar to ours in creating an end-to-end GPU-centric simultaneous compute and visualisation environment. Although our system is designed for EEG brain imaging, with little modifications, it can be readily used for ECG-based multi-channel body-surface potential mapping [14, 15, 16] applications as well.

The structure of our paper is the following. Section 2 introduces fundamental EEG imaging concepts and describes two illustrative methods, the (*i*) surface Laplacian map calculation and the (*ii*) EEG forward problem that is indispensable in brain activity source localisation. Both approaches aim at detecting the spatial and temporal patterns of cortical brain activity. The theory behind both methods is described briefly, along with key details of their traditional MATLAB implementations. Section 3 describes the rationale behind using graphical processors for increasing processing speed and discusses massively parallel GPU implementations of both algorithms. Section 4 describes the core architecture of our software and explains the key design concepts that resulted in a GPU-centric EEG processing and visualisation framework. Section 5 provides details of the visualisation subsystem and finally, Section 6 discusses our results.

2. EEG imaging methods. EEG imaging is a cost-effective and non-invasive measurement method that maps the activity of the brain at very high temporal resolution. Its use is indispensable in epilepsy diagnosis and treatment, and in cognitive experiments that aim to understand the neural mechanism of various brain processes. The electric discharge of neurons generate a small but measurable potential distribution on the scalp surface, which is the basis of EEG brain imaging.

The electric potential distribution measured on the scalp is the effect of many simultaneously active neurons in the brain. These firing neurons are normally considered to be found in the cortical area and are represented as electrical current generators (dipoles) with a direction normal to the cortical surface. The electric field is largely influenced by the conductivity properties of the various parts (cerebrospinal fluid, skull, scalp) of the head. Most importantly, the relatively low conductivity of the skull results in a lateral spread of the potential distribution, which results in a “blurred” or “smeared” image on the scalp, as shown in Fig. 2.a. In addition, the measured signals have low signal-to-noise ratio and suffer from the presence of unwanted artefacts, such as the effect of eye and other muscle movements, skin resistivity changes, etc. As a consequence, the potential map is rarely used as recorded, but (i) first cleaned by performing additional filtering and other artefact removing pre-processing steps, then (ii) subsequent operations are employed that increase the spatial resolution of the EEG image and/or determine the location and amplitude of the current sources in the brain that generated the measured potential distribution. Two representative methods of the latter group are the calculation of the surface Laplacian and the source localisation based on the forward/inverse solution, which we describe briefly in the next sections. For practical reasons, these methods are often executed on a simplified, hypothetical 3 or 4-layer spherical head model, in which the brain, skull and scalp layers are represented by concentric spheres of varying radii, as illustrated in Fig. 2.b. While several methods exist that use realistic head models, in this paper we focus on the spherical head model only.

2.1. The surface Laplacian. Laplace imaging can be used to ‘sharpen’ the scalp potential images, i.e. to reduce the above-mentioned smearing effect. The surface Laplacian [17, 18, 19] – the second derivative of the surface potential – estimates the current source density (CSD) of the brain that is directly related to the activity level of cortical areas. The surface Laplacian is frequently used in event-related experiments, in which normally a multi-trial average of a 5-10 second time window of brain activity is analysed. The advantage of the Laplacian is that it is independent of the reference electrode choice, relatively insensitive to eye-movement artefacts and generates a relative topographic image that is well suited to the recognition of activity patterns. The Laplacian of the potential field f in the three-dimensional Cartesian coordinate system is expressed as

$$\Delta f = \nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \tag{2.1}$$

where f is the potential function represented by the discrete potential values measured at the electrodes. Fig. 2.2.a illustrates the power of the surface Laplacian in highlighting scalp current sources that are otherwise

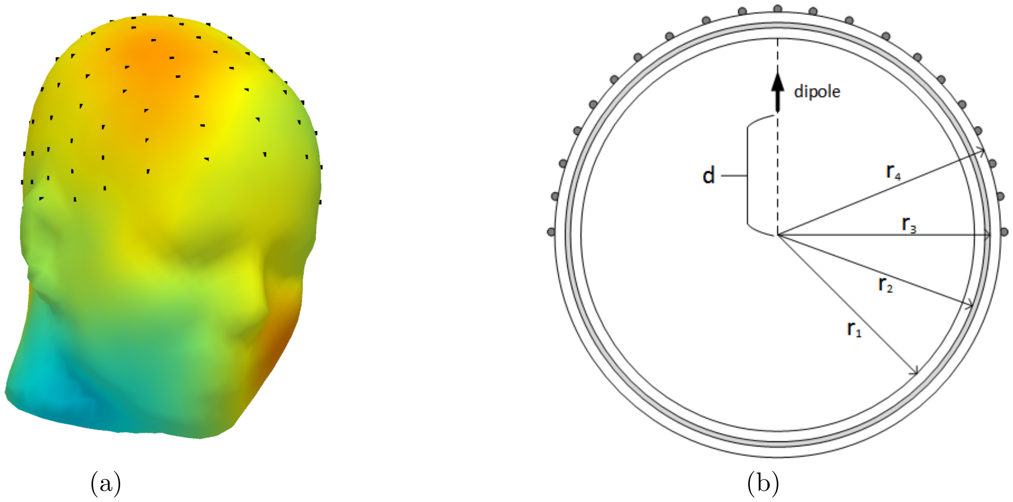


Fig. 2.1. *(a) EEG scalp potential map: A typical potential map displayed on a realistic head model along with the measuring electrodes. Red and blue colour represents positive and negative potential values, respectively. Face and neck areas should be ignored as they are the result of projecting values from the spherical interpolation. (b) 4-layer spherical head model: The spherical head model with a single radial activity source (electric dipole) with eccentricity d, and four structural layers (brain, inner and outer skull, scalp).*

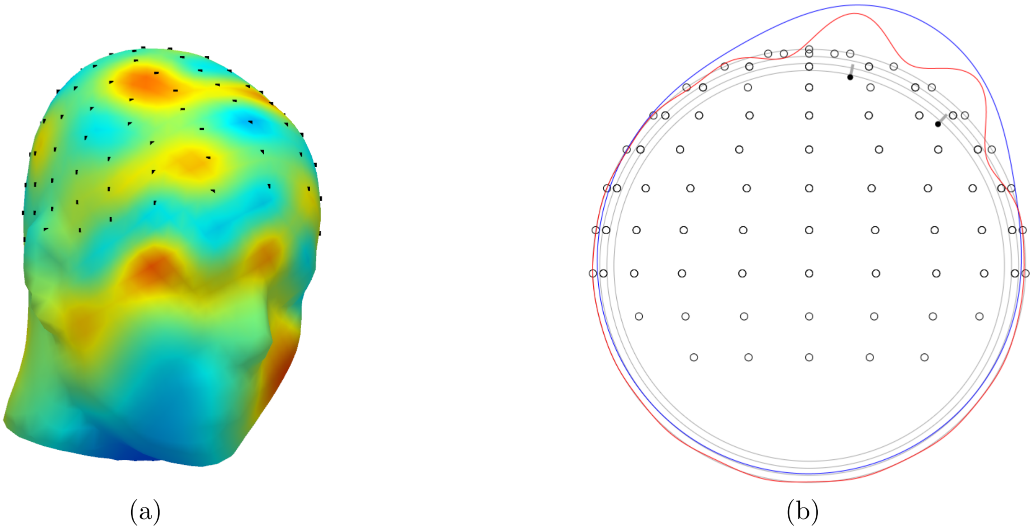


Fig. 2.2. *(a) The surface Laplacian of the scalp potential: Laplace map derived from the scalp potential distribution. (b) Difference in the activity source separation capability of the potential (blue) and Laplacian (red) maps: Comparison of the potential (blue line) and Laplace (red line) maps of a two radial dipole source configuration.*

invisible on the potential map of Fig. 2.a. Fig 2.2.b shows the focusing effect of the Laplacian on the cross section of the spherical model. The blue and red curves represent the potential and the Laplacian interpolated values, respectively. Although there are two active radial dipoles near the brain surface, the potential map does not indicate two sources, whereas the Laplace result clearly shows the presence of two dipoles.

One of the most widely used algorithm for computing the surface Laplacian on a sphere was proposed by Perrin *et al* [8, 20] that uses spherical splines for the calculation of the Laplacian. For a given surface point E

on the sphere, the surface Laplacian (a.k.a. the current source density), C, is given as

$$C(E) = \sum_{i=1}^{N} c_i h\big(cos(E, E_i)\big), \tag{2.2}$$

where E_i is the i^{th} electrode on the surface of the sphere and the c_i s are the solutions of the following system of equations:

$$\mathbf{GC} + \mathbf{T}c_0 = \mathbf{Z} \tag{2.3}$$

$$\mathbf{T'C} = 0 \tag{2.4}$$

where $T' = (1, 1, \dots, 1)$, $C' = (c_1, c_2, \dots, c_n)$, $Z' = (z_1, z_2, \dots, z_n)$, and $G = (g_{ij}) = \Big(g\big(cos(E_i, E_j)\big)\Big)$, where z_i is the potential measured at electrode E_i. The potential and Laplacian interpolation functions $g(x)$ and $h(x)$ are defined as the sum of the following series

$$g(x) = \frac{1}{4\pi} \sum_{n=1}^{\infty} \frac{(2n+1)}{n^m (n+1)^m} P_n(x) \tag{2.5}$$

and

$$h(x) = -\frac{1}{4\pi} \sum_{n=1}^{\infty} \frac{-(2n+1)}{n^{m-1} (n+1)^{m-1}} P_n(x) \tag{2.6}$$

where $P_n(x)$ is the n^{th} degree Legendre polynomial and m is the stiffness parameter of the spline. The summation theoretically is infinite, in practice, it is sufficient to take only the first, typically $N = 15$ or $N = 50$ terms.

In the widely used MATLAB-based CSD Toolbox [7] package, the calculation of the surface Laplacian is carried out in two steps; first, the G and H matrices are computed using Eqs. 2.5 and 2.6, than the Laplacian is calculated for the entire *electrodes* × *samples* data window by solving Eq. 2.4. The outline of the MATLAB implementation is given below:

```
// high-level structure of the Laplacian (CSD) computation
M = electrode coordinate information;
[G,H] = GetGH(M);
D = interval of measured potential data;
D = D';
X = CSD (D, G, H);
```

where the functions getGH() and CSD() are defined as follows.

```
// calculation of the G, H matrices
function [G,H] = GetGH (M, m)
...
N = 50;                              % set N iterations
G(nElec,nElec) = 0; H(nElec,nElec) = 0;  % claim memory for G- and H-matrices
for i = 1:nElec; for j = 1:nElec;
   P = zeros(N);                     % compute Legendre polynomial
   for n = 1:N;
      p = legendre(n,EF(i,j));
      P(n) = p(1);
   end;
   g = 0.0; h = 0.0;                      % compute h- and g-functions
   for n = 1:N;
      g = g + ( (( 2.0*n+1.0) * P(n)) / ((n*n+n)^m    ) );
      h = h + ( ((-2.0*n-1.0) * P(n)) / ((n*n+n)^(m-1)) );
   end;
   G(i,j) =  g / 4.0 / pi;                % finalize cell of G-matrix
   H(i,j) = -h / 4.0 / pi;                % finalize cell of H-matrix
end; end;

// calculation of the Laplacian
function [X, Y] = CSD(Data, G, H, lambda, head)
...
Gi = inv(G);                              % compute G inverse
for i = 1:size(Gi,1);                     % compute sums for each row
   TC(i) = sum(Gi(i,:));
end;
sgi = sum(TC);                            % compute sum total
fprintf('Sample points: %d\n', nPnts);
for p = 1:nPnts
   Cp = Gi * Z(:,p);                      % compute preliminary C vector
   c0 = sum(Cp) / sgi;                    % common constant across electrodes
   C = Cp - (c0 * TC');                   % compute final C vector
   for e = 1:nElec;                       % compute all CSDs ...
      X(e,p) = sum(C .* H(e,:)') / head;  % ... and scale to head size
   end;
   if nargout > 1; for e = 1:nElec;       % if requested ...
      Y(e,p) = c0 + sum(C .* G(e,:)');    % ... compute all SPs
   end; end;
end;
```

2.2. The forward solution. While the surface Laplacian can help in focusing the scalp potential image, it does not provide information about the exact cortical location of the neuronal activity sources. Source localization methods can help to identify the location and magnitude of the activity sources (dipoles) based on the measured potentials. The difficulty lies in the problem that several dipole combinations can generate identical scalp potential maps. This is the bioelectrical *inverse problem* which has no unique solution except when a priori constraints can be applied.

The basis of the inverse solution is that one or more hypothetical dipoles are placed inside the brain and we compute the potential image generated by the given dipole — this is called the *forward problem* — which is then compared to the measured values. During this process, the position, orientation and magnitude of the dipoles are varied until an acceptable solution is found. Several solution methods have been proposed for solving the inverse problem; reviews and comparisons of these can be found in [21, 22, 23, 24].

The calculation of the forward solution — the potentials generated by a given dipole — is based on computing the electrical field generated by that dipole using a chosen head model. The simplest head model is the spherical model that can contain up to five layers. The advantage of potential calculations using the spherical head model is that they have a closed form solution, consequently their associated computational cost is relatively modest. This is crucial in the localisation process, during which a large number of dipoles need to be tested. Various algorithms have been developed for the multi-layer spherical head model forward calculation [25]. One of these is the 4-layer spherical forward solution algorithm proposed by Cuffin and Cohen [9] that is part of the Fieldtrip MATLAB toolbox [6]. The computation of the generated potential values is based on the following equation that calculates the effect of the x component of a source dipole vector. Similar equations are used for the y and z components.

$$V = \frac{P_x \cos(\varphi)}{4\pi\sigma_4 R^2} \sum_{n=1}^{\infty} \frac{(2n+1)^4 f^{(n-1)} (c*d)^{(2n+1)} P_n^1\big(\cos(\theta)\big)}{n\Gamma} \tag{2.7}$$

where P_n^1 is the associated Legendre polynomial,

$$\begin{aligned}\Gamma = {} & d^{(2n+1)}\big\{b^{(2n+1)} n(k_1-1)(k_2-1)(n+1) + \\ & c^{(2n+1)}(k_1 n + n + 1)(k_2 n + n + 1)\big\}\big\{(k_3 n + n + 1) + (n+1)(k_3-1)d^{(2n+1)}\big\} + \\ & (n+1)c^{(2n+1)}\big\{b^{(2n+1)}(k_1-1)(k_2 n + k_2 + n) + \\ & c^{(2n+1)}(k_1 n + n + 1)(k_2 - 1)\big\}\big\{n(k_3-1)(k_3 n + k_3 + n)d^{(2n+1)}\big\}\end{aligned} \tag{2.8}$$

and R is the radius of the scalp sphere, f is the coefficient for the dipole position (eccentricity), $k_1 = \sigma_1/\sigma_2$, $k_2 = \sigma_2/\sigma_3$, $k_3 = \sigma_3/\sigma_4$, whereas b, c, d are radius coefficients for each layer given as $b = r_1/r_4$, $c = r_2/r_4$, $d = r_3/r_4$, $R = r_4$.

```
// function for computing the forward solution
function [lf, vol] = eeg_leadfield4(R, elc, vol)
...
   % given a fixed volume conductor, these only need to be computed once for all electrodes
   const = (2*n+1).^4.*f.^(n-1) ./ (vol.t.*4*pi*c4*r4^2);
   % note that variable vol includes the Gamma term
   for i=1:Nchans
      % convert the position of the electrodes to spherical coordinates
      [phi, el] = cart2sph(elc(i,1), elc(i,2), elc(i,3));

      % change from colatitude to latitude and compute the cosine
      cos_theta = cos(pi/2-el);

      % the series summation starts at zero
      s_x = 0;
      s_z = 0;

      for n=1:Nmax
         P0  = plgndr(n,0,cos_theta);     % zero'th order Legendre
         P1  = plgndr(n,1,cos_theta);     % first order Legendre
         s_x = s_x + const(n)*P1/n;       % s_y is identical
         s_z = s_z + const(n)*P0;
       end

    lf(i,1) = -cos(phi) * s_x;
    lf(i,2) = -sin(phi) * s_x;            % s_y is identical to s_x
    lf(i,3) = 1         * s_z;
end
...
```

2.3. Computational cost. In this section, we list the execution times of the surface Laplacian and forward solution computation using the CSD Toolbox and Fieldtrip MATLAB implementations described in the brevious sections. These results serve as baseline performance indicators that highlight the performance

TABLE 2.1

Calculation time [sec] of the **G** *and* **H** *matrices (CSD Toolbox)*

n	Number of electrodes		
	64	128	256
15	5.74	22.84	92.50
50	34.00	137.51	552.12

TABLE 2.2

Total surface Laplacian calculation time [sec], N=15, in MATLAB (CSD Toolbox)

Number of samples	Number of electrodes		
	64	128	256
1	5.74	22.84	92.51
512	5.81	23.03	92.89
1024	5.89	23.15	93.28
2048	6.04	23.47	94.07
5120	6.50	24.40	96.33
7160	6.80	25.03	97.94

problems of traditional sequential implementations, and indicate what speedups are required for achieving online processing and visualisation. All values are obtained with an Intel Core i7-3820 2.70 GHz processor.

The CSD Toolbox surface Laplacian execution times are listed in Tables 2.1 and 2.2. Since this implementation requires an initialisation step, Table 2.1 lists the calculation time of the **G** and **H** matrices for different numbers of electrodes and summation limits, while Table 2.2 shows the overall execution time including the **G**, **H** matrix and the sample window Laplacian calculations for $N = 15$. It is evident — even at this summation limit, N — that the computation of the surface Laplacian for a single sample can take 10s of seconds. For $N = 50$, the execution time reaches the order of minutes. Note that in our measurement setup the sampling frequency is 2048 Hz, thus the evaluation of 1 second of EEG data requires the processing of 2048 samples.

Table 2.3 contains illustrative execution times obtained with the Fieldtrip Toolbox using 64 and 128 electrodes and varying number of dipoles. The forward solution is a truly critical operation since it has to be calculated many times during the solution of the inverse problem. In the most general case, the location, orientation and magnitude parameters of each dipole are adjusted iteratively until the best fit is found, therefore several iterations are required. Although a dipole count of 1,000 or 10,000 might seem large, in a spherical head model more than 5,000, in a realistic cortical model, more than 100,000 dipoles might be required in order to achieve sufficient spatial resolution.

3. Embracing GPU technology. The traditional approach to speeding up the illustrated MATLAB algorithms is to implement them in C/C++, potentially using such a parallel implementation technology that can effectively scale over a range of CPU cores. Unfortunately, in traditional computers the achievable parallelism – hence speedup – is limited by the number of CPU cores. In commodity personal computers, one cannot expect speedup values larger than 8. On HPC servers using state-of-the-art Xeon processors, the potential speedup can reach up to 16. Note that we ignore large multi-node supercomputers with hundreds or thousands of CPU cores as they are neither typical nor practical within the medical field. The execution times listed in Tables 2.1 to 2.3 show clearly that orders of magnitude larger speedup values are required if the target is to reach near real-time execution speed.

A rough estimate for the required computational performance can be made as follows. Let us assume a forward calculation problem for E = 128 electrodes, D = 5200 dipoles, S = 2048 samples/sec with K = 1578 operations for computing the effect of one dipole on one scalp electrode. The required number of operations per second is $E \times D \times S \times K = 1.28 \times 5.2 \times 2.048 \times 1.578 \times 1011 = 2.045 \times 1012$, which gives approximately 2 teraflops. A similar estimate for the surface Laplacian with E = 128 electrodes, P = 5120 surface points, S = 2048 samples/sec, and K = 1200 operations per surface point results in $1.28 \times 5.12 \times 2.048 \times 1.2 \times 1011 = 1.61 \times 1012$ operations, or 1.61 teraflops. This level of performance cannot be achieved with 8-10 CPU cores. Graphical processing units, however, provide a cost and energy-efficient high-performance computing platform for EEG processing.

Table 2.3
Calculation time [sec] of the 4-layer forward solution in MATLAB (Fieldtrip Toolbox)

Number of samples	Number of dipoles					
	1	10	100	500	1000	10000
64	0.25	0.52	3.34	15.93	31.90	314.86
128	0.32	0.95	7.22	35.13	70.03	700.02

3.1. GPU overview. Graphical processors, due to their origin in 3D graphics, are designed for massively parallel execution of vertex/pixel level instructions. Today's GPUs are general-purpose, programmable computing engines. State-of-the-art GPU chips can contain over 3000 cores and their computational performance can exceed 6 teraflops (using single-precision operations). GPUs can efficiently solve large data-parallel problems if the problem maps effectively to a large number of GPU cores and is based on executing the same instruction sequence for the data set. As illustrated in Figure 3.1, the GPU program (called kernel) is executed under the control of a host program. The kernel represents a thread of execution that is executed in multiple copies at the same time. Kernels access data from the GPU device and on-chip memory during execution, hence data must be transferred between the host system and the GPU before and after the kernel execution. Kernel threads can be arranged into 1D, 2D or 3D grids in order to efficiently map a problem onto the physical cores. Threads are executed in warps (groups of 32 threads) under the control of an on-chip thread scheduler. The GPU card uses a non-uniform access memory; the various layers of the memory hierarchy (registers, shared memory, constant and global memory) have very different access times. Register and shared memory operations are the fastest. Shared memory is also accessible to all threads within a block and can be used to communicate values among the threads. Global memory access is very expensive, thousands of instructions can be executed within a single read/write cycle. Even more expensive is the data transfer operation between the host computer and the GPU.

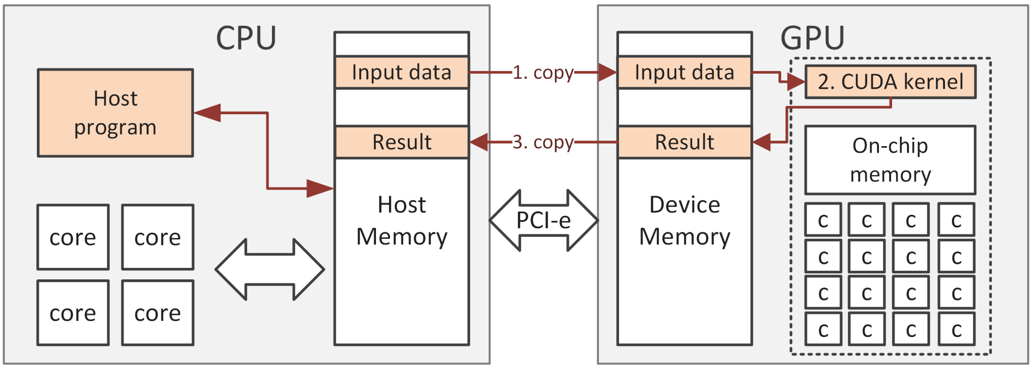


Fig. 3.1. *Illustration of the typical GPU kernel execution steps, including host–GPU data transfer.*

3.2. Parallel implementation strategy. Our primary aim in creating parallel GPU implementations for the Laplacian and the forward computation was to investigate various performance optimisation approaches and test how fast we can execute these algorithms on graphics cards. In order to have maximum flexibility and maximum control over the hardware during our implementation and optimisation steps, we have excluded OpenACC [26] and library-based (e.g. cuBLAS) approaches in our solution and developed the programs at a lower abstraction level using CUDA [27].

As the first step, we re-implemented both algorithms in C and in Java. They provided the foundation for the parallel implementations. Then, we created a data-parallel implementation of both the Laplacian and the forward algorithms. The goal of the first implementation was to numerically validate the parallel versions and to identify performance critical points. Finally, we experimented with various data partitioning schemes, memory and instruction scheduling optimisations to improve performance. Our aim was to create implementations that scale over a large number of cores, minimise memory transfer and the number of cycles not spent with computation.

3.2.1. The GPU surface Laplacian. In Perrin's original surface Laplacian algorithm and in its CSD Toolbox MATLAB implementation, the Laplacian is only computed for the electrode coordinates. The other points are linearly interpolated during plotting the results. In our system, we wanted to compute the values at every vertex position of the head as the output of the Laplacian kernel is directly fed into the OpenGL pipeline's vertex shader. Our implementation computes interpolated values for each surface point (vertex), hence our main loop is executed S times, where S typically varies between 5,120 and 15,000, depending on the head model mesh. Since each surface point calculation requires all 128 electrode coordinates, our first implementation was distributed into S independent threads, each computing one surface point value. The outline of the Laplace kernel is as follows.

```
__global__ void laplace_kernel(
   const int numPoints, const float *d_surface_x, const float *d_surface_y,
   const float *d_surface_z, const float *d_electrodes_x, const float *d_electrodes_y,
   const float *d_electrodes_z, const float *d_potential, float *d_laplacian, float* mX)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   ...
   float interpolVal = 0.;
   for (int j = 0; j < 128; ++j) {
      interpolVal += laplace_interpol_device(d_surface_x[i], d_surface_y[i],
      d_surface_z[i], d_electrodes_x[i], d_electrodes_y[i],
      d_electrodes_z[i]) * mX[j];
   }
   d_laplacian[i] = interpolVal / ((sphereRadius ) * (sphereRadius));
}
```

The function `laplace_interpol_device()` is our GPU device function executing one step of the Laplace interpolation calculation; it calculates the Legendre polynomial and the value of $h(x)$ based on 2.6.

In the second implementation version, the problem was partitioned into an S block by E thread grid, where S is the number of surface points and E is the number of electrodes. This partition enables us to compute the partial Laplacian terms in the series of Eq 2.2 in parallel ans store the results in a shared memory array that is available to all threads of a block. These partial results are then summed to provide the final result for each surface point by a reduce-sum operation, implemented using the `__shuffle_down()` CUDA function. This implementation further reduced the execution time of the Laplacian.

3.2.2. The GPU forward solver. The forward algorithm is similar in structure to the Laplacian algorithm, in that it computes the potential for each electrode from the combined effect of the dipoles placed in the head model. The effect of each dipole on one specific electrode can be evaluated independently from the others, but the final result must be computed as the superposition of the partial results, therefore requiring a reduce-sum operation. Consequently, we used a block-partitioning scheme for the dipoles × electrodes data domain.

Our first parallel forward solver implementation was based on the work reported in [28]. The underlying structure of the reported algorithm is a dipoles × electrodes thread grid, in which each thread calculates the effect of one dipole on one scalp electrode. We have added several performance optimizations to this algorithm. Using a more optimized grid allocation strategy, restructuring code to optimize memory access and exploiting on-chip instruction level parallelism, we were able to almost double the performance of the original algorithm.

Our next modification for improving the overall execution time was to use a more efficient baseline algorithm. The forward algorithm proposed by Sun [29] uses an interpolated function to eliminate the lengthy computation of the Legendre sums in the original Cuffin & Cohen algorithm. This algorithm reduces the number of floating point operations to 140 per steps, bringing another ten-fold increase in performance[29]. The outline of this kernel is as follows. Input parameters are the x,y,z coordinates of the dipole, x,y,z dipole moments, x,y,z

coordinates of the electrodes (here called sensors).

```
_global__ void kernel_potential_Sun(float *dipx, float *dipy, float *dipz,
   float *momx, float *momy, float *momz, float *senx, float *seny,
   float *senz, float *out, int dipole_num, int sensor_num)
{
    int dipole_id = blockIdx.x;
    int sensor_id = threadIdx.x;

   __shared__ float potentials[128];

   ...
   vr = cn[1]*x+cn[2]*f*(1.5f*x2 - 0.5f)+a[0]*(Q-1.0f) * rsqrt(p)
        +a[1]*(x-f)*Q3+a[2]*(f*(x2+f2-2.0f)+x*f12)*Q5+cn[3]*f2*x*(2.5f*x2-1.5f)
        +a[3]*(x+f*((5.0f*x2-4.0f)+f*((x*(x2-9.0f))+f*((10.0f-2.0f*x2)-fx-f2))))*Q7;

   vf = __fsqrt_rn(1.0f - x2) * (cn[1] + 1.5f * fx * cn[2] +
        a[0] * Q * (1.0f + Q)/(Q - fx *
        Q + 1.0f) + a[1] * Q3 + a[2] * Q5 * (f12-f2+fx) +
        cn[3] * (5.0f * x2 - 1.0f) * f2*.5f + a[3]*(1.0f  +
        f2*(4.0f * f2 - fx + x2 - 10.0f) + 5.0f * fx) * Q7);

    potentials[sensor_id] += C*(t0*vf + r0*vr)/R2;

    __syncthreads();
    float sum = potentials[sensor_id];
    sum = blockReduceSum(sum);
    if (threadIdx.x == 0) out[dipole_id] = sum ;
}
```

All operations are executed on single-precision variables and where possible, higher-performance NVIDIA CUDA fast mathematical functions were used, such as the rsqrt for the reciprocal square root or the intrinsic function __fsqrt_rn for square root.

4. Combining visualisation and computation – the stream architecture. Traditional EEG processing systems follow the batch processing style of computation; data is input from a file, a set of algorithms are executed on the data and the results are saved into output files. Intermediate data is stored in memory. These systems are typically CPU-centric, i.e. the CPU controls the entire execution flow as well as performs the heavy mathematical computations. This type of software architecture is the result of abstractions based on the common computer architecture model of the past few decades that are often commonly referred to as the "*Denial Architecture*" [30]. These systems pretend that computers operate sequentially and that memory architecture is flat and access is uniform. GPU architecture and the CUDA computing model explicitly expose the underlying parallel hardware components and encourages developers to take advantage of data locality and restructure instruction execution flow to discover new opportunities for increasing performance.

In addition, our GPU kernel implementations were used as co-processor functions with explicit data movement before and after the kernel calls. This resulted in a performance bottleneck, which was caused by executing visualisation calls and additional processing steps on the CPU.

Our software architecture is explicitly parallel. The underlying design is based the stream computing paradigm, in which data is processed in a parallel pipeline. Measurement data is pumped from the EEG measuremnt device or data file and passed through various loosely coupled processing nodes. The nodes are connected by FIFO channels and output data to further processing or visualization nodes. The processing nodes can be executed by CPU threads or GPU kernels. The execution of the algorithms are triggered by the arrival of data and is carried out with maximum level of hardware parallelism. The pipeline operation and the coupling of CUDA kernels with the OpenGL graphics pipeline helped to remove any unnecessary data movement (by keep as much data as possible on the GPU).

Fig. 4.1 illustrates the high-level, abstract view of our architecture. The CPU system refers to the host computer that acts as a coordinator of the environment. Although it can contain several processing nodes, the goal is that most of the computation is carried out on the GPU and the host, ideally, only coordinates the execution process. The GPU system is responsible for computational and visualisation tasks. The output of this subsystem is the set of monitors comprising the display wall environment. Although not shown in the figure, but – if required – data can be fetched from the GPU and stored locally in the file system for future reference.

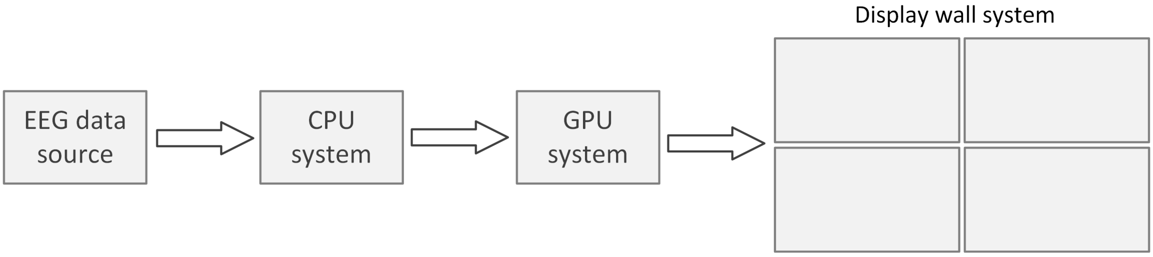


Fig. 4.1. *Simplified view of the CPU-GPU EEG processing and visualisation pipeline.*

The more detailed view of the architecture is depicted in Fig. 4.2. On the CPU side, green blocks mark objects that encapsulate the building blocks of the EEG processing system, e.g. head model, electrodes, computation, data source, etc. Although our system is implemented in Java, the principles can be carried over to implementations in C++ or other object-oriented languages. These objects allow the developer to configure the processing pipeline and parametrise the building blocks. The solid red arrow indicates the EEG stream data path, i.e., communication channels through which data travels from the source to the visualisation pipeline. The GPU system is accessed via interface layers; we use a JOGL wrapper layer for performing OpenGL calls and JCuda for calling CUDA functions from the Java objects.

We have avoided the use of any open-source visualization toolkit in order to retain maximum design freedom. Since the primary visualization aim is the display of time-varying potential or potential-derivative data on a static 2D or 3D geometry, head layer and electrode cap geometry data are cached into the GPU memory on initialization. Only potential data is updated continuously when the EEG marker position is changed. The OpenGL pipeline receives scalar values (potential or Laplacian results) as input and transforms these into vertex colour information via internal colour lookup tables implemented in the shader programs.

Potential data is processed and transformed by the GPU using CUDA kernels. The largest potential performance penalty in a GPU-based computational and visualization system is the host-GPU data transfer. If the GPU-computed results need to be copied back to the host in order to be displayed by a visualization system that will send the same data back to the GPU as part of the graphics calls, a major performance penalty is encountered. The CUDA-OpenGL interoperability architecture allows us to avoid this by directly modifying OpenGL data structures from inside the CUDA kernels.

Our algorithms are designed for scalability. As new GPU processors appear with more compute cores, the implementation automatically scales. If needed, the system can run with several GPU cards installed, in which case computation and visualization can be assigned to separate GPU devices.

5. Visualisation. The visualisation system is built around our display wall system consisting of four 46” Samsung edge-lit display panels arranged in a 2 × 2 grid. The resolution of the panels is 1920 × 1080 pixels, providing an overall 4K resolution drawing canvas for our application. The system is driven by an Intel Core-i5 hosted NVIDIA Quadro K4200 card (1344 cores, 2.1 teraflops). The surface area of the display wall – as shown in Fig. 5.1 – is sufficient to simultaneously display the EEG measurement plots, the 2D and 3D potential

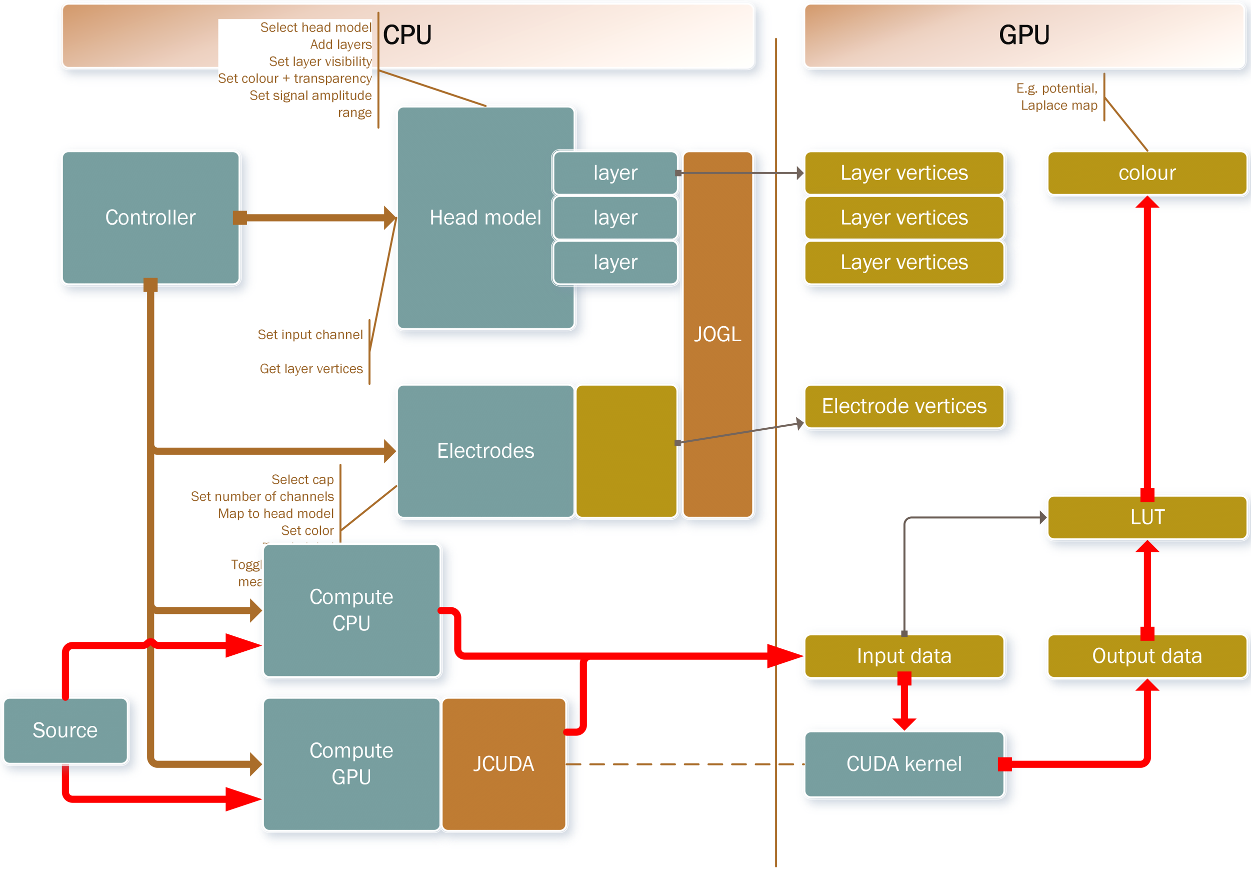


Fig. 4.2. *The high-level view of the GPU stream architecture. Solid arrows indicate data movement, thick red arrow depicts live data update path (stream). Data copied to the GPU memory are available to both the OpenGL and the CUDA backend. Processing node may or may not have GPU implementation parts. Their number and order are not limited.*

and Laplacian maps, as well as MRI images and (if available) video recordings. The visualization subsystem is implemented in OpenGL and JavaFX. OpenGL is used for all 3D operations, whereas JavaFX is for image views and general user interface elements.

The EEG plot (Fig. 5.2) displays the measured or pre-processed EEG data and allows the user to change the signal amplitude range and time axis resolution. The current sample to be processed is indicated by and selected with a marker (red vertical line). The marker can be moved to arbitrary positions to examine the activity map. It is also possible to play back the measurement and recompute the activity maps on the fly at the selected speed. Fig. 5.2 also illustrates the result of the surface Laplacian computation and its visualization as a 2D plot. The focusing effect the of the Laplacian is clearly visible.

5.1. Polygon mesh model. The core component of the visualisation subsystem is the polygon-mesh head model. Although the computational algorithms utilise the spherical head model, our display module is designed to support patient-specific realistic head models. Individual head models are created using the Freesurfer [31] and Brainstorm packages. Freesurfer is used to segment the patient's MR image into scalp and brain (white matter and cortical surface) layers. The Freesurfer output surfaces are then further processed with Brainstorm to generate the inner and outer skull surfaces.

