Future Generation Computer Systems 29 (2013) 451–459

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Graph partitioning algorithms for optimizing software deployment in mobile cloud computing

Tim Verbelen *, Tim Stevens, Filip De Turck, Bart Dhoedt

Ghent University – IBBT, Department of Information Technology, Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium

ARTICLE INFO

Article history:
Received 8 September 2010
Received in revised form
19 June 2012
Accepted 14 July 2012
Available online 20 July 2012

Keywords:
Distributed systems
Graph algorithms
Deployment optimization
Cloud computing
Mobile computing

ABSTRACT

As cloud computing is gaining popularity, an important question is how to optimally deploy software applications on the offered infrastructure in the cloud. Especially in the context of mobile computing where software components could be offloaded from the mobile device to the cloud, it is important to optimize the deployment, by minimizing the network usage. Therefore we have designed and evaluated graph partitioning algorithms that allocate software components to machines in the cloud while minimizing the required bandwidth. Contrary to the traditional graph partitioning problem our algorithms are not restricted to balanced partitions and take into account infrastructure heterogenity. To benchmark our algorithms we evaluated their performance and found they produce 10%–40% smaller graph cut sizes than METIS 4.0 for typical mobile computing scenarios.

1. Introduction

Nowadays, the emergence of cloud computing is leading to a new paradigm of utility computing [1], where computing power is offered on an on-demand basis. Users are able to access applications, storage and processing over the Internet, via services offered by cloud providers on a pay-as-you-use scheme. The advantages for the end users are reduced cost, higher scalability and improved performance in comparison to maintaining their own private computer systems, dimensioned for peak load conditions. Moreover the elasticity of the cloud reduces the risks of overprovisioning (underutilization) or underprovisioning (saturation) [2].

The usage of the cloud is not only beneficial for web-based applications, but can also be used for other applications composed of many service components following the service-oriented programming paradigm. Some of these service components may have high needs regarding CPU power or memory consumption, and should therefore be executed on dedicated server machines in the cloud rather than on a regular desktop PC or mobile terminal, for example recognition components in an object recognition or speech to text application.

The adoption of the cloud paradigm poses the problem where to deploy software components, given the many options in terms of available hardware nodes in even moderate scale data centers. This deployment optimization is important to both the cloud user and the cloud provider in order to reduce costs. All components need to be deployed on a machine with sufficient CPU power while the communication overhead between different machines is preferably minimized as this introduces extra latency and network load. This problem can be modelled as a graph partitioning problem where a weighted graph of software components has to be partitioned in a number of parts representing the available machines. Moreover the optimal deployment can change over time, and thus in order to realise an optimal deployment a fast algorithm is desired.

An important scenario in this respect is cloud overspilling. In this scenario, a company offloads work from its own private infrastructure to a public cloud infrastructure on peak moments, as shown in Fig. 1. This enables the company to dimension its infrastructure for the average workload instead of the peak workload, reducing the underutilization and the cost of the private infrastructure. This situation is typical in digital document processing where one faces strict month-end or year-end deadlines, and thousands of batches of documents are to be processed. Typically, document processing systems support workflows consisting of a few tens of components (including content ingest, reformatting, layouting, merging, versioning, logging, output generation and printing components). As many of these components come in different versions, and potentially need to be instantiated for each customer separately, the number of components in such a scenario quickly amounts to a few hundreds.

A second use case is situated in the area of integrated simulation tools for engineering purposes [3]. These integrated tools typically involve multi-physics simulation (e.g. structural analysis, acoustic

[*] Corresponding author.
E-mail address: tim.verbelen@intec.ugent.be (T. Verbelen).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.07.003

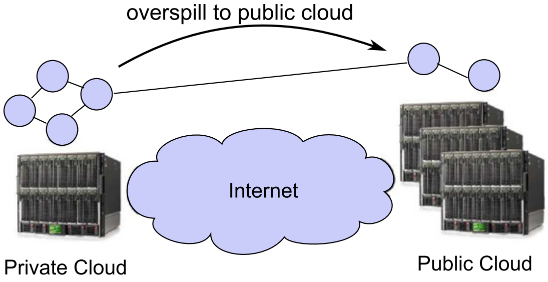


Fig. 1. Work is offloaded from private infrastructure to a public cloud on peak moments, reducing underutilization (and the cost) of the private infrastructure.

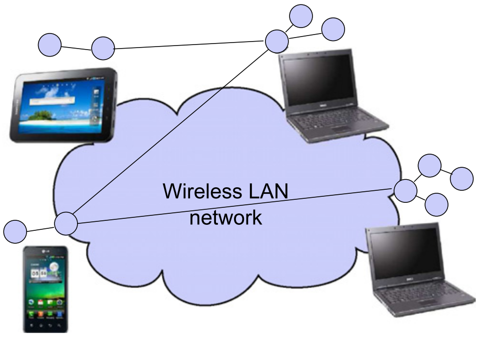


Fig. 2. A cloudlet consisting of four devices connected by a wireless network sharing their resources. Application components are distributed among all devices in the cloudlet, in order to enhance the user experience of all users.

simulation and engine dynamics simulation in the case of designing a new automobile engine), and the number of simulation tools involved can easily amount to 10–20 for small engineering projects to a few 100 individually deployable components for a realistic engineering project. In such engineering endeavours, often parameter sweeps are executed to optimize the design (or to assess the sensitivity of the resulting design performance w.r.t. these parameters), necessitating multiple instances of these simulation components running concurrently, in order to arrive at realistic design times. Again, we end up with a component graph containing a few 100–1000 components.

This overspilling problem also arises in the context of mobile computing, where the cloud can be used to enhance the capabilities of a mobile device. Due to the restricted CPU power of mobile devices, the idea is to offload parts of the application at runtime to a cloud infrastructure [4]. The question then is which parts to offload and to deploy on which machines in the cloud – and how many – in order to spread the load while keeping the needed bandwidth low. In this case, the complexity of the partitioning problem depends on the granularity of the offloading, as offloading can be done on component [4], Java class [5] or method [6] level, resulting in graphs of tens, hundreds or thousands of components.

A special case of deployment optimization occurs when multiple mobile devices connected via a wireless network share their resources in order to enhance the user experience of all users, in a so called "cloudlet" [7], as shown in Fig. 2. This is the case when no internet uplink is available for offloading to the cloud, or when cloud offloading is not beneficial due to a high WAN latency. Because the bandwidth is a scarce resource and shared between all devices in the wireless LAN, a global optimization is needed taking into account all application components of all devices.

As an use case, we mention a mobile augmented reality application. When casting such an application into a component framework, the number of independently deployable components amounts to 5–10 [7], with different components for tracking camera movements, building a 3D map of the environment, recognizing objects, detecting collisions between objects, rendering a 3D overlay, etc. Other applications, such as 3D games, are reported to consist of 10–20 components [8]. Assuming a few tens of users connected to the same cloudlet and hence sharing computing and network resources, the number of components easily exceeds 100. As these users are connected through heterogeneous devices (a mix of low-end and high-end devices), an optimal deployment guaranteeing a minimal quality of experience for all users should be aimed for.

In this paper we present algorithms to partition a software application, composed of a number of components, on a number of interconnected machines in the cloud with different capacities while minimizing the communication cost between the components. In Section 2 related work regarding graph partitioning and task allocation on the grid is discussed. Section 3 more formally describes our problem and in Section 4 algorithms are proposed that solve the problem. In Section 5 the different algorithms are evaluated and compared regarding solution quality and execution time, and the influence of different parameters is discussed. In view of the use cases mentioned in this introduction, we focus on graphs containing 100–1000 components