The module allows one to select which head layers to display, adjust the transparency value of each layer if multiple layers are chosen. The results are fed into the OpenGL pipeline as a float value per vertex and converted into a vertex colour within the shader program. At the moment, the standard Jet (Fig. 5.3.a) and Blue-White-Red (Fig. 5.3.b) colour lookup tables are supported. It is also possible to choose on which layer

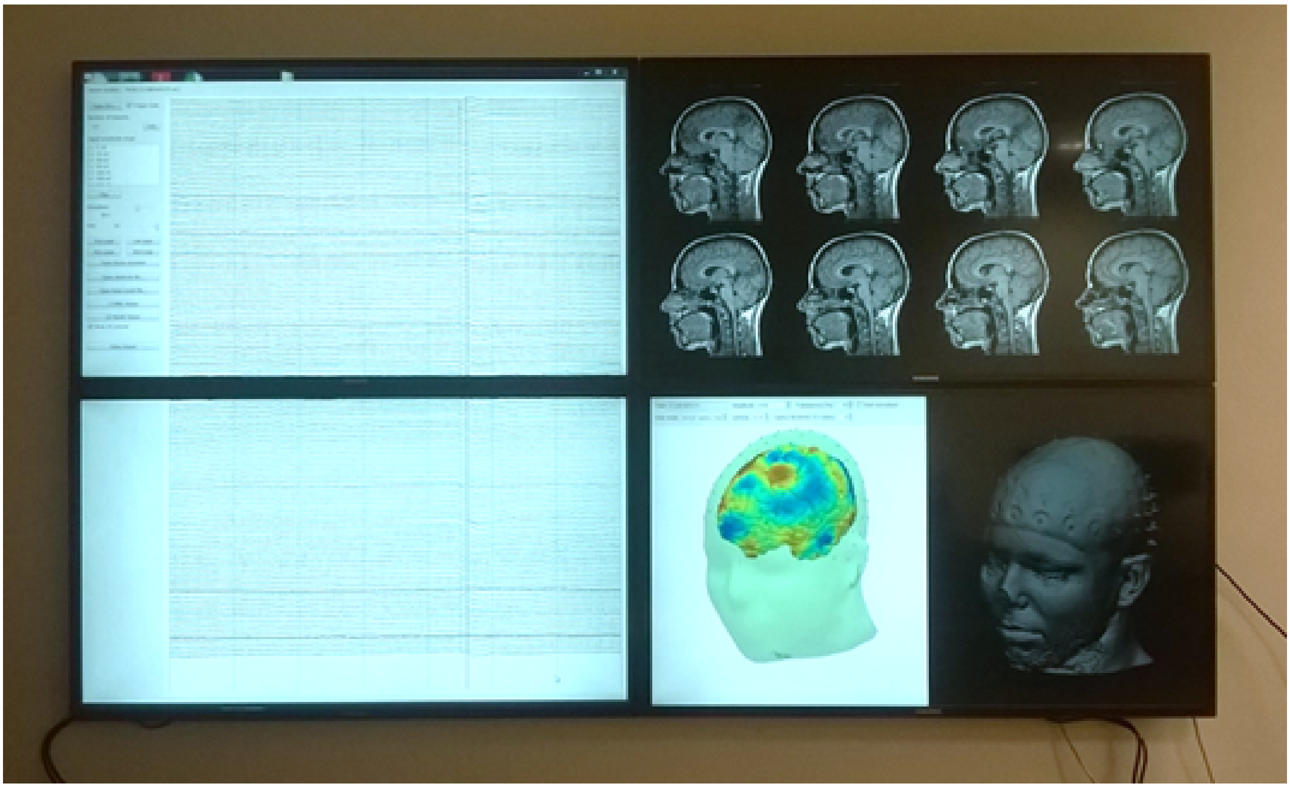


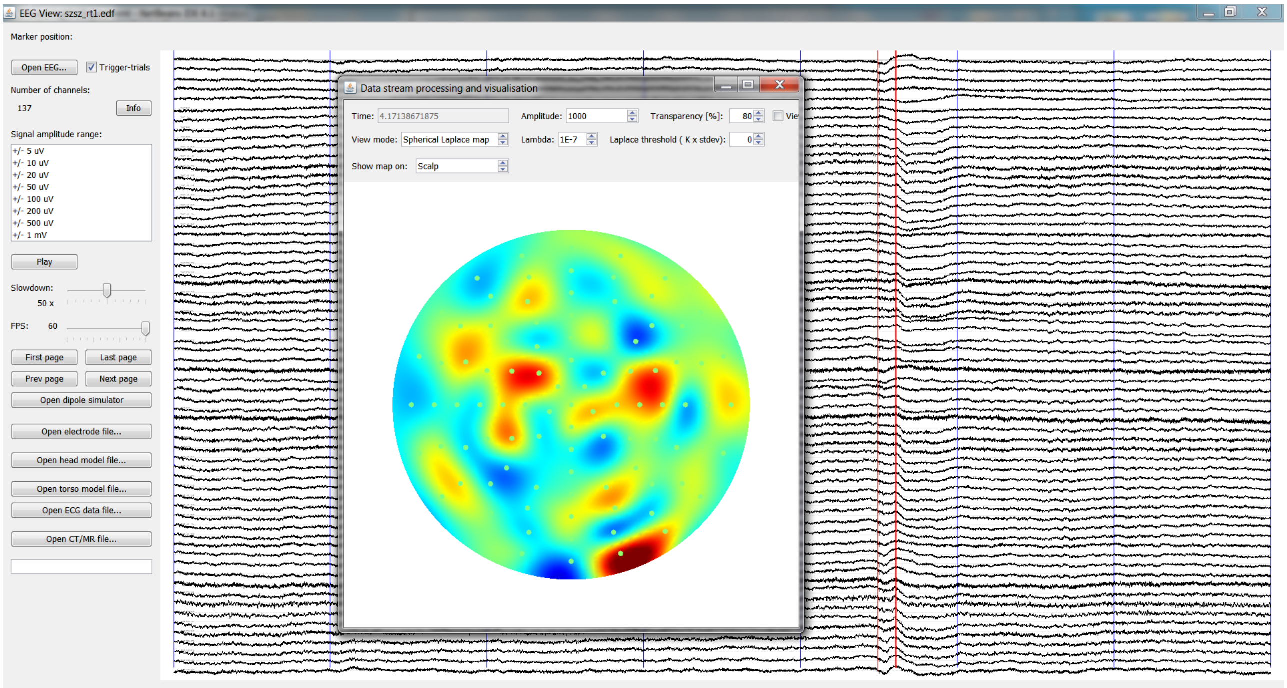
Fig. 5.1. *The EEG visualisation system in operation showing the EEG chart, surface Laplacian map, MRI slice and 3D volume rendering modalities.*

to visualise the result of the processing steps (potential or Laplacian maps), as shown in Figs. 5.3.a and 5.4.a. While at the moment cortical visualisation is the result of a simple projection of the scalp values, it creates the basis of more complex cortical imaging algorithms, such as distributed dipole model based inverse solutions, intracranial EEG visualisation and so on. The polygon view module also supports the usual rotation and zoom functions, hence any part of a surface can be examined in detail (Fig. 5.4.b).

5.2. MRI Viewer. In order to assist the users with anatomical data, the system also includes an MRI slice viewer module. The head can be viewed in axial, sagittal or coronal orientation, as well as in a combined view of these three orientations. The user can select how many slices to view (from 1 to 5 × 10 slices) – see Fig. 5.5 –, and can page or step through slices. If required, image contrast and intensity can be adjusted for the slices. The supported input formats are DICOM and Nifti. Images are input using the LONI Java Image IO Plugins [32] and displayed as JavaFX `ImageView` instances. Image slices are pre-processed (sorted, transformed) based on metadata in the MRI file.

5.3. 3D volumetric visualisation. On top of the slice viewer, the visualisation module also supports volumetric rendering of MRI brain images. A 3D texture-based ray-casting algorithm is implemented in OpenGL. Several options are incorporated for voxel visualisation, such as iso-surface or maximum intensity projection. A head model using the iso-surface approach is shown in Fig. 5.6. Note that the image was taken from a patient wearing the EEG electrode cap. The visibility of the cap provides the basis for a practical feature in our system. A common problem in EEG processing is the accurate registration of the EEG electrode coordinates on the patient's head. Normally these coordinates are estimated from the standard electrode coordinate map after measuring standard fiducial points (nasion, preauricular left and preauricular right) [33] or measured by expensive 3D digitising or scanner devices [34, 35].

In our system we implemented a 3D picking capability in the iso-surface volumetric rendering mode. As


EEG View: szsz_rt1.edf

Marker position:

☑ Trigger-trials

Number of channels: 137

Signal amplitude range:

+/- 5 uV
+/- 10 uV
+/- 20 uV
+/- 50 uV
+/- 100 uV
+/- 200 uV
+/- 500 uV
+/- 1 mV

Slowdown: 50 x

FPS: 60

Data stream processing and visualisation

Time: 4.17138671875

Amplitude: 1000

Transparency [%]: 80

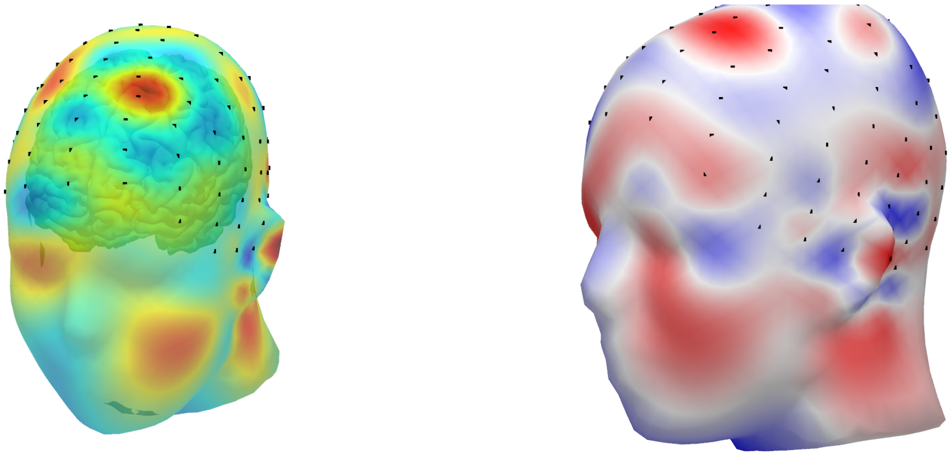
View mode: Spherical Laplace map

Lambda: 1E-7

Laplace threshold (K x stdev): 0

Show map on: Scalp

Fig. 5.2. *Screenshot of the prototype showing the multi-channel EEG plot and the 2D Laplacian map computed from the potentials.*


(a)

(b)

Fig. 5.3. *(a) Laplacian with Jet colour map: The surface Laplacian displayed on the scalp surface using the Jet colour lookup table. Red indicates positive, Blue negative values. (b) Laplacian with Red-Blue colour map: The surface Laplacian displayed on the scalp surface using the Blue-White-Red colour lookup table. Red indicates positive, Blue negative values.*

shown in Fig. 5.6, the centre of each electrode can be located on the 3D model. Each electrode centre is picked and the ray-surface intersection point is calculated and returned as the electrode coordinate. In order to achieve accurate scalp intersection calculation, for each electrode, the surface threshold is modified to remove the electrode 'cone' (Fig. 5.6). The processed electrode is marked with a small semi-transparent sphere to track progress. The system supports electrode coordinate labelling by using a user-selected electrode cap coordinate map that is used to locate the closest set of neighbour electrodes for the pick location.

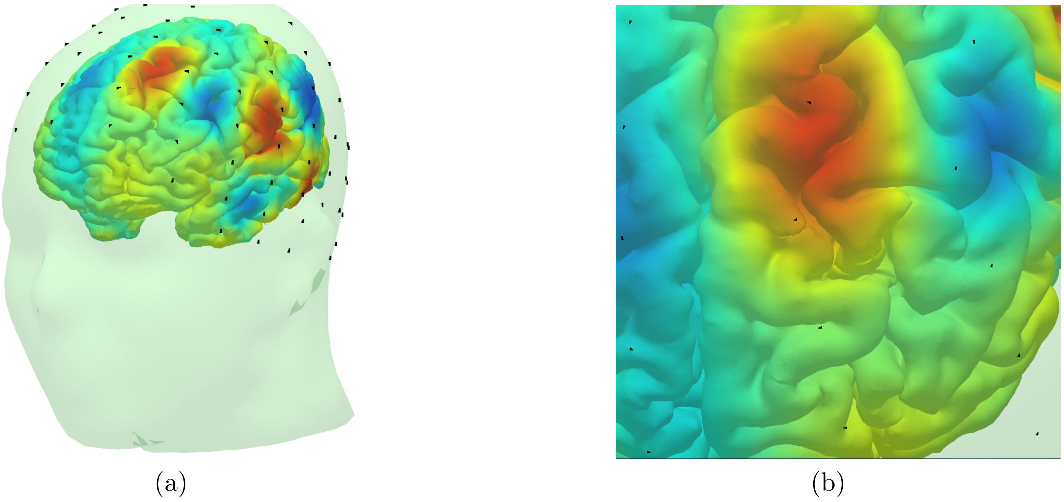


FIG. 5.4. *(a) The Laplacian map displayed on the cortex using back projection from the scalp; (b) Zoomed-in view of the cortical layer.*

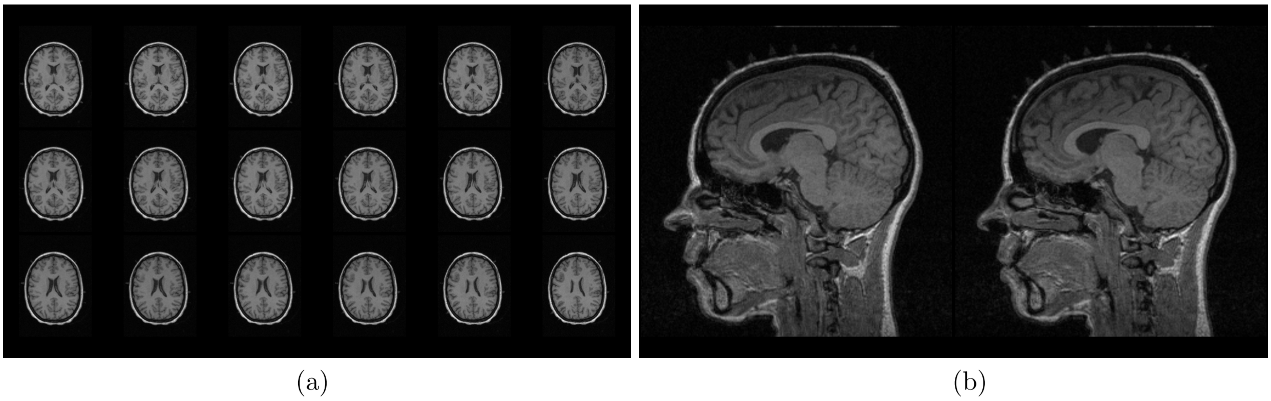


FIG. 5.5. *MRI slice viewer componen: (a) Axial slices displayed in a 3 × 6 arrangement; (b) Sagittal slices in a 1 × 2 layout arrangement*

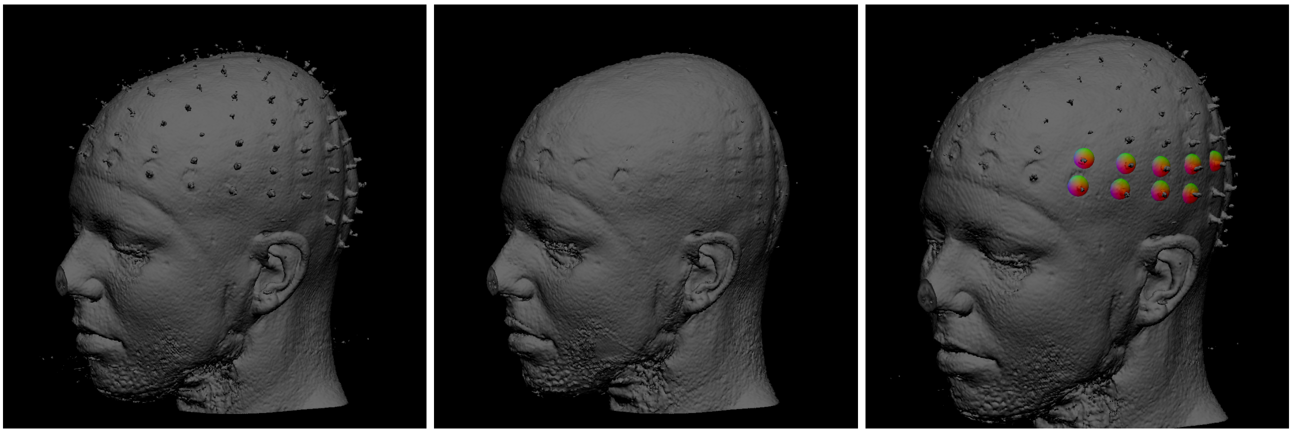


FIG. 5.6. *3D volumetric reconstruction of the patient MRI slices with different surface threshold (left and middle) values, and progress in the electrode localisation 'pick' operation (right).*

Table 6.1
GPU-based surface Laplacian execution times (128 electrodes)

Number of surface points	Execution time [$msec$] n=15	n=50
1024	0.164	0.361
2048	0.166	0.379
4096	0.172	0.469
8192	0.189	1.033
16384	0.341	3.482
32768	0.677	5.941

Table 6.2
GPU-based spherical forward solution execution times

Number	Execution time [$msec$]			
of	Cuffin & Cohen		Sun-Stok	
dipoles	E=64	E=128	E=64	E=128
1	0.341	0.720	0.144	0.497
10	0.402	0.891	0.152	0.578
100	0.626	1.337	0.293	1.086
1000	1.721	3.374	0.444	1.761
10000	8.852	17.002	0.752	3.001

6. Results and conclusions. In addition to the novel, stream-based EEG processing software architecture and the wide range of visualisation capabilities presented in Sections 4 and 5, the most notable result of our development is the achieved computational performance for the selected two key EEG processing algorithms. Our system has achieved considerable increase in computational performance. Tables 6.1 and 6.2 show the achieved execution times with the parallel GPU Laplace and forward implementations. Results were obtained using an NVIDIA Quadro K4200 GPU card having 1344 cores.

As shown in Table 6.1, using 128 electrodes and up to 2048 ($N = 50$) or 8192 ($N = 15$) surface points, the system can compute and visualize the Laplacian map in real-time at up to 2 kHz sampling frequency. Note that execution times are given in milliseconds, not in seconds! When further increasing the number of surface points, the system will work in soft real-time mode, still simultaneously computing and visualising results but at a lower rate.

The performance results for the forward problem are similarly impressive, as shown in Table 6.2. Using several code optimization steps we were able to improve the performance of the GPU-version of the Cuffin-Cohen algorithm [28]. Using Suns algorithm we could further decrease the execution time and reach millisecond range for 128 channels and 5000 dipoles.

We described a novel, GPU-based stream-oriented EEG processing software system. We demonstrated that the GPU architecture is well suited to simultaneous EEG visualization and processing. The computational performance offered by current GPUs make it possible to carry out simultaneous computation and visualization in real-time, or online if live measurement is performed. We were also interested in exploring new architectural and performance optimization methods that could lead towards the development of EEG software systems with higher flexibility and several orders of magnitude larger performance than what is available today. The removal of unnecessary host-GPU data transfers, optimised memory access and instruction execution strategies all contributed to the high achieved computational performance.

We have illustrated using two important EEG imaging algorithms, how a GPU-based stream-oriented system can be created, and showed that computational performance up to three orders of magnitude higher than MATLAB can be achieved while providing simultaneous interactive 2D/3D visualization. For both algorithms, the single-precision GPU implementation produced results with $RMSE < 2.6 \times 10^{-3}$ when compared to the MATLAB implementations.

The high-performance simultaneous processing and visualisation system integrated with a 4K resolution 2×2 display wall environment creates a very fast, efficient, yet cost-effective brain imaging system. Future development plans include the integration and coupling of other imaging modalities such as MRI and fMRI in our framework, and implementing fast realistic head-model based cortical imaging algorithms. Our fast implementation of the surface Laplacian can also serve as a basis for high-resolution cognitive studies and BCI applications. The software architecture and the system can be easily adapted for use in ECG-based body-surface potential mapping applications or similar computationally intensive visualisation areas.

Acknowledgements. The authors wish to thank Prof Zoltan Nagy and the National Institute for Clinical Neurosciences (Budapest, Hungary) for providing access to patient MRI and EEG measurement data, as well as their students, Andras Gergely Nagy (MSc) and Daniel Kenyeres (BSc) for their invaluable help in the development of the system described in this paper.

REFERENCES

[1] C. M. Michel, M. M. Murray, *Towards the utilization of EEG as a brain imaging tool*, NeuroImage, 61 (2012), pp. 371-385.

[2] L. Ding, G. A. Worrell, T. D. Lagerlund, and B. He, *3D source localization of interictal spikes in epilepsy patients with MRI lesions.*, Phys. Med. Biol., vol. 51, no. 16, pp. 4047-62, Aug. 2006.

[3] B. Alkonyi, C. Juhász, O. Muzik, E. Asano, A. Saporta, A. Shah, and H. T. Chugani, *Quantitative brain surface mapping of an electrophysiologic/metabolic mismatch in human neocortical epilepsy.*, Epilepsy Res., vol. 87, no. 1, pp. 77-87, Nov. 2009.

[4] N. Srinivasan, *Cognitive neuroscience of creativity: EEG based approaches*, Methods, vol. 42, no. 1, pp. 109-116, May 2007.

[5] Z. A. Acar and S. Makeig, *Neuroelectromagnetic forward head modeling toolbox.*, J. Neurosci. Methods, vol. 190, no. 2, pp. 258-70, Jul. 2010.

[6] R. Oostenveld, P. Fries, E. Maris, and J. Schoffelen, *FieldTrip: Open Source Software for Advanced Analysis of MEG, EEG, and Invasive Electrophysiological Data*, Computational Intelligence and Neuroscience, vol. 2011, Article ID 156869, 9 pages, 2011. doi:10.1155/2011/156869

[7] J. Kayser and C. E. Tenke, *Principal components analysis of Laplacian waveforms as a generic method for identifying ERP generator patterns: I. Evaluation with auditory oddball tasks.*, Clin. Neurophysiol., vol. 117, no. 2, pp. 348-368, Feb. 2006.

[8] F. Perrin, J. Pernier, O. Bertrand, and J. F. Echallier, *Spherical splines for scalp potential and current density mapping*, Electroencephalogr. Clin. Neurophysiol., vol. 72, no. 2, pp. 184-187, 1989.

[9] N. Cuffin and D. Cohen, *Comparison of the Magnetoenchephalogram and Electroencephalogram*, Electroencephalogr. Clin. Neurophysiol., vol. 47, pp. 132-146, 1979.

[10] R. Ramalho, P. Tomás, and L. Sousa, *Efficient Independent Component Analysis on a GPU*, in 2010 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 1128-1133.

[11] M. Corraine, S. Lopez, and L. Wang, *GPU Acceleration of Transmural Electrophysiological Imaging*, pp. 849-852, 2012.

[12] C. Dinh, J. Ruhle, S. Bollmann, and D. Gullmar, *A GPU-accelerated Performance Optimized RAP-MUSIC Algorithm for Online Source Localization.*, Biomedizinische Technik (Berl.) (2012), 57

[13] T. De Marco, F. Ries, M. Guermandi, and R. Guerrieri, *EIT forward problem parallel simulation environment with anisotropic tissue and realistic electrode models.*, IEEE Trans. Biomed. Eng., vol. 59, no. 5, pp. 1229-39, May 2012.

[14] R. Trobec, *Computer analysis of multichannel ECG*, Comput. Biol. Med., vol. 33, no. 3, pp. 215-226, May 2003.

[15] B. He, G. Li, and J. Lian, *A spline Laplacian ECG estimator in a realistic geometry volume conductor.*, IEEE Trans. Biomed. Eng., vol. 49, no. 2, pp. 110-7, Feb. 2002.

[16] G. Li, J. Lian, and B. He, *Spatial resolution of body surface potential and Laplacian pace mapping.*, Pacing Clin. Electrophysiol., vol. 25:(4 Pt 1), pp. 420-9, Apr. 2002.

[17] P. L. Nunez and R. Srinivasan, *Electric Fields of the Brain: The Neurophysics of EEG*, 2nd Edition. Oxford University Press, USA, 2005.

[18] P. L. Nunez and K. L. Pilgreen, *The spline-Laplacian in clinical neurophysiology: a method to improve EEG spatial resolution.*, J. Clin. Neurophysiol., vol. 8, no. 4, pp. 397-413, Oct. 1991.

[19] P. L. Nunez and A. F. Westdorp, *The surface laplacian, high resolution EEG and controversies*, Brain Topogr., vol. 6, no. 3, pp. 221-226, Mar. 1994.

[20] *Corrigenda*, Electroencephalogr. Clin. Neurophysiol., vol. 76, no. 6, pp. 565-566, Dec. 1990.

[21] R. Grech, T. Cassar, J. Muscat, K. P. Camilleri, S. G. Fabri, M. Zervakis, P. Xanthopoulos, V. Sakkalis, and B. Vanrumste, *Review on solving the inverse problem in EEG source analysis.*, J. Neuroeng. Rehabil., vol. 5, p. 25, Jan. 2008.

[22] C. M. Michel, M. M. Murray, G. Lantz, S. Gonzalez, L. Spinelli, and R. Grave de Peralta, *EEG source imaging.*, Clin. Neurophysiol., vol. 115, no. 10, pp. 2195-222, Oct. 2004.

[23] H. Hallez, B. Vanrumste, R. Grech, J. Muscat, W. De Clercq, A. Vergult, Y. DAsseler, K. P. Camilleri, S. G. Fabri, S. Van Huffel, and I. Lemahieu, *Review on solving the forward problem in EEG source analysis.*, J. Neuroeng. Rehabil., vol. 4, p. 46, Jan. 2007.

[24] O. Hauk, *Keep it simple: a case for using classical minimum norm estimation in the analysis of EEG and MEG data.*, Neuroimage, vol. 21, no. 4, pp. 1612-21, Apr. 2004.

[25] P. Berg and M. Scherg, *A fast method for forward computation of multiple-shell spherical head models*, Electroencephalogr. Clin. Neurophysiol., vol. 90, no. 1, pp. 58-64, Jan. 1994.

[26] *OpenACC Home — www.openacc.org* [Online]. Available: http://www.openacc.org/. [Accessed: 20-Dec-2015].

[27] NVIDIA Corp., *Cuda C Programming Guide*, Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide. [Accessed: 20-Dec-2015]

[28] N. B. Bangera and D. Lewine, *Accelerated large scale spherical model forward solutions for the EEG / MEG using CUDA*, in GPU Technology Conference, 2010, vol. 23, no. 11, p. Poster.

[29] M. Sun, *An efficient algorithm for computing multishell spherical volume conductor models in EEG dipole source localization.*, IEEE Trans. Biomed. Eng., vol. 44, no. 12, pp. 1243-52, Dec. 1997.

[30] B. Dally, *The End of Denial Architecture and The Rise of Throughput Computing*, Electronics, 2009.

[31] B. Fischl, *FreeSurfer*, Neuroimage, vol. 62, no. 2, pp. 774-81, Aug. 2012.

[32] *LONI Java Image I/O Plugins — Laboratory of Neuro Imaging.* [Online]. Available: http://www.loni.usc.edu/Software/IO_Plugins. [Accessed: 20-Dec-2015].

[33] U. Windhorst and H. Johansson, *Modern Techniques in Neuroscience Research.* Springer Science & Business Media, 2012.

[34] *xensorTM — ANT Neuro.* [Online]. Available: https://www.ant-neuro.com/products/xensor. [Accessed: 20-Dec-2015].

[35] S. S. Dalal, S. Rampp, F. Willomitzer, and S. Ettl, *Consequences of EEG electrode position error on ultimate beamformer source reconstruction performance*, Front. Neurosci., vol. 8, p. 42, Jan. 2014.

[36] M. J. Kilgard, *Modern OpenGL Usage: Using Vertex Buffer Objects Well*, White paper, Available: https://www.researchgate.net/publication/240320227_Modern_OpenGL_usage_using_vertex_buffer_objects_well

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016

DOI 10.12694/scpe.v17i2.1157
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 79–90. http://www.scpe.org © 2016 SCPE

IMPLEMENTATION OF A HORIZONTAL SCALABLE BALANCER FOR DEW COMPUTING SERVICES

SASKO RISTOV[*] KIRIL CVETKOV[†] AND MARJAN GUSEV[‡]

Abstract. Cloud, fog and dew computing concepts offer elastic resources that can serve scalable services. These resources can be scaled horizontally or vertically. The former is more powerful, which increases the number of same machines (scaled out) to retain the performance of the service. However, this scaling is tightly connected with the existence of a balancer in front of the scaled resources that will balance the load among the end points. In this paper, we present a successful implementation of a scalable low-level load balancer, implemented on the network layer.

The scalability is tested by a series of experiments for a small scale servers providing services in the range of dew computing services. The experiments showed that it adds small latency of several milliseconds and thus it slightly reduces the performance when the distributed system is underutilized. However, the results show that the balancer achieves even a super-linear speedup (speedup greater than the number of scaled resources) for a greater load. The paper discusses also many other benefits that the balancer provides.

Key words: Cloud computing, dew computing, distributed computing, large scale, load balancer, performance

AMS subject classifications. 68M14

1. Introduction.

Information and communication technology offers a huge support for the business today by speeding up the internal business processes. However, a company must be prepared for the business processes that interact with its customers and handle the peaks in the load [12].

The exponential growth of Internet of Things (IoT) and Big Data processing lead to the necessary scalability of resources at multiple levels. Skala et al. [26] present a three-layer scalable platform to facilitate the rapidly developing complex distributed computer systems and meet the performance, availability, reliability, manageability, and cost.

Today's companies either build their own virtualized data centers (usually enterprises) or rent on-demand resources organized in virtual machine (VM) instances (usually small and medium enterprises) from some cloud service provider. In either way, they require achieving maximum performance with minimized costs. However, the companies with their applications and services are challenged to orchestrate resources in real time in order to serve the dynamic load by their customers. One mechanism to instantly orchestrate the resources in the virtualized environment or cloud computing and maximize the performance by minimizing the utilized hardware resources is to introduce and implement an appropriate load balancing strategy [18].

Adding more computing resources and balancing the load can solve the peaks and various load balancers with different approaches for balancing techniques exist. The efficient and effective load balancer should have satisfied several functional requirements: balancing all incoming clients' packets to the available endpoint servers, having a configurable port where the requests should be sent their, having a pool of active endpoint servers with IP addresses that will handle the incoming packets, and the introduced load balancer between the clients and endpoint servers should improve the overall response time of the clients' requests.

Horizontal scaling is impossible without load balancing. Even more, the cloud computing based on advanced applications, high-performance computing, with huge storage and networking demands, can not be imagined without load balancing. It is one of the four layers in the cloud application architecture [15, 16]. Rimal et al. [19] describe the load balancing techniques that are used by various cloud service providers and solutions, including Amazon, Google, SalesForce, Azure, Eucalyptus, OpenNebula, etc. Most of these techniques are implementations of a conventional Round Robin schema, weighted selection mechanisms, HAProxy, Sticky session, SSL Least Connect, Source address, cluster server load equalization, high-performance protocols over EC2, hardware load balancing, cloud controllers, etc.

[*] Ss. Cyril and Methodious University, FCSE, Rugjer Boshkovikj 16, 1000 Skopje, Macedonia (sashko.ristov@finki.ukim.mk).

[†] Ss. Cyril and Methodious University, FCSE, Rugjer Boshkovikj 16, 1000 Skopje, Macedonia (kiril_cvetkov@yahoo.com).

[‡] Ss. Cyril and Methodious University, FCSE, Rugjer Boshkovikj 16, 1000 Skopje, Macedonia (marjan.gushev@finki.ukim.mk).

Load balancing was a widely used technique prior to the introduction the cloud computing paradigm and the virtualization technique. The performance of load balancing techniques in the cloud is analyzed by several authors [11, 13].

Radojevic and Zagar [17] grouped them into three groups: Session switching at the application layer, Packet-switching mode at the network layer and Processor load balancing mode. On the other hand, Heinzl and Metz [9] classified the load balancers as hardware and software, and commercial and open source.

There are two main techniques in load balancing: centralized on a specific server or distributed on several servers. The former is controlled by a single central node with all the other endpoint servers communicate with it. Bhadani and For example, Chaudhary [3] proposed the Central Load Balancing Policy for Virtual Machines (CLBVM) solution, while the CLBDM solution [17] considers server load and application performance on top of the Round Robin Algorithm. However, the centralized load balancing approaches in cloud computing have design limitation and communication overhead, thus failing to offer full scalability features [1].

Simjanoska et al. [25] proposed an architecture for a Low Level Load Balancer (L3B). Their architecture can distribute server load among several machines that are integrated over the communication link [24]. The proposed architecture is realized on a network level. It dynamically balances the network packets of the clients among all end point servers and after the computation, it forwards the responses to the clients that send the requests. All these transformations are conducted on the network layer, that is, by changing the information in the incoming and outgoing packet's header.

In this paper, we analyze dew computing services build as modular applications, with big data access, including various smart objects and embedded systems, sensors and other rich mobile clients. Wang [28] explains the dew architecture as an extension of the client server concept, which uses cloud server instead of the classical server, but includes a dew server that realizes a lot of processing and works in companion with the cloud server. In this context, a dew server acts as a cloudlet server that includes local data processing [2].

We have developed a naive implementation of the proposed L3B architecture, which unfortunately did not yield the expected results [21]. The successful implementation of redeveloped balancer according to the same L3B architecture was presented in [5]. The results of the conducted experiments not only that confirmed the successfulness of the implementation, but they show a superlinear speedup, that is, we achieved a speedup of up to 6.5 by using two endpoint servers compared to the case when using only one endpoint server. This paper extends the application domain of our centralized L3B balancer, which offers a high level of scalability when the number of end point servers has been increased (scaled). The conducted experiments lead towards very interesting conclusions. The analysis shows the load where the end point services achieve the best performance shown as speed. Again, a superlinear speedup is achieved as load increases. This important result means that the customers will achieve better performance for the same cost, or pay less for scaled resources to achieve the same performance.

The rest of the paper is organized as follows. Section 2 briefly describes the background. The successful implementation of the L3B balancer is presented in Section 3. Section 4 presents the testing methodology that is used in the experiments to prove the L3B scalability when it is used before two different endpoint servers: 1) web servers with static web pages and 2) computation intensive endpoint servers. The results for the former test case are presented and discussed, along with theoretical analysis in Section 5, while the results for the latter test case are presented in Section 6. Section 7 analyzes and discusses the additional benefits that the successful implementation of L3B balancer provides. Finally, the conclusions and plan for future work are presented in Section 8.

2. Background.

This section briefly describes the architecture that provides dew computing services and the architectural concept with a naive implementation of the L3B balancer.

2.1. Dew Computing Services.

A typical modern computing scenario realizes the thin client concept with a browser that has access to the Internet and all the processing is realized by cloud servers. This scenario does not use the potential of the processing and storage of the wirelessly connected small personal and mobile devices. The advances in technology can once again return the fat client idea, but now within the cloud computing environment setting the dew computing concept.

Transferring the servers to the edge of the network is not a new concept, it is a basis of the fog computing concept that extends Cloud computing and services to the edge of the network. Stojmenovic and Wen [27] discuss that the infrastructure of this new distributed computing allows applications to run as close as possible to sensed actionable and massive data, coming out of people, processes, and things. The idea to distribute servers to the edge of the mobile operator eases the transfer of the information from distributed IoT devices to the cloud servers, reducing service latency and improving the overall quality of service (QoS) with location awareness, mobility, and wireless access [4].

There were several other solutions to cope with WAN delays and approach cloud servers. For example, the cloudlet concept uses a smaller data center positioned locally where increased processing demand arises. The main obstacles the personalized mobile devices are facing include poor resources, such as battery life, weight, heat dissipation, etc. These constraints limit the possibilities of efficient use and initiate the need for offload of computationally expensive tasks to nearby servers in cloudlets or cloud servers [2]. The concept of cloudlets, as trusted, resource-rich computers, located near the mobile users was introduced by Satyanarayanan et al. [23].

The fog computing approach actually is an answer of the communication community to the rise of IT business to deliver IT-based services. The mobile operators have found a new way to deliver extended value-added services to their customers by setting computing resources on the edges of their network, not just for communication needs, but also for initial data processing.

The idea behind the dew computing is using the resources as much as possible before the processing is transferred to the cloud server. It uses the dew computing architecture providing micro services in collaboration with macro services, or dew services in collaboration with cloud services.

This is especially important in the IoT era, where a lot of sensors will stream data and require huge data processing with sufficient volume, velocity, veracity and all the characteristics of Big Data.

Dew services usually require smaller resources and, for example, can perform smaller audio, image or data compression tasks.

Another IoT problem concerns the Internet and cloud server availability. In such cases, the user cannot access its data and can hardly synchronize the computing tasks [28]. The cloudlet and dew architecture concepts enable an environment of smaller cloudlet and/or dew servers that can cope with problems of Internet or cloud server availability. Wang and Pan [28] discuss that the dew server and its related databases have two functions to provides the client with the same services as the cloud server and to synchronize corresponding databases.

2.2. The L3B architectural concept. The L3B's concept is presented in Fig. 2.1 [21]. The L3B balancer is hosted on a machine in front of a pool of endpoint servers and its main goal is to balance the clients' requests among the available servers. The latency due to balancer's additional layer should be compensated with a faster response time of the endpoint servers.

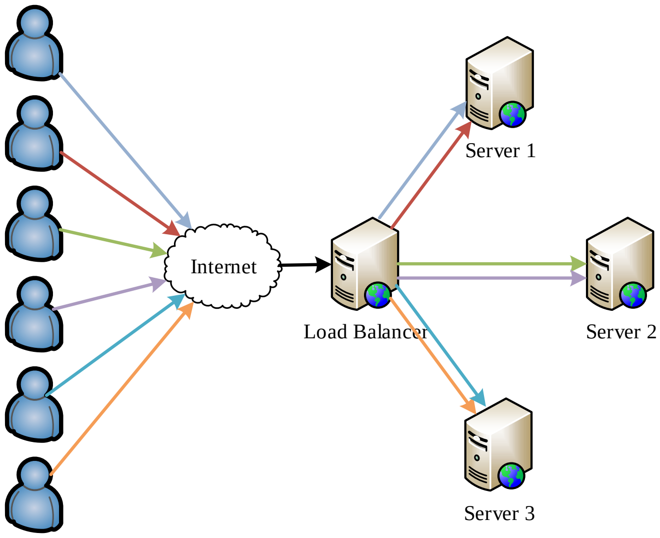


Fig. 2.1. *L3B System architecture*

The L3B balancer is designed with two main modules: *Resource Management Module (RMM)* and *Packet Management Module (PMM)* [25]. The main objective of the former is to provide new available endpoint resources according to the current clients' load and endpoint utilization. PMM balances the incoming packets according to the received information of RMM about available and utilized resources. This paper focuses on the implementation of the PMM module. Its design is presented in Fig. 2.2.

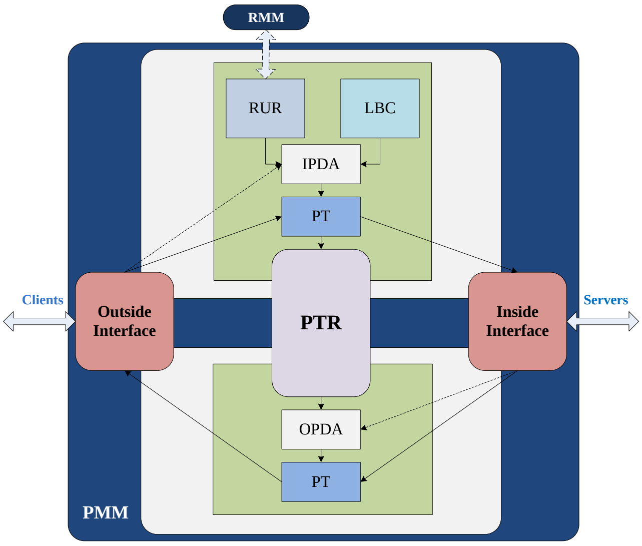


Fig. 2.2. *Design of the PMM module*

PMM manages the client's requests, i.e., it forwards an incoming packet to a particular endpoint by using some balancing technique. After the target endpoint server responds back, PMM forwards again the response to the client that has initiated the request.

When a client request is received at the Outside Interface (OI), it sends information to the Input Packet Decision Agent (IPDA). which decides where to forward the request (which endpoint server), by using relevant information by the Resource Utilisation Repository (RUR) and Load Balancing Configuration (LBC) about current utilisation of the endpoint servers and load balancing configuration, correspondingly. The realized decision is sent to the Packet Translation (PT) agent, which proceed with the IP packet header translation using the NAT/PAT (Network Address Translation / Port Address Translation). These translations are stored in the Packet Translation Repository (PTR) as they can be used to forward the response back to the client. Finally, the transformed packet is forwarded to the Inside Interface (II) in order to be forwarded to the target endpoint server. The response packets are transformed similarly by the PT, by using the decision of the Output Packet Decision Agent (OPDA), according to the information stored in PTR. More details can be found in [25].

The L3B was developed as a Java implementation according to the proposed architecture since modern Java virtual machines' performance is similar to C or C++ [14]. However, the results were not promising [21]. That is, introducing the L3B balancer reduced the performance even worse than the case without balancer. This motivated us to analyze various proposals how to make the L3B more efficient. The next section presents the improved L3B architecture that supports all functional requirements and provides better performance than its predecessor.

3. The new implementation. Since the naive implementation of L3B did not yield the expected results, we proceed to improve the L3B implementation. This section presents the development of the new improved implementation.

3.1. The new implementation's concept. Fig. 2.3 presents the new improved implementation, based on creation of virtual client socket. When a packet arrives at the L3B server, L3B creates a ”Virtual Client”,

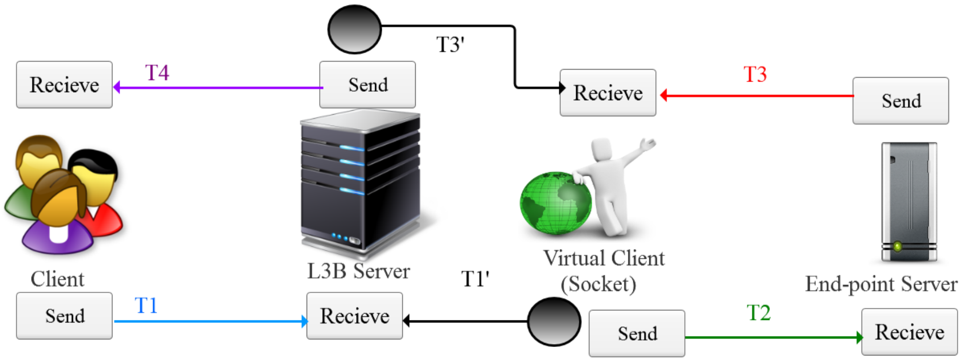


Fig. 2.3. *Improved L3B implementation*

which is a creation of a socket between new client on a certain port, and a port and IP address with the endpoint server, presented as T_1 in the figure. Now, the L3B takes the memory (packet) from the received data in the state and prepares for transmission on the server (T_1'). The next step is to forward the packet to the endpoint server, shown as T_2 in the figure. In the next step, denoted by T_3 in the figure, the server processes the request and returns the reply to the virtual client. Then the L3B prepares to send to the client everything that it has previously received and takes its memory (T_3 VirtualClient.Send = Balancer.Recieve). Further on, the final activity, denoted by T_4 is totally the same as previous case.

3.2. The implementation details. A robust, reliable and correct package transmitter is used from source to destination in our concept. Java Network Sockets features are extended in order to realize the load balancing concept.

The implementation is done in Java JDK 7.0, by using network sockets. The algorithm provides package forwarding using Sockets for TCP communication. It will be the same algorithm if a UDP communication is used, but in this case, the Datagram transfer should be used, instead of Sockets.

The details of the implementation are presented in Fig. 3.1. The L3B architecture is implemented with several classes:

- *Client* class, which sends the client's request to Balancer;
- *Balancer* class, which receives request from client;
- *ServerInput* class, which sends client's request to multiple servers;
- *ServerOutput* class, which receives the responses from the servers; and
- *PackageForwarder* class, which creates instances of ServerInput class and ServerOutput class as well as responds with output data to the client.

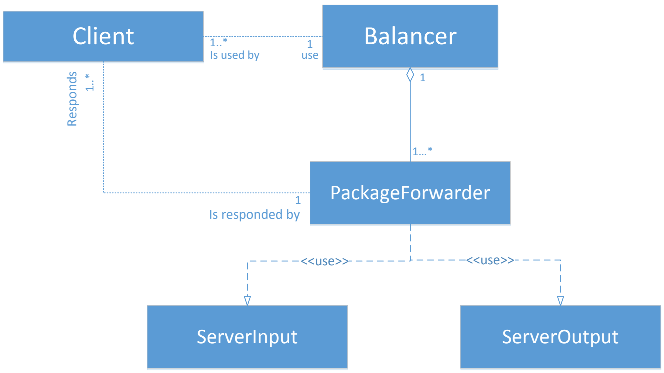


Fig. 3.1. *The details of the implementation*

4. Testing Methodology. This section presents the testing methodology that is used in the experiments to prove the L3B scalability. Series of experiments are conducted to determine the performance of the L3B balancer for various load and scaling.

The environment consists of up to 10 same servers, each with Intel(R) Xeon(R) CPU X5647 @ 2.93GHz with 4 cores and 8GB RAM, all connected on 1Gb LAN to exclude the network impact [10]. One server acts as a client, the second as a balancer while others up to eight servers act as endpoint servers. All servers are installed with CentOS 6.5.

Two different type of experiments were conducted:

- Proof of concept for horizontal scaling with an endpoint server that delivers static web page; and
- Proof of concept for horizontal scaling with a computation intensive load as an endpoint server.

4.1. Testing the L3B balancer for web page end point. A client requests a packet with constant web page content, which size is $56KB$. The different number of concurrent requests are then initiated to create various loads. We choose this page size in order to be smaller than the IP packet limit of 64KB. Usually, the web requests and responses are smaller than $64KB$.

The concurrent requests are simulated with SOAPUI. Each test case consists of sending N concurrent requests per second, such that N varies in each test case starting from $N = 5$ until $N = 100$ requests, by increasing N with step 5. Each test case lasts 60 seconds.

4.2. Testing the L3B balancer for computation intensive end point. The client server is installed with the client benchmark software. It sends an array of 300 integer numbers to the server's listener, which shuffles 10.000 times randomly chosen two elements of the array. The idea of this benchmark tool is to utilize only the resources on the server side, without generating huge network traffic, that is, a situation where the load balancer is applicable.

Total of 9 experiments are conducted with a different number of scaled resources. The first experiment consists of measuring the nominal performance of a single server (an unbalanced endpoint server), while the other 8 experiments are conducted by using $p = 1, 2, \ldots, 8$ servers with one L3B balancer in front of them. Each experiment consists of several test cases, each of which sends different load $N = 25, 50, 75, \ldots, 975, 1000$ (number of concurrent requests) by the client machine.

Several performance parameters are measured and calculated in each test case. *Response Time* T_iB is measured, where index i presents the number of scaled endpoint servers and B denotes that L3B is used in front of those endpoint servers. *Speed* $V_iB = N/T_iB$ expresses the number of handled requests per corresponding response time, while $S = T_1/T_iB$ expresses the *Speedup* achieved in the scaled system compared to the nominal one and $E_i = S_i/i$ expresses the corresponding $Efficiency$. The expected speedup according to the Gustafson's Law is linear ($S_i = i$), while the expected efficiency is 100% ($E_i = 1$).

Several L3B's behaviors will be checked. Firstly, the L3B latency will be checked, that is, the difference between response time T_1 of the experiment without L3B and response time T_1B of experiment with L3B that uses also one endpoint server. Secondly, the behavior of the L3B will be examined while loaded with a various number of requests, that is, to check the points where additional scaling should be used. Finally, the regions with the best performance will be analyzed.

5. Performance analysis of the L3B implementation with web page end point. This section presents the results of the experiments to confirm the successfulness of the new implementation for web page end point. It also discusses the speedup of the improved L3B implementation. We conduct two experiments based on a configuration using two endpoint servers

5.1. Results of the experiments for the L3B implementation with web page end point. Fig. 5.1 shows the results of the experiments that evaluated the performance of the new improved L3B implementation over the system which does not use L3B balancer. The results show that using the L3B is always better for each number of concurrent requests $N > 30$.

Let us explain why the scenario without the L3B balancer (4 CPU cores, 8GB RAM) is better comparing with the scenario with L3B (8 CPU cores and 16GB RAM) for $N \leq 30$, which uses double resources. Introducing L3B adds a constant delay per packet because a packet goes through the additional server (the L3B server) and virtual client (socket). For small server load N this delay is in the range of the server response time. But, when

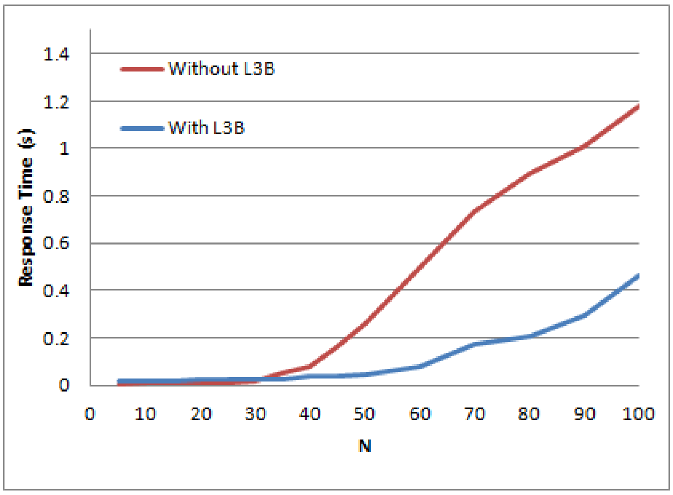


FIG. 5.1. *Results of the experiments for the L3B implementation*

the total load is increased, the single server in the first scenario (without L3B) will be over-utilised, which will decrease its performance while the second scenario (with L3B) will still work in the normal mode with better performance. The next section analyzes the scalability and achieved speedup by using the L3B balancer to balance the load among endpoints in more detail.

5.2. Theoretical analysis. According to the Gustafson's law [8], adding more computing and processing resources (CPUs) to some parallel or distributed system will improve its performance limited with linear speedup (the speedup that is equal to the number of scaled resources). The *Speedup* $S(p)$ is defined in (5.1) as a ratio of the response time of the nominal system T_1 (without L3B with 1 endpoint server) and the response time of the scaled system T_p (p endpoint servers and the L3B in front of them).

$$S(p) = \frac{T_1}{T_p} \tag{5.1}$$

Our experiment is conducted using L3B with $p = 2$ endpoint servers. That is, N client requests are sent to a single endpoint server (as described in Section 4), while N client requests are sent to two endpoint servers in the second scenario (with L3B), thus balancing the requests by $N/2$ per endpoint server.

5.3. Case study with scaling factor 2. Fig. 5.2 presents the speedup that the improved version of L3B achieves over the scenario without L3B. Three regions are observed:

- *Slowdown* region, where the speedup $S(p) = S(2) < 1$;
- *Sublinear* speedup region, where the speedup $S(p) = S(2) < p = 2$;
- *Superlinear* speedup region, where the speedup $S(p) = S(2) > p = 2$;

The slowdown region appears in the range $N < 30$ and the speedup in this region has a constant value. The servers in both scenarios are under-utilised, and the L3B reduces the system performance because it introduces a delay. When a load is increased, the first scenario is over-utilised, while the L3B balances the load and thus keeping both servers still in the normal working mode (going into Sublinear and Superlinear speedup regions). However, when the load reaches $N = 60$, that is, 30 requests per server, both servers become over-utilised and thus the speedup starts to decrease. Increasing the load $N > 100$ started to generate errors in the first scenario.

5.4. A superlinear speedup phenomenon. A superlinear speedup region is achieved in distributed environment. The superlinearity is a well-known phenomenon in the cache intensive algorithms [7] because of using more CPU cache memory in parallel implementations compared to the sequential, where the communication among CPUs is low. The superlinear speedup is achieved in the cloud virtual environment, despite the additional virtualisation layer [6, 20].

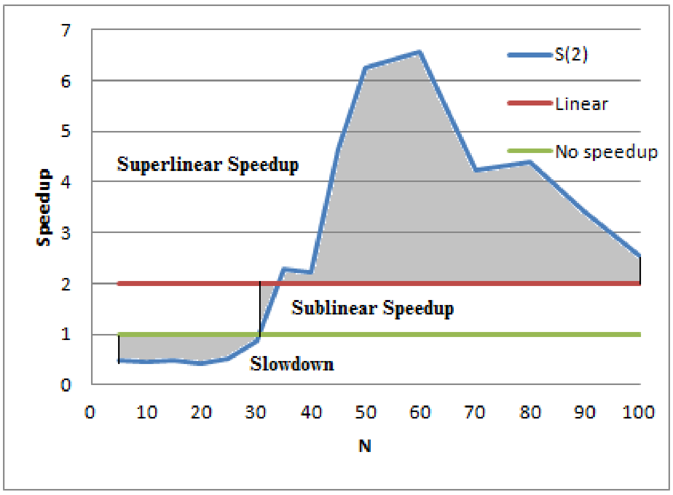


Fig. 5.2. *The achieved speedup of the improved L3B when using two endpoint servers*

However, the reason why the superlinear speedup region appears in this distributed environment is totally different. The nature of web servers is such that increasing the incoming requests dive the server into CPU over-utilisation, while balancing the load among more servers keep the servers in the normal mode, and thus generating the superlinear speedup, despite the delay that the L3B introduces.

6. Experimental Results for horizontal scaling with a computation intensive load as end point. This section presents the results of each test case of the experiments for all four test parameters: T, V, S and E.

Fig. 6.1 compares the response time of all experiments. As expected, the response time of scaled systems is lower that the nominal system with only one endpoint and without using load balancing. When comparing the T_1 and T_1B curves, one can observe that there is a region ($N < 550$) where $T_1B < T_1$, although it was expected that the L3B server will introduce additional latency. We explain this paradox with the fact that we retain the same session between the balancer and the endpoint, which reduces the utilized RAM memory of the endpoint server.

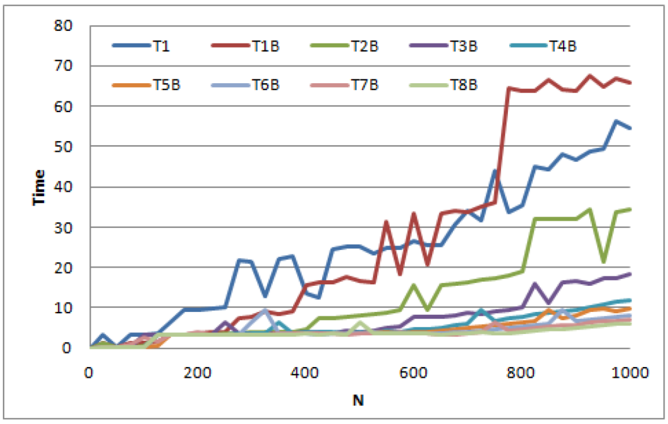


Fig. 6.1. *Execution time for different number of concurrent messages and servers*

The results for the Speed V_i are presented in Fig. 6.2. Three regions are observed for each experiment going from left to the right: A huge peak on the left, a rising region and then a falling region. The first region exists because the fact that was previously explained for the response time. After this peak, the linearly growing of the speed is observed, which also proves the scalability. The top of the speed is moved to the right for about $N = 100$ for each scaled system. The falling region appears for such load when the existing endpoint servers are not enough to handle the particular load, and in this case, these resources should be scaled.

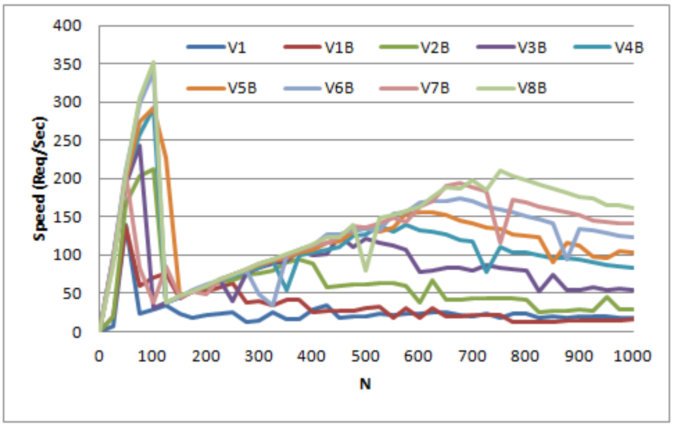


FIG. 6.2. *Speed for different number of concurrent messages and servers*

Fig. 6.3 presents the achieved speedup for scaled systems as a function of the number of messages. Similar regions are observed as those for speed for each scaled experiment compared to the experiment without the balancer. The scalability of the balancer is also observed, that is, the greater speedup is achieved while using greater scaling (more endpoint servers). Even more, the speedup is greater for a greater load. We observe that there is a superlinear speedup region for each experiment, such that each region is moved to the right compared to the experiment with a smaller number of end points.

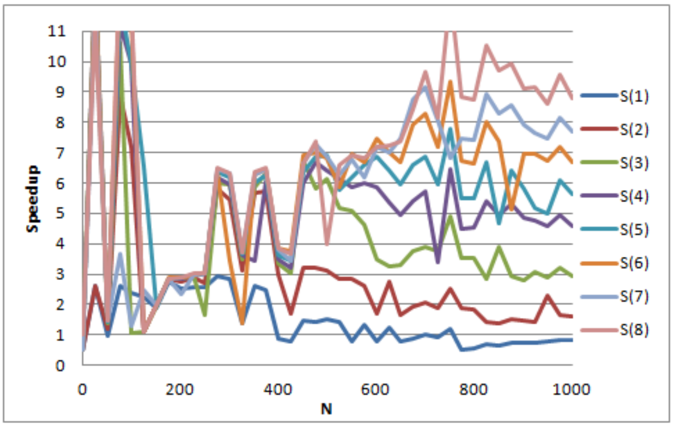


FIG. 6.3. *Speedup for different number of concurrent messages and servers*

The results of the last parameter E is shown in Fig. 6.4. This is the measure of the achieved speedup compared to the number of used scaled resources. That is, Fig. 6.4 shows which experiment achieves the greater performance price trade-off because the price for rented resources (VMs) of the most common cloud service providers is linear.

The Efficiency curves have two different regions: the left region, where mostly the experiments with smaller scaling show superlinear efficiency ($E(i) > 1$ and some region even $E(i) > 2$), and the right region where also superlinear efficiency is achieved, but now those experiments that use greater scaling.

7. Analysis and Discussion. This section discusses the benefits that the final implementation of L3B offers. The results that were presented in the previous section show that there are several additional benefits,

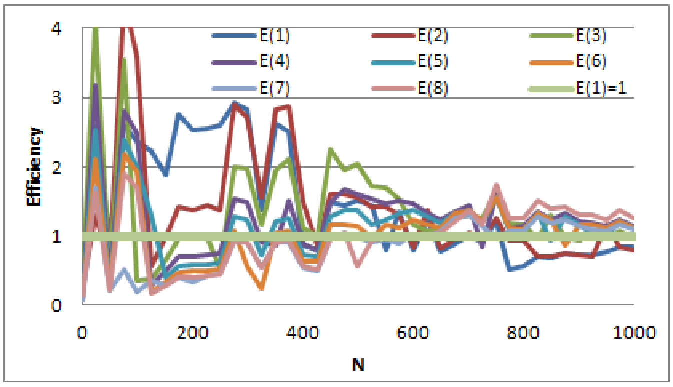


Fig. 6.4. *Efficiency for different number of concurrent messages and servers*

apart of its scalability, along with some further challenges.

The L3B latency is even smaller than the expected one, that is, there is a region where the server is even better by using the balancer in front of only one endpoint. The maximum achieved speedup is 2.93 by using the same endpoint server. This paradox exists because the clients open sessions with the balancer, while the L3B balancer keeps the firstly opened session to the endpoint server and sends the requests through it, which reduces the utilized resources at the endpoint server and improves its performance. Normally, this region ends when the endpoint server is over-utilized, when the performance is reduced and is smaller than the case without the balancer.

Another important benefit is the achieved speedup, which not only that proved the L3B's scalability, but there is a region of superlinear speedup in distributed environment. The superlinearity is a well-known phenomenon, such as the one in parallel systems for cache intensive algorithms [7]. These algorithms are executed in parallel systems that use more CPU cache memory compared to the serial hardware systems. Also, the communication among CPUs is low. Apart of the additional virtualisation layer, the superlinear speedup is also achieved in the cloud virtual environment [20].

However, the reason why the superlinear speedup region appears in this distributed environment is totally different. Ristov et al. [22] modeled the scalable web services by determining five regions, and one of them is the superlinear region. It means that increased number of the incoming requests dive the endpoint server into over-utilisation, while balancing the same client load among increased number of endpoint servers keeps them in the normal mode. Thus, it improves the overall performance and provides the superlinear speedup, despite the latency that the L3B introduces, which is also a positive factor in some region (explained previously).

The maximal achieved speedup is sometimes even as twice as the linear speedup. For example, maximal achieved speedup for the experiment with two endpoint servers is 5.80, rising to 12.26 for the scaling factor of eight. However, this rising of the maximal speedup does not follow the trend with the maximal efficiency, and shows a decreasing trend, from 2.9 for the experiment with two endpoint servers to 1.53 in tests with a scaling factor of eight.

The L3B has a limit in the network packet size. That is, the L3B scalability and additional benefits can be achieved only for the client-server systems where the requests do not exceed the maximal network (IP) packet size. Still, most services, for example, web services, do not exceed it. However, this feature will be implemented in the future and the performance will be measured again.

The testing environment (client-server model and the benchmark tool) is chosen such that L3B balancer will be applicable. Otherwise, for example, if the clients send huge messages with small computation or memory demanding, not only the L3B balancer, but all other centralized balancers would reduce the performance with their latency and bottleneck. However, these systems prefer using a DNS-based (Domain Name Services) balancing the load.

Skala et al. [26] discuss that Dew Computing, which is placed at the lowest level of the distributed computing hierarchy below the fog and cloud computing, offers information-oriented processing rather than data-oriented.

Cloud and fog computing operate on huge amounts of raw data, which is context-free while dew computing is context-aware giving the meaning of data that is processed. They suggest a new context layer in an extended OSI model on top of the application layer to cope with the necessities of the IoT. Our solution for a balancer works on a network level of the OSI model and fits in this scenario.

8. Conclusion and Future Work. The presented load balancing technique is intended for horizontal scaling of small scale servers, such as cloudlets and dew servers. Since the naive implementation of the L3B balancer did not yield the expected results, we have improved the implementation and redeveloped the improved L3B, which proved to be scalable with improved performance. Even more, a superlinear speedup region is achieved with the superlinear speedup up to 6.5 when scaling the resources by 2.

This paper presented the case study with introducing L3B in front of only two endpoint web servers, loading them with a single static web page. To test a wider domain of provision of dew services we conducted experiments with a larger number of endpoint servers with computationally intensive tasks. This case study also achieved superlinear speedup, with efficiency greater than two for a smaller load, and greater than 1.5 for those experiments that use greater scaling, which even more improve the contribution of our L3B balancer. We can conclude that the balancer is much better for computation intensive algorithms with a small latency, such as servers providing dew services.

This implementation of L3B promises a lot and thus it will be upgraded even more. An additional plan is to upgrade the intelligent balancer to balance the load keeping the endpoint web servers in the superlinear speedup region where the maximum speedup and the best performance are achieved. Therefore, this solution could be used by the cloud service providers to improve the price - performance trade-off for the scaled resources.

Also, this version of L3B works well only for requests (messages) that do not exceed the size of IP packet. We will upgrade the L3B balancer to support an arbitrary message size, and measure its scalability. In this direction, the RMM module should be implemented to become an elastic load balancer hosted in a cloud due to its scalability.

Additionally, further research will be towards using the L3B in the heterogeneous environment, that is, using the vertical and diagonal scaling, as well, since sometimes they achieve greater speedup than horizontal.

Acknowledgment. This work was partially financed by the Faculty of Computer Science and Engineering at the "Ss. Cyril and Methodius University", Skopje, Macedonia.

REFERENCES

[1] D. Ardagna, S. Casolari, and B. Panicucci, *Flexible distributed capacity allocation and load redirect algorithms for cloud systems*, in Cloud Computing (CLOUD), 2011 IEEE International Conference on, July 2011, pp. 163-170.

[2] A. Bahtovski and M. Gusev, *Cloudlet challenges*, Procedia Engineering, 69 (2014), pp. 704-711.

[3] A. Bhadani and S. Chaudhary, *Performance evaluation of web servers using central load balancing policy over virtual machines on cloud*, in Proceedings of the Third Annual ACM Bangalore Conference, COMPUTE '10, ACM, 2010, pp. 16:1-16:4.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, *Fog computing and its role in the internet of things*, in Proceedings of the first edition of the MCC workshop on Mobile cloud computing, ACM, 2012, pp. 13-16.

[5] K. Cvetkov, S. Ristov, and M. Gusev, *Successful implementation of l3b: Low level load balancer*, in Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on, May 2015, pp. 199-203.

[6] M. Gusev and S. Ristov, *Superlinear speedup in Windows Azure cloud*, in Cloud Networking (IEEE CLOUDNET), 2012 IEEE 1st International Conference on, Paris, France, 2012, pp. 173-175.

[7] M. Gusev and S. Ristov, *A superlinear speedup region for matrix multiplication*, Concurrency and Computation: Practice and Experience, 26 (2013), pp. 1847-1868.

[8] J. L. Gustafson, *Reevaluating amdahl's law*, Communication of ACM, 31 (1988), pp. 532533.

[9] S. Heinzl and C. Metz, *Toward a cloud-ready dynamic load balancer based on the apache web server*, in Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2013 IEEE 22nd International Workshop on, June 2013, pp. 342-345.

[10] M. B. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko, *Comparison of perfor- mance of web services, ws-security, rmi, and rmi-ssl*, J. Syst. Softw., 79 (2006), pp. 689-700.

[11] N. J. Kansal and I. Chana, *Cloud load balancing techniques: A step towards green computing*, IJCSI International Journal of Computer Science Issues, 9 (2012), pp. 238-246.

[12] A. Murua, I. Gonzalez, and E. Gomez-Martinez, *Cloud-based assistive technology services*, in Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on, Sept 2011, pp. 985-989.

[13] K. Nuaimi, N. Mohamed, M. Nuaimi, and J. Al-Jaroodi, *A survey of load balancing in cloud computing: Challenges and algorithms*, in Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on, Dec 2012, pp. 137-142.

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, Fourth Edition: The Hardware/Software Interface, Morgan Kaufmann, 2009.

[15] D. Petcu, G. Macariu, S. Panica, and C. Craciun, *Portable cloud applications - from theory to practice*, Future Generation Computer Systems, 29 (2013), pp. 1417-1430.

[16] D. Petcu and A. V. Vasilakos, *Portability in clouds: approaches and research opportunities*, Scalable Computing: Practice and Experience, 15 (2014), pp. 251-270.

[17] B. Radojevic and M. Zaga, *Analysis of issues with load balancing algorithms in hosted (cloud) environments*, in MIPRO, 2011 Proceedings of the 34th International Convention, May 2011, pp. 416-420.

[18] M. Randles, E. Odat, D. Lamb, O. Abu-Rahmeh, and A. Taleb-Bendiab, *A compara- tive experiment in distributed load balancing*, in Developments in eSystems Engineering (DESE), 2009 Second International Conference on, 2009, pp. 258-265.

[19] B. Rimal, E. Choi, and I. Lumb, *A taxonomy and survey of cloud computing systems*, in INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on, Aug 2009, pp. 44-51.

[20] S. Ristov and M. Gusev, *Performance vs cost for Windows and linux platforms in Windows Azure cloud*, in 2013 IEEE 2nd International Conference on Cloud Networking (CloudNet) (IEEE CloudNet'13), San Francisco, USA, Nov. 2013.

[21] S. Ristov, M. Gusev, K. Cvetkov, and G. Velkoski, *Implementation of a network based cloud load balancer*, in Computer Science and Information Systems (FedCSIS), 2014 Fed- erated Conference on, Sept 2014, pp. 775-780.

[22] S. Ristov, M. Gusev, and G. Velkoski, *Modeling the speedup for scalable web services*, in ICT Innovations 2014, A. M. Bogdanova and D. Gjorgjevikj, eds., vol. 311 of Advances in Intelligent Systems and Computing, Springer International Publishing, 2015, pp. 177-186.

[23] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, *The case for vm-based cloudlets in mobile computing*, Pervasive Computing, IEEE, 8 (2009), pp. 14-23.

[24] L. Schubert, M. Assel, and S. Wesner, *Resource fabrics: The next level of grids and clouds*, in Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 In- ternational Multiconference on, Oct 2010, pp. 677-684.

[25] M. Simjanoska, S. Ristov, G. Velkoski, and M. Gusev, *L3B: Low level load balancer in the cloud*, in EUROCON, IEEE, Zagreb, Croatia, 2013, pp. 250-257.

[26] K. Skala, D. Davidovic, E. Afgan, I. Sovic, and Z. Sojat, *Scalable distributed computing hierarchy: Cloud, fog and dew computing*, Open Journal of Cloud Computing (OJCC), 2 (2015), pp. 16-24.

[27] I. Stojmenovic and S. Wen, *The fog computing paradigm: Scenarios and security issues*, in Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on, IEEE, 2014, pp. 1-8.

[28] Y. Wang, *Cloud-dew architecture*, International Journal of Cloud Computing, 4 (2015), pp. 199-210.

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016

DOI 10.12694/scpe.v17i2.1158
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 91–102. http://www.scpe.org © 2016 SCPE

AN EXTENSIBLE SOFTWARE-AS-A-SERVICE SOLUTION FOR DISTRIBUTED INFRASTRUCTURES

JEDRZEJ RYBICKI AND BENEDIKT VON ST. VIETH[*]

Abstract. Distributed research infrastructures embrace resources, services, and users. From those, services display the highest churn rate. The reasons are twofold. First, new versions or new services emerge, gain popularity, and have to be provisioned by infrastructure operators. Secondly, the demand to make research reproducible inclines the users to share the software they used to obtain their results. Coming to grips with these high churn rates is not easy, but has the potential to speed-up scientific discovery.

In this paper we will describe an extensible Software-as-a-Service (SaaS) solution conceived to facilitate seamless exchange, evolution, and deployment of software services in distributed environments. In opposite to ordinary SaaS solutions, the portfolio of the software services in our approach is extensible. We implemented means for users and developers to make their services readily available in the distributed infrastructure with minimal overhead. In this process, not only the actual software is made available but also the knowledge of how to install and configure it is conveyed, thus running instances of the new service can be provided right away. We will share the experience gained during the implementation of the extensible SaaS solution for DARIAH-DE research infrastructure. It was used to provision on-demand instances of software used in digital humanities. Subsequently, the same solution was reused in a completely different context to provide on-demand instances of UNICORE Grid middleware for training and testing purposes. The operation of our extensible SaaS, yielded additional requirements and extensions. In particular, the need for monitoring and measuring the resource usage were identified and addressed. The presented insights can help to address the problem of exchange and provision of scientific software. Other distributed infrastructures can incorporate them to improve the scalability, maintainability, user-friendliness, and sustainability of their platforms.

Key words: service sharing, distributed systems, cloud computing

AMS subject classifications. 68M14

1. Introduction. Digital methods have become commonplace in modern science. Their popularity is amplified by the recent establishment of the so-called data-driven science, which is expected to provide scientific insights by applying software solutions on digital data. The obvious prerequisite for such approaches is the availability and accessibility of the data. A problem which is acknowledged and addressed (if yet-not-completely-solved). The less obvious problem is how to share the scientific software used for analyzing the data. We have presented the initial solution to this problem in our priory publication [23]. In this paper we will recap our experiences and show that the paradigm is also applicable outside of its original context.

Research software is often a custom-made solution, developed in the course of a research project as an after-effect of answering concrete scientific questions. Other researchers might, nevertheless, benefit from reusing it to solve similar problems or for the validation of the results by applying the software to a different set of data [14]. Sharing software has the potential to accelerate scientific discovery. But even more important is the fact that software which is not sharable is, in the long run, not sustainable. Scientific discoveries done with such a software are, in worst case, not reproducible and thus useless. We believe that sharing data and software is a corner stone for doing science today. In this paper we will brush aside the social challenges of sharing software and data, and concentrate on technical solutions to facilitate such sharing.

It should be stressed that open software repositories like `GitHub` or `sourceforge` do not solve the stated problem completely. The software is only usable when it runs. Installing, configuring, and satisfying the software dependencies often proves to be harder than expected. Especially, when it is conducted on a different platform than it was developed on. Furthermore the knowledge on how to perform these steps is often only present in the heads of the software authors or within the project it was developed for (e. g. closed wiki). Hence the first requirement for effective software sharing is the availability of the abstract, platform-independent, yet executable deployment descriptions. Such descriptions could be, for instance, system images, executable installation scripts, configuration management tools, or container-based solutions. We will discuss the applicability of these approaches in Sect. 2.

The proliferation of distributed research infrastructures [5, 17] adds a new dimension to the problem of sharing scientific software. Such infrastructures offer a quick and cost-effective way of sharing resources. They

[*] Forschungszentrum Juelich GmbH, Juelich, Germany ({j.rybicki, b.von.st.vieth}@fz-juelich.de)

excel at enabling access to data and often provide some compute resources. The later is offered either in a Infrastructure-as-a-Service or Software-as-a-Service manner (for the definition of the terms see [19]). None of these approaches are optimal from the user's perspective. IaaS solutions put quite a large burden on the users: they have to install, configure, and maintain not only the software they require but also the whole system (e. g. do system upgrades). Furthermore, the created instances cannot be easily shared: this remains in power of the user who created them and is usually technically challenging. The higher-level abstractions in form of Software-as-a-Service are much more attractive to researchers: The effort on their part is reduced and they can directly apply the software in their scientific endeavors. But the effort still exists: The installation work has to be done by the computer centers in the infrastructure. From software sharing perspective the problems are twofold. Firstly, due to the rapid development of digital methods, researchers constantly require new versions or new types of software in the SaaS portfolio. Secondly, the distributed research infrastructures span across multiple resource providers, thus high level of heterogeneity has to be coped with, making the installation harder. To this end, the demanded software deployment descriptions must be platform-independent and portable. To account for the constantly changing requirements, the descriptions should be executable. Finally, the system administrators must be able to audit them, to verify that they are correct, trustworthy, and secure. In this paper, we will show how the deployment descriptions can be used to build a extensible Software-as-a-Service solution, which allows for an easy extension of the offered software portfolio.

The motivation for looking into a particular software service can be manyfold. In many cases it can be just curiosity, a need for testing its applicability, or training purposes. Our solution is primary made for these applications. We will show in this paper that it can even provide on-demand instances of quite a complicated Grid middleware stack. They are configured to start, and provide basic functionality, making it possible to get an impression of how particular software works. The configuration might, however, not necessarily be sufficient for long-running, multi user, installations. There are examples in the literature of more sophisticated, higher-level approaches, e. g., [2] which can be used complementary to our extensible SaaS.

The high development pace of cooperative science together with the increasing importance of interdisciplinarity, bears one more use case for sharing research software: exchanging software between e-Infrastructures. This use case reinforces the previously mentioned requirement of making software deployment descriptions executable, auditable, portable, and sharable. The potential users of research software from different disciplines will have to invest some time to understand how to use it. The time should not be increased by forcing the user to understand how to install, and deploy the software.

This paper is structured as follows. Related work is discussed in Sect. 2. We provide a short introduction in the technology which we will use as a basis for the software sharing in Sect. 3. Section 4 presents how we used executable software deployment descriptions to implement an extensible Software-as-a-Service solution for DARIAH-DE. Section 5 provides a short summary of further applications and extensions of our solution. We conclude our paper with a summary in Section 6.

2. Related work. Let us first reflect on how software reuse can be achieved. In the first step, the scientific software has to be made discoverable. This step involves some social challenges, for instance willingness to share and use software, or efficient communication between the parties, involving software citation. In this paper, however, we focus on the technical challenges and assume that open-source repositories are sufficient for discovery. As stated above, the software is only useful if it can be applied to a new problem, and this is only possible when the software runs. Hence, the discovered software has to be installed. The process is conducted either locally on the researchers machine or (in a more modern fashion) on a remote machine running somewhere in the e-Infrastructure Cloud. The installation usually involves the configuration of the operating system (OS) and the provision of run-time environment, additional libraries, and actual software. The knowledge on how to achieve this might be, to some extent, conveyed in the service documentation. Experience shows, that the documentation is seldom up-to-date and does not cover all the corner cases encountered in the real-life deployments e. g., library dependencies or support for differing operating systems. When it comes to a migration of the service, the process must be started again, often by a different person. Even such a down-to-earth, recurring situation like the OS upgrade for the underlying machine might render a service unusable and result in the need for a re-installation. Since the process described above is quite laborious and error-prone, let us now discuss ways in which it can be either simplified or conducted in such a way that other researchers can use the software without repeating the installation process.

2.1. Virtual machines. One way of sharing “already-installed” software is to prepare images from which virtual machines can be spun off. The most obvious case for such applications are Clouds offering an IaaS interface [4, 20], although local deployments of images are conceivable as well. VM image preparation involves repeating the above steps in a virtual machine while accounting for special properties of the given Cloud. The later is often subsumed under the term: *contextualization.* When the image is created and uploaded to a Cloud image repository, it can be instantiated by other users to obtain running instance of the application. VM images suffer, however, under some limitations. First of all, images are black boxes: one has to trust the creator and has very limited capabilities of auditing the actual content of the image. Thus it is hard to safe-guard that the images do not include some malicious software or are not misconfigured so that running instances can be easily hacked or destroyed by malicious third parties. Virtual images include a complete software stack comprised of the required software, dependencies, operating system, and kernel. This results in a high overhead in terms of both image size (what makes the sharing of images harder) and performance. The inclusion of the operating system results in the need for periodic updates of the image: for instance to apply new security patches. The further problem with images is, that they are infrastructure-specific: they strongly depend on the hypervisor used and contextualization solution provided in the particular Cloud. Clouds include their own image repositories but moving images between the infrastructures, i. e., between different Clouds, remains an unsolved problem. A problem with obvious ramification for software sharing.

2.2. Configuration management tools. One way to create trustworthy and reproducible service deployments that are, to some extent, infrastructure, hypervisor, and OS agnostic, is to use configuration management tools like Puppet [21] or Chef [6]. With these tools it is possible to describe the service deployment in a declarative way. One defines which packages should be installed, which configuration changes should take place or which commands should be executed instead of applying all these steps manually. The tools can be used to provide a generic description of the service deployment that interested partners can apply to virtual machines or servers. Configuration management tools do not offer any isolation nor virtualization by themselves. Here the same limitations with respect to overhead of running VMs apply. On the upside, the descriptions are human-readable and can be reviewed and adjusted before the actual deployment, this contributes to their trustworthiness. Configuration management tools enable reproducible service deployments, but they fail to some extent at abstracting the underlaying platform, for this the developer has to spent additional effort.

One example for this is the deployment of an Apache HTTP server [3], a software often used to provide webservices. Listing 1 shows an excerpt from a deployment description that is used to deploy the software by installing (section `package` on Listing 1), configuring (`file`), and starting (`service`) it. Puppet brings an abstractions for package managers and it also provides wrappers for different `init` systems, but it has some limitations with respect to the different package, path, and service naming. To make them portable, one has to prepare deployment descriptions for all the supported platforms. The initial lines of the Listing 1 show how to cope with the fact that the same software is available under different names in RedHat and Debian repositories.

Listing 1
Puppet example for deploying Apache httpd

```
if $::osfamily == ’RedHat’{
  $user = ’apache’
  $name = ’httpd’
} elsif $::osfamily == ’Debian’ {
  $user = ’www-data’
  $name = ’apache2’
} else {
  fail("Unsupported os: ${::osfamily}")
}

file { "/etc/$name/$name.conf":
  owner => $user
  ensure => ’present’
}
package { $name:
  ensure => ’latest’
}
service { $name:
  ensure   => 'running'
}
```

We establish that tools like Puppet solve a slightly different problem. They are perfectly suited for managing the configuration (changes) of software running in a distributed infrastructure. For this purpose, they allow for configuration changes of the running machines (such changes are then automatically applied on the managed machines). The scientific software sharing, on the other hand, will be rather used to create quickly disposable instances of given services. The instances will be used, e. g., to verify the researcher's hypothesis or validate previous results and they will be discarded afterwards. We argue that, at least in some of the cases, there will be no need to manage the configuration changes in one instance, for the modified setup new instance will be spun off. Researcher will not be willing to spent much effort into the configuration management when it would be possible to simply, quickly, and cheaply create new instances. This assumption is especially true in Cloud environments. Since running instances costs money, they incentivize an approach of "run-and-discard."

2.3. Dynamic provisioning of research software. Another dimension of our work is the actual problem of software sharing and exchange regardless of the technology used. The problem is not new. The advent of affordable pay-per-use Cloud offerings in form of either public [4] or private Clouds [20], paved the way for many solutions to this problem. The works in this field cover a broad spectrum of subjects, starting from the general problem of reproducibility and repeatability, and how Clouds can help to alleviate it [11]. A lot of work was put into providing re-usable tools to support data-driven research, ranging from generic solutions like dedicated Linux distributions similar to bioKepler ones [29], up to Cloud-based provision workflow systems Galaxy Cloud [1]. The recent works on this field [2] concentrate on providing and executing data-intensive workflows. This work bears some similarities with our approach but operates on higher level of abstraction. From a technical point of view, on which we concentrate in this paper, there are no arguments preventing the usage of our solution to provide software packages constituting the workflows.

3. Container-based virtualization. Container-based virtualization solutions like Linux VServer [28] or Docker [9] became very popular for building, shipping, and running applications across many machines. They constitute the perfect basis for an efficient sharing of scientific software. In this section, we will first provide the reader with some basic information on how Docker works, and also argue why it better suits our use cases than the previously described system images and configuration management tools. The section will be concluded with a proposal on how the software sharing life cycle could be implemented in a distributed infrastructure.

3.1. Introduction to Docker. Docker is used as the basis for our software-sharing solution. This section will equip the reader with enough information about this technology to understand the rest of the paper. More details can be found in the extensive Docker documentation [9].

Docker is a lightweight, open-source, virtualization solution. In contrast to full-stack virtualization solutions, Docker images do not include a guest operating system kernel. It lowers the overhead in terms of both performance and size of the images, allowing near-native performance. Docker is based on mature Linux kernel technologies (`cgroups` and `namespaces`, among the others) to isolate independent application containers. Docker images use layered AuFS file system, enabling sharing libraries between images on one hand, and inspecting the changes done in the images, on the other. The later feature permits a rudimentary provenance tracking of images.

Let us now briefly discuss the Docker terminology and usage. Applications running with Docker are called containers, they are created from images. Images should be understood as hierarchical templates. It is possible to start with one image, add some software, and save the result as a new image. It is also easily possible to share images by using image repositories either private or public ones, like the `Docker Hub`. There are two main ways of shipping applications with Docker. In the first one (let us call it interactive), the developer starts a basis Docker image (e. g.: official Debian Linux image) and installs application, its dependencies, and everything else by issuing ordinary Linux commands. As soon as the application is installed, a snapshot of the running container can be created. Such a snapshot is, in fact, an image from which new containers can be created. An alternative way of creating images follows a declarative approach. The developer can describe the installation steps in a

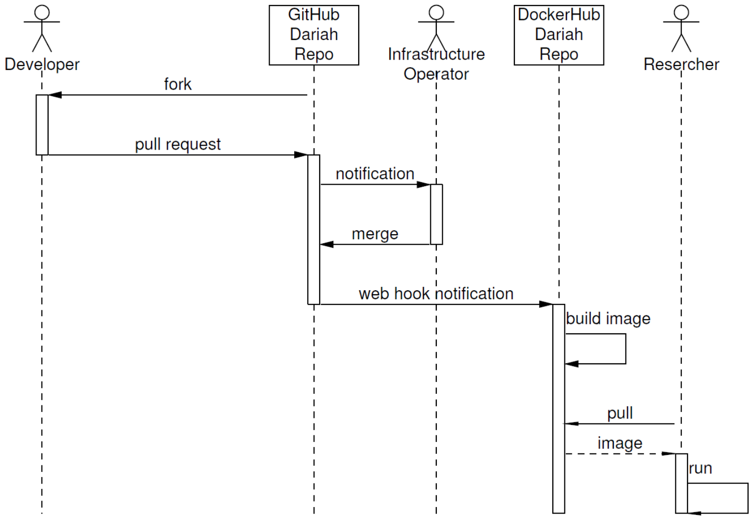


FIG. 3.1. *Proposed software sharing life cycle*

special `Dockerfile`. An image is created from such a description by issuing the `docker build` command. A generalization of the declarative approach are automatic builds. To setup an automatic build one has to publish the `Dockerfile` in a public code repository (like `GitHub`). Afterwards a web hook has to be created, so that each time the `Dockerfile` is modified, a new image build is automatically triggered. Regardless of the way in which an image was created, it is possible to view all the changes done in the image during the installation of the software. The automatic builds go an extra mile in terms of the trustworthiness of images: each user can review not only the content of the image but also the way they were created, since the description is publicly available.

Docker has a very important abstraction of data volumes which allows to separate software and data. The data are not included in the image, as it often happens in VM images. Data volumes, on the other hand, enable an easy injection of the data into the running containers. Hence the software in a container can be used for analyzing various sets of data. The separation of the software and the data diminishes one of the common barrier for sharing software: Researcher might be afraid of leaking out the data in the software they publish.

Docker is available on almost all Linux-based platforms, there is a support for MacOS and recently announced support on Windows platforms. Since the images encapsulate the application and all its dependencies, starting new containers on a new system is like starting a statically linked, self-contained program: no modification in the host system are required.

3.2. DARIAH-DE software sharing life cycle. The first application of our extensible SaaS solution took place in the DARIAH-DE distributed, research infrastructure. This use case is pretty typical for software sharing, thus explaining it in detail will help the reader to understand the trade-offs of software sharing, real-world applicability of Docker, and the design of the actual extensible Software-as-a-Service which we present later in this paper. Most of these points are relevant also outside of the DARIAH context.

DARIAH, the Digital Research Infrastructure for the Arts and Humanities, aims to enhance and support digitally-enabled research and teaching across the arts and humanities. It comprises a number of national initiatives, there is also a Germany-based effort, DARIAH-DE [5], where the described work was conducted. Efficient software sharing is essential for digitally-enabled research of its users. Also sharing software between infrastructures is becoming relevant in this context [16]. Software developed in one part of DARIAH might be passed over to and reused by other national efforts.

The technology for creating executable and deployable descriptions of software must be integrated into the scientific workflows and play well with existing e-Infrastructures. On Fig. 3.1 we present how such an integration could be conducted. We differentiate between three roles: developers, infrastructure operators, and researchers.

We have also two central components: a code repository, where the deployment descriptions are stored, and `Docker Hub` repository where the images are stored and pulled from.

The process of making scientific software available kicks off with the developer wanting to advertise and make his software available. He can propose the deployment description by follow typical `GitHub` cooperation workflow. It includes forking of the official `GitHub` repository of DARIAH-DE Docker files [13], adding a new description (or updating an existing one), and creating a pull request.[1] The operators of the repository are notified about the pull request. They can review the proposed solution, adjust it when required, or even reject it if it does not work. The DARIAH-DE `GitHub` repository is connected to the `Docker Hub` via web hooks. In short, this mechanism triggers an HTTP request to the `Docker Hub` each time the `GitHub` repository is modified. Upon such a request, a new build of the provided Dockerfile is started. The infrastructure operators can view the details of the build and act accordingly in case of a failure. Successfully built images are published into the official DARIAH-DE image repository [10] in the `Docker Hub`. The researchers can explore the repository, select the software they are interested in, pull the images and start it on their machines. Pulling and starting requires just one command (`docker run`). The `Docker Hub` allows for storing additional information about the software. The developer can describe the typical applications or provide a link to a more extensive documentation.

The workflow described above can be repeated multiple times, e. g., each time a new version of the software is released. Moreover, the roles can be distributed or handed-over: Both repositories allow for registration of organizations. Such organizations can have multiple administrators. Hence it is not required to designate just one infrastructure operator but rather a group of people can be delegated for this task. Similarly, all the `GitHub` users can fork, and modify the Dockerfiles. The trustworthiness of the images is guaranteed by the reviews conducted by the operator who has to accept the pull requests. Of course, each researcher can view the publicly available descriptions. The images (and Dockerfiles) are publicly available and they can be used by the researchers from outside of DARIAH-DE. All interested parties can found Dockerfiles in DARIAH-DE `GitHub` repository [13] and Docker images can be retrieved from DARIAH-DE `Docker Hub` [10].

Fig. 3.1 depicts the simplest workflow, which ends with a local deployment of the software. It is sufficient to explain how the workflow works but clearly the local deployment is not always the optimal solution. In the next section we will show how an extensible SaaS solution can be build, based on the proposed workflow.

4. Implementing extensible Software-as-a-Service. So far we have shown that Docker provides a means for a fast deployment of software and how `Docker Hub` and `GitHub` can be used to define a workflow for sharing researcher software in a decentralized yet trustworthy manner. In the optimal case, the user has to issue just one command (`docker run`) to obtain a running instance on her system. The optimal case requires some changes on the users machine, at least the installation of Docker is required. Even if easily possible, the local deployments are not always the best option. One reason for the remote deployments would be efficiency: remote instances can run close to the resources they require. Many data-driven projects require long-lasting software runs. This is hard to perform on the local researcher's machine. Fortunately, e-Infrastructures like DARIAH-DE provide resources which can be used by the researchers for this purpose.

In this section we will show how the availability of deployable software description (as those provided by Docker) can facilitate an extensible Software-as-a-Service solution. This service will offer all the benefits of the SaaS approach (i. e. very low effort is required to use scientific software shared by other researchers) whilst allowing for quick and easy extensions of the software portfolio. We present our prototype solution which runs on low-level compute resources already available in the DARIAH-DE infrastructure. Our goal was a system which provides the researchers with an instance of the software she requires in just one request through a web page.

4.1. Actors. Fig. 4.1 depicts the architecture of the implemented extensible Software-as-a-Service. It includes lots of loosely-coupled components. Messaging is used for exchanging information between components in an asynchronous way. In production we use a `amqp`-based product RabbitMQ [22]. Let us now briefly discuss the roles and responsibilities of the single actors of our architecture.

The *Facade* is the entity facing end-users. It is an entry point from which the commands (like create a new instance of a given software product) are issued. The Facade is stateless and it is possible to run multiple

[1] Readers who are not familiar with the `GitHub` commands are referred to the documentation [12]

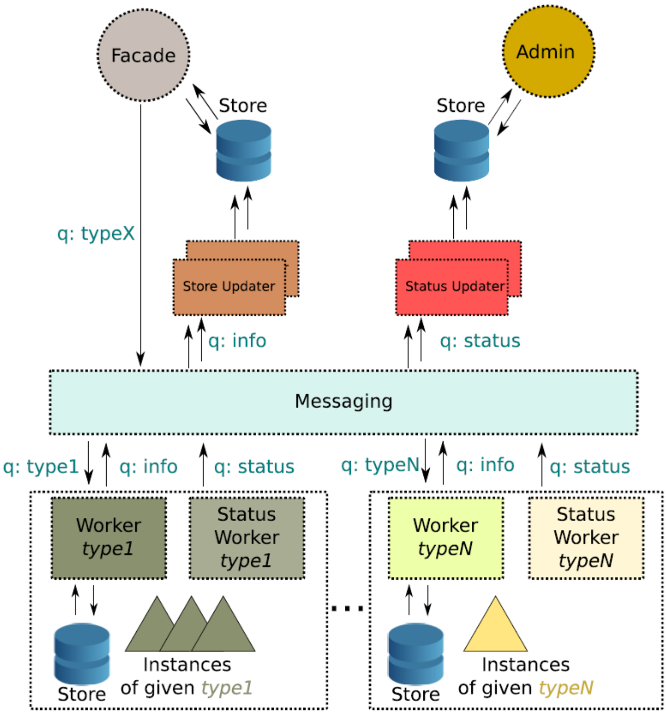


Fig. 4.1. *Architecture of the extensible Software-as-a-Service*

Facades simultaneously. They can offer interfaces of different types, e. g., a html-based for browser access or a json-based programmatic API.

The Facade has access to the messaging bus and a read-only access to the *Store*. Store is a system-wide cache with information about:

1. types of service registered in the system,
2. instances running in the system.

The content of the cache is updated by the *Store Updaters*. These entities subscribe for information about creation of new instances and registration of new types. Upon reception of such messages, Store Updaters write the information in the Store. It should be stressed that the content of the Store itself can be lost as it will be periodically republished. Store is solely used to speed-up the response times.

The *Workers* embody the main functionality of the system. Those are the entities which can manage instances of given service types (e. g. instances of `eXist` or `neo4j` databases). Each Worker upon its creation (and later periodically) informs other components about its capabilities (i. e. supported service type) and its current state (i. e. managed instances). Workers receive commands from users (via Facade) through the messaging system. Next section will be devoted to the communication problem and includes example message exchange.

4.2. Communication with workers. The critical part of our system is the communication between the components. It is dynamic, asynchronous, and subject-based. We required the possibility to change the system capabilities during the run-time. In particular, to add new software to the portfolio of the offered Software-as-a-Service. This is done by adding (or removing) Workers. To become active in the system, Worker needs access to the messaging bus. The incoming Worker publishes its “specialty” (i. e. instance type that it can provide) and subscribes for the messages in this instance-type-queue (like create new instance, delete instance, etc.). Listing 2 shows the initialization message published by a worker able to create and manage instances of `neo4j` graph database. Both the name of the worker (`neo4j`) and the description field are visible to the user. The message also includes a status field (and a human readable status message), a timestamp, and (empty in this case) environment field. The later one is a feature connector, which can be used to request additional information required to create new instances. An example would be a request for the password value for the database.

Listing 2
Worker initialization message

```
{
    "available": true,
    "name": "neo4j",
    "description": "neo4j is a graph database",
    "status": "Worker available",
    "ts": 1447977918.137527,
    "environment": []
}
```

As can be seen on Fig. 4.1, for each instance type supported by the system, there is a separate queue. When a user requires a new instance of a given type it sends a message to the respective queue (via Facade). An example of such a message is presented on Listing 3 and is pretty self-explanatory. Client is able to give instances its own ids by which the instances can be referred to in the future, in this example we used a UUID algorithm to generate such an id.

Listing 3
Create instance message

```
{
    "subject": "create_instance",
    "instance": {
        "type": "neo4j",
        "id": "e6da7ab7-9076-41d7-b0c2-c724a92712ab",
        "environment": []
    }
}
```

The message is routed to the proper Worker, it creates an instance and sends a response to the messaging system. This response contains details required for connecting to the just-created instance of a concrete service. Such details typically include IP address, protocol and port, username and password. An example of a message sent by a `neo4j` Worker in response to a request like presented above is shown on Listing 4. We see that `neo4j` is accessible via both, HTTP and HTTPS. A user can directly connect to her new service instance via provided URL.

Listing 4
Instance created response

```
{
    "subject": "instance_info",
    "instance": {
        "type": "neo4j",
        "id": "e6da7ab7-9076-41d7-b0c2-c724a92712ab",
        "status": "created",
        "created": 1447978020.125227,
        "ts": 1447978920.137527,
        "urls": [
            "http://134.194.198.39:9011",
            "https://134.194.198.39:9010"
        ],
        "environment": []
    }
}
```

Workers republish their state periodically (i. e. instance types that they support and state of managed instances). The timestamps included in the messages are used to implement soft-state status management, workers which do not published their status for a longer time will be consider not active or overloaded. A worker has also a possibility to gracefully leave the system by publishing an explicit non-active status message.

Each worker has a dedicated queue for the incoming commands and all workers share a common response queue (the `info` queue on Fig. 4.1).

4.3. Implementation and operation. We have created a prototype of the extensible Software-as-a-Service called DARIAH Meta Hosting. The implementation was done in Python and uses a set of frameworks and well-known products: Flask for the web front end, MongoDB as storage, and RabbitMQ for messaging. We offer two interfaces: one is a human-friendly web site, the other is a programmatic json-based API. We envision the later one to be used to script automatic deployments of software, e. g., in data processing workflows.

One feature of our design was the division of the system into small, autonomous parts connected by messaging. The advantage of such a design was the ability to change the parts independently, for instance to scale up. We use Docker for deploying single parts and the Docker-based orchestration tool `Docker Compose` [7] to create test deployments including all the parts required. An example of such a deployment of our system is show on Listing 5. It can be used to deploy a testing system (or a production system, after some modifications) on every research infrastructure supporting Docker. Docker Compose is also able to scale out the system by spinning off additional instances. In this self-test, the solution we use for the software sharing was successfully applied to the software we have implemented. Our prototype is deployed on a OpenStack Infrastructure-as-a-Service cloud offered in DARIAH-DE. So far the complete system is placed in one data center, but it should be possible to extend it beyond this one entity. This would only require to make the messaging system accessible from the outside and deploy additional workers in different data centers.

Listing 5
Docker Compose description of a test deployment

```
messaging:
  image: rabbitmq:3.4
db:
  image: mongo:latest
  command: --smallfiles
autho:
  image: httpprincess/authorizator:latest
updater:
  image: httpprincess/store-updater:latest
  links:
    - db
    - messaging
neoworker:
  image: benedicere/metahosting-worker
  links:
    - messaging
  environment:
    - DB_LOCAL_COLLECTION=local-instances
    - WORKER_NAME=neo4j
    - WORKER_DESCRIPTION=neo4j is a graph database
    - WORKER_IMAGE=dariah/neo4j:latest
  volumes:
    - ./neo4j-worker.ini:/app/dockerworker.ini
facade:
  image: httpprincess/http-facade:latest
  links:
    - db
    - messaging
  command: python /app/client.py
```

For our prototype we have implemented a generic Docker worker. The deployment of Docker-based instances is usually conducted in the same fashion, regardless of what software is confined in the image. Hence the Worker is initialized with a name of the Docker image containing the software it will be responsible for (see Listing 5). We create an instance of Worker for each software package offered in the extensible SaaS. To scale out the system it is also possible to have multiple workers offering the same type of the software. The requests will be then divided among the workers in a Round-robin fashion, allowing for fair load distribution.

At this stage it should be stressed, that the proposed architecture is extensible. The Workers abstract the technology used for creating the instances. It would be perfectly possible to write a worker which would use

Puppet-based deployment descriptions or any other technology that might come around in the future.

An important feature for the operation of the service was the integration with the DARIAH-DE Authentication and Authorization Infrastructure (AAI). It is based on Shibboleth with project specific Attribute Providers. Flask does not provide a native means for Shibboleth-based authentication, therefore a new plug-in had to be developed. The effort paid out since it potentially opened the extensible SaaS also beyond the group of DARIAH-DE users. There is a Europe-wide federation of Shibboleth-based AAI Infrastructures called eduGAIN [18]. With the current solution it is possible to allow all members of this federation to access the DARIAH-DE Meta Hosting. This means that potentially all European researchers can use it to share their software. This point was crucial for DARIAH-DE since (as already explained) it is part of bigger project federation of national DARIAH efforts.

The integration with the DARIAH-DE AAI solution was important also for the sake of integrating Meta Hosting with other DARIAH-DE services. By sharing common authentication and authorization mechanism it would be possible to offer a seamless user experience of using multiple services at the same time. An example of such a usage would be an analysis of digital texts collected in the TextGrid repository [26] with the Voyant [24] tools running in the Meta Hosting as implemented in DARIAH-DE DigiVoy tool [8]. It should be stressed that this paper is written from the perspective of the resource provider, the actual use of the resources (i. e. the research tasks tackled on the provided resources) are part of a different domain and could be subject of a future work.

Another extension that was implemented for operational reasons was the status monitoring part. The astute reader may notice some elements on Fig. 4.1 which were not yet explained. In particular elements called: Admin, and Status Worker. The later entity is responsible for monitoring local resource usage and status of Workers. The collected information is passed through a dedicated messaging queue to the Status Store, from which it is presented to the infrastructure operators. All in all, it is very similar to the information flow in case of instance creation and management. There is, however, an important difference. The information in the Status Store are persistent and should not be lost. The reason is simple, the information might be used for accounting and infrastructure debugging purposes. The administrator is able to track down the resource usage and usage induced by single users. Some pieces of this information are also send to the central monitoring system of DARIAH-DE to inform users about the health status of the Meta Hosting service.

5. Extending SaaS. A test of the extensibility of our solution was to provide more complex software through the extensible SaaS solution. We have tested this scenario with a Grid middleware. Uniform Interface to Computing Resources (UNICORE) [25] is a ready-to-run Grid system to make distributed compute and data resources accessible in a seamless way. It has a broad spectrum of applications, for instance, the US-based infrastructure project XSEDE [30], or Human Brain Project [15]. The far-reaching flexibility of the middleware comes at the cost of not-straight-forward deployments, resulting also from the number of components they contain. The installations in the aforementioned big infrastructures are long-running and have to be tailored to specific requirements and resources. Although it would be possible to at least support such installations with Docker images, our use case was a little bit different. There are situations where users want to “train” with the middleware, without the risk of depleting resources provided by the infrastructures, also the infrastructure operators prefer to get the users familiar with the system before allowing the actual access to the scarce resources. The extensible SaaS can be applied exactly for such training purposes. It allows a quick provision of a preconfigured UNICORE middleware, giving the users a possibility to learn how to use it and discard the instances afterwards.

The prerequisite for the inclusion of the UNICORE into extensible SaaS was the availability of the Docker images. As already explained, Docker is a lightweight virtualization solution to provide images with executable services inside. The typical philosophy of the tool is to provide one image for one service. UNICORE is composed of number of service components. Also the way UNICORE is configured (lots of XML files) and how the services mutually authenticate to each other (with x.509 certificates) makes it hard to use with Docker. Luckily, it allows an all-in-one installation which should not be used for production deployments but is sufficient for testing and training purposes. It was used to create the Docker images [27]. Solely, the availability of the proper image was sufficient to include UNICORE in the portfolio of the SaaS. We have created new deployment of the extensible SaaS which can be used for training purposes. We also reuse the Shibboleth-based authentication plug-in, this time the users were not limited to the DARIAH-DE user base.

6. Conclusion. The two main problems addressed in this paper were scientific software sharing and efficient, seamless, user-friendly access to resources in distributed infrastructures. Sharing of research software is the central component of reproducible data-driven science. We have argued that the problem is manyfold. It involves both technical and social challenges. We focused on the former ones. To this end, we firstly reviewed the technologies which could be used for software sharing. We discussed the software sharing with the selected technology in the context of e-Infrastructure: DARIAH-DE. The presented software sharing workflow is based on established and well-known technologies. It achieves a high level of trust in a decentralized environment.

In the second part of this paper we have shown how the presented software sharing solution can be used to leverage an extensible Software-as-a-Service system. It enables a quick and easy deployment of the software on IaaS resources available in the e-Infrastructure. To underpin the extensibility of the proposed architecture we show that it can be applied beyond its original purpose of sharing software in digital humanities. Based on DARIAH-DE Meta Hosting we have created a test bed of the UNICORE Grid middleware for training purposes. We shared the experiences gained during the implementation and operation of the working prototypes.

Our observation is that both addressed problems: efficient access to resources in distributed infrastructures and user-friendly software sharing intervene to high extent in case of DARIAH-DE. The workflow for software sharing produces artifacts which can be instantly deployed on the available resources with help of the extensible SaaS. For the second discussed use case of dynamic provisioning of UNICORE instances, this is not the case. There is no sharing workflow. It shows that the extensible SaaS is compatible with different sharing approaches. By contrast the sharing workflow can be used also beyond the extensible SaaS, for instance, to create service instances on local resources.

Acknowledgment. The work on Meta Hosting was partially funded by the German Federal Ministry of Education and Research (BMBF) under the project DARIAH-DE (fund number 01UG1110A-M).

REFERENCES

[1] E. Afgan, D. Baker, N. Coraor, H. Goto, I. M. Paul, K. D. Makova, and J. Taylor, *Harnessing cloud-computing for biomedical research with Galaxy Cloud*, Nature Biotechnology, 29 (2011), pp. 972–974.

[2] E. Afgan, K. Krampis, N. Goonasekera, K. Skala, and J. Taylor, *Building and provisioning bioinformatics environments on public and private clouds*, in MIPRO 15: 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, May 2015, pp. 223–228.

[3] *Apache HTTP server project.* `http://httpd.apache.org/`. [Online; accessed: 2015-12-20].

[4] *Amazon Web Services.* `http://aws.amazon.com/`. [Online; accessed: 2015-12-20].

[5] T. Blanke, M. Bryant, M. Hedges, A. Aschenbrenner, and M. Priddy, *Preparing DARIAH*, in IEEE 7th International Conference on E-Science, Dec 2011, pp. 158–165.

[6] *Chef.* `https://www.chef.io/`. [Online; accessed: 2015-12-20].

[7] *Docker Compose.* `https://www.docker.com/docker-compose`. [Online; accessed: 2015-12-20].

[8] *DIGIVOY.* `https://de.dariah.eu/digivoy`. [Online; accessed: 2016-01-29 (in German)].

[9] *Docker.* `https://www.docker.com/`. [Online; accessed: 2015-12-20].

[10] *DARIAH-DE Docker Hub Account.* `https://hub.docker.com/r/dariah/`. [Online; accessed: 2016-01-27].

[11] J. T. Dudley and A. J. Butte, *In silico research in the era of cloud computing*, Nature biotechnology, 28 (2010), pp. 1181–1185.

[12] *GitHub.* `https://github.com/`. [Online; accessed: 2015-12-20].

[13] *Official DARIAH-DE GitHub Account.* `https://github.com/DARIAH-DE`. [Online; accessed: 2016-01-27].

[14] C. Goble, *Better software, better research*, IEEE Internet Computing, 18 (2014), pp. 4–8.

[15] *Human Brain Project.* `http://www.humanbrainproject.eu/`. [Online; accessed: 2015-12-20].

[16] M. Hedges, H. Neuroth, K. M. Smith, T. Blanke, L. Romary, M. Kster, and M. Illingworth, *TextGrid, TEXTvre, and DARIAH: Sustainability of Infrastructures for Textual Scholarship*, Journal of the Text Encoding Initiative, (2013), pp. 3–13.

[17] D. Lecarpentier, P. Wittenburg, W. Elbers, A. Michelini, R. Kanso, P. Coveney, and R. Baxter, *EUDAT: A new cross-disciplinary data infrastructure for science*, International Journal of Digital Curation, 8 (2013), pp. 279–287.

[18] D. López, *eduGAIN: Federation interoperation by design*, in TERENA Networking Conference, May 2006.

[19] P. Mell and T. Grance, *The NIST definition of cloud computing*, Tech. Report 800-145, National Institute of Standards and Technology, 2011.

[20] *OpenStack.* `http://openstack.org/`. [Online; accessed: 2015-12-20].

[21] *Puppet.* `http://puppetlabs.com/`. [Online; accessed: 2015-12-20].

[22] *RabbitMQ.* `http://www.rabbitmq.com/`. [Online; accessed: 2015-12-20].

[23] J. Rybicki and B. von St Vieth, *DARIAH Meta Hosting: Sharing software in a distributed infrastructure*, in MIPRO 15: 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, May 2015, pp. 217–222.

[24] S. Sinclair and G. Rockwell, *Voyant Tools*. `http://voyant-tools.org`. [Online; accessed: 2016-01-29].

[25] A. Streit, D. Erwin, T. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and P. Wieder, *UNICORE: From project results to production grids*, in Grid Computing The New Frontier of High Performance Computing, vol. 14 of Advances in Parallel Computing, Elsevier, 2005, pp. 357–376.

[26] *TextGrid Repository*. `http://textgridrep.de/`. [Online; accessed: 2016-01-29].

[27] B. von St. Vieth, *UNICORE Docker Image*. `https://hub.docker.com/r/benedicere/unicore/`. [Online; accessed: 2016-01-29].

[28] *Linux-VServer*. `http://linux-vserver.org/`. [Online; accessed: 2015-12-20].

[29] J. Wang, D. Crawl, and I. Altintas, *A framework for distributed data-parallel execution in the Kepler scientific workflow system*, in ICCS 12: 1st International Workshop on Advances in the Kepler Scientific Workflow System and Its Applications, 2012, pp. 41–42.

[30] *Extreme Science and Engineering Discovery Environment*. `http://www.xsede.org`. [Online; accessed 2015-12-20].

Edited by: Enis Afgan
Received: December 21, 2015
Accepted: March 31, 2016

DOI 10.12694/scpe.v17i2.1159
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 103–114. http://www.scpe.org © 2016 SCPE

CLOUDFLOW - ENABLING FASTER BIOMEDICAL PIPELINES WITH MAPREDUCE AND SPARK

LUKAS FORER [§,*] ENIS AFGAN[†] HANSI WEISSENSTEINER[*§] DAVOR DAVIDOVIC[‡] GUENTHER SPECHT[§]
FLORIAN KRONENBERG[*] AND SEBASTIAN SCHOENHERR[*]

Abstract. For many years Apache Hadoop has been used as a synonym for processing data in the MapReduce fashion. However, due to the complexity of developing MapReduce applications, adoption of this paradigm in genetics has been limited. To alleviate some of the issues, we have previously developed Cloudflow - a high-level pipeline framework that allows users to create sophisticated biomedical pipelines using predefined code blocks while the framework automatically translates those into the MapReduce execution model. With the introduction of the YARN resource management layer, new computational processing models such as Apache Spark are now plugable into the Hadoop ecosystem. In this paper we describe the extension of Cloudflow to support Apache Spark without any adaptions to already implemented pipelines. The described performance evaluation demonstrates that Spark can bring an additional boost for analysing next generation sequencing (NGS) data to the field of genetics. The Cloudflow framework is open source and freely available at https://github.com/genepi/cloudflow.

Key words: Apache YARN, Pipeline Framework, Spark, Cloud Computing

AMS subject classifications. 68M14

1. Introduction. Since the advent of high-throughput technologies in the field of molecular biology (i.e. Next Generation Sequencing (NGS)), a growing amount of data is produced and needs to be analysed. Thus, molecular biology has evolved into a big data science, where the bottleneck is no longer the production of raw data in the laboratory, but its subsequent analysis and interpretation. Due to the variety of data, users need to carefully select the suitable processing framework that fits their data structure and processing task best [8]; further, the size of the data makes analysis parallelization desirable. Fortunately, a large number of conceptual approaches exist on how to deal with the data boost [14]. One promising approach for efficient data parallelization is Apache Hadoop with its YARN (Yet Another Resource Negotiator) architecture [15]. Within Hadoop, users can focus on the functional parallelization of their problem while benefiting from the scalable Hadoop architecture stack in the background. However, writing Apache Hadoop applications requires custom code development and domain expertise, which has led to poor adoption of this parallelization model in biomedical research. More specifically, the MapReduce model is quite restrictive requiring users to break an existing workflow into a number of *map* and *reduce* steps, often a challenging task. Additionally, the reusability of the map and reduce functions are limited, resulting in a use-case specific implementation and therefore time-intensive solution for every problem. To alleviate some of the pressing issues, we developed Cloudflow [6] - a framework that simplifies pipeline creation in biomedical research, especially in the field of genetics. Cloudflow supports a variety of NGS data formats and contains a rich collection of built-in operations for analyzing such kind of datasets (e.g. quality checks, mapping reads or variation calling). The main concept behind our approach is to break complex data analysis steps into three basic operations. All further use-case specific operations are built by implementing or extending one of the basic operations. Pipelines are then composed by creating a sequence of these operations. The framework itself translates the set of pipeline operations into one or more jobs and decides which of the operations are executed in the map or the reduce phase. Thus, Cloudflow hides the complexity and the implementation details of MapReduce jobs, allowing scientists to build pipelines with an intuitive method.

Initially, Cloudflow provided a framework to reuse existing Hadoop blocks for MapReduce. With the rise of the previously mentioned Apache Hadoop resource manager YARN [15], new non-batch processing models such as Apache Spark can also be run within a Hadoop cluster. The success of Apache Spark can be seen within new projects such as Adam [9], VariantSpark [11], SparkSeq [16] and especially the ongoing initiative on porting

[*] Division of Genetic Epidemiology, Medical University of Innsbruck, Innsbruck, Austria (lukas.forer@i-med.ac.at)

[†] Department of Biology, Johns Hopkins University, Baltimore, MD, USA

[‡] Center for Informatics and Computing, Ruder Boskovic Institute, Zagreb, Croatia

[§] Institute of Computer Science, Research Group Databases and Information Systems, Innsbruck, Austria

GATK to Spark (GATK 4). However, similar to MapReduce, writing Spark pipelines can be a challenging task that prevents domain experts from using such models in their daily work. In the future, it is likely that Apache Spark will substitute MapReduce as the general-purpose processing engine of Hadoop. To support this transition, we extended Cloudflow to support Spark and evaluated three genetics data-analysis pipelines that have been developed on both MapReduce and Spark engines. The results demonstrates that our contribution helps reduce the development time and increases reusability of the code over different use cases and even on different data processing engines.

2. Apache Hadoop. Apache Hadoop is an initiative for distributed computing on a cluster infrastructure. It is well known for providing an open-source implementation of the batch processing model MapReduce, initially developed by Google in 2004. In 2013, Apache Hadoop introduced a new resource management system - YARN - that allows multiple data processing models to be integrated on the same cluster, allowing different processing models to share common underlying resources. In YARN, the MapReduce processing library is now referred to as MapReduce 2 (MR2). MR2 uses the required distributed file system (HDFS) as the default data location for data input and output.

Compared to MapReduce 1 (MR1), the two main components (JobTracker, TaskTracker) have been reorganized: The *JobTracker* has been broken up into three services (ResourceManager, Application Master, JobHistoryServer). The *ResourceManager* is a YARN service to run applications (e.g. MapReduce, Impala or Spark). The *Application Master* is started for each job and includes all parts to manage e.g. a MapReduce job. The *JobHistoryServer* is responsible for providing information about already finished jobs. Instead of using a *TaskTracker* to start individual tasks, the YARN *NodeManager* service manages all resources on one node and starts individual *containers* (former tasks). As mentioned, this setup provides multiple processing models within the same cluster, including MapReduce and Spark.

One of the important changes within YARN is how resources are managed. Within MapReduce, the amount of concurrent tasks on each node have been specified separately for map and reduce (*mapred.tasktracker.map.tasks.maximum*, *mapred.tasktracker.reduce.tasks.maximum*). Now with YARN, one can only specify the amount of cores on each node (*yarn.nodemanager.resource.cpu-vcores*) and the amount of available memory (*yarn.nodemanager.resource.memory-mb*), independently of map and reduce. Using these two parameters, YARN runs the amount of appropriate containers per node. Since Cloudflow provides support for both the MapReduce and Spark processing model, they will be introduced shortly in the coming paragraphs.

2.1. MapReduce. The aim of MapReduce is to develop a simple and scalable method to process large datasets on several machines in parallel [5]. The main idea behind this distributed programming model is that a long-time calculation is split into a *map* and a *reduce* phase which contains all the logic behind the calculation and is specified by the user. The underlying framework itself takes care of parallelization, task scheduling, load-balancing and fault-tolerance. Due to its simplicity, MapReduce is used to solve many scientific problems where large-scale computing is needed. Moreover, MapReduce is ideal for parallel batch processing of terabytes of input data. The data-flow of a MapReduce program consists of several steps, where only the map and reduce function are problem-specific and the other steps are loosely coupled with the problem and generalized. In the first step, the input data set is split into key/value pairs and a user defined map function is executed for each pair:

$$map(key, value) \rightarrow list((key_i, value_i)), where_i = 0...n$$

The map function reads a pair, performs some problem-specific calculations and produces zero or n intermediate key/value pairs for each input pair. In the next step, the intermediate key/value pairs are grouped by similar keys and a merged list of all values for this key is created. Finally, the user-defined reduce function is applied to the intermediate key/list pairs:

$$reduce(key_i, list(value_i)) \rightarrow list(outvalue)$$

The values created by the reduce function are the final outputs of a MapReduce job.

2.2. Apache Spark. The cluster computing framework Spark was initiated at UC Berkeley AMPlab in 2009 and is an Apache top-level project since 2014 [17]. Spark can run in standalone mode, or over existing cluster managers like Apache Mesos or YARN. Therefore, no Hadoop cluster is required as long as a shared file system is provided (e.g. NFS). In addition to the batch data processing model, streaming data processing model is supported by using a micro-batch model. Currently Spark offers four main libraries: Spark SQL and Dataframes for working with structured data, Spark Streaming for fault-tolerant stream processing, a machine learning library called MLlib, with a collection of algorithms already provided and an API for graph processing called GraphX.

Spark's main differentiation to MapReduce is the in-memory cache based on the Resilent Distributed Datasets (RDD) concept. Thereby it has lower launching overheads compared to MapReduce [16]. While MapReduce is ideal for batch processing of large datasets, Spark profits from RDDs caching and is therefore able to handle tight-coupled problems faster than MapReduce. If the data fits in the memory, Spark outperforms MapReduce. In contrast to the two-stage disk based Map and Reduce phase, Spark allows multi-stage in-memory primitives, based on RDDs transformation and action operations. While *map* can be modeled with a lazy transformation that calls a function for each element in a data set and returns another RDD, *reduce* is an action operation triggering a computation (aggregation) and sending either all the RDDs elements back to the main program (*driver*) or write them on the disk. The RDD's operation concept is represented in Figure 2.1.

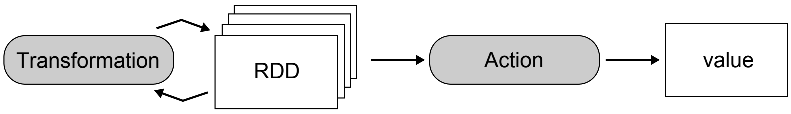


Fig. 2.1. *Apache Spark's RDD Operations.*

Spark based software in the genomic data analysis is currently evolving, by exploiting the increased memory of cluster systems. First implementations of the Smith-Waterman local sequence alignment approach was realized within SparkSW [18]. VariantSpark [11] employs Spark MLlib machine learning library using the k-means clustering for population structure deconvolution.

2.3. Related Work. To simplify the implementation of MapReduce jobs, Apache Pig has been introduced as a higher level interface [12]. It is based on HDFS and MapReduce and allows a fast implementation of data flows with already available data operations such as *join*, *filter* or *group by*. Data flows are specified in the Pig Latin language that describes a directed acyclic graph (DAG) and defines how data should be processed. A further advantage of Apache Pig is the ability to check the data flow on optimizations, e.g. if two grouping statements can be combined. The costs to write code in Apache Pig are lower than setting up a Java project and implementing the corresponding MapReduce functions. But, as stated earlier, Apache Pig includes only a limited number of operators and writing Apache MapReduce jobs directly in Java yields advantages in speed compared to Apache Pig.

The pipeline framework FlumeJava [4] is based on the concept of immutable parallel collections. This kind of data-structure can be used to process and analyze their items in parallel. The end user can either use one of the predefined functions or can combine them with its own. Each function is implemented as parallel for-each loop, which is then translated by the framework into a series of MapReduce jobs. During the translation, the framework itself decides if such a function should be executed locally (i.e. sequentially) or remotely (i.e. parallel). Apache Crunch (https://crunch.apache.org/) is a freely available open source implementation of FlumeJava.

3. Cloudflow. Cloudflow [6] is a framework to simplify the creation of analysis pipelines by encapsulating complex data analysis steps as simple operations. This approach helps hide the complexity and the implementation details of complex data-parallel pipelines. Moreover, the concept of using basic operations increases the reusability and enables testing of the operation logic on a local workstation by using existing unit testing frameworks.

Since Cloudflow used initially Apache MapReduce for pipeline execution, it offers parallel data processing, data reliability and fault tolerance out of the box. This fact is especially important in the field of Cloud Computing, where infrastructure often relies on commodity hardware and nodes can fail on a regular basis (e.g. due to miss-configuration or hardware failures). At the same time, the architecture of Cloudflow is independent from MapReduce. As we show in this paper on the example of Apache Spark, it provides parallelization constructs and abstraction interfaces that can be used to extend the system by implementing other parallel programming models (see Figure 3.1).

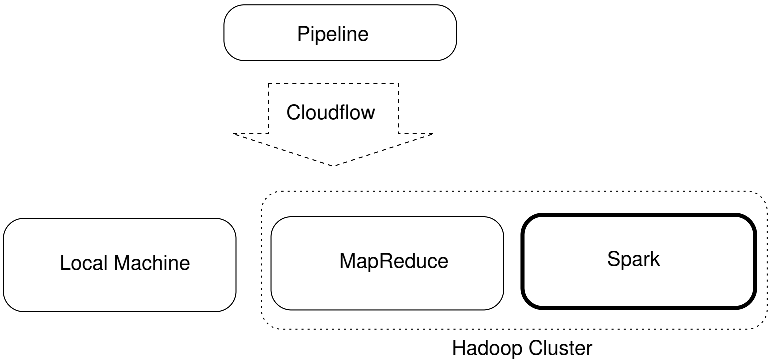


Fig. 3.1. *The architecture of Cloudflow enables to test the pipeline during development on the local machine and to execute the same pipeline on a Hadoop cluster for big data analaysis.*

Instead of developing a new declarative language for the pipeline composition, we developed a clear Java API. The proposed framework implements different patterns to speed up pipeline creation, to be extensible, and to support test-driven pipeline development. The following section gives an overview on the abstraction, explains the basic operations in detail, and shows how pipelines are created.

3.1. Data Types and Basic Operations. Cloudflow operates on records consisting of a key/value pair, whereby different record types are available (e.g. *TextRecord*, *IntegerRecord*, *FastqRecord*). A loader class is responsible for loading the input data and converting it into an appropriate record type. As mentioned earlier, Cloudflow supports three different basic operations: *transform*, *summarize*, and *group.* These are used to analyze and transform records.

The **transform operation** is used to analyze one input record and to create between 0 and n output records. The user implements the computational logic for this operation by extending an abstract class. This child class provides the function implementation that is then executed by the Cloudflow framework for all input records in parallel:

```
Class MyTransformer extends Transformer {
    public void transform(Record) {
       doSomethingInParallel();
       emit(new Record());
    }
}
```

The **summarize operation** is used on a list of records, whereby records with the same key are grouped. Thus, the signature of the process method has the key and a list of records as an input:

```
Class MySummarizer extends Summarizer {
   public void summarize(Key, List<Record>) {
       doSomethingInParallel();
       emit(new Record());
    }
}
```

The **group operation** is a special operation, which takes a list of records as an input and creates a Record group with the same key. The Cloudflow framework automatically inserts a group-operation between a transform- and a summarize-operation. This ensures, that output records of the transform operation are compatible with the input records of the summarize operation. The group-operation is realized by using the shuffle phase of a MapReduce job or $reduceByKey$ transformation in Spark.

Based on these operations, the user defines pipelines by building sequences of operations (see Section 3.3); a pipeline must start with a transform operation, all further operations are optional and can be used in arbitrary order.

3.2. Extended Operations. Complex operations are built by combining one or more basic operations. We already implemented several standard operations that are helpful for the analysis of text or numerical data records:

- Filter: this operation is a special transform operation, which emits the record to the subsequent operation iff a user-defined condition is fulfilled.
- Split: the transform operation calculates for each input record a new split level (i.e. a new key). This key is used by the group operation to create chunks, which can then be analyzed by a user-defined summarize operation.
- Aggregation (sum, mean): This defines a group operation followed by a summarize operation to aggregate all values with the same key (e.g. calculates the mean of all values). One record with the aggregated value is then emitted to the subsequent operation.
- Executor: this summarize operation writes all grouped values into a file on the local disk. It is then used as the input of an external UNIX command line program. Based on the lines of the output file, new records are created and emitted to the subsequent operation.

Since Cloudflow's operations are based on the Composite pattern, all these extended operations can also be used as a basis for new operations. In addition, this enables to split complex operations into several sub-operations, which improves testing and maintenance.

3.3. Pipeline Composition. The user builds pipelines by connecting several operations with compatible interfaces. For this purpose the Cloudflow framework implements the Builder pattern, which enables (a) building complex pipelines, (b) providing type safety and (c)support for implementing domain specific builders (see Section 4.1). In addition, the Builder pattern ensures that only a valid sequence of operations can be created (e.g. after the group-by operation, a summarize operation has to be added).

```
Class LineToWords extends Transformer {
   public void transform(TextRecord rec) {
    String[] words = rec.getValue().split()
    for (String word: words){
      emit(new IntegerRecord(word, 1));
    }
   }
}
pipeline.loadText(input)
        .transform(LineToWords.class)
        .sum()
        .save(output);
```

FIG. 3.2. *WordCount example using Cloudflow.*

To help the user and accelerate the pipeline composition process, Cloudflow comes preconfigured with a set of useful operations. This has the advantage that even the default WordCount example can be broken down into a few simple operations and is defined in a single line of code (see Figure 3.2). In the first step, a text file is loaded from HDFS (loadText). Then, for each record (i.e. line of input) the application-specific *LineToWords* operation is executed, which splits the line into words and creates a new record for each word. In the last step, a predefined sum operation is executed. The operation extends the pipeline by a group-by operation and a summarize-operation in order to sum up all the values for a certain key. For frequently used operations (e.g. sum, mean or count), we created special builder functions, which extend the pipeline and improve the code readability by keeping the code simple.

3.4. Pipeline Execution on MapReduce. Before the execution, Cloudflow checks the compatibility of input and output records of consecutive operations. This ensures that only valid and executable pipelines are submitted to the cluster.

If the pipeline is executable and valid, then the operation sequence is translated into an execution plan. The execution plan specifies if an operation is executed in the map or in the reduce phase. Based on this plan, Cloudflow creates one or more MapReduce jobs and configures them to execute the user-defined operations in the correct order. In this translation step, Cloudflow tries to minimize the number of MapReduce jobs by combining consecutive transform operations and by executing all transform operations after a summarize operation in the same reducer instance (see Figure 3.3).

For additive summarize operations (e.g. sum), Cloudflow takes advantage of Hadoop's combiner functionality. The idea of this improvement is to combine the key/value pairs that are generated by all the map tasks on the same machine into fewer pairs. Thus, the number of pairs that are transferred between mapper and reducer are minimized, which results in a positive effect on the network bandwidth since unnecessary communication is avoided.

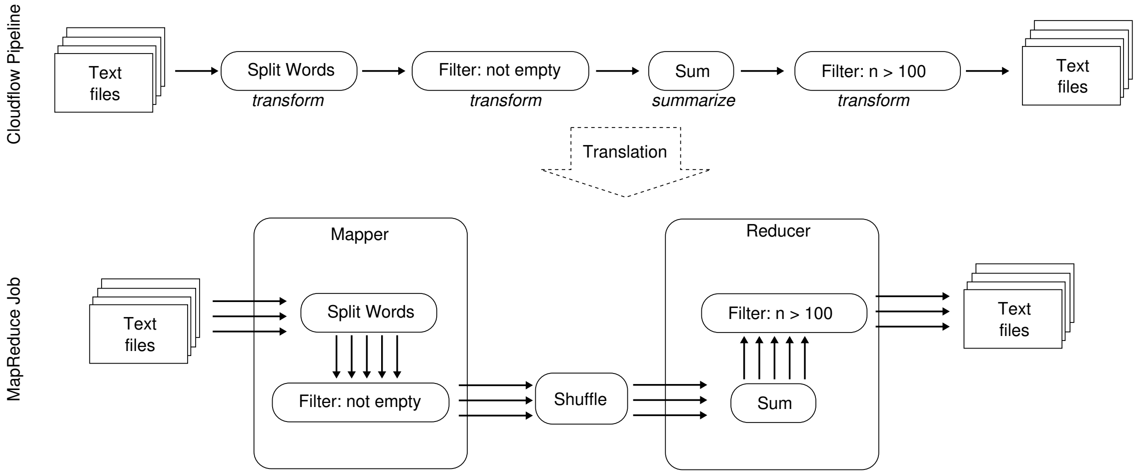


FIG. 3.3. *Cloudflow translates the operation sequence automatically into an executable MapReduce job.*

3.5. Pipeline Execution on Spark. Since the architecture of Cloudflow is independent from MapReduce, the provided parallelization constructs and abstraction interfaces can be used to extend the system by using Spark as an alternative parallelization framework. Thus, we implemented a new pipeline executor that takes a Cloudflow pipeline as input and translates the execution plan to a sequence of transformations. This transformations are using the Spark API in order to operate on Resilient Distributed Datasets (RDD) and take advantage of in-memory technologies.

The main task of this process is to combine all consecutive transform operations and to execute them step by step in the *flatMapToPair* stage of a Spark job. Next, the summarize operation defines the logic behind the *groupByKey* operation. In the last step, all transform operations are then executed using Spark's *map* transformation. Thus, the created Spark job needs two stages to execute a single Cloudflow pipeline (see Figure 3.4).

To optimize the execution time of the generated Spark job, Cloudflow tries to minimize the number of memory intensive *groupByKey* operations. This is achieved by using *reduceByKey* for most summarize-operations or by mapping this operations to the equivalent build-in transformation of Spark (e.g. count, sum, mean).

3.6. Bioinformatics Support. Cloudflow provides a variety of already implemented utilities, which facilitate the creation of pipelines in the field of Bioinformatics (especially for NGS data in Genetics). For that purpose, we implemented, based on HadoopBAM [10], several record types and loader classes in order to process FASTQ, BAM and VCF files. Moreover, we created several operations and filters for the analysis of biological

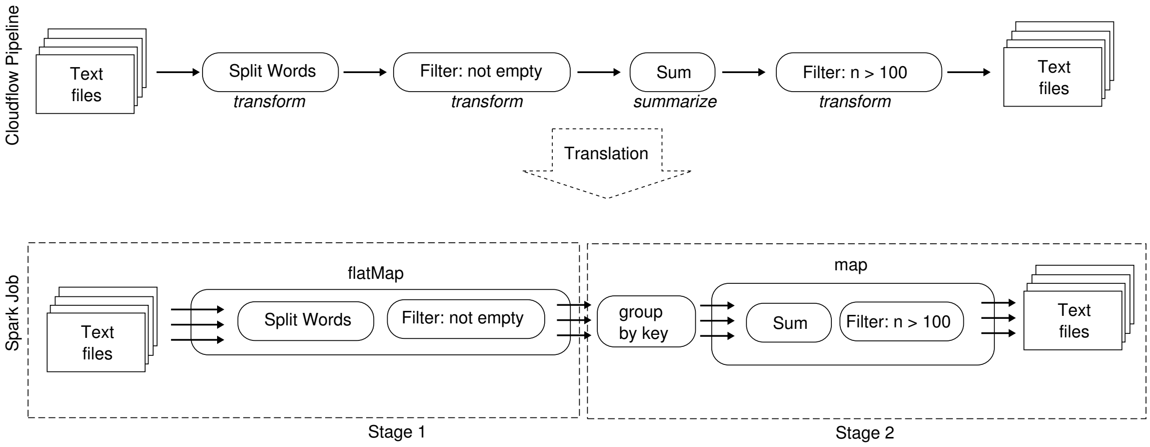


Fig. 3.4. *The Spark pipeline executor takes a Cloudflow pipeline as input and translates the execution plan to a sequence of Spark transformations.*

datasets. For example, a typical quality control pipeline for VCF files can be implemented by simply combining several built-in operations. First, we apply predefined filters to discard variations that are monomorphic, marked as duplicates, or are Insertions or Deletions (InDels). For all records passing the filters, Cloudflow applies a summarize operation that calculates the call rate for each variation (see Figure 3.5).

```
Class CallRateCalc extends Transformer {
  public void transform(VcfRecord record) {
    VariantContext snp = record.getValue();
    float call = callRate(snp);
    emit(new FloatRecord(snp.getID(), call);
  }
}

pipeline.loadVCF(input)
        .filter(MonomorphicFilter.class)
        .filter(DuplicateFilter.class)
        .filter(InDelFilter.class)
        .transform(CallRateCalc.class)
        .save(output);
```

Fig. 3.5. *VCF Quality Control Pipeline using Cloudflow.*

3.7. Deploying Pipelines as a Service. Cloudgene [13] is a web-based platform to create and execute workflows consisting of Hadoop YARN, Apache Pig and command line-based programs. It can be seen as an additional layer between Hadoop and the end-user that hides the complexity of the framework. Therefore, Cloudgene is the perfect candidate to provide Cloudflow pipelines as a service. Such pipeline can be integrated into the workflow platform by utilizing Cloudgene's plugin interface. No adaptation to the source code is needed, while only a simple plain text file including a header, input parameters, output parameters and the definition of the workflow itself need to be created. When launching Cloudgene, the manifest file is loaded and the client interface is automatically rendered using information from the file. As Cloudgene supports different technologies, it is possible to parallelize the calculations using Cloudflow and to visualize the results using R.

Cloudflow requires a compatible Hadoop YARN cluster for executing pipelines. CloudMan [2, 3] makes it possible to easily procure and configure a virtual compute cluster on a cloud infrastructure. The procured platform delivers a dynamically scalable Slurm and Hadoop cluster along with a number of higher level bioinformatics applications. With its ability to be launched and managed via a web browser on a number of clouds, customized as necessary, and easily shared with collaborators, CloudMan makes it possible to readily utilize cloud resources in a research environment. The approach on how Cloudgene and CloudMan can be combined efficiently has already been demonstrated [7], paving the path for readily executing Cloudflow pipelines.

4. Evaluation. As shown in [6], Cloudflow has practically the same execution time compared to Apache Crunch and the overhead between a Cloudflow pipeline and a plain MapReduce job is negligible. To evaluate the here presented extension, we implemented three different data-analysis pipelines using Cloudflow. The results of our experiments demonstrate that Spark brings a speed-up in the execution time compared to MapReduce. The following sections describe the experiments in detail. For more extensive pipeline examples, take a look at the source code repository at https://github.com/genepi/cloudflow. All the experiments have been executed on a 6 node physical cluster (5 cores & 32 GB RAM each) running the latest Cloudera Version (CDH 5.1.1) and Hadoop YARN 2.6.

4.1. WordCount. We extended our previously introduced WordCount example from [6] and executed it with different input datasets to compare the execution times of the existing Cloudflow MapReduce implementation with the new Spark implementation (see Figure 4.1).

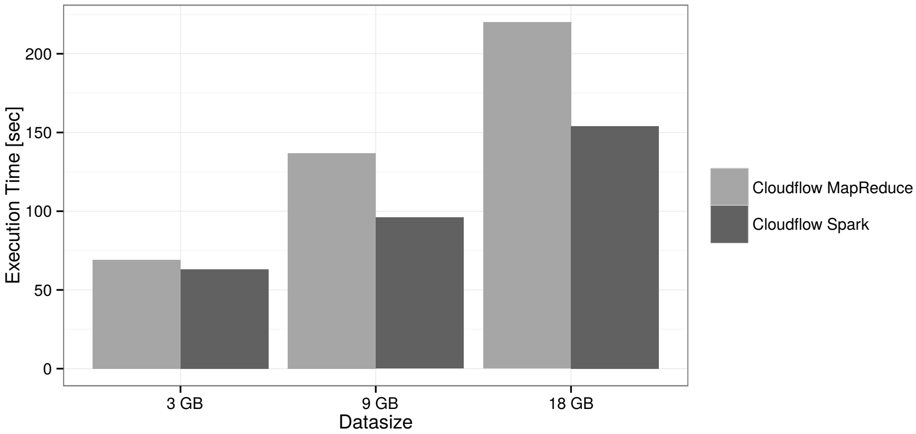


FIG. 4.1. *Execution time of the WordCount use case*

The results of this initial experiment show that overall the new Spark executor is faster than the existing MapReduce implementation. Moreover, the results demonstrate that the Spark WordCount pipeline implemented within Cloudflow scales well with the amount of input data.

4.2. Quality Control of Sequencing Data. When working with NGS data, the quality of raw data needs to be checked before subsequent downstream analysis (e.g. read mapping/alignment) can be achieved. Factors such as read errors, insertions or deletions of bases must be considered that finally results in the most accurate genome position for each read. The mean base quality per position is an important metric to get an overview about sequencing data quality.

The parallelization of the pipeline is realized by writing a transform operation, which calculates a quality value for each base of the input read. Next, a summarize operation is used to calculate the mean value of all qualities for a certain position (see Figure 4.2).

We executed the pipeline with different BAM files taken from the 1000 Genomes Project [1] to test and validate the pipeline with real-world input data. Three different datasets were used: (1) sample HG00097 high coverage chromosome 11 [1 GB]; (2) sample HG00096 high coverage whole exome [8 GB] and (3) sample HG00096 low coverage whole genome [15 GB]. The results of this experiment show that the new Spark executor has a speed-up of 2 compared to the MapReduce implementation (see Figure 4.3).

```
class GetQuality extends Transformer<BamRecord, IntegerRecord> {
    public void transform(BamRecord record) {
    for (int pos = 0; pos < (record.getSequenceLength(); pos++) {
        emit(new IntegerRecord(pos, (record.getQualityAtPos(pos));
    }
}
BioPipeline pipeline = new BioPipeline("Check Base Quality");
pipeline.loadBam(input).apply(GetQuality.class).mean().save(output);
```

FIG. 4.2. *BAM Quality Control Pipeline using Cloudflow.*

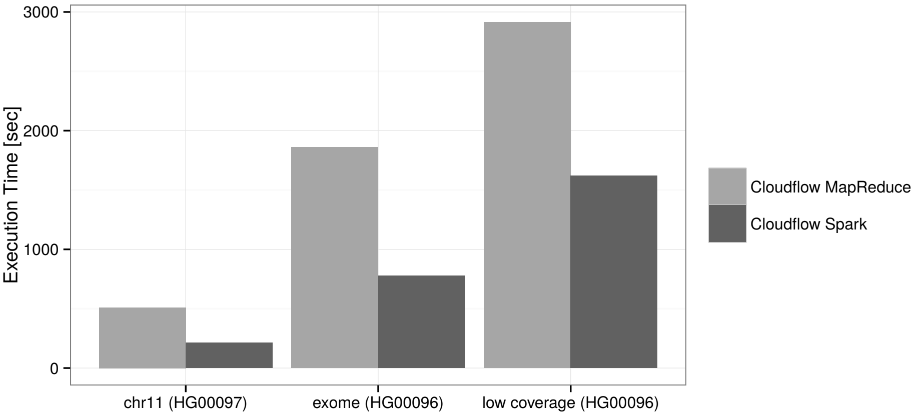


FIG. 4.3. *Execution time of the BAM Quality Control Pipeline with different sample from the 1000 Genomes Project*

For this pipeline Cloudflow uses the *groupByKey* transformation to calculate the average of all qualities. This transformation shows especially a good performance if all values for one key can fit into memory. However, too many values resulting in disc swaps and finally resulting in a execution time similar to MapReduce. This could lead to a bottleneck for extremely large datasets.

4.3. Sequence Length Distribution. This workflow generates the distribution of all sequence lengths within one BAM sample. This is especially useful for currently upcoming technologies such as the third generation Single Molecule Sequencing on PacBio or Oxford Nanopore systems to check for long reads or to determine appropriate reads as seeds within mapping algorithms.

The parallelization of the pipeline is realized by writing a transform operation that creates a new record for each read where the key represents the length of the read and the value is set to a constant. The summarize operation sums up all values per key, which finally represents the total number of reads with a certain length (see Figure 4.4).

```
class CalcReadLength extends Transformer<BamRecord, IntegerRecord> {
    public void transform(BamRecord record) {
        emit(new IntegerRecord((record.getSequenceLength(), 1);
    }
}
BioPipeline pipeline = new BioPipeline("Check Read Length");
pipeline.loadBam(input).apply(CalcReadLength.class).sum().save(output);
```

FIG. 4.4. *Sequence Length Distribution using Cloudflow.*

The pipeline was executed with the same datasets as described in Section 4.2. Again this experiment shows the expected speed-up when the Cloudflow pipeline is executed on Spark instead of MapReduce (see Figure 4.5).

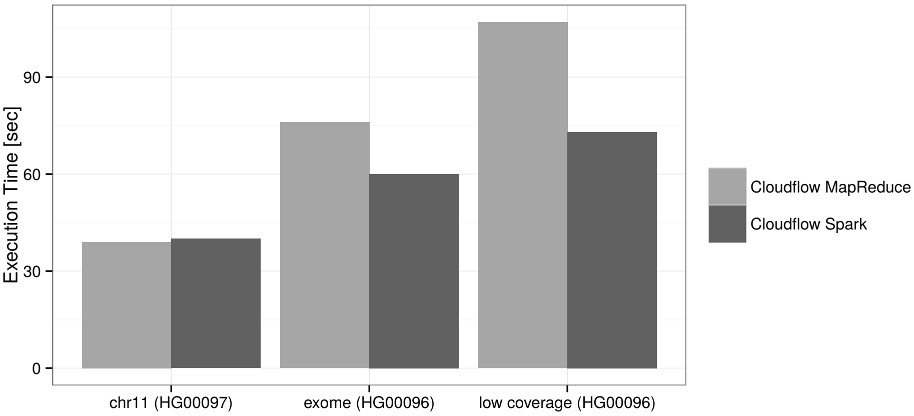


Fig. 4.5. *Execution time of the Sequence Length Distribution pipeline with different samples from the 1000 Genomes Project*

Compared to the previous pipeline, Cloudflow uses the *reduceByKey* transformation to sum up all values. This enables Spark to combine records with the same key on each partition before shuffling the data and sending it to other nodes. Thus, no memory overflow is produced and the pipeline scales linear with the amount of data.

4.4. Scalability. We tested the scalability of the new Spark executor in terms of the number of machines. For that experiment, we executed the same Cloudflow pipeline on the Hadoop cluster using 2, 4 and 6 nodes and compared the measured execution times with those of the MapReduce executor (see Figure 4.6). The results show that the overhead of executing a Cloudflow pipeline on a small MapReduce cluster (in our case two nodes) is much higher than using the same cluster with Spark. However, after a cluster size of 4 nodes both executors scale identically. Thus, the user benefits from the Spark executor when the used cluster is small.

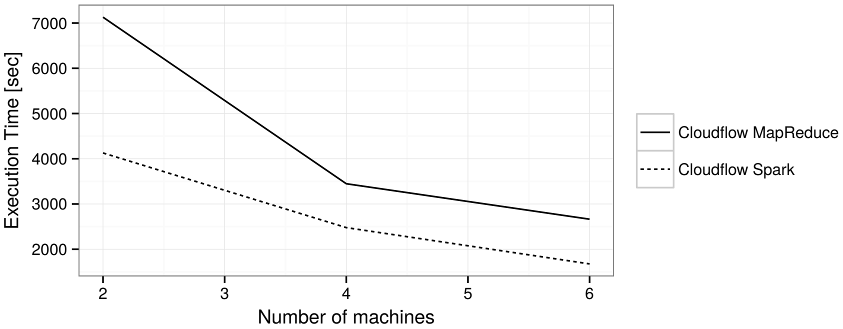


Fig. 4.6. *Execution time of the WordCount use case*

5. Discussion and Future Work. With Coudflow, an approach was presented that enables separation of the development of analysis pipelines from the underlying parallelization model. The extension of Cloudflow to Spark as an alternative execution engine has several advantages: (1) the developer has the possibility to create pipelines without thinking about the underlying technical details of the used parallelization framework; (2) the developer is not limited to one specific parallelization framework and can evaluate its pipeline using different frameworks without additional effort; (3) the end-user has the possibility to decide where an existing pipeline should be run depending on its computing infrastructure. Moreover, users take advantage when new parallelization models are integrated into Cloudflow and can use them without changing the source code of the pipeline.

However, such a generic approach also has some limitations in terms of optimizations. Currently, Cloudflow does not take advantage of all in-memory features provided by Spark. Our implementation minimizes the number of memory intensive transformations (e.g. *groupByKey*) only for some predefined 'summarizers' (e.g. sum and count). Moreover, no caching-support is implemented, which is essential for machine learning algorithms. To overcome this issue we will implement additional features in future releases to support even more use cases in the field of bioinformatics. Beside the optimizations of the translation processes, we plan to integrate new file formats such as ADAM and CRAM, which will enable further development of scalable services.

6. Conclusion. Cloudflow's overall aim is to simplify the development of complex analysis pipelines by using abstract operations. Therefore, operations need only be written once and can be re-used for future MapReduce and Spark usage. The major advantage of Cloudflow lies in the provision of validated operations, especially in the area of genetics, and its extensibility.

The contribution of this paper was an extension of Cloudflow that enables executing the same pipeline on Spark as on MapReduce. The experiments and evaluation show that (1) the same pipeline can be executed on Spark without changing the existing source code, (2) Spark leads to a speed-up for most pipelines and (3) the architecture of Cloudflow is extensible and therefore new analysis tools and file-formats based on Spark (e.g. ADAM) can be easily integrated in the future.

Acknowledgments. This work was, in part, supported by the "Scalable Big Data Bioinformatics Analysis in the Cloud" grant from the Croatian Ministry of Science, Education, and Sport and the Austrian Federal Ministry of Science and Research (BMWF) and by the FP7-PEOPLE programme grant 277144 (AIS-DC).

REFERENCES

[1] G. R. Abecasis, A. Auton, L. D. Brooks, M. A. DePristo, R. M. Durbin, R. E. Handsaker, H. M. Kang, G. T. Marth, and G. A. McVean. *An integrated map of genetic variation from 1,092 human genomes.* Nature*,* 491:56–65, 2012 Nov 1 2012.

[2] E. Afgan, D. Baker, N. Coraor, B. Chapman, A. Nekrutenko, and J. Taylor. *Galaxy cloudman: delivering cloud compute clusters.* BMC bioinformatics*,* 11(Suppl 12):S4, 2010.

[3] E. Afgan, D. Baker, N. Coraor, H. Goto, I. M. Paul, K. D. Makova, A. Nekrutenko, and J. Taylor. *Harnessing cloud computing with galaxy cloud.* Nature biotechnology*,* 29(11):972–974, 2011.

[4] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. *Flumejava: easy, efficient data-parallel pipelines. In* ACM Sigplan Notices*,* volume 45, pages 363–375. ACM, 2010.

[5] J. Dean and S. Ghemawat. *Mapreduce: simplified data processing on large clusters.* Communications of the ACM*,* 51(1):107–113, 2008.

[6] L. Forer, E. Afgan, H. Weissensteiner, D. Davidovic, G. Specht, F. Kronenberg, and S. Schönherr. *Cloudflow - A framework for mapreduce pipeline development in biomedical research. In* 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia, May 25-29, 2015*,* pages 172–177, 2015.

[7] L. Forer, T. Lipic, S. Schonherr, H. Weisensteiner, D. Davidovic, F. Kronenberg, and E. Afgan. *Delivering bioinformatics mapreduce applications in the cloud. In* Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*,* pages 373–377. IEEE, 2014.

[8] T. Lipic, K. Skala, and E. Afgan. *Deciphering big data stacks: An overview of big data tools. In* Big Data Analytics: Challenges and Opportunities (BDAC-14)*,* 2014.

[9] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. *Adam: Genomics formats and processing patterns for cloud scale computing.* Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley, Dec 2013.

[10] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. *Hadoop-bam: directly manipulating next generation sequencing data in the cloud.* Bioinformatics*, 28(6):876–877, 2012.*

[11] A. R. O'Brien, N. F. W. Saunders, Y. Guo, F. A. Buske, R. J. Scott, and D. C. Bauer. VariantSpark: population scale clustering of genotype information. BMC Genomics*,* 16(1):1052+, Dec. 2015.

[12] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. *Pig latin: a not-so-foreign language for data processing. In* Proceedings of the 2008 ACM SIGMOD international conference on Management of data*,* pages 1099–1110. ACM, 2008.

[13] S. Schönherr, L. Forer, H. Weissensteiner, F. Kronenberg, G. Specht, and A. Kloss-Brandstätter. *Cloudgene: A graphical execution platform for mapreduce programs on private and public clouds.* BMC bioinformatics*,* 13(1):200, 2012.

[14] O. Spjuth, E. Bongcam-Rudloff, G. C. Hernández, L. Forer, M. Giovacchini, R. V. Guimera, A. Kallio, E. Korpelainen, M. M. Kańduła, M. Krachunov, et al. *Experiences with workflows for automating data-intensive bioinformatics.* Biology direct*,* 10(1):1–12, 2015.

[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. *Apache hadoop yarn: Yet another resource negotiator. In* Proceedings of the 4th annual Symposium on Cloud Computing*,* page 5. ACM, 2013.

[16] M. S. Wiewiorka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski. *SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision.* Bioinformatics*,* 30(18):2652–2653, Sept. 2014.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster computing with working sets. In* Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*,* HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[18] G. Zhao, C. Ling, and D. Sun. *Sparksw: scalable distributed computing system for large-scale biological sequence alignment. In* Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*,* pages 845–852. IEEE, 2015.

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016

DOI 10.12694/scpe.v17i2.1160
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 115–128. http://www.scpe.org © 2016 SCPE

ENABLING SCALABLE DATA PROCESSING AND MANAGEMENT THROUGH STANDARDS-BASED JOB EXECUTION AND THE GLOBAL FEDERATED FILE SYSTEM

SHAHBAZ MEMON[†][§] MORRIS RIEDEL[†][§] SHIRAZ MEMON[†][§] CHRIS KOERITZ[‡] ANDREW GRIMSHAW[‡] AND HELMUT NEUKIRCHEN[§]

Abstract. Emerging challenges for scientific communities are to efficiently process big data obtained by experimentation and computational simulations. Supercomputing architectures are available to support scalable and high performant processing environment, but many of the existing algorithm implementations are still unable to cope with its architectural complexity. One approach is to have innovative technologies that effectively use these resources and also deal with geographically dispersed large datasets. Those technologies should be accessible in a way that data scientists who are running data intensive computations do not have to deal with technical intricacies of the underling execution system. Our work primarily focuses on providing data scientists with transparent access to these resources in order to easily analyze data. Impact of our work is given by describing how we enabled access to multiple high performance computing resources through an open standards-based middleware that takes advantage of a unified data management provided by the the Global Federated File System. Our architectural design and its associated implementation is validated by a usecase that requires massivley parallel DBSCAN outlier detection on a 3D point clouds dataset.

Key words: UNICORE, Genesis II, statistical data mining, data processing, distributed file system, security, standards, parallel processing

AMS subject classifications. 68M14

1. Introduction.

An ever increasing number of datasets from scientific experimentation such as earth observatories or computational simulations generate an enormous amount of information for discovering useful knowledge. In order to analyze data, the area of statistical data mining provides useful methods and tools to extract and explore useful patterns or prediction models. The field of statistical data mining comes with intuitive methods to learn from data, using a wide variety of algorithms for clustering, classification and regression. Several implementations are available, for example, Matlab, R, Octave [3], or scikit-learn. Mostly, these tools offer serial implementation of the algorithms, which is quite challenging (i.e. insufficient memory, extremely long running times, etc.) for processing the volume of data having terabytes or petabytes of magnitude. Considering that amount, the resources running the data processing tools require large number of processors, as well as much more primary and secondary storage. Therefore, parallel tools and platforms such as Hadoop [15] implementing the map reduce paradigm [18] and selected massively parallel algorithm developments based on the MPI and OpenMP environments are commonly used.

We observe mainly tools for (High Performance Computing) HPC and High Throuput Computing (HTC) paradigms evolving concurrently, but each supporting their own set of requirements. Scientific communities, either from biology, physics and medicine adopt more conservative approaches in order to retain their focus on scientific findings and as such traditional HPC environment still play a major role in the relatively new realm of 'big data'. Given the stability of HPC environments and its benefits using locally parallel filesystems with parallel I/O techniques motivates our work to enable straightforward job executions managed by HPC sites that seamlessly access data from a distributed file system service which has not been traditionally supported in HPC-based execution services.

We validate our approach with a use case from earth science using 3D points cloud obtained from devices that measure a large number of points of an object surface (i.e. in our case the inner city of Bremen). The data analysis of this dataset has the goal to cluster special data points and identify any noise elements. In this paper we describe the architectural design and implementation necessary to run data analysis jobs on HPC resources through standards-based UNICORE middleware [10] and uses data from the Global Federated File System [28] that we derive from the architecture of the Extreme Science and Engineering Discovery Environment (XSEDE) [24]. UNICORE is a HPC middleware and deployed on production on XSEDE supercomputing sites, whereas

[†] Juelich Supercomputing Centre, Forschungszentrum Juelich GmbH Juelich, Germany

[‡] Department of Computer Science,University of Virginia Charlottesville, USA

[§] School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland the GFFS is a distributed network file system which is an integral component of the Genesis II platform, that is also consider to be a middleware element in XSEDE. Our architectural design overcomes the limit that the data is hosted by the GFFS cannot be easily made available to the job executions that perform data clustering over the points cloud data set.

The paper is structured as follows. Section 2 describe the basic background of the technologies and standards used as part of our research. Section 3 lists a detailed requirement analysis we obtained during the course of the integration effort. Section 4 describes the security model and the implementation we derived for supporting the requirements from Section 3. Section 5 offers detailed insights on our architectural design and its realization that enable the UNICORE and GFFS integration while addressing the selected requirements. Section 6 takes a massively parallel data analysis application in order to validate our work based on a real world use case. Section 7 provides a brief overview of the related work, and the paper concludes in Section 8.

This article is a joint and extended version of [34] and [40].

2. Background. This section gives a brief background of the technologies, algorithms and standards that supported our work.

2.1. UNICORE. UNICORE is an HPC middleware which is built upon the priniciples of Service Oriented Architecture (SOA). It realizes compute, information and data functions through a set of stateful web services [42]. These services are designed in such a way that they enable seamless access to heterogeneous high performance computing resources. In this sense, the middleware layer to these clusters provides access and location transparency to compute and and thus offers scientists a unique environment hiding low level technical complexities (i.e. avoiding writing and submitting error-prone scheduler dependent job scripts). The compute access transparency enables an abstraction of different flavors of resource management systems (sometimes also referred to as schedulers), such as SLURM [45] or Torque [6], and more notably through a unified and standard interface. Figure 2.1 depicts the basic UNICORE architecture that is composed of layers with distinct functionality, including Client, Services and Target System Interface.

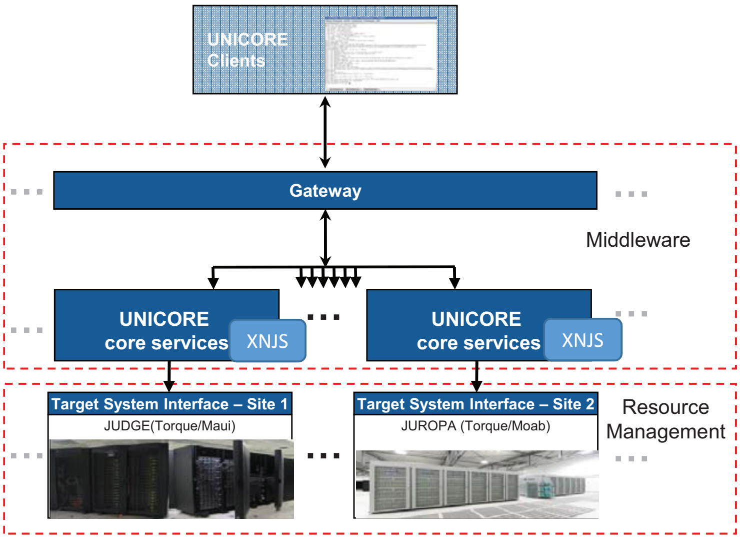


FIG. 2.1. *Basic UNICORE architecture with example deployments of two HPC systems in Juelich (JUDGE and JUROPA).*

The client layer provides API and end user interface, which include interfaces for constructing and sending client requests to remotely deployed services. The client side API is useful for scientific communities which are not necessarily using the UNICORE's povided interface, but instead their own clients such as their application specific science portals or gateways (e.g. UltraScan Scientific Gateway [33]. Hence, all important functionality of the middleware services can be invoked through the direct client API interaction. The end user interface offers a rich client interface called UNICORE Rich Client (URC) [19], with advanced user controls to compose and orchestrate scientific workflows. The second variant is a command line client called UNICORE Command Line (UCC), which provides an interface for advanced users who know the low level details of writing job request scripts to be executed jointly on the batch system. For a more detailed architecture explanation we refer to [10].

The Services layer plays a vital role in enabling job execution and data management by means of SOAP [16] over XML based web services. Not only the Services layer implement the core functionalities, but also the hosting environment which can host and deploy stateless and stateful web services, for instance, WS-I (Web Services Interoperability) [5] and WS-RF (web Services Resources Framework) [42]. Job management functionality implements a complete life cycle through which job passes, and that includes submission, monitoring and data staging. The job management functionality is supported by UNICORE's embedded scheduling and execution framework, called XNJS (Extended Network Job Supervisor) [43]. It manages the incoming middleware requests against the hosted application and resource capabilities (for example number of available nodes, processors per node etc.). The Services layer gives a configuration based interface to expose underlying cluster resource and environment, so that XNJS can perform resource match making upon the client initiated job requests. After validating the job request, the XNJS component formats the job to the generic UNICORE protocol, and then sends it to the resource manager specific implementation of Target System Interface. This is the layer where the generic UNICORE script gets translated to the request formatted according to the batch system.

As a summary, a simple job execution sequence comprises of, client job submission to the Services layer, then the request is forwarded to XNJS, and then it is communicated to the Target System tier. This tier in turn directly interacts with the low level batch system, and fetch job statuses, and manage underlying running file transfers during the job's execution life cycle.

2.2. The Global Federated File System. The Genesis II Global Federated File System (GFFS) [28] is a distributed file system that provides researchers with tools for securely managing and sharing their scientific data. The GFFS offers a set of interfaces that manage jobs and provide access to the required scientific data. This is achieved through the GFFS-Queue component, which is also based on SOA wherein standards-based interfaces are adopted for storing and accessing remote compute and data resources. In order to support federation across different organizational entities, the GFFS provides a hierarchical file system structure with standard namespace locations for storing user profiles, groups, directories, and service elements such as meta-scheduling queues and Basic Execution Service endpoints (BES) [29] for processing jobs. Figure 2.2 provides an overview of the integrated architecture with Genesis II and UNICORE Basic Execution Service (BES) endpoints interacting with the GFFSs root container.

The GFFS implements many of the standard Unix commands (such as `cp` and `mv`) in a console mode through the so called grid shell. There is also a GUI view of the GFFS, which supports rich drag and drop file management. The GFFS also provides a FUSE file system interface [2] that allow users to mount the GFFS on a Unix directory and operate on files in the GFFS as if a user is interacting with her local file system. The GFFS has an export feature like NFSv4 that allows users to share part of their own file system visible within the GFFS, and to other users part of the broader federated infrastructure. The GFFS Queue is a metascheduler that supports submission of multiple jobs for subsequent distribution to the execution service endpoints connected to the queue. The GFFS Queue provides researchers with a mechanism for managing and controlling their computations via a GUI as well as with familiar command line tools such as "qstat" and "qkill". Jobs will be distributed to BES resources automatically by the GFFS Queue, but can also be rescheduled as needed. The XSEDE project benefits from the GFFS by giving researchers a way to securely share data with their colleagues and by providing a high level view of the computational resources available at the XSEDE infrastructure through the GFFS Queue.

2.3. Standards. RNS (Resource Namespace Service) [36] is an Open Grid Forum initiative that standardizes the naming of distributed resources that form an infrastructure. It has a simple set of operations for managing grid and cluster services mapped as file system operations such as `rm`, `mkdir` or `cp`. The RNS specification provides client applications to couple WS-Addressing [22] based endpoints with human readable notations. For instance, a job execution service managing multiple jobs can be represented as a parent directory and individuals jobs are child directories which may further contain the contents of their working directories.

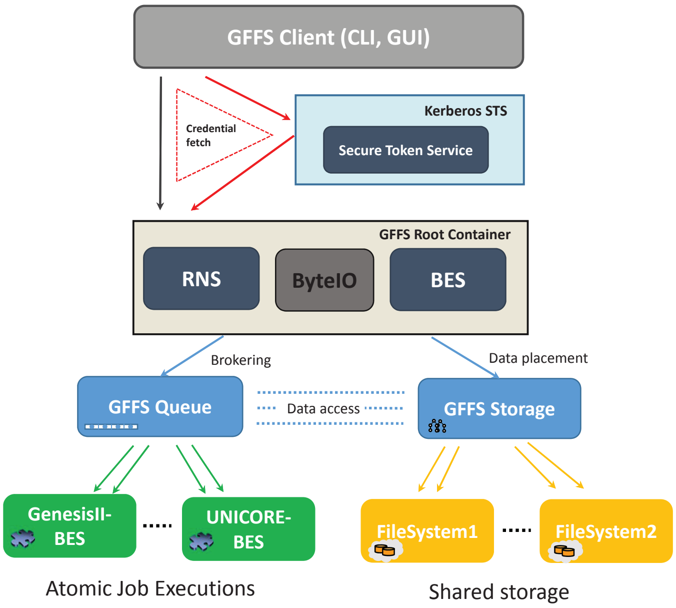


Fig. 2.2. *The Global Federated File System (GFFS) architecture.*

Considering the stateful service based endpoints, wherein web service resources can have a nested heirarchical structure, the RNS representation is very helpful in providing an access and location transparency to underlying resources. The GFFS mentioned earlier implements the core RNS and its Web Services Resource Framework (WSRF) [37] rendering to access service endpoints. The statefulness gives individual access which is very much analogous to the domain of distributed remote objects. As far as the data transfer is concerned the GFFS primarily uses the ByteIO [35] standard. ByteIO provides a set of interfaces to interact with bulk data sources and sinks. The ByteIO standard enables large amount of data in an efficient way. It has two interfaces, RandomByteIO and StreamableByteIO. RandomByteIO provides an interface to access bulk data in a stateless and random manner. This interface is normally being called when a client transfer large files. StreamableByteIO interface allows data transfer data in a stateful manner. It is normally used for accessing short files, in most cases standard outputs of managed jobs.

The Job Submission and Description Language (JSDL) [8] is an XML-based comprehensive data model for specifying computational job requirements consisting of application, resources and data concepts. UNICORE and Genesis II clients specify job requirements in the JSDL format. UNICORE server side implements most of the JSDL, and also its related profiles and extensions.

The JSDL specification has a generic model for representing multiple type of resource settings, such as HPC and HTC. The JSDL model further provides a set of profiles which imposes constraints on requirements according to the type of resource architecture. These requirements may include, file staging modalities, parallel execution environments and parametric jobs. This paper mainly targets HPC resource types which normally use parallel execution environments and a resource management system. HPC resource profile [12] enable users to specify more internal HPC architecture specific requirements in combination with restrictions on the execution service. The HPC file staging profile [44] captures data movement specific elements to be used within heterogeneous cluster environments. This profile impose constraints on using HPC specific data staging attributes as part of the job submission request. For instance, the request may contain FTP user name and password credentials for the BES instance to carry out third-party file transfers on user's behalf.

The related technologies presented above provide a base for providing a seamless and robust middleware platform to tackle big data challenges. One of the common big data processing machinery requirement is to have an iterative execution for discovering optimal set of algorithm parameters. Specifically, data clustering algorithms such as K-Means or DBSCAN require certain parameters before their processing. In the case of iterative execution multiple runs with a varying set of parameters are required. If we combine these runs into a single composite job then this kind of job is called parametric. UNICORE as our base execution middleware supports parametric jobs through the JSDL Parameter Sweep extension [23]. This specification provides an intuitive model to parametrize multiple parts of job request per se. It may allow client application to sweep over a list of arguments, file transfer locations and environment variables specified under the JSDL request. The sweep model can represent different kind of iterations, either be it a element-wise iteration on a set or a counter with configurable stepping factor. There are two major kind of sweeps the specification provides, Document sweep and File sweep. The Document sweep provides a model to modify a requested JSDL instance. The File sweep is an advanced model which presents a data structure to modify the contents of the files imported before the job execution phase. This kind of sweep is applicable to text based files. UNICORE middleware implements both kind of sweeps through its client API and command line client (UCC). This specification is used to automate the iterative data clustering on the points cloud data set we use in this paper.

OGSA-Basic Execution Service (BES) [8] is an Open Grid Forum (OGF) initiative which provides a web services-based interface for managing and monitoring computational jobs in HPC and HTC environments. For a job submission use case BES interface accepts a JSDL instance and its related profiles as a parameter and then runs it on back-end resource. The focus of this paper is based on a scenario in which UNICORE server expose it computing capabilities via BES model, and Genesis II client use this interface to invoke remote calls on the UNICORE endpoint.

2.4. Unsupervised Learning. While the overall architectural design in this paper is applicable to many learning models, our work is using parallel version of the Density Based Spatial Clustering for Application with Noise (DBSCAN) algorithm [21]. The focus is rather on the access of the algorithm implementation through the UNICORE middleware and the GFFS based file system. Therefore, this section briefly introduces the DBSCAN method. Goetz et al. describes more details on the algorithm implementation in [27], which gives more detail on what parallelization strategies are used to achieve scalability and high performance while analyzing large data sets. DBSCAN [21] is an unsupervised density based clustering algorithm. The cluster based on density is represented by a number of points MinPoints within a specified radius Epsilon. These are the important user defined parameters of the algorithm to identify the clusters.

i) Core point: A central point in a dense region, it has more than a specified number of minimum points MinPoints within its neighborhood (or radius) Epsilon.

ii) Border point: A point that lies on the border of the dense region, it fewer than minimum points MinPoints within its neighborhood (or radius) Eps

iii) Noise point: A point that is neither a core point nor a border point

DBSCAN intrinsically enables maximizing the local point density recursively. It sets apart from other clustering algorithms as it detects the clusters of arbitrary shapes and sizes. Notably, it is resistant to noise and suitable for finding anomalies or filter specific noise signals from the data. We use the parameter-based DBSCAN learning algorithm as a specific example of how our architectural design and its implementation can be generically used by a wide variety of learning algorithms in this paper.

3. Requirement Analysis. During the course of transparent integration for bridging both technologies, we identified multiple requirements which not only aims at superficially combining them, but also some extensions in the UNICORE services layer which will help to tackle "big data" challenges from machine learning and data mining. The integration has taken XSEDE infrastructure as an example, but the implementation is applicable to any distributed computing and storage infrastructure. The major integration requirements from both the technologies' perspective are summarized together with relevant technology environments in Table 3.1. They lie in the following areas. R1) Secure Trust Delegation: As in a distributed service interaction a user interacts with a portal or meta scheduler which then forwards the request to a service that takes care of the job execution and also calls upon data management services to pull and fetch data. While the user is participating in the very beginning, then the following phases are to be done by other services, require some kind of trust delegation which the user entity assigns to the target job execution and data management services to act on her behalf. In our scenario a user communicates a data oriented job request with a set of input data staging elements, therefore trust delegation has to be implemented by the UNICORE platform to understand the GFFS user requests. R2) Openness: In any kind of communication between a user and the GFFS or UNICORE, it

Table 3.1
Summary of Requirements

#	Requirements	
	Description	*Environment*
R1	Secure Trust Delegation	GFFS and UNICORE
R2	Openness	OGSA-BES, JSDL
R3	Data transport	RNS, BYTEIO
R4	Infrastructure integration	XSEDE, MYPROXY
R5	Transparency	UNICORE Server
R6	Parameter sweep	JSDL Parameter Sweep
R7	Extensible resource and job model	GFFS and UNICORE

should support standards-based protocols, so that users or services from different middleware backgrounds can easily interact through UNICORE and the GFFS client-side APIs. R3) Data transport: As UNICORE jobs are intended to use data from the GFFS, the running jobs should be able to upload and download data from the file system space. R4) Infrastructure integration: XSEDE-based identity management should be understandable to both layers of job submission and data management middlewares. R5) Transparency: The jobs managed by UNICORE in an HPC environment which accesses data from the GFFS data should not know the physical location and also on how the data is structured across data nodes within the file system space. R6) Parameter sweep: This requirement is very specific to jobs which require re-running the same application but with different parameters:these are called composite or parametric jobs. In a parametric job, the execution middleware iterates through a set of parameters provided by a user job submission request and creates a separate job internally for each parameter combination. In this case UNICORE middleware should be capable of interpreting and incarnating parametric jobs. R7) Extensible resource and job model: As supercomputing architectures are evolving to support data and network intensive applications, the hosting middleware should be adaptive to new changes and thus possess an extensible model for users specifying sophisticated requirements. For instance number of GPGPUS or use of execution environment.

4. Interoperable Security Model. The GFFS security model is based on the Security Assertion Markup Language (SAML) [41] standard, and takes the UNICORE SAML profile [14] as a reference implementation. The GFFS extends the UNICORE's SAML profile to represent the trust delegation chains of the Genesis II security model. A delegation chain encompasses that a user has delegated some level of trust to a service in the GFFS in order to achieve a task, such as processing a job. Mostly, delegation operations include three entities, (1) a grid user (for our example, called U, (2) a TLS connection identified by an X.509 certificate (called C), and (3) a grid resource (called R). Longer delegation chains usually contain all three of these types of entities as the first links in the chain.

In the GFFS, the first entity U is always a user defined as a grid STS (Secure Token Service) object. This entity is the prime mover for any operations that are performed in the GFFS. The user's rights within the grid's access control list permission system dictates what that user can and cannot do with regard to every grid resource.

The client software must authenticate as the grid user U to obtain services from a GFFS container. This is where the second security entity C comes in; it is the connection by the client software at the behest of the user. Initially, the client credentials only contain C, as one makes the connection before STS authentication occurs. In the XSEDE login process, this connection is always based on X.509 credentials obtained from a certificate authority service, the so called XSEDE MyProxy server. Thus, it can be a well-known identity within both Genesis (via a Kerberos-based STS) and UNICORE (via the grid-mapfile). In the example, the client software first authenticates to MyProxy by using user name and password to obtain the certificate C. Then, the client authenticates to the Kerberos STS in the GFFS to obtain grid user U.

After the user authenticates, the client software's credential wallet will contain the first delegation of trust, U→C. This states that the grid user U trusts the TLS connection C to act on its behalf. Afterward, all of the actions taken by the TLS connection C are understood to be U's actions. Any access that provides U with

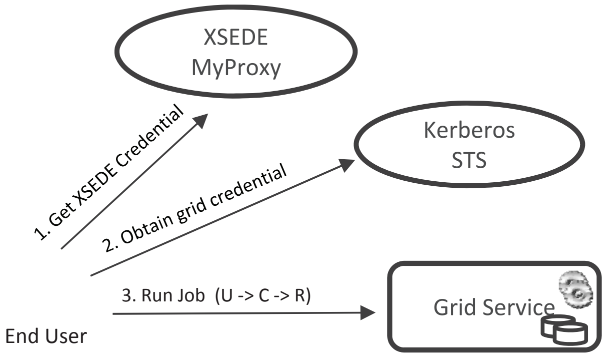


Fig. 4.1. *End User interaction with security and grid services.*

permissions will also be granted to C. That may include submitting a job to a BES named R. This is the second point where trust is delegated; the certificate for TLS connection C signs a new trust delegation that expresses "C trusts R" to perform a job execution. This extends the length of the delegation chain by one, so that it now has all three entities involved in two trust delegation objects. This can be represented with delegation arrows such as:

- First delegation: U→C (The grid user trusts the TLS connection)
- Second delegation: C→R (The TLS connection trusts the resource)
- Full Chain: U→C→R (The user U trusts the connection C which trusts R to run a job)

This new delegation chain can be presented by resource R when it needs to do further actions on the grid user's behalf. Moreover, actions might include storing file staging results back to RNS space (U→C→R→D, where D is a Data folder or possibly submitting the job to another BES for final processing. The important point is that entity D is just another resource to which trust can be delegated, and the chain can be continued in that manner for as long as its delegation depth limit allows. Figure 4.1 depicts a typical interaction of an end user with the security services and a target grid (execution or data management) service.

One challenge while interoperating between the GFFS and UNICORE integration arose due to a difference in interpretation of the SAML assertions. The delegation chains in the GFFS are tightly-coupled, and do not allow mixing and matching of individual entities. This is not directly provided by the UNICORE SAML implementation, which permits the receiver to mix and match any delegations provided in a message (U→C→R is considered to be two separable delegations U→C and C→R). In the GFFS model, a delegation chain must be used in its entirety or not at all.

To address this difference, the GFFS implementation of SAML adds a unique identifier to each SAML assertion. A chain such as U→C→R is built by embedding the identifier of the U→C assertion in the C→R assertion. To make this cryptographically secure, the signature of the U→C assertion is also embedded in the C→R assertion before C→R itself is signed. This enforces the connections between the GFFS delegation chains while still leveraging the UNICORE's Security Assertion Markup Language (SAML) implementation. Upon reception, the chains are reassembled and any assertions that are referenced by a longer delegation chain are removed from the pool of available assertions.

The Genesis delegation chain model supports having multiple chains in a credential wallet. This supports the user possessing multiple different types of identity and authorization on resources. The users will always have their own identity as a credential, which allows them access to resources where the user has been given explicit permission. The user will also usually have at least one group credential, which allows them access to portions of the grid file system. Additional group credentials may convey access to different BES or queue resources within the grid. Thus the credential wallet approach supports a flexible authorization appropriate to the variety of grid resources, possibly across multiple administrative domains, that may be required for the user's work.

The signing of credentials ensures that it is computationally infeasible to create a fraudulent credential chain where a new identity is inserted into the credential chain. Each credential records the signature of the prior element in the chain, along with its unique identifier. Thus an attacker would have to compute a valid XML digital signature inside a valid trust delegation object, where the unique id is also properly signed by that signature.

To ensure that the credential wallet cannot be easily compromised and used for playback attacks (where the valid credentials of a user are stolen and used by a different user), all credentials must be "anchored" with the current TLS session credential of the grid client. At least one link in the credential chain must be identical to the TLS session certificate. This ensures that playback is very difficult indeed, since the stolen credentials must be based on the TLS session key that the user was employing at the time the valid credentials were minted. This mitigates attacks upon the server, where the container database is compromised. Attacks using an entire set of stolen client credentials are also somewhat mitigated, since the TLS session certificate is based on a short-lived key pair.

With respect to the requirements mentioned in Table 3.1, the given security model implementation covers R1-Secure Trust Delegation. XSEDE's MyProxy access is provided to allow XSEDE users run job with their infrastructure credentials. This feature relates to R4-XSEDE MyProxy integration.

5. Integrated Architecture and Implementation. We have extended UNICORE's server tier to accept the incoming requests incoming from Genesis II remote clients. The remote clients here implies the GFFS's GFFS-Queue component which is an entry point for a user to submit job. The GFFS-Queue acts as a meta scheduler that schedules the user's request based on its resource requirements on a set of available BES-based computing endpoints. Even though UNICORE understands BES protocol, but still the execution service should know how to interpret, authenticate, and authorize the incoming GFFS Queue requests. A separate UNICORE server extension is implemented that is invoked when server finds a security token containing GFFS–related information in the incoming client request. The extension validates the SAML chain by looking into every element of the chain. These elements are entities (described in the previous section) which contain every stakeholder including end user or service through which the request was passed. The standards-based access and the validation of incoming requests required to trigger the data transfers serve the requirements R2-Openness and R3-Data transport. The user doesn't need to provide the actual physical location of the GFFS hosted data, instead she uses the symbolic RNS qualified hierarchical paths. This feature is inclined to support R5-Transparency.

Before a Genesis II client is able to send jobs to a UNICORE BES endpoint, a Genesis II container should recognize and link the UNICORE BES instance into the RNS space. The linking is achieved through the Endpoint Reference Minting process. An EPR (Endpoint Reference) is the basic component of the RNS. Every location in the GFFS namespace has an EPR that identifies (1) where the resource lives and (2) the X.509 certificate that represents the resource. Minting an EPR is the process of creating a new EPR as an XML document that represents an external resource, such as a UNICORE BES instance. The process of minting an EPR combines the URI where the resource is located with the X.509 certificate expected as the resource's identity (which it would report over a TLS connection). Once an EPR is minted, the EPR's XML document can be stored locally as a file or added as a new link to the grid namespace. When linked into the grid, a user with appropriate credentials can see the entry in the GFFS files system and can use it to obtain whatever services the resource provides.

The user's XSEDE identity is extracted from the delegation chain, and the retrieved identity is validated against the authorization store of the UNICORE server deployment. If the user is found under the authorization store then the required user context is created for carrying out GFFS data staging invocations. After the context creation phase, the server extension releases its control and job moves to the next phase of execution. In the beginning of the execution phase the request is processed further to carry out the GFFS-based data stagings of jobs. Figure 5.1 shows the job request encoded in JSDL containing application requirements and data staging elements pointing to the GFFS space. Note that the job will be executed on the UNICORE site, therefore Genesis II-BESes are not involved in this sequence.

The GFFS-based data transfer is realized through an XNJS extension. The GFFS download component prepares a command to pull data from the GFFS. The command (such as '`grid cp remote-source job-working-directory`') will be forwarded to the Target System Interface (TSI) that invokes the Genesis II '`grid`' command gracefully and downloads the data to the job's working directory. In a likewise manner the

GFFS upload takes care of uploading output files to the remote GFFS location. Any failure will make UNICORE fail the job and abandon any further processing related to it. For every job which is sent by the Genesis II client UNICORE server extension maintains an additional folder in each of the job's working directory that contain user's contextual information.

The pictorial representation of a simple job submission sequence is shown in Figure 5.1. In the first step, the Genesis II GUI client asks a user to log-in through an XSEDE provided credentials. After the authentication phase (step 2), she uploads data to the GFFS folder (step 3). In step 4 the user then submits a job request with application details, and location of the data to be downloaded. In a similar manner, output data paths are also specified. The user then selects a UNICORE endpoint and run the job. Steps 5 and 7 shows a submission of request to the TSI and the target batch system. As soon as the job has been submitted to the execution service, the client continuously monitors the job until it reaches to a terminal state. Once the job is finished successfully the output is fetched back from the remote job working directory which is located on cluster's file system to the GFFS space. Steps 6 and 8 depicts the TSI and the GFFS interaction. For the sake of brevity only a sequence of major steps are being highlighted.

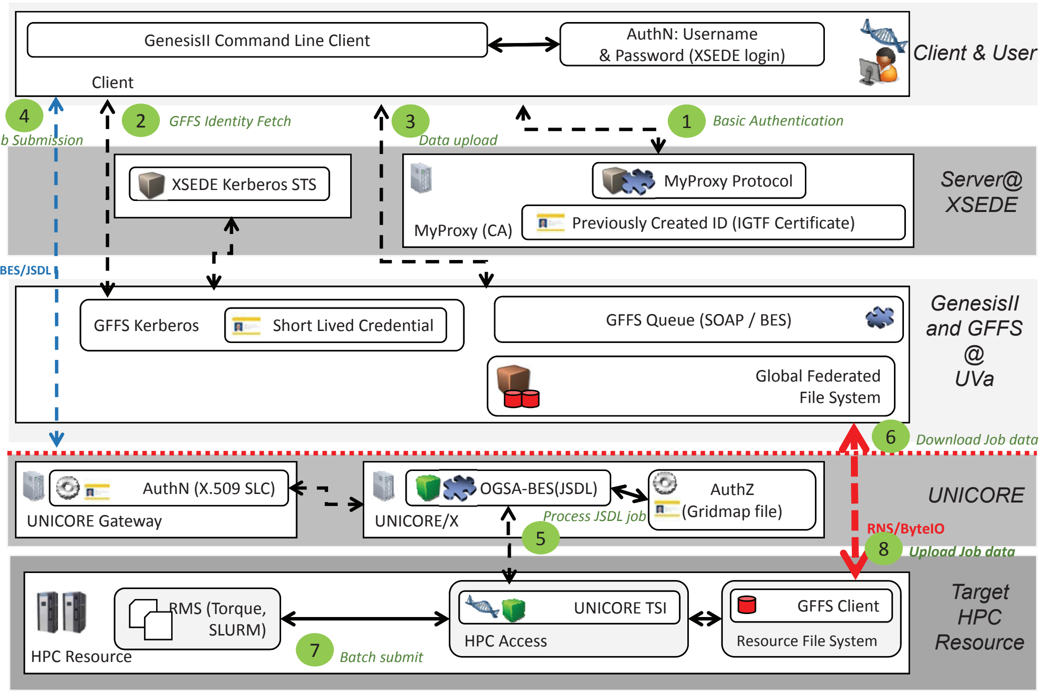


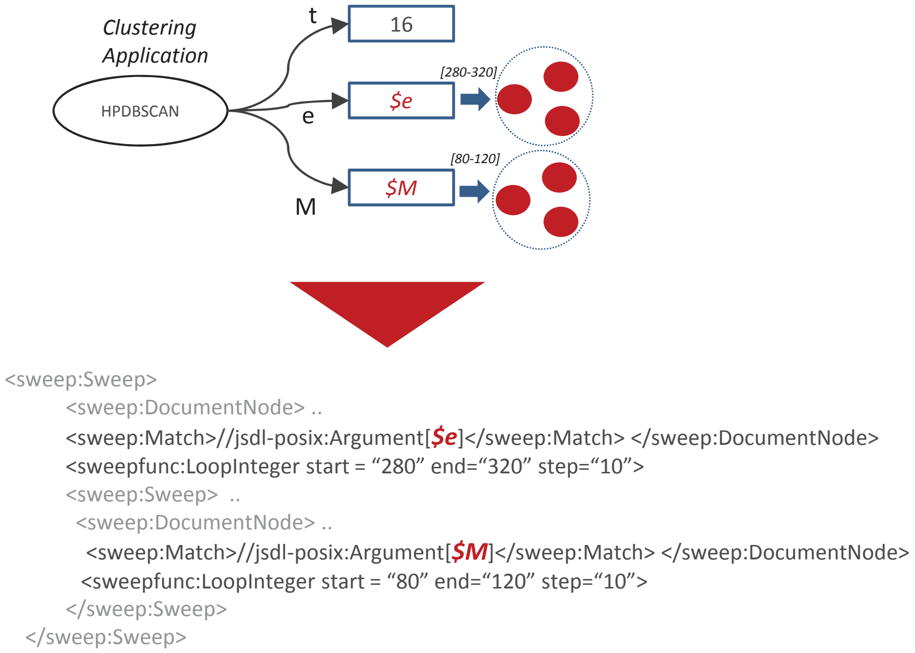
Fig. 5.1. *The UNICORE and the GFFS integration showing a job submission sequence with data staging.*

Another building block for supporting semi-automated data analytics is to allow user running jobs of parametric nature. Specially, the use case presented in the next section needs multiple runs required to identify optimal set of application parameters. By using the JSDL Parameter Sweep extension implementation of UNICORE [32] users can run multiple jobs as a single request. This is a very useful feature that has a positive impact on the overall data analysis life cycle. Considering, if manually running many jobs and compiling resultant outputs by hand, it will take user's considerable time just for book keeping the previous and next set of runs. We experimented by sending parametric jobs via Genesis II client and UNICORE BES implementation, and compared this model with manual SSH based submissions. By following this approach the user's administrative and usability overhead has significantly reduced. Figure 6.1 shows the snapshot of the used JSDL Parameter Sweep instance for the application.

6. Usecase: Point Cloud Anomaly Detection. Anomaly or outlier detection algorithms primarily identify a set of data points that appear to be different than the remaining data. There are different data clustering approaches which help data scientists to discover anomalies and a meaningful set of clusters from data. Several methods exist, for instance K-Means and Agglomerative clustering, have been used in commercial and scientific domains.

In this paper we place our focus on the DBSCAN [21]] algorithm which allows to reduce noise factor of the 3D point cloud dataset. A point cloud dataset captures objects in three dimensional space representing the external surface of objects by a point cloud. In our case, we use a data set that contains a point cloud for landscape elements, such as different kind of buildings, monuments or bridges, of the city of Bremen, Germany. This points cloud has approximately 81 million data points. We use DBSCAN to detect outliers in particular noise artefacts produced by the 3D scanner when recording the 3D point cloud. In practice if the dataset is processed using serial algorithm, it may take a couple of days. Therefore, it is imperative to have a parallel implementation of DBSCAN, which not only improves the performance, but also uses storage and memory requirements in an efficient manner. Another requirement is to have an implementation that adequately exploits execution environments of HPC. In order to support the application, HPDBSCAN [27] implementation is used. It is an initiative of Juelich that provides parallel implementation of DBSCAN. For efficient data storage and access, it uses the HDF5 data format.

We deployed this application on XSEDE infrastructure. We specifically used the BlackLight cluster that is deployed at Pittsburgh Supercomputing Center (PSC). A UNICORE service instance has been deployed and linked with the XSEDE-wide GFFS. Before the job execution, the dataset is placed on the GFFS node at the Indiana University's Mason cluster. The data staging was done by using the UNICORE's GFFS extension that copies data from the file system space to the local job's working directory.



```
<sweep:Sweep>
        <sweep:DocumentNode> ..
        <sweep:Match>//jsdl-posix:Argument[$e]</sweep:Match> </sweep:DocumentNode>
        <sweepfunc:LoopInteger start = "280" end="320" step="10">
        <sweep:Sweep>  ..
          <sweep:DocumentNode> ..
            <sweep:Match>//jsdl-posix:Argument[$M]</sweep:Match> </sweep:DocumentNode>
          <sweepfunc:LoopInteger start = "80" end="120" step="10">
        </sweep:Sweep>
   </sweep:Sweep>
```

Fig. 6.1. *HPDBSCAN representation in the JSDL Parameter Sweep format depicting application with arguments: epsilon (e) and MinPoints (M) sweeping through a range of values.*

The identification of anomalies from the point cloud dataset is the main objective of the clustering application. This requires to find an optimal set of application arguments: MinPoints M, and epsilon (also called radius) e which influence the clustering of new point cloud instances or different variants of the same data, respectively. In terms of data mining this phase is called post-processing. The discovery of optimal arguments is achieved by analyzing each of the completed job's output which contains the cluster distribution and noise factor. Within the output, the criterion is to select the job configuration containing the minimum noise factor combined with the best cluster distribution. The whole process of optimization requires multiple manual runs

Table 6.1
The user perspective of the Data analysis lifecycle using manual and automated mechanisms.

Access Mode / Execution Phase	Data Transfer	Data Processing	Post Processing
Manual	SCP, GridFTP, ByteIO, FTP	Job script for every different resource and batch system	Create script manually for every variation
Middleware (UNICORE & the GFFS)	Automated through the supported data transfer protocols	One JSDL instance for all kind of backends	Single JSDL-PS template for the specified parametric variations

of the same application but with different M and e values. In order to avoid that users need to run these multiple jobs manually, the extended JSDL Parameter Sweep implementation which is provided by UNICORE's execution back-end was used. This allows using just a single job request which is not only more convenient for the user, but also faster, reproducible and less error-prone. The parameter sweep implementation serves the requirement R6-Parameter Sweep. Even though the user submits only one job, multiple child jobs are automatically generated according to the number of parameter iterations and nested sweeps. Figure 6.1 shows the sample HPDBSCAN JSDL job description making use of the parameter. The sweep factor of epsilon (e) and MinPoints (M) shown in Figure 6.1 will spawn 25 jobs in total with each generating a separate output.

Table 6.1 summarize the steps user need to perform data analysis in manual (script-based) and middleware-hosted environment. It is also evident from the illustration that the use of JSDL and JSDL-PS is more intuitive and avoids a need to write custom job requests for each flavor of the target resource management system. Furthermore, the data transfer event here applies to the pre-execution and fetch outputs phase.

7. Related Work. In this section we present the related job execution middle- ware technologies which are integrated with distributed file systems as well as work related to DBSCAN.

GridFTP [31] is one of the major data transfer protocols used in today's scientific and commercial data infrastructures. Specifically, GlobusOnline [25] data transfer service is mainly using this protocol to move data across widely distributed endpoints. UNICORE's GridFTP extension [7] helps scientists to submit job executions on UNICORE and using GridFTP-based data endpoints for data stagings. From the implementation perspective, both the GridFTP and GFFS extensions are integrated following the same approach, that is by using the XNJS programmatic interfaces. In the case of GridFTP integration, the clients requiring UNICORE servers to perform data staging on user's behalf, need to send at least X.509 proxy certificate chain along with the job submission request. For the GFFS the entities communicating the job request to the server must send a set of SAML assertions.

The GlobusOnline [25] service is a web portal to help end users perform GridFTP based high performance data transfers across different data endpoints. From the data management aspect, GlobusOnline and the GFFS are sharing a common set of features. By bridging the data access and processing (i.e. the job submission and execution, and execution service mount-point) services simultaneously distinguish the GFFS from GlobusOnline. The processing part is capable of attaching high performance (GenesisII) and high throughput computing (UNICORE) in a standard way. A very positive aspect of GlobusOnline is usability as it offers a ready to use data transfer service through a common web brower, whereas the GFFS user interaction is native desktop-based, which is not very intuitive and responsive as compare to browser-based applications.

ARC [20] is a middleware suite used by high throughput computing communities. ARC's integration with [26] and DDM (Distributed Data Management) [13] solutions are mostly used by the ATLAS [13] particle physics community at the Large Hadron Collider. dCache and DDM are distributed data management platforms providing storage and retrieval of huge amounts of data. dCache and the GFFS share mostly the same set of scenarios, but the major difference is that the GFFS expose its interface via RNS and ByteIO, whereas dCache is accessed through the SRM [9] interface.

WS-PGRADE / gUSE (grid and cloud user support environment) [30] is an open source scientific gateway framework that allows access to heterogeneous grid and cloud resources. gUSE provides a client extension in the form of DCI bridge [1] to the GFFS by invoking Genesis II clients. It is much similar to the way UNICORE integrates the GFFS. The frame- work provides access to UNICORE and ARC job submission services through the OGSA-BES interface.

In the context of worfklow (e.g. Taverna [38], Kepler [11], etc.) enabling data mining methods on distributed computing infrastructures. Da Silva et al. [17] describe workflows with serial implementation of DBSCAN. According to our understanding their approach is not using the parallel DBSCAN implementation and in contrast to our approach that is intended for production usage in a high performance computing environment, the paper rather describes a research project than a production implementation.

PDSDBSCAN-D [39] is an implementation of DBSCAN, based on the MPI and OpenMP frameworks. According to [27] the HPDBSCAN application is more performant on various earth science data sets, among which the points cloud data is one. It performs better due to efficient pre-processing of spatial cells and use of density-based chunking to balance the local computation load on each node. Furthermore, HPDBSCAN uses the HDF5 [4] data format to store data and uses its library for achieving better parallel input and output performance.

8. Conclusion. In this paper, we have derived and implemented an integrated architecture which covers a set of requirements for providing transparent, secure and interoperable data processing tasks. Also provide these tasks access to the datasets managed by the Genesis II's Global Federated File System (GFFS). This is mainly achieved by the technical integration of UNICORE and the GFFS. The most important requirements are: R1 expresses a need for a secure trust delegation model, but should be standards-based and extensible (R2). As part of the integration the GFFS uses the SAML-based profile provided by the UNICORE's execution service. On the other hand, we extended UNICORE's identity validation that understand the GFFS-based requests containing multi-chain delegation assertions. R2 is also fulfilled by having the standards-based job execution and data management interfaces through the OGSA-BES and RNS specifications, respectively. The ByteIO standard is used to manage the data transport, thus the functionality implements R3.

While jobs are managed by UNICORE execution services, its internal service implementation is taking care of any status update delays through time out based probes against the target resource management system. In addition to that, the execution service also handles gracefully if the parallel file system on which the job's working data is stored becomes temporarily unresponsive, quite normal in production environment. The requirement R5 is served in this case.

For the Extensible resource and job model requirement R7, the resource model of the OGSA-Basic Execution Service (BES) and Job Submission and Description Language (JSDL) standards are extensible. But it will be only helpful if the compliant implementations are with minimal effort supporting the standard-allowed extensions. The technologies in our focus, UNICORE and the GFFS, are providing server and client side APIs to easily extend the resource model. This feature will be much more useful for community specific science gateways and next generation infrastructures with varying requirements. The XSEDE infrastructure has been used to demonstrate our implementation and data analysis excursion. This would require any technology and users entering the domain of an infrastructure should abide by its security model and its policies. With the GFFS client and UNICORE-based server, we used XSEDE-provided credentials to execute data processing jobs on a production deployment.

In our observation, most of the machine learning and data mining job submissions are parametric in nature, thus they need to be running multiple times. UNICORE's standard-based parameter sweep implementation helps to support our point cloud data clustering tasks. If we are able to represent the HPDBSCAN application requirement through JSDL and its parameter sweep extension, then any other data mining application can easily be supported. For the sake of implementation validity, we are analysing other methods of data mining, for example classification algorithms. One usability issue with the UNICORE's parametric sweep implementation is to produce a single job output based on some user specified criteria, which is currently not supported. The realization of this feature will reduce an overhead for data scientists to manually sort and merge the resultant job outputs. We intend to support this feature through a rule-based convergence of all the results from different parametric jobs into a single meaningful output. Another useful aspect is to avoid submitting multiple jobs to the batch system and rather use its internal feature of chaining multiple jobs. By enabling this feature the management and monitoring of complex job composites can be much more intuitive and usable.

REFERENCES

[1] *DCI Bridge Manual - v3.7.4.* http://sourceforge.net/projects/guse/files/3.7.4/Documentation/. [Online; accessed 29-Dec-2015].

[2] *FUSE (Filesystem in userspace).* https://github.com/libfuse/libfuse. [Online; accessed 29-Dec-2015].

[3] *GNU Octave.* https://www.gnu.org/software/octave/. [Online; accessed 31-Dec-2015].

[4] *Hierarchical data format version 5.* http://www.hdfgroup.org/HDF5. [Online; accessed 29-Dec-2015].

[5] *The OASIS Web Services Interoperability (WS-I).* http://www.ws-i.org/. [Online; accessed 29-Dec-2015].

[6] *TORQUE Resource Manager.* http://www.adaptivecomputing.com/products/open-source/torque-resource-manager/. [Online; accessed 31-Dec-2015].

[7] *UNICORE/X Manual.* https://www.unicore.eu/documentation/manuals/unicore6/files/unicor ex/unicorex-manual.html. [Online; accessed 29-Dec-2015].

[8] A. Anjomshoaa et al., *Job Submission Description Language (JSDL) Specification, Version 1.0.* Open Grid Forum, GFD-R.136, July 2008.

[9] A. Sim et al., *The Storage Resource Manager Interface Specification, Version 2.2.* Open Grid Forum, GFD-R-P.129, May 2008.

[10] A. Streit et al., *Unicore 6 recent and future advancements*, Annals of Telecommunications, 65 (2010), pp. 757–762.

[11] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, *Kepler: an extensible system for design and execution of scientific workflows*, in Proceedings. 16th International Conference on Scientific and Statistical Database Management., June 2004, pp. 423–424.

[12] B. Dillaway et al., *HPC Basic Profile, Version 1.0.* Open Grid Forum, GFD-R-P.114, Aug 2007.

[13] G. Behrmann, D. Cameron, M. Ellert, J. Kleist, and A. Taga, *ATLAS DDM integration in ARC*, Journal of Physics: Conference Series, 119 (2008).

[14] K. Benedyczak, P. Bała, S. van den Berghe, R. Menday, and B. Schuller, *Key aspects of the UNICORE 6 security model*, Future Generation Computer Systems, 27 (2011), pp. 195 – 201.

[15] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY, *Hadoop: a framework for running applications on large clusters built of commodity hardware.* http://hadoop.apache.org/. [Online; accessed 29-Dec-2015].

[16] D. Box et al., *Simple Object Access Protocol (SOAP) 1.1.* http://www.w3.org/TR/2000/NOTE-SOAP-20000508/. [Online; accessed 29-Dec-2015].

[17] R. F. da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny, *Toward fine-grained online task characteristics estimation in scientific workflows*, in Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science, WORKS '13, New York, NY, USA, 2013, ACM, pp. 58–67.

[18] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Commun. ACM, 51 (2008), pp. 107–113.

[19] B. Demuth, B. Schuller, S. Holl, J. Daivandy, A. Giesler, V. Huber, and S. Sild, *The unicore rich client: Facilitating the automated execution of scientific workflows*, 2013 IEEE 9th International Conference on e-Science, 0 (2010), pp. 238–245.

[20] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen, *Advanced resource connector middleware for lightweight computational grids*, Future Gener. Comput. Syst., 23 (2007), pp. 219–240.

[21] M. Ester, H. peter Kriegel, J. Sander, and X. Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise*, AAAI Press, 1996, pp. 226–231.

[22] D. B. et al., *Web Services Addressing (WS-Addressing).* http://www.w3.org/Submission/ws-addressing/. [Online; accessed 29-Dec-2015].

[23] M. D. et al., *JSDL Parameter Sweep Job Extension.* Open Grid Forum, GFD-R-P.149, May 2009.

[24] F. Bachmann et al., *XSEDE Architecture Level 3 Decomposition*, Dec 2012.

[25] I. Foster, *Globus online: Accelerating and democratizing science through cloud-based services*, IEEE Internet Computing, 15 (2011), pp. 70–73.

[26] P. Fuhrmann and V. Gülzow, *dCache, Storage System for the Future*, in Euro-Par 2006 Parallel Processing, W. Nagel, W. Walter, and W. Lehner, eds., vol. 4128 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 1106–1113.

[27] M. Götz, M. Richerzhagen, C. Bodenstein, G. Cavallaro, P. Glock, M. Riedel, and J. A. Benediktsson, *On Scalable Data Mining Techniques for Earth Science*, Procedia Computer Science, 51 (2015), pp. 2188–2197.

[28] A. Grimshaw, M. Morgan, and A. Kalyanaraman, *GFFS - The XSEDE Global Federated File System*, Parallel Processing Letters, 23 (2013), p. 1340005.

[29] I. Foster et al., *OGSA Basic Execution Service (BES), Version 1.0*, Nov 2008.

[30] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, and I. Marton, *WS-PGRADE/gUSE Generic DCI Gateway Framework for a Large Variety of User Communities*, Journal of Grid Computing, 10 (2012), pp. 601–630.

[31] I. Mandrichenko, W. Allcock, and T. Perelmutov, *GridFTP v2 Protocol Description.* Open Grid Forum, GFD-R-P.047, May 2005.

[32] S. Memon, S. Holl, B. Schuller, M. Riedel, and A. Grimshaw, *Enhancing the performance of scientific workflow execution in e-science environments by harnessing the standards based parameter sweep model*, in Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery, XSEDE '13, New York, NY, USA, 2013, ACM, pp. 56:1–56:7.

[33] S. Memon, M. Riedel, F. Janetzko, B. Demeler, G. Gorbet, S. Marru, A. Grimshaw, L. Gunathilake, R. Singh,

N. Attig, and T. Lippert, *Advancements of the ultrascan scientific gateway for open standards-based cyberinfrastructures*, Concurrency and Computation: Practice and Experience, 26 (2014), pp. 2280–2291.

[34] S. Memon, M. Riedel, C. Koeritz, and A. Grimshaw, *Interoperable job execution and data access through unicore and the global federated file system*, in Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on, May 2015, pp. 269–274.

[35] M. Morgan, *ByteIO Specification 1.0.* Open Grid Forum, GFD-R-P.87, Oct 2006.

[36] M. Morgan, A. Grimshaw, and O. Tatebe, *RNS Specification 1.1.* Open Grid Forum, GFD-R-P.171, December 2010.

[37] M. Morgan and O. Tatebe, *RNS 1.1 OGSA WSRF Basic Profile Rendering 1.0.* Open Grid Forum, GWD-R.172, December 2010.

[38] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, *Taverna: lessons in creating a workflow environment for the life sciences*, Concurrency and Computation: Practice and Experience, 18 (2006), pp. 1067–1100.

[39] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary, *A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, Los Alamitos, CA, USA, 2012, IEEE Computer Society Press, pp. 62:1–62:11.

[40] M. Riedel, M. Goetz, M. Richerzhagen, P. Glock, C. Bodenstein, A. Memon, and M. Memon, *Scalable and parallel machine learning algorithms for statistical data mining - practice amp; experience*, in Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on, May 2015, pp. 204–209.

[41] S. Cantor et al., *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0.* OASIS Standard saml-core-2.0-os, March 2005.

[42] S. Graham et al., *Web Services Resource 1.2 (WS-Resource)*, April 2006.

[43] B. Schuller, R. Menday, and A. Streit, *A versatile execution management system for next-generation unicore grids*, in Euro-Par Workshops, 2006, pp. 195–204.

[44] G. Wasson and M. Humphrey, *HPC File Staging Profile, Version 1.0.* Open Grid Forum, GFD-R-P.135, Jun 2008.

[45] A. B. Yoo, M. A. Jette, and M. Grondona, *Slurm: Simple linux utility for resource management*, in Job Scheduling Strategies for Parallel Processing, Springer, 2003, pp. 44–60.

Edited by: Enis Afgan
Received: December 21, 2015
Accepted: March 31, 2016

DOI 10.12694/scpe.v17i2.1161
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 129–141. http://www.scpe.org © 2016 SCPE

A PARALLEL ALGORITHM FOR THE STATE SPACE EXPLORATION

LAMIA ALLAL[*], GHALEM BELALEM[†], PHILIPPE DHAUSSY[‡], AND CIPRIAN TEODOROV[§]

Abstract. Model checking has long been used as a means of verification of formal specifications. This is a verification technique of dynamic systems that explores all possible states of the system. It determines whether the given system satisfies its specification. This technique suffers from the state explosion problem when traversing all possible states of systems. Parallel and/or distributed approaches are used to cope with the state space explosion problem. In this article, we propose a synchronized parallel algorithm of exploration based on a fixed number of threads. We present many experiments for a comparison between our parallel approach and the algorithm proposed for a parallel exploration in $SPIN$. We show by an experimental study that our parallel approach gives encouraging results.

Key words: Model checking, state explosion problem, parallel exploration, memory space, reachability graph, parallel algorithms, execution time, software verification.

AMS subject classifications. 68Q60, 68W10

1. Introduction.

Verification by model checking is an automatic verification technique, to verify that a system satisfies a given specification. This is a commonly used tool in situations, where, it is essential to certify the proper functioning of a system. Thus, the model checking tools are widely used in high-tech industries, for verification of electronic circuits, or even in aeronautics, to ensure safety of embedded systems. Any algorithm of model checking is based on two steps: (1) exploration of reachable states of the system; and (2) verification of the specifications in this state space. These two steps can sometimes be performed simultaneously, which is called on the fly model checking. Exploration is a computing process which determines a sequence of actions, making it possible to achieve a desired goal. A good exploration means, the achieving and the storage of a large number of states without exceeding the available memory resources [1] and in finite time. The state space can be described by an initial state and a set of transitions. A succession of states produced by actions forms a path within the state space [2, 3].

In the case of realistic examples, the number of states can be enormous. For example, in an n-bit counter, the number of states is exponential in the number of bits (2^n). Reachability analysis is limited by the state explosion problem [4, 5, 6]. This problem occurs, when the state space to be explored is large and cannot be explored by the algorithms for lack of capacity memory resources, or an important time because the memory space needed to carry out exploration is higher than the memory space contained in the machine. Many researches have been done to fight the state explosion problem, by taking advantage of a distributed environment, by increasing the computational power and using large available memory [27].

In this article, we are interested in the analysis of execution time needed to carry out exploration. During the reachability analysis, it is essential to take into account the concept of time because, even by distributing the states graph on multiple machines (at Amazon or another provider) will not settle the problem, due to the exponential growth of the number of system-states, so we need in this case to treat the temporal explosion. We present a comparative study between two algorithms for a parallel exploration. For that, we used four models: Peterson [7], Dining philosophers [8], Producer-consumer [9] and counters.

Outline of the paper: the article is divided into 7 sections, the second section presents some works which offer solutions to the state explosion problem. Section 3 presents model checking and reachability analysis. Section 4 is devoted to the definition of the synchronized parallel algorithm (SPA). The fifth section presents the parallel algorithm for state exploration in $SPIN$. The sixth section is devoted to the experiments performed for a comparison of the SPA algorithm and the algorithm proposed for $SPIN$ model checker. Section 7 presents a comparison between two parallel approaches for reachability analysis.

[*] Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 Ahmed Ben Bella, Oran, Algeria (allal.lamia@gmail.com).

[†] Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 Ahmed Ben Bella, Oran, Algeria (ghalem1dz@gmail.com).

[‡] Lab-STICC UMR CNRS 6285 ENSTA Bretagne, Brest, France (philippe.dhaussy@ensta-bretagne.fr).

[§] Lab-STICC UMR CNRS 6285 ENSTA Bretagne, Brest, France (ciprian.teodorov@ensta-bretagne.fr).

2. Model checking and reachability. Model Checking is a verification technique, based on the exhaustive state space exploration of systems, in searching of behaviors that do not verify its specification. A model checker can be seen as a black box, which accepts as input a system as well as a property expressed on this system and returns an answer, indicating if the property is checked or not. When verifying properties, through explicit-state model checking, all the possible behaviors of the system are enumerated and the properties are checked. The algorithms implemented include two phases, a construction of the state space then an exploration of this state space in searching of errors. The state space is represented as a graph which, describes all the possible evolutions of the system. Nodes of the graph represent the states of the system and the arcs represent the transitions between these states [2, 3]. The reachability analysis consists on the exploration of models state by state, each state and all its successors are stored in memory. Exploration finishes when all the states are visited. An exploration algorithm, with each step of its execution, can either visit a new node, or an explored node. *Fig. 2.1* gives the organizing chart of the basic sequential reachability for performing a breadth first search.

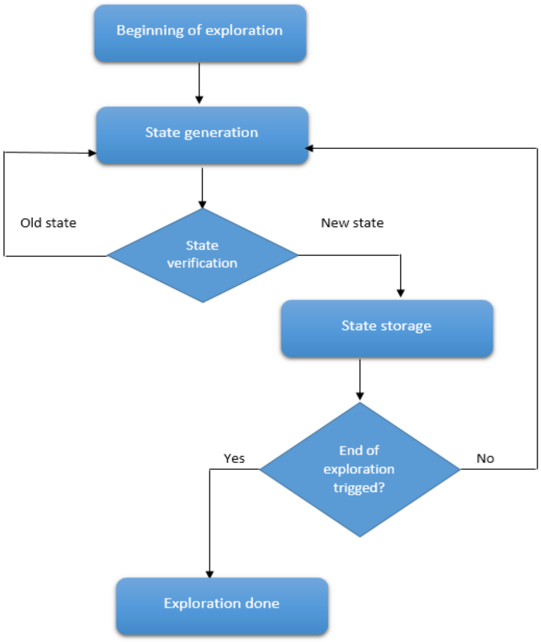


FIG. 2.1. *Sequential Reachability using a breadth first search*

3. Related work. Many solutions have been proposed for the state explosion problem. In this section, we present five solutions. Each one of them is running on a different architecture (distributed, parallel, sequential). These solutions are based on methods (states compression, partial order reduction, bit state hashing) and different data structures. Each solution aims to improve the performances in execution time and memory capacity.

In [10], the authors present a new solution to the state explosion problem. The approach is based on the concept of scalability. When checking systems by model checking, this problem can occur if the models are large. The authors propose to optimize the model checker Divine [10], so that the network can be scaled up in number of nodes, which can check larger systems. This is the main advantage of this method. In previous experiments, Divine was evaluated on a small number of nodes, it was soon determined that performance, of several Divine algorithms did not scale well, with a high number of nodes, because of the number of messages that pass through the network. The optimizations focus on improving two distributed algorithms (OWCTY and MAP) included in Divine, which allow for a good exploration of the model. Each node will process a part of state space. A major drawback of this approach, is the number of messages exchanged through the network because large messages are sent in priority which could cause an overload on the network.

The solution described in [11] allows a distribution of state space exploration, during the verification of models by model checking using $SPIN$ [12, 13]. Each node has a set V, that contains the explored states, and a queue U, to store the unvisited states. An advantage of this approach, is that the proposed algorithm is compatible with 3 methods for state space exploration (states of compression, partial order reduction, state bit hashing). A drawback of this method is the absence of a considerable gain for each model explored.

The approach proposed in [14] is based on a sequential algorithm. Its objective is the storage of states in their compressed form (this constitutes the advantage of this method), only the difference between the previous state and the following state is stored. The first generated state (initial state) is stored in an explicit way, the other states are stored in a compressed form in hash tables. The states are decompressed, to verify if a state was already visited or not, for that, it is necessary to add the most recent changes for each state, until a state stored in explicit form is reached. The disadvantage of this method, is the backtracking function, which represents an overload because the execution time can increase quickly.

The solution presented in [15] is based on an algorithm executed in parallel by multiple processors. The authors presented a model checker called PMC (Parallel Model Checker), to verify properties written in CTL* [16] that combines both of language Computation Tree Logic (CTL) [18] and Linear Temporal Logic (LTL) [17]. This model checker verifies, models whose behavior is represented as and/or trees. The proposed solution is inspired by the parallel algorithm, for pure reachability analysis [19]. Inggs & Barringer [15], presented a parallelization of processing model checking using a shared memory architecture. PMC uses a dynamic load balancing technique. Each process has its own unbounded private stack and bounded shared stack for storing work items. Shared stacks are protected by locks to synchronize read and write access to it. During model checking, processes have to interact with each other, via the shared memory, to divide new work items and to avoid duplication of work. The advantage of this method is the presence of private and shared stacks for each processor.

Inggs & Barringer [15] presented the results for three types of experiments. The drawback of this approach, is the use of a shared memory, which requires synchronization between processes.

In [20], the authors propose a parallel algorithm for the construction of state space. The architecture used is a shared memory multiprocessor architecture. States are stored in the local hash tables (lockless hash table). This constitutes the advantage of this method. Each processor has a private stack and a shared stack. Distribution and coordination of states between the processors is made through a location table, which contains the list of states that have been visited. It is used to dynamically allocate works on the processors. Its primary role is to return true, if the state was visited and the number of process running, false otherwise. Collisions may occur, if the key returned by the hash function, is the same for both states. This constitutes the drawback of this approach.

4. Synchronized parallel algorithm (SPA). A parallel machine is essentially a set of processors that cooperate and communicate. A parallel algorithm runs on a parallel computer. The instructions are executed simultaneously, which can lead to a considerable gain in execution time. An important task in a parallel approach, is the assignment of work to threads, to have a load balancing between threads.

The Synchronized parallel algorithm (SPA), is based on the use of a fixed number of threads and uses 2 sets of states: K and $Q[i]$. Set K, is the set of visited states, this set is shared by all the threads, then, the access is synchronized. Successors' states are stored in Queue $Q[i]$, where i varies from 1 to N (N is the number of threads). $Q[i]$ is a FIFO queue where each thread i processes states in $Q[i]$. The size of this queue is unlimited. Each pointer $Q[i]$, points to a linked list of nodes (value, link to the next node). Each time a state is generated, a thread will be randomly selected and the state will be stored in its list (we can't have at the same time, removal and adding operations). Random function was used, to load balancing states on subsets $Q[i]$. Synchronization between different threads, is performed on the lists of each thread during states adding, or states removal. We have a fixed number of threads, so we need to be able to determine when all states have been reached, to stop the exploration.

Fig. 4.1 shows the threads synchronization on the Queue $Q[i]$ containing the states to be processed. At the first step, thread 1 generates the initial state. For each next state, a thread is generated randomly and the state

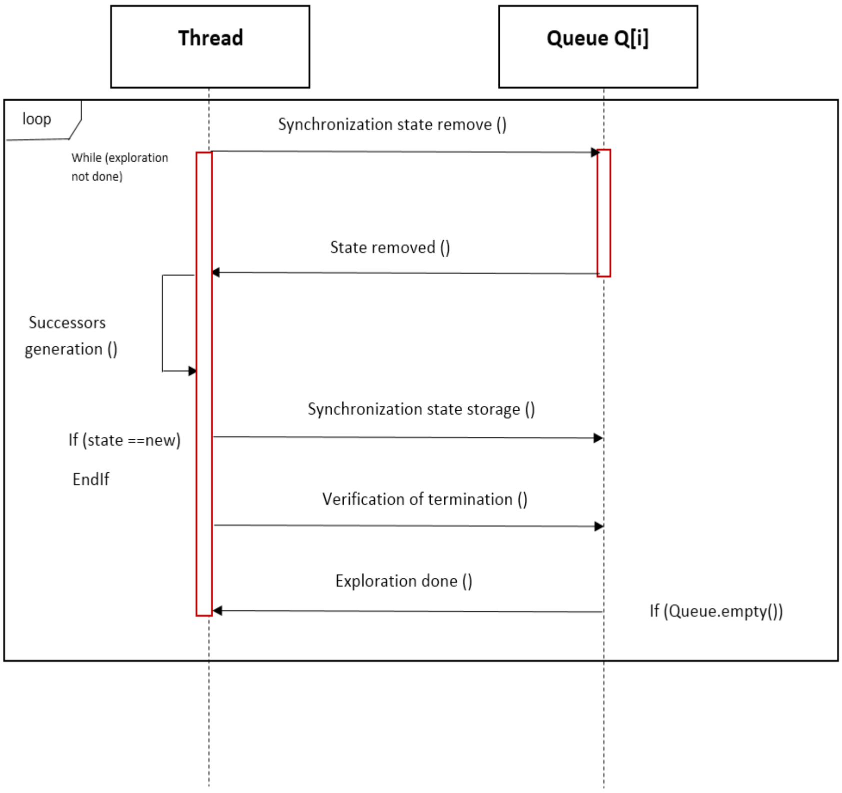

Thread
Queue Q[i]
loop
While (exploration not done)
Synchronization state remove ()
State removed ()
Successors generation ()
Synchronization state storage ()
If (state ==new)
EndIf
Verification of termination ()
Exploration done ()
If (Queue.empty())

Fig. 4.1. *Threads Synchronization on the queue $Q[i]$*

is stored in its list. After this step, each thread i where $Q[i]$ is not empty, will remove a state from the queue by the synchronization state remove function, only one thread can execute this function at the same time (critical section). The successors of this state are generated by the successors generation function. After that, a test is realized on each next state, if the state is new, it is stored randomly in the queue $Q[k]$ of the thread k, by executing the state storage function, otherwise it goes to the next state. Termination verification is triggered, to check whether, termination of exploration has been reached, by the verification of termination function. If the queue $Q[i]$ is empty, exploration ends, otherwise the process will be repeated as long as the queue is not empty.

The algorithm of parallel exploration is presented in what follows.

At the beginning of the exploration, the initial state is generated, then its successors are observed. At this time, for each new configuration, a thread is chosen randomly (line 9 of *Algorithm 1*). Data used are shared between threads, synchronization are made on the set k containing all visited states and on the queue $Q[i]$ for processing current states.

To explain the exploration process, we have taken an example of counters (*Fig. 4.2*). The representation of a counter that is incremented and decremented, is carried out using a deterministic automaton consisting of a single state (both initial and terminal). The state s is incremented up to N, s can be decremented at each step down to 0, which represents the initial state. If we add another counter, there could be a state vector with 2 cells, therefore, with N counters, we will have a state vector composed of N cells, corresponding to n automaton (*Fig. 4.3*). For example, by fixing the specific value to 5 and having 6 counters, the total number of states to be explored is equal to 46656 ($(the\_specific\_value + 1)^{the\_number\_of\_counters}$). The number 1 corresponds to the minimum value 0. *Fig. 4.3* presents a part of the reachability graph of 6 counters that can be incremented or decremented on each step. For each explored state, a set of generated successors is generated. The first step, is the generation of initial state (0 0 0 0 0 0) by thread 1. From the initial state, six states are discovered, because

Algorithm 1 Synchronized Parallel Algorithm (SPA)

```
$exploration\_done \leftarrow false$
For each (w: 1..N)
do
    for (each s in 1 .. N) do
        (Synchronized) delete s from $Q[w]$
        for (each successor s of s) do
            if ((Synchronized) s not in K) then
                (Synchronized) add s to K
                w= choose Random 1 .. N
                (Synchronized) add s to $Q[w]$
            end if
        end for
    end for
    if (w==1) then
        if (all $Q[1..N]$ == NULL) then
            done ← true
        end if
    end if
while !exploration_done
```

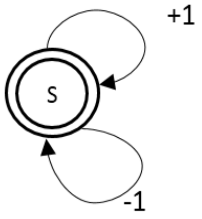


Fig. 4.2. *Automaton of a counter that increments and decrements*

at each transition, a counter can be incremented. A transition from one state to another occurs, when a counter is incremented or decremented. The initial state is stored in the set K, because the state is new and in the Queue $Q[1]$, to explore its next configurations (states). Thereafter, all its successors are visited and stored in their turn in both sets, to do that, we need to choose randomly, a thread x for each successor c. Each next configuration c is stored in $Q[x]$. The exploration stops when the queue $Q[i]$ is empty, i varies from 1 to N.

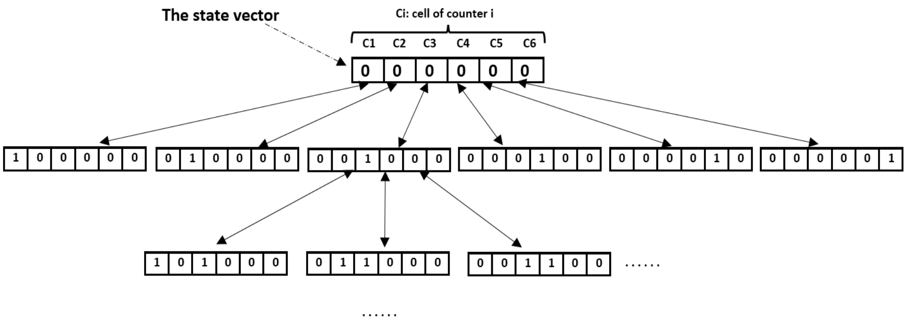


Fig. 4.3. *A part of Reachability graph of 6 counters incremented up to 5 and decremented down to 0*

5. Parallel exploration in SPIN. $SPIN$ is a tool for verification and simulation of concurrent systems. To be studied, a concurrent system is first described in Promela (Process Meta Language), the $SPIN$ verification language [26]. The algorithm (*Algorithm 2*) proposed in [22], is based on the use of a three dimensional queue $Q[t][i][j]$ for storage of states, whose successors have not been observed yet. It is composed of 3 parameters: t, i and j. The parameter t, is varied from 0 to 1, it allows states to pass from current states to future states. At each step of exploration, all states from $Q[t][i][j]$ are processed and their successors are stored in $Q[1-t][i][j]$, corresponding to the configurations that will be observed at the next step. A lockless hachtable [23], was used in order to avoid waits between different threads. An important task in the algorithm proposed in [22], is to determine when all states have been explored to stop exploration.

Algorithm 2 Parallel Exploration Algorithm in $SPIN$

```
$done \leftarrow false$
$t \leftarrow 0$
Search (i: 1..N)
do
    for (each state in 1..N) do
        Delete s from Q[t][i][j]
        for (Each next configuration c of s) do
            if ¬ (S.Contains(c)) then
                S.add(c)
                k ← Choose Random from 1..N
                add state to $Q[1-t][k][i]$
            end if
        end for
    end for
    Wait()
    if (i==1) then
        wait until all threads are idle
        if (all $Q[1-t][i][j]$ == NULL) then
            done ← true
        else
            Notify all threads
            t ← 1-t
        end if
    end if
while !done
```

In this algorithm, the parameters i and j vary from 1 to N, where N is the number of threads. The size of this queue is unlimited. The current states to be generated are treated from $Q[t][i][j]$ and successors are add to $Q[1-t][k][i]$ with k, a thread selected randomly from N. The conceptual difference between the SPA and $SPIN$ algorithm is that, in $SPIN$, a three dimensional queue is used, which consumes more memory space. SPA algorithm is based on the use of a one dimensional queue. A thread that ends exploring its existing states, waits for all threads to finish their treatment, to pass to future states. In the SPA algorithm, there is no waiting between threads, threads wait when adding or removing state in (from) the queue.

6. Experimental study. In this article, we presented a parallel approach for state space exploration. We carried out four experimental studies on 4 different models, 3 models from BEEM database [21] and a counter model. The objective is to make a comparison between two parallel approaches. This comparison is based on the execution time of the exploration step. The experiments are performed by varying some metrics. We study the behavior of these approaches by experiments. The experiments were performed on an i7 machine with 8 cores, it operates at a frequency of 2 MHz, with 16 GB of physical memory. We have implemented both parallel algorithms using java platform [25]. We realized the specification of these models in Java. We have fixed the

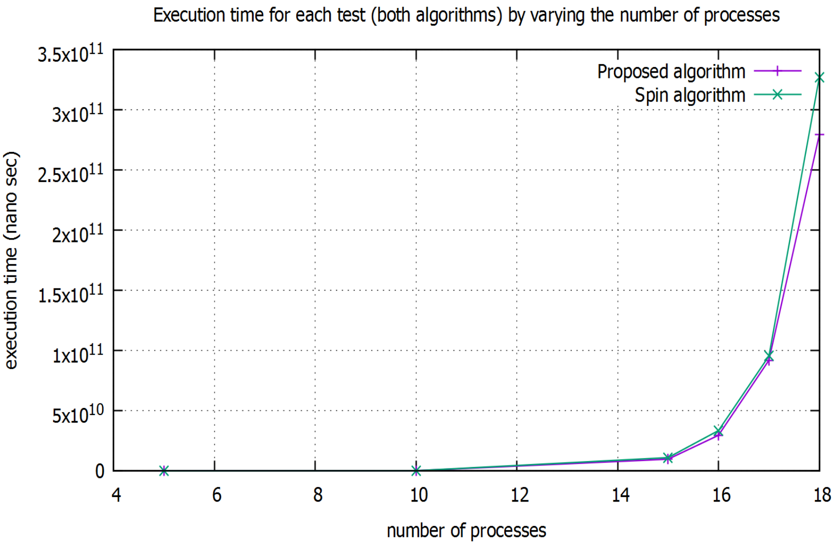


Fig. 6.1. *Execution time (both parallel approaches) by varying the number of processes*

Table 6.1
Execution time, configurations number, and processes for both algorithms

Processes	**Configurations**	**Exe. time** SPA **algo (seconds)**	**Exe. time** $SPIN$ **algo (seconds)**
5	352	0.017	0.022
10	47104	0.20	0.21
15	3473408	9	11
16	7929856	29	33
17	17956864	91	95
18	40370176	279	327

number of threads at each experiment. The launch of threads has been realized using the class thread. The number of states is given at the end of each experiment by the set K (set of visited states). The size of this set is equal to the number of states. No information is calculated or communicated to each state, because, the purpose of reachability analysis, is to load all the states in memory. The number of threads to perform exploration is the same for both algorithms on each experiment.

6.1. Experiment 1: Peterson model. Peterson's algorithm [7], is a mutual exclusion algorithm for concurrent programming. This algorithm is based on an active wait approach. It consists of two parts: the input into the critical section and the output from it. We have made a parallel comparison between, our approach, and the parallel algorithm developed in $SPIN$ model checker [22], using Peterson model [7].

We made 6 different tests, by varying the number of concurrent processes (5, 10, 15, 16, 17, 18), and estimated execution time of reachability analysis for each test (see *Fig. 6.1* and *Table 6.1*). The number of states varies from 352 to 40370176, regarding to the number of processes accessing a critical section. At each test, we fixed the number of created threads (process machine) for a parallel execution (from 2 to 8). At the last test (with 18 processes accessing a critical section), we used 8 threads to fully exploit all machine resources. From the results provided by this experiment, SPA algorithm shows better performance in execution time (*Fig. 6.1*). The comparison was made on the execution time estimated to perform experimentations. In the algorithm presented in [22], whenever a thread finishes processing its tasks (lists of current states), it waits for other threads, therefore, it takes more time because it is based on the processing of states step by step. Having 18 processes accessing a critical section, the synchronized parallel algorithm shows better performance, because, for each process added (process in critical section), more configurations will be observed, because, the state

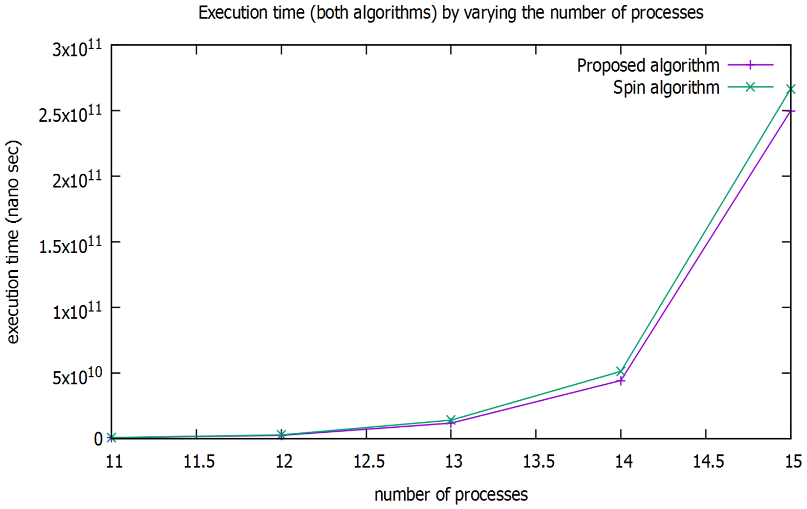


Fig. 6.2. *Execution time (both parallel approaches) by varying the number of processes*

Table 6.2

Execution time, configurations number, and processes for both algorithms

Processes	**Configurations**	**Exe. time** SPA **algo (seconds)**	**Exe. time** $SPIN$ **algo (seconds)**
11	393660	0.86	0.88
12	1240029	2	3
13	3897234	11	14
14	12223143	44	51
15	38263752	249	266

vector will change. The state vector, is an array, where each cell corresponds to the identifier of a process in critical section.

To interpret these results in execution time, we calculated the gain (in percentage) obtained by our proposed algorithm from each experience (21.45, 3.31, 12.10, 11.75, 3.64, 14.63). From these results, we estimate the average gain obtained from all tests, using Peterson model specification, $T_{average\_peterson} = 11.15\%$.

6.2. Experiment 2: Dining philosophers model. The dining philosophers model [8], is used to solve the synchronization problems between different processes. we realized the specification of this model in Java, and then, we performed an exploration on this model using both parallel algorithms. The result about execution time estimated, is given in *Fig. 6.2* and *Table 6.2.* We realized the experiments on the same machine, we have varied the number of processes in critical section from 11 to 15, with a step of 1 and we measured at every experiment, the execution time estimated during exploration. The number of states increases regarding to the number of concurrent processes, all possible case are exploited (all possible states are explored).

Having 15 concurrent processes, the number of configurations (states) is about 38.263.752. We can notice a performance gain, using the proposed approach compared to the parallel exploration in $SPIN$ model checker. The average gain obtained using our algorithm is about $9.90\% \simeq 10\%$. Concerning the model specification and reachability analysis, our algorithm shows better performance compared to $SPIN$ algorithm. The gain increases by increasing the number of processes in critical section, therefore, the proposed algorithm scales well.

6.3. Experiment 3: Producers consumer model. The Producers/Consumer model [9], is a classic example of two synchronization processes, one that produces information he deposited in a limited buffer size, and one that removes one by one to consume. We have done the specification of this model, using several producers and one consumer. We realized 5 tests, by varying the number of producers and the table size, in which data are stored. The comparison is made on execution time (*Fig. 6.3* and *Table 6.3*). In the first test,

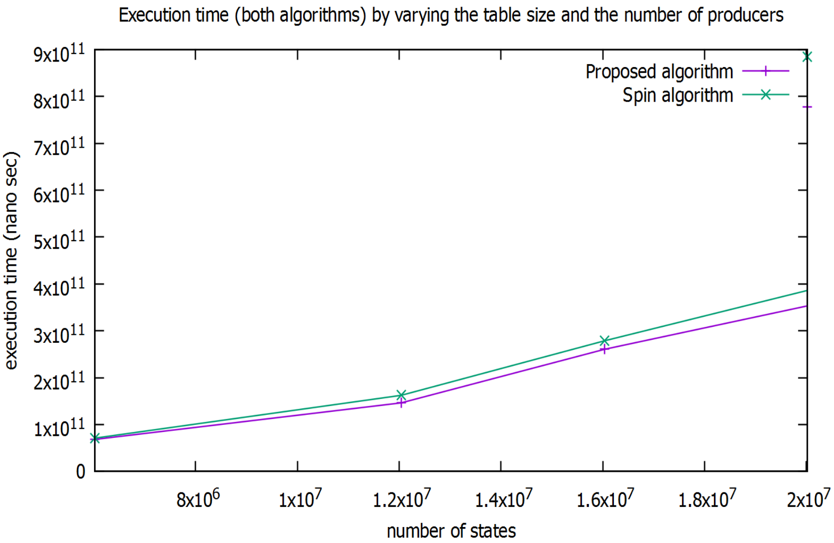


Fig. 6.3. *Execution time (both parallel approaches) by varying the table size and the number of producers*

Table 6.3
Execution time, and configurations number for both algorithms

Configurations	**Exe. time *SPA* algo (seconds)**	**Exe. time *SPIN* algo (seconds)**
6020000	68	70
12040001	146	162
16040001	260	278
30060001	586 ($\simeq 10min$)	656 ($\simeq 11min$)
20020001	777 ($\simeq 13(min)$)	885 ($\simeq 15(min)$)

we used 600 producers and the table size was fixed at 10000 cells. The reachability graph corresponding to this experiment is composed of 6.020.000 states (configurations). The experiments and the number of states at each test, are given in *Table 6.4*.

Table 6.4
Experiments parameters and the total number of states obtained

Producers	**Table size (cells)**	**Number of states after reachability analysis**
600	10000	6.020.000
600	200000	12.040.001
800	20000	16.040.001
1000	30000	30.060.001
2000	20000	20.020.001

Comparing the results given by both approaches, we note that our algorithm shows better results in execution time, compared to the parallel exploration algorithm in *SPIN* in each experiment. The average gain obtained by performing a comparison by *SPA* algorithm, is estimated by 8.62%.

6.4. Experiment 4: Counters model. We have made a parallel comparison between our approach, and the parallel algorithm developed in *SPIN* model checker [22], using counters model. We used 5 counters, incremented from 0 to a maximum value, and made 6 tests. In the first one, the counters are incremented from 0 to 22. At each test, we incremented a maximum value by a step of 3 (from 0 to 22, from 0 to 25, ...). The result, is shown in *Fig. 6.4* and *Table 6.5*.

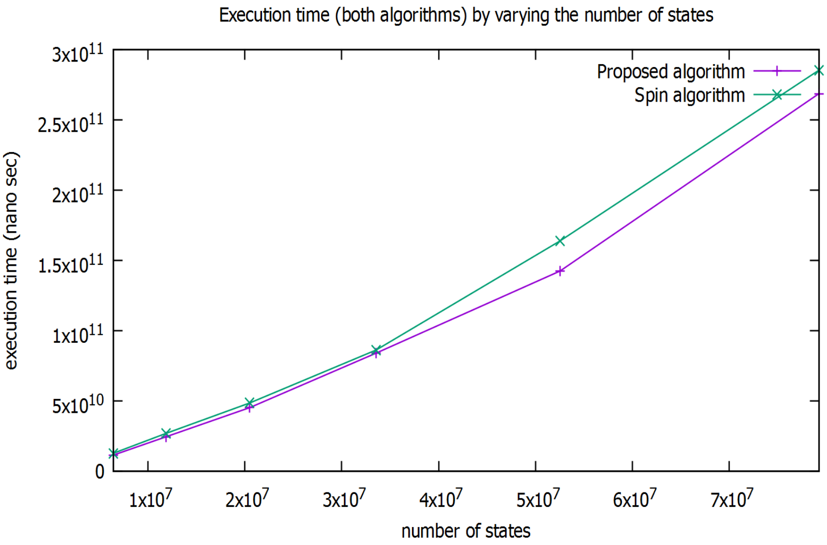


Fig. 6.4. *Execution time (both parallel approaches) by varying the number of configurations*

Table 6.5
Execution time, configurations number, and max_value for both algorithms

Max_value	**Configurations**	**Exe. time** SPA **algo (seconds)**	**Exe. time** $SPIN$ **algo (seconds)**
22	6436343	11	13
25	11881376	24	27
28	20511149	45	48
31	33554432	84	86
34	52521875	142	164
37	79235168	268	285

To calculate the number of states, we need the number of counters and the maximum value of counter. Then, this number is calculated by: $(max\_value+1)^{number\_of\_counters}$. Regarding to the experiment shown in *Fig. 6.4*, our algorithm gives better results, by increasing the maximum value of counters. Both parallel approaches show significant results, when the number of configurations is high, with a gain provided by the synchronized parallel algorithm.

Despite the gain of 10%, we can notice at each experiment, that the gap between both curves (SPA and $SPIN$), becomes important when the number of processes increases, which predicts that these curves move further for larger number of processes, and therefore, the gain obtained by our approach will be more significant.

7. Positioning our parallel approach and Discussion. To position our parallel algorithm, with the integrated parallel algorithm in $SPIN$, which is very used in the exploration of states using model checking techniques, we studied the conceptual difference between both algorithms (SPA and $SPIN$), and the complexity of these two algorithms [24].

Conceptual difference between both algorithms: We realized in the previous section, a series of experiments. The SPA algorithm showed in each one, better results. This is due to the fact, that in the $SPIN$ approach, several loops are executed each time we alternate between current and future states: having N threads, for each thread i, all $Q[t][i][j]$ are explored (j varies from 1 to N). The treatment of visited states, for this thread, ends only after all queues are empty. The process is the same for other threads. In the proposed approach, each thread i manages its queue $Q[i]$, so there is no time lost. In $SPIN$, at each step (alternation between current and future states), a thread that ends its processing, waits for all other threads. A time is lost at this level, which represents the major drawback of this algorithm ($SPIN$ algorithm), especially if the wait is

Table 8.1
Data structures specified in the SPA algorithm

Data structures	Memory space used
exploration_done	32 bits
pointer queue $Q[i]$	32 bits
Linked list (queue $Q[i]$)	(32 + 32) * s bits
K (set of reached states)	(32 + 32) * t bits (key and data)

long. Waiting in the SPA algorithm are in the access to the queue $Q[i]$ for removal, or addition of states. This time is smaller, compared to the wait between threads in $SPIN$. This is demonstrated by the results obtained.

Study of the algorithmic complexity: an algorithm, is a sequence of actions performed from an initial state, to a final state in a finite time. We study the complexity to predict the execution time of an algorithm, and to compare two algorithms performing the same treatments. The complexity of an algorithm, is determined through a description of the behavior of algorithms. The complexity of an algorithm can be evaluated in time (speed), and in space. In this article, we focus on the study of the execution time. We conducted a study of the complexity, for both parallel algorithms: SPA algorithm, and the algorithm integrated in $SPIN$, for this, we have defined execution time for each type of instruction:

- ae: state assignment
- ce: comparison of states
- s: number of next states per state
- q: maximum number of states in $Q[i]$ or M
- p: number of threads
- w: waiting time per thread (idle)

Taking each operation, we estimate the time needed to achieve the reachability analysis, by each algorithm:

- **complexity of the synchronized parallel algorithm "SPA" (*Algorithm 1*, section 4)**: complexity of the proposed algorithm C_{App} is estimated by:

$$C_{App} = ae + p(ae + s.q(ce + 3.ae)) + ce + p.ce)) = O(qp) \tag{7.1}$$

- **complexity of the parallel algorithm proposed in [22] (*Algorithm 2*, section 5)**: complexity C_{SPIN} is estimated by:

$$C_{SPIN} = 2.ae + p(ae + s.p.q(ce + 2.ae)) + w(p - 1) + q.ce.ae = O(qp^2) \tag{7.2}$$

According to these complexities obtained by equations (7.1) and (7.2), we can notice that our algorithm has order of $O(qp)$ time complexity, the complexity of the algorithm proposed in [22] is around the square, estimated to $O(qp^2)$. In conclusion, we can say and confirm that our proposed algorithm, for the exploration of states, can be used to explore a large number of states, in a linear time.

8. Memory space used in reachability analysis. Model checking, is a technique based on verification of the correctness of a system, with respect to a desired behavior and properties. Exploration consists on exploring each state (we have to iterate algorithm many times), therefore, having systems composed of large number of states, that tend to grow exponentially, in the number of its processes and variables, this case leads to a state space exploration for large systems. In this section, we analyzed the memory space used in both algorithms, by counting the total number of data structures used.

- **Memory space used in the synchronized parallel algorithm "SPA" (*Algorithm 1*, section 4)**: An integer value, is specified in the source code of any program as a sequence of digits. Usually, variables are stored on 32 bits, therefore, to analyze the used memory space, we have to count the number of data structures declared in the algorithm. The data structures specified in the algorithm, are listed in *Table 8.1.* exploration_done is a boolean variable, it is stored on 32 bits, we have N pointers of queue $Q[i]$, with N referring to the number of threads and each $Q[i]$ (i varies from 1 to N), points to a liked list of states. A number of next states is unknown. We have N linked lists, and each one contains nodes,

TABLE 8.2
Data structures specified in the parallel algorithm in SPIN

Data structures	**Memory space used**
done	32 bits
t	32 bits
pointer queue $Q[i]$	32 bits
list of points (thread's pointer to other threads)	32 * N bits
Linked list (for one thread)	(32 + 32) * s * N bits
S (set of reached states)	(32 + 32) * t bits (key and data)

each node contains two fields, an integer value and a link to the next node. Therefore, the memory used for states storage in the queue $Q[i]$ is (32 + 32) * s * N (s is the maximum number of states in a list of a thread). A set K contains explored states.

The size of K is determined by: (32 + 32) * t, with t the total number of states explored by reachability analysis. By having these information, we can estimate the state space storage, used by our approach, expressed in bits: $Memory\_used_{SPA\_app}$ = 32 + (32 * N) + ((32 + 32) * s * N) + ((32 + 32) * t) = $32(1 + N + (2 * s * N) + (t * 2)) = 32(1 + N(1 + (2 * s) + (t * 2))$.

- **Memory space used in the parallel algorithm in** $SPIN$ **(*Algorithm 2*, section 5)**: In the parallel algorithm developed in $SPIN$ model checker, 2 sets are specified, $Q[i]$ (3 dimensional queue) and S (set of reached states). The data structures specified in the algorithm are listed in *Table 8.2.* done is a boolean variable that indicates whether all states have been reached or not. The variable t, is stored on 32 bits. There are N pointers of queue $Q[i]$, each one points to a list of pointers (N pointers), linking to linked lists. Each thread, maintains N lists of states, the memory space used for states storage in the queue $Q[i]$ is (32+32) * s * N (s is the maximum number of states in a list of a thread). The memory space, is allocated for storage of current and future states, therefore, space is allocated for $Q[0][i][j]$ and $Q[1][i][j]$.

  The size of S is determined by: (32 + 32) * t (t is the total number of states explored). We can estimate the state space storage used by the parallel algorithm proposed in [22] in bits: $Memory\_used_{algo\_SPIN}$ = 32 + 32 + 2 (32 * N) + 2 (32 * N * N) + (2* N) *((32 + 32) * s * N) + ((32 + 32) * t) = $32(1 + 1 + (2 * N) + (2 * N^2) + 4 * s * N^2 + (t * 2)) = 32(2 + N(2 + (2 * N) + (4 * s * N)) + (t * 2))$.

A less memory space used is necessary to fight the state explosion problem. From this analysis, we can conclude that our algorithm uses less memory than the parallel algorithm developed in $SPIN$.

9. Conclusion and future work. Model checking is a set of an automatic verification techniques of temporal properties on reactive systems. It takes as input, a system of transitions and a formula from some temporal logic, and answers if the abstraction satisfies or not the formula. This technique suffers from the state explosion problem, where systems become too large. In this article, we have proposed a parallel approach to the state space exploration. We realized many experiments, for a comparison between our algorithm, and the algorithm proposed in [22]. As first experiment, we measured the performance of both parallel algorithms, using Peterson model [7] then we compared the results. We showed that our approach gives better results. In the second experiment, we measured performances in terms of execution time, obtained by both parallel algorithms, using Dining Philosophers model [8], we calculated and noticed that our approach produces better results. In the third experiment, we made a comparison between both parallel algorithms, using Producer-Consumer model [9], we noticed an improvement of performance in execution time, reported by our approach. In the last experiment, we used counters model, and concluded that our approach gives better results. We work on the execution of experiments on a distributed environment, to improve performance, we are about to conduct tests, using models observed in the article.

REFERENCES

[1] N. ABED, S. TRIPAKIS, AND J. M. VINCENT, *Resource-Aware Verification Using Randomized Exploration of Large State Spaces*, Model Checking Software. In Proceedings of the 15th International SPIN Workshop, August 10-12, 2008, Los

Angeles, CA, USA, Proceedings, pp. 214–231.

[2] M. C. Boukala, and L. Petrucci, *Towards distributed verification of petri nets properties*, In Proceedings of the First international conference on Verification and Evaluation of Computer and Communication Systems VECoS '07, May, 2007, Algiers, Algeria, pp. 13–24.

[3] S. Christensen, L. M. Kristensen and T. Mailund, *A Sweep-Line Method for State Space Exploration*, In Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '2001, April, 2001, Genova, Italy, pp. 450–464.

[4] E. M. Clarke, W. Klieber, M. Novček, and P. Zuliani, *Model Checking and the State Explosion Problem*, In Proceedings of the 8th Laser Summer School on Software Engineering, September, 2011, Elba Island, Italy, pp. 1–30.

[5] N. Guan, Z. Gu, W. Yi, and G. Yu, *Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs*, In Proceedings of the 14th Asia and South Pacific Design Automation Conference, ASP-DAC '09, January, 2009, Pacifico Yokohama, Yokohama, Japan, pp. 715–720.

[6] R. Pelánek, *Fighting State Space Explosion: Review and Evaluation*, In Proceedings of the 13th on Formal Methods for Industrial Critical Systems, FMICS '08, September, 2008, L'Aquila, Italy, pp. 37–52.

[7] G. L. Peterson, *Myths About the Mutual Exclusion Problem*, Inf. Process. Lett., 12(3): 115–116, 1981.

[8] K. M. Chandy, and J. Misra, *The Drinking Philosophers Problem*, ACM Trans. Program. Lang. Syst., 6(4): 632–646, 1984.

[9] K. Jeffay, *The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*, In Proc. ACM/SIGAPP Symp. on Applied Computing, 1993, pp. 3796–804.

[10] K. Verstoep and H. E. Bal and J. Barnat and L. Brim, *Efficient Large-Scale Model Checking**, IEEE International Parallel and Distributed Processing Symposium, IPDPS'09, May, 2009, Rome, Italy, pp. 1–12.

[11] F. Lerda, and R. Sisto, *Distributed-Memory Model-Checking With SPIN*, In Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, July, 1999, Trento, Italy, pp. 22–39.

[12] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, 23(5): 279–295, 1997.

[13] P. Dhaussy, J. C. Roger, and F. Boniol, *Reducing State Explosion with Context Modeling for Model-Checking*, In Proceedings of the 13th International Symposium on High Assurance Systems Engineering, November, 2011, Boca Raton, Florida, USA, 1990, pp. 130–137.

[14] A. Mukherjee, Z. Tari, and P. Bertok, *Memory Efficient State-space Analysis in Software Model-checking*, In Proceedings of the Thirty-Third Australasian Conference on Computer Science, ACSC '10, January, 2010, Brisbane, Australia, pp. 23–32.

[15] C. P. Inggs, and H. Barringer, *CTL* Model Checking on a Shared-memory Architecture*, Formal Methods in System Design, 29(2): 135–155, 2006.

[16] E. A. Emerson, and J. Y. Halpern, *"Sometimes" and "not never" revisited: on branching versus linear time temporal logic*, Journal of the ACM (JACM) - The MIT Press scientific computation series, 33(1), pp. 151–178, 1986.

[17] A. Pnueli, *The temporal logic of programs*, In Proceedings of the 18th Annual Symposium on Foundations of Computer Science, October, 1977, Providence, Rhode Island, USA, pp. 46–57.

[18] E. M. Clarke, and E. A. Emerson, *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*, In Logic of Programs, Workshop, May, 1981, pp. 52–71.

[19] T. Heyman, D. Geist, O. Grumberg, and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, In Proceedings of the 12th international Conference on computer aided verification, CAV2000, July, 2000, Chicago, Illinois, USA, pp. 20–35.

[20] R. T. Saad, S. D. Zilio, and B. Berthomieu, *Mixed Shared-Distributed Hash Tables Approaches for Parallel State Space Construction*, In Proceedings of the 10th International Symposium on Parallel and Distributed Computing, ISPDC '11, July, 2011, Cluj-Napoca, Romania, pp. 9–16.

[21] R. Pelánek, *BEEM: Benchmarks for explicit model checkers*, In Proceedings of the 14th International Spin Workshop on Model Checking Software, Springer Verlag, LNCS Vol. 4595, 2007, pp. 263–267.

[22] G. J. Holzmann, *Parallelizing the Spin Model Checker*, In Proceedings of the 19th International Conference on Model Checking Software, SPIN'12, 2012, Oxford, UK, pp. 155–171.

[23] C. P. Inggs, and H. Barringer, *Effective State Exploration for Model Checking on a Shared Memory Architecture*, Electr. Notes Theor. Comput. Sci., No. 4, 2002, pp. 605–620.

[24] A. M.C.A. Koster and M. Tieves, *Network design with compression: complexity and algorithms*, INFORMS Computing Society Conference (INFORMS ICS), 2015.

[25] K. Arnold, and J. Gosling, *The Java Programming Language (2Nd Ed.)*, INFORMS Computing Society Conference (INFORMS ICS), ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[26] G. J. Holzmann, *The Model Checker SPIN*, IEEE Trans. Softw. Eng., 23(5): 279–295, May, 1997.

[27] M. C. Boukala, and L. Petrucci, *Distributed CTL Model-Checking and counterexample search*, International Journal of Critical Computer-Based Systems (IJCCBS), 3(1), January, 2012.

Edited by: Roman Trobec
Received: December 21, 2015
Accepted: March 31, 2016

DOI 10.12694/scpe.v17i2.1162
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 143–153. http://www.scpe.org © 2016 SCPE

PROVENANCE BASED CHECKPOINTING METHOD FOR DYNAMIC HEALTH CARE SMART SYSTEM[*]

ESZTER KAIL[†] KRISZTIÁN KARÓCZKAI[‡] PÉTER KACSUK[§] AND MIKLÓS KOZLOVSZKY[¶]

Abstract. Smart systems in telemedicine frequently use intelligent sensor devices at large scale. Practitioners can monitor non-stop the vital parameters of hundreds of patients in real-time. The most important pillars of remote patient monitoring services are communication and data processing. Large scale data processing is done mainly using workflows. Some workflows are working in real-time, more complex ones are running for days or even for weeks on parallel and distributed infrastructures such as HPC systems and cloud. In HPC environment high number of failures can arise during health care smart systems workflow enactment, so the use of fault tolerance techniques is unavoidable. The most frequently used fault tolerance technique is checkpointing. The effectiveness of the checkpointing method depends on the checkpointing interval. In this work we give a brief overview of the different checkpointing techniques and propose two new provenance based checkpointing algorithms which uses the information stored in the workflow structure to dynamically change the frequency of checkpointing and can be efficiently used for dynamic health care smart systems.

Key words: scientific workflow, health care, fault tolerance, checkpointing

AMS subject classifications. 68M14

1. Introduction.

Smart systems are by definition miniaturized devices that incorporate functions of sensing, actuation and control. Telemedicine frequently uses intelligent sensor devices at large scale. The deployed health care devices (such as ECG, pulse meters, blood glucose meters, etc.) at patient's home are firstly sensing then later transmitting huge amount of data, thus building up communication intensive smart system networks. A single practitioner can easily monitor vital parameters of hundreds of patients 24/7 in real time. Remote patient monitoring has several advantage as: the patient is monitored location independently, not restricted physically to a single place, patient is not connected tightly to the doctor, or to the hospital, on the contrary patients can be monitored during their everyday life, within their normal environment, medication can be adapted to the patient's normal lifestyle and not to an artificial situation when they are hospitalized. The most important pillar of telemedicine services is communication. Well matured technologies, practices, industry standards and reliable infrastructure are available for sensor data transport and storage.

The medical diagnosis requires more than just simple data collection and visualization of sensor data. The sensor data should be pre/post-processed (data filtering, cleaning and analysis of basic sensor data patterns). Sensor data processing is by definition a complex task, and it is mainly a special Big Data challenge both for engineers and for data processing systems. The elementary data size can vary from the very small as a few bytes to some thousands of bytes, however, if we look at the number of already available data sources and their data sending frequency, it turns out, that even simple real-time health care monitoring tasks (e.g.: measure pulse, 6-12 channels ECG -600 Hz-, and SpO2) requires more processing resources than conventional systems can effectively handle.

Data processing workflows used in health care smart systems are data and computational intensive, thus they may require long execution time, which in certain cases can even last for days. During long intervals it is inevitable to adapt to the dynamically changing environment which can be caused by unwanted input data, crash faults or network problems. In one of our earlier works [5] we defined the main requirements of dynamic workflow execution systems as: the ability to react to or to handle unforeseen scenarios raised during the workflow enactment phase, to adapt to new situations, to change the abstract or concrete workflow model

[*] This work was supported by EU project SCI-BUS

[†] Óbuda University, John von Neumann Faculty of Informatics, Biotech Lab Bécsi str. 96/b., H-1034, Budapest, Hungary (kail.eszter@nik.uni-obuda.hu).

[‡] MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary

[§] MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary and University of Westminster, 115 New Cavendish Street, London W1W 6UW

[¶] Óbuda University, John von Neumann Faculty of Informatics, Biotech Lab Bécsi str. 96/b., H-1034, Budapest, Hungary and MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary or to give faster execution and higher level performance according to the actual environmental conditions and intermediary results. We can define three main areas of dynamism which are optimization of the workflow execution according some criteria, user-steering (user or administrator interaction during execution) and fault tolerance behavior [4]. In this work we delve mainly into fault tolerance behavior.

Fault tolerance is the ability of a system to perform its functions even in the presence of a failure [2]. There are two main groups of failures that could arise during enactment. The first group includes the crash faults or fail-stop faults which may come with faulty system components that result in complete data loss. The other group consists of byzantine faults which result the system components to behave unpredictably and maliciously. Byzantine failures can occur, e.g., due to software bugs, (transitional or permanent) hardware malfunction, or malicious attack. In our work we consider only crash faults where the complete state of the actual task and environment must be restored.

Before we go more detailed into fault tolerance we need to define three important terms:

- Checkpointing is a technique to save the complete status of an executing program or job and to restore and restart from this so called checkpoint (which is basically a snapshot of the program's state) later if the original program or job was halted due to system failures.
- Breakpoint: is an intentional stopping or pausing place in a program. It is used extensively by software developers and testers to stop the program whenever a certain point in the program is reached. For each breakpoint, conditions can be added to control in finer detail whether the program should stop.
- Watchpoint: It is a special conditional breakpoint that stops the program when the value of an expression changes. Watchpoints are not set for functions or lines of code, but are set on variables. When those variables are read or written, the watchpoint is triggered and program execution stops.

Fault tolerance policy can be reactive and proactive. While the aim of proactive techniques is to avoid situations caused by failures by predicting them and taking the necessary actions, reactive fault tolerance policies reduce the effect of failures on application execution when the failure effectively occurs. Most of the scientific workflow management systems are working reactively (e.g.: gUSE WMS), and steering workflows according to the available situation. There are several solutions in the literature for fault tolerant behavior and other complementary methods in its connected fields [1]. To achieve fault tolerant behavior the most widely adopted methods are:

- Checking and monitoring, which is a key factor in failure detection.
- Checkpointing and resubmission, where the system state is captured and saved based on predefined parameters (i.e.: time interval, number of instructions) and when the system undergoes some kind of failure the last consistent state is restored and computation is restarted from that point on [2].
- Replication, where critical system components are duplicated using additional hardware, or with scientific workflows critical tasks are replicated and executed on more than one processor. We can differentiate active and passive replication. Passive replication means that only one primary processor is invoked in the execution of a task and in the case of a failure the backup ones take over the task processing. In the active form all the replicas are executed at the same time and in the case of a failure the replica can continue the execution without intervention. The idea behind task replication is that replication size r can tolerate $r - 1$ faults while keeping the impact on the execution time minimal. We call r the replication size [12].

  While this technique is useful for time-critical tasks its downsides lays in the large resource consumption, so our attention is focused on mainly checkpointing methods in this work.

1.1. Motivation. Our main aim is to realize a more dynamic workflow management system to decrease overall processing/checkpointing time at the workflow level. In this paper we build up an Adaptive Provenance Based (APB) checkpointing model and propose two new checkpointing algorithms. Provenance in general carries information about the source, origin, and processes that are involved in producing data. The main target of collecting provenance data is support driven thus at the end the system effectively provides reusability and reproducibility for the user/scientist/developer communities. However, provenance can also support fault tolerant behavior by providing statistics about historical executions, such as failure rates or distribution and by storing the intermediary results generated by each tasks of the workflow [3].

To analyze the effects of the length of the checkpointing interval more precisely, we have based our approach on the fact, that the different paths constituting the workflow may have different time requirements and constraints, moreover longer checkpointing intervals may cause longer rework time in the case of a failure. For example if the workflow consists of only a single task or two or more sequentially ordered tasks (Fig. 1.1) then the aim can be the minimal execution time or to meet a soft or hard deadline. In this case the deadline determines whether we should use optimal checkpointing interval, which expectedly result in minimal execution time, or we have a little spare time where the fault tolerance parameters can be adjusted.

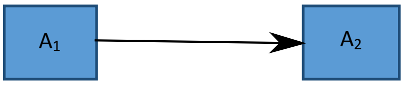


Fig. 1.1. *Workflow with two sequentially ordered tasks*

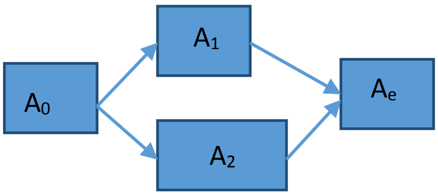


Fig. 1.2. *Workflow with two parallel executable, heterogeneous tasks*

Similarly if we have two or more parallel paths in the workflow model as in Fig. 1.2 we can take advantage of the different time requirements and constraints concerning the individual paths. Arising a failure during task A_1 or task A_2 the effect of the failure can be different, thus different fault tolerant parameters can be used.

Our main contribution is to dynamically adapt the checkpointing frequency based on the workflow structure in order to minimize the overall cost of checkpointing in time, network usage and storage capacity while still keeping to the soft or hard deadline.

1.2. Paper Organization. Our paper is organized as follows. After the introduction we give an overview about the existing theoretical checkpointing methods, and we also introduce some examples about the used solutions in the existing scientific workflow management systems. In Section 3 we introduce and detail our checkpointing method with algorithmic description. In Section 4 we provide use cases focusing on different (more-and-more complex) workflow structures and we also demonstrate the usefulness of our algorithms in an often used health care workflow. Finally, after the conclusions the bibliography closes our work.

2. Related Work.

2.1. Theoretical Model. Concerning dynamic workflow execution fault tolerance is a very important issue and checkpointing is the most widely used methods to achieve fault tolerant behavior. We investigated the different algorithms in order to give a brief overview of them.

According to the level where the checkpointing occurs we differentiate:

- application level checkpointing,
- library level checkpointing,
- system level checkpointing methods.

Application level checkpointing means that the application itself contains the checkpointing code. The main advantage of this solution lies in the fact, that it does not depend on auxiliary components however it requires a significant programming effort to be implemented while library level checkpointing is transparent for the programmer. Library level solution requires a special library linked to the application that can perform the checkpoint and restart procedure. System level solution can be implemented by a dedicated service layer that hides the implementation details from the application developers but still give the opportunity to specify and apply the desired level of fault tolerance [6].

From another perspective we can differentiate coordinated and uncoordinated methods. With coordinated checkpointing (synchronous) the processes will synchronize to take checkpoints in a manner to ensure that the resulting global state is consistent. This solution is considered to be domino-effect free. With uncoordinated checkpointing (independent) the checkpoints at each process are taken independently without any synchronization among the processes. Because of the absence of synchronization there is no guarantee that a set of local checkpoints result in having a consistent set of checkpoints. It may lead to the initial state due to domino-effect.

The frequency of the checkpointing interval also imposes many opportunities in checkpointing algorithms. Young in [7] has already in 1974 defined his formula for the optimum periodic checkpoint interval which is based on the checkpointing cost and the mean time between failures (MTBF) with the assumption that failure intervals follow an exponential distribution.

Di et al in [8] has also derived a formula to compute the optimal number of checkpoints for jobs executed in the cloud. His formula is generic in a sense that it does not use any assumption on the failure probability distribution.

The drawback of these solutions lies in the fact that the checkpointing cost can change during the execution if the memory footprint of the job changes, or depending on network reachability issues or when the failure distribution changes. Thus static intervals may not lead to the optimal solution. By dynamically assigning checkpoint frequency we can eliminate unnecessary checkpoints or where the danger of a failure is considered to be severe it can introduce extra state savings.

Meroufel and Belalem [9] proposed an adaptive time-based coordinated checkpointing technique without clock synchronization on cloud infrastructure. Between the different VMs jobs can communicate with each other through a message passing interface. One VM is selected as initiator and based on timing it estimates the possible time interval where orphan and transit messages can be created. There are several solutions to deal with orphan and transit messages, but most of them solve the problem by blocking the communication between the jobs during this time interval. However blocking the communication increases the response time and thus the total execution time of the workflow which can lead to SLA violation. In Meroufel's work they avoid blocking the communication by piggybacking the messages with some extra data so during the estimated time intervals it can be decided when to take checkpoint or logging the messages can resolve the transit messages problem.

The initiator selection is also investigated in Meroufel and Belalems another work [10] and they found that the impact of initiator choice is significant in term of performance. They also propose a simple and efficient strategy to select the best initiator.

Di et al also propose a new adaptive algorithm to optimize the impact of checkpointing regarding the checkpointing or restarting costs in [8].

Theresa et al in their work [11] propose two dynamic checkpoint strategies: Last Failure time based Checkpoint Adaptation (LFCA) and Mean Failure time based Checkpoint Adaptation (MFCA) which takes into account the stability of the system and the probability of failure concerning the individual resources.

This paper introduces a novel checkpointing method that is based on workflow model structure and failure statistics gathered about resources from historical executions. It extends related work on workflow structure analyses which focuses mainly on workflow similarity issues concerning the efficient storing and sharing of reproducible workflows [18], on scheduling mechanisms, and also on workflow execution time estimation problems [17]. Our work also promotes researches in fault tolerance by including the information obtained from the workflow structure into the actual state analysis and thus into the checkpointing interval determination.

2.2. Model implementations. Most of the existing workflow management systems are using some sort of mechanism to introduce fault tolerance in the system. As an example we are listing here two approaches:

2.2.1. gUSE. gUSE [16] (grid and cloud user support environment) is an open source science gateway (SG) framework, developed by Laboratory of Parallel and Distributed Systems (LPDS) at the MTA SZTAKI. It provides a generic purpose, workflow-oriented graphical user interface to create and run (DAG like) workflows on various Distributed Computing Infrastructures (DCIs) including clusters, grids, desktop grids and clouds.

- gUSE breakpoint support
  In gUSE breakpoints can be assigned to a job in two places: before submission of a job and after termination of a job. In the former case the user can check and modify the next running job instance of the workflow instance directly before starting it, while in the latter case the user has additional information about the job instance. A waiting time can also be set concerning the actual breakpoint.

- gUSE checkpointing support
  The suspend operation executes a job instance level check pointing, saving only those job items, which terminated properly (being in state finished) and sets back the system to be ready to continue its working, using the outputs of the properly terminated job instances.

2.2.2. Taverna. Taverna [15] is an open source and domain-independent Workflow Management System, developed by the myGrid team. It provides a desktop authoring environment (Taverna Workbench) and enactment engine for scientific workflows.

- Taverna breakpoint support
  Taverna has breakpoint support, including the editing of intermediate data. Breakpoints can be placed during the construction of the model at which execution will automatically pause or by manually pausing the entire workflow. However in Taverna the e-scientist cannot find a way to dynamically choose other services to be executed on the next workflow steps depending on the results.

2.2.3. Pegasus. The Pegasus system [14] was developed at the University of Southern California, with the aim to help workflow-based applications to be executed in a number of different environments including desktops, campus clusters, grids, and clouds.

- Pegasus checkpoint support
  Pegasus uses a detailed provenance database to keep track of what has been done including the locations of data used and produced, and which software was used with which parameters. When errors occur, Pegasus tries to recover when possible by retrying tasks, and when all else fails, generates a rescue workflow containing a description of only the work that remains to be done. A rescue DAG only skips jobs that have completely finished. It does not continue a partially running job unless the executable supports checkpointing. With this rescue DAG Pegasus implements a workflow level checkpointing similar to the suspend feature of the gUSE system.

3. Proposed Model - APB Chekpointing.

To achieve a more dynamic workflow management system which is capable to effectively resolve faults during program execution we apply a provenance based checkpointing method (Adaptive Provenance Based - APB Checkpointing) at workflow management system level. We are focusing on DAG (directed acyclic graph, which is a directed graph with no directed cycles) based workflow enactment such as used in gUSE. We have identified the following environmental conditions where the proposed APB model can work effectively.

3.1. Environmental Conditions.

- The system resources are monitored and failures can be detected as soon as possible, therefore the fault detection time (t_f) does not add high latency to the overall makespan of the workflow execution ($t_f = 0$ considered during our research).
- Task A_j cannot be started before it has received the output from all its predecessors and the results of a Task A_i can only be sent to its successor tasks after the task has been finished. Concerning a simple workflow as in Fig. 1.1 task A_e can only be started after the successful termination of both tasks A_1 and A_2.
- There is an ideal case so that tasks can be executed as soon as all the results from the predecessor tasks are ready and available. The system resources are inexhaustible in number, so the system can allocate the required number of resources to execute all the tasks parallel that are independent from each other.
- The system supports the collection of provenance data, therefore the intermediary results generated by the individual tasks are saved and in case of failure they can be easily retrieved. Thus there is no need to take checkpoints at the end of the tasks, and there is no need to take global checkpoints, since in the case of failure only the effected task should be rolled back.
- The system also support provenance data about failure statistics, so the probability of failures for a certain period of time is available for each resource component taking into account the aging factor as well.

3.2. General notation. Workflows in general and also scientific workflows can be represented as directed acyclic graphs (DAGs) $W = (N, E)$, where the nodes (N) represent the computational tasks or jobs and the directed edges (E) represent the dependency between them. The dependency can be data dependency, and control dependency. Scientific workflows are mainly data driven, so we are focusing on data flow oriented workflows in our paper. In this case the output of a Task A_i gives the input of a Task A_j if there exists an $A_iA_j \in E$ directed edge in the workflow. The list of symbols used in this paper can be seen in Table 3.1.

Table 3.1
Notation of used symbols

T_c	The checkpointing interval
T_{opt}	The optimal checkpointing interval
X_i	Optimal number of checkpoints during the execution of a task A_i
C	Checkpointing cost (considered constant)
$T(A_i)$	Execution time of task with T_{opt} and $E(Y)$
T_f	Mean time between failures (MTBF)
$E(Y)$	Expected number of failures during the execution of a task
T_l	Loading time, to restore the last saved checkpoint state
t_f	Fault detection time, the time to detect the failure
A_0	First or entry task of the workflow
A_e	Last or exit task of the workflow

For our APB model we propose two new checkpointing algorithms that with the help of monitoring the resources and executions dynamically adjust the checkpointing interval based on the task's dependency factor and the already occurred failures.

In the proposed algorithms there is no need to take global checkpoints of the workflow, and therefore there is no need of synchronization of any kind (based on time or based on communication channels between the processors). The parallel threads of the workflow may run on different type of computing infrastructures (for example on virtual machines of different cloud providers) therefore it would be a complex challenge to solve the synchronization between them.

3.3. Algorithms. The primary goal of our algorithms is to minimize the checkpointing overhead (time, resource) while still keep to the soft-deadline of the workflow and the performance level at a satisfactory level.

Young [6] and Di [7] have already proved that the optimal checkpointing interval, or the optimal number of checkpoints concerning the execution of a single task can be computed by Equation 3.1 and 3.2.

In their investigations Di et al. have declared that with equidistant checkpointing model, constant checkpointing cost (referred to as time overhead and denoted by C) and task restarting cost the total wallclock time of a task can be written as the sum of the calculation time, the checkpointing costs and the recovery time after failures. Their calculation was based on the assumption that the rework time after a failure of a task is about half of the checkpointing interval. By minimizing the expected time of execution they get their optimal value.

In both equations the fault detection time is considered $t_f = 0$. Equation 3.1 is a more general form, because it does not depend on any probability distribution, unlike the Young (3.2) formula which needs to assume that failure intervals follow an exponential distribution.

In both equations C is the checkpointing cost, T_{opt} is the checkpointing interval, T_f is the mean time between failures, $T(A_i)$ is the expected time of execution, $E(Y)$ is the expected value of failures during the execution of a task.

$$X = \sqrt{\left(T(A_i) \cdot \frac{E(Y)}{2C}\right)}, \tag{3.1}$$

$$T_{opt} = \sqrt{2CT_f}. \tag{3.2}$$

Our first algorithm is a static solution, while the other one is an adaptive one. The main difference between them lies in the fact, that the static algorithms does not change the checkpointing interval during the execution of the workflow it only adjust the length of the intervals before workflow submission. The second, adaptive solution may adjust the frequency of the checkpointing before each task execution. While the first algorithm gives adaptivity at workflow level, the other one proposes a real task level adaptive solution.

In our proposed algorithms we use (3.1) as a starting point to compute the checkpointing intervals. The main idea is that there is a dependency factor between the tasks. Namely if a failure occurs during the execution of a task A_i then it not only has a local effect on the task itself, but has a global effect also on the whole workflow concerning the execution time. Since if a failure occurs during the execution of task A_i then it has to be re-executed from the last checkpoint. It means the execution of the task ends later, so it may cause all of the successor tasks of task A_i to wait for the results. This can result the whole workflow execution to last longer. Similarly if we decrease the frequency of checkpointing, in other words we increase the length of the checkpointing interval T_c, then it has also a local effect, but the scope of its effect can extend for more tasks or even for all tasks as well.

3.3.1. Static Workflow Level (SWL) Algorithm. We define local cost (3.3) of increasing the checkpointing interval of task A_i, which is the execution time overhead of a task if the number of checkpoints is decreased by 1:

$$C_{local} = \frac{ET(A_i) + (\frac{T_c - T_{opt}}{2}) \cdot E(Y) - C}{ET(A_i)}. \tag{3.3}$$

We define the global cost (3.4) of increasing the checkpointing interval of a task A_i, which is the execution time overhead of the whole workflow, if the number of checkpoints of task A_i is decreased by 1:

$$C_{global} = \frac{(\frac{T_c - T_{opt}}{2}) \cdot E(Y) - C + rank(A_i) + brank(A_i)}{rank(A_0)}, \tag{3.4}$$

where the $rank()$ (3.5) function is a classic formula that are used in tasks scheduling [12] [13]. Basically the $rank()$ function is the critical path from task A_i to the last task, and can be computed recursively backward from the last task. For simplicity we have introduced the $brank()$ (3.6) function, which is the backward $rank()$ value from task A_i backward to the entry task A_0. It is the longest distance from the entry task to task A_i excluding the computation cost of the task itself. It can also be calculated recursively downward from task A_0.

$$rank(A_i) = T(A_i) + max_{A_j \in succ(A_i)} \cdot rank(A_j), \tag{3.5}$$

$$brank(A_i) = max_{A_j \in pred(A_i)} \cdot (brank(A_j) + T(A_j)). \tag{3.6}$$

Before submitting the workflow at first the optimal checkpointing interval should be calculated for each task based on the failure statistics of the resource (expected value of the failures that can arise during execution) and the estimated (or retrieved from provenance database) execution time of the task. Than the adjusted checkpointing intervals for all tasks can be calculated. Concerning those tasks that are part of the critical path or one of the critical paths of the workflow the checkpointing interval should remain the optimal value. However along all other paths between A_0 and A_e in the abstract model of the workflow the checkpointing interval should be adjusted. If inequality 3.7 stands for a task A_i it means, that for that task the checkpointing interval can be increased without effecting the total wallclock time of the workflow. The cumulative costs along a path should be taken into account.

The pseudocode of the WLS static algorithm can be seen in Table 3.2 and WLS works as follows: The algorithms analyzes the tasks in a so called topological ordering. In a topological ordering task A_i precedes A_j if during execution task A_i should also precede task A_j because task A_j or any predecessors of task A_j uses its results as an input parameter. WLS is a fair algorithm, it tries to share the spare time equivalently. Given a path s_j between A_0 and A_e the algorithm makes attempts to adjust the checkpointing interval of the tasks belonging to this path in execution ordering

Table 3.2

Pseudo code of the WLS algorithm

```
FOR $i : 1$ TO $N$ DO
  $T_{adj}(A_i) = T(A_i)$
  $brank_{adj}(A_i) = max_{A_j \in pred(A_i)} \cdot (brank_{adj}(A_j) + T_{adj}(A_j))$
While $\exists i$:   $X_i > 1$ AND $brank_{adj}(A_i) + rank(A_i) < rank(A_0)$ DO
  FOR $i : 1$ TO $N$ DO
    IF $brank_{adj}(A_i) + rank(A_i) < rank(A_0)$ THEN DO
      $T_{adj}(A_i) = C_{local}$
      $X_i = X_i - 1$
    ELSE $T_{adj}(A_i) = T(A_i)$
```

$$brank(A_i) + rank(A_i) + \frac{T_c - T_{opt}}{2} \cdot E(Y) - C < rank(A_0). \tag{3.7}$$

The first three lines of the algorithm in (Table 3.1) initializes the modified execution time $T_{adj}(A_i)$ and cumulative execution time $brank_{adj}(A_i)$ values concerning the individual paths based on the optimal checkpointing interval. Next, in topological ordering the checkpointing intervals are adjusted. When the checkpointing intervals for all tasks have been adjusted once, the algorithms repeats it until the workflow structure enables it.

This algorithm is based on the assumption that when faults occur during a checkpointing interval (between two consecutive checkpoints) the expected average time loss is half of the checkpointing interval and also based on the a-priori calculated (or from provenance retrieved) expected values of failures and execution times of the individual tasks. This algorithm is executed once before submitting the workflow and after that the checkpointing intervals are not modified. It gives a workflow level, static solution for optimizing checkpointing costs.

3.3.2. Adaptive task level (ATL) algorithm. If the expected number of failures are already met during the execution of the actual task or this was the case during one of the predecessors of the actual task, then it may be possible, that default checkpointing intervals should not have been increased, because the cumulative overhead of the occurred faults can negatively affect the whole workflow execution time. Or the shoe is on the other foot, the predecessors have been executed without or less failures and thus a lot of free time remained to (expectedly) meet the soft or hard deadline and thus to minimize the overhead caused by frequent checkpointing. To take earlier faults and real execution conditions (changing failure distributions or execution times) into account we need information about the realistic execution time of the tasks. With provenance support the real execution time and actual failure statistics can be obtained and the algorithm can adjust the checkpointing interval dynamically. This adaptive algorithm is based on the assumption that failures are detected as soon as possible, and system reacts immediately.

Before executing the individual tasks the checkpointing intervals are adapted to the new situation.

The ATL adaptive algorithm works as follows. Before submitting a task A_i the estimated $brank(A_i)$ values are updated with real execution time of the predecessor tasks and based on the actual values of the critical path or paths the checkpointing intervals for the remaining tasks of the workflow is recalculated.

4. Use Cases. As it was already shown in the previous section the effect of the length of the checkpointing interval concerning the individual tasks can be different on the whole workflow based on the assumption that when a failure occurs during the execution of a task the average rework time is half of the checkpointing interval.

4.1. Basic workflow structures. Concerning a single task workflow or a workflow, constituting of sequentially ordered tasks, when we increase the length of the checkpointing interval it surely has a global effect on the makespan of the workflow. However, if soft or hard deadline T_D is given, and $T(A_i) < T_D$ then we have the opportunity to reduce the number of checkpoints, so that execution time still remains less or equal then the predefined deadline. Thus in this simple case both the static (SWL) and adaptive (ATL) algorithms can be used.

Concerning a workflow with two parallel executable, heterogeneous tasks (1.2) we can differentiate two cases:

- If $T(A_2) >> T(A_1)$ than A_2 should be executed with optimal checkpointing and A_1 can be executed without any checkpoints. If A_1 fails for the first time it has still enough time to be executed once more with minimal checkpoints, until the remaining time is only enough for an optimal execution.
- If $T(A_2) > T(A_1)$ than A_2 should be executed with optimal checkpointing and A_1 should be executed with less frequent checkpoints (static (SWL) algorithm) or with adaptive checkpointing (ATL) algorithm).

Concerning a workflow with two parallel executable, homogeneous tasks as a starting point we do not have spare time because the other task may generate its result on time. The only way we may minimize the checkpointing overhead is to use the adaptive (ATL) algorithm. The two tasks are started with optimal checkpointing and when a failure occurs during the execution of one of the two tasks then the other task can use the adaptive algorithm with the increased makespan of the first task as a deadline.

Sequentially and parallel ordered tasks can form arbitrary constructed workflow. As our algorithms calculates the $brank()$ value along the paths of the workflow starting from the entry task A_0 to the exit task A_e, it can be used for arbitrary constructed DAGs.

4.2. Dynamic Health Care Workflow. The workflow depicted in Fig. 4.1 is often used in health care smart systems.

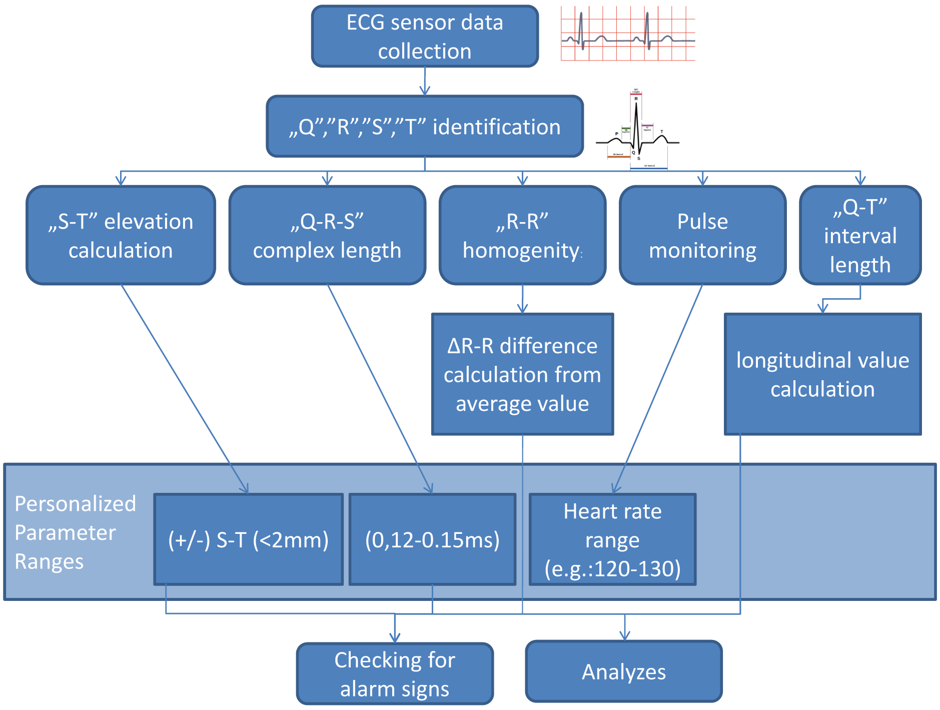

ECG sensor data collection
„Q","R","S","T" identification
„S-T" elevation calculation
„Q-R-S" complex length
„R-R" homogenity
Pulse monitoring
„Q-T" interval length
ΔR-R difference calculation from average value
longitudinal value calculation
Personalized Parameter Ranges
(+/-) S-T (<2mm)
(0,12-0.15ms)
Heart rate range (e.g.:120-130)
Checking for alarm signs
Analyzes

Fig. 4.1. *Cardiac Workflow*

4.2.1. Workflow description. Workflow inputs: ECG sensor data and general+personalized parameter ranges: (normal/green zone/, small parameter value deviation/yellow zone/, large parameter value deviation /red zone/).

Workflow outputs: inform external system (alarm signal/no alarm signal), analyzed results.

The cardiac workflow receives the acquired ECG sensor data and automatically identify special landmark points on the ECG curves. As a next step it calculates special parameters like the QRS complex length, or

QT interval. These small calculations have to be done mostly at each ECG signal cycle. After the calculations the parameter values are evaluated against the pre-defined parameter ranges. These (general and personalized) parameter ranges specify what is a ,,normal" parameter value (defined as green zone), what is slightly (defined as yellow zone) or significantly abnormal value (defined as red zone).

The result of the parameter value evaluation is the basic input of an external decision support application which can realize actions (such as alarming) according to the received information.

4.2.2. Usability of our checkpointing algorithm. The workflow depicted in Fig. 4.1 can be used in numerous scenarios. For example to monitor heart activity for a smaller period of time (for monitoring patients after stroke during physical training or load, or monitoring professional football players during a match), but also for monitoring longer periods of time. One of its long-term usage is the Holter ECG. Holter ECG monitor records electrical signals from the heart via a series of electrodes attached to the chest during 24 h period of time or even during several months as well. The number and position of electrodes varies by model, but most Holter monitors employ between three and eight. These electrodes are connected to a small piece of equipment that is attached to the patient's belt or hung around the neck, and is responsible for keeping a log of the heart's electrical activity throughout the recording period. There are also available intelligent systems that already make some analyzes on the recorded data, but the widespread used model are only capable to log data and after the measurement period to send these data to a computing resource where these data are analyzed. In general a 24 h interval is logged on Holter appliances. At the time the cardiologist retrieves the ECG sensor data collection from the Holter, he or she needs the analyzed results very fast, so the workflow should be enacted as fast as it can, and in general it is calculated not for a single patient, but for a group of patients or for even a big population used for cardiological researches.

The workflow model consists of several distinct paths that can be enacted separately and parallel. These distinct paths need different calculation times so our algorithm can be efficiently used here without increasing the total makespan of the workflow execution.

5. Conclusion. Smart systems in telemedicine frequently use intelligent sensor devices at large scale. Their data processing tasks are often realized in workflows. In HPC environment high number of failures can arise during workflow enactment, so the use of fault tolerance techniques is unavoidable. In this paper we investigated the different checkpointing techniques, which are the most widely used proactive fault tolerant methods. We gave a brief overview of the different checkpointing perspectives with special attention on those solutions where the checkpointing intervals are periodic or it changes adaptively during execution. We proposed a workflow level (SWL) and a task level (ATL) provenance based dynamic algorithm where the checkpointing frequency is adjusted in order to eliminate blind checkpoints while still maintaining soft deadlines.

We have compared the two algorithms and the ATL algorithm outperforms the SWL one in adaptivity support, since the static algorithm cannot adjust the checkpointing intervals to the actual environmental conditions (failures, execution times). SWL cannot guarantee soft or hard deadline if the number of failures exceeds the expected value. However, the ATL algorithm is more compute intensive (it must be executed before each task submission) and in most of the cases all the required provenance data are not surely available. Moreover, retrieving data from a provenance database may have high overheads as well.

In our future work we will work out a more sophisticated algorithm which based on available statistics of failures can also adaptively determine the length of the forthcoming checkpointing interval that predictively can avoid faults and still keeps the global cost of the failure limited by the predefined soft or hard deadline.

Acknowledgments. This work was supported by EU project SCI-BUS (SCIentific gateway Based User Support). The SCI-BUS project aims to create customized workflow based gateways connected to Distributed Computing Infrastructures (DCIs) fostering HPC and HTC services without the need to deal with the underlying infrastructures' details.

REFERENCES

[1] A. Bala, and I. Chana, *Fault tolerance-challenges, techniques and implementation in cloud computing*, IJCSI International Journal of Computer Science Issues vol. 9, 2012.

[2] R. Garg and A. K. Singh, *Fault Tolerance in grid computing: State of the art*, International Journal of Computer Science & Engineering Survey vol. 1, pp. 88-97, 2011.

[3] E. Kail, A. Bánáti, P. Kacsuk, and M. Kozlovszky, *Provenance based adaptive and dynamic workflows.*, 15th IEEE International Symposium on Computational Intelligence and Informatics, pp. 215-219, IEEE Press, Budapest, 2014,

[4] E. Kail, P., Kacsuk, and M. Kozlovszky, *A Novel Approach to User-steering in Scientific Workflows*, Proceedings of CGW14, 2014

[5] E. Kail, A. Bánáti, K. Karóczkai, P. Kacsuk, and M. Kozlovszky, *Dynamic workflow support in gUSE*, MIPRO, 2014 Proceedings of the 37th International Convention.

[6] R. Jhawar, V. Piuri, and M. Santambrogio, LU-*Fault Tolerance Management in Cloud Computing: A System-Level Perspective*, IEEE Systems Journal 7, vol 2, 2013.

[7] J.W. Young, *A first order approximation to the optimum checkpoint interval*, Communications ACM, 1974.

[8] S. Di, Y. Robert, F. Vivien, D. Kondo, Cho-Li Wang, and F. Cappello, *Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism*, 112. ACM Press, 2013.

[9] B. Meroufel, and G. Belalem, *Adaptive time-based coordinated checkpointing for cloud computing workflows*, Scalable Computing: Practice and Experience, Vol 15, No 2, 2014.

[10] B. Meroufel, and B. Ghalem, *Policy Driven Initiator in Coordination Checkpointing Strategies*, http://www.wseas.us/e-library/conferences/2014/Istanbul/TELEDU/TELEDU-20.pdf.

[11] A. Lidya, S. Therasa, G. Sumathi, and S. A. Dalya, *Dynamic Adaptation of Checkpoints and Rescheduling in Grid Computing*, International Journal of Computer Applications vol. 3, 2010.

[12] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, *Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems*, 12th IEEE International Conference on High Performance Computing and Communications (HPCC), 2010 pp.434,441, 1-3 Sept. 2010.

[13] H. Topcuoglu, S.Hariri, and M. Wu, *Performance-effective and low-complexity task scheduling for heterogeneous computing*, in IEEE Transactions on Parallel and Distributed Systems, vol.13, no.3, pp.260,274, Mar 2002

[14] https://pegasus.isi.edu [acc. 01.02.2016].

[15] http://www.taverna.org.uk [acc. 01.02.2016].

[16] http://guse.hu [acc. 01.02.2016].

[17] I. Pietri, G. Juve, E. Deelman, R. Sakellariou, *Performance Model to Estimate Execution Time of Scientific Workflows on the Cloud* Proceedings of WORKS14

[18] J. Starlinger, S. Cohen-Boulakia, S. Khanna, S.B. Davidson et al., *Layer Decomposition: An Effective Structure-based Approach for Scientific Workflow Similarity*, 2014 IEEE 10th International Conference on e-Science, vol.1, pp.169,176, 20-24 Oct. 2014

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016

AIMS AND SCOPE

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

Expressiveness:

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

System engineering:

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

Performance:

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

Applications:

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

Future:

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

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

INSTRUCTIONS FOR CONTRIBUTORS

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

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

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

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

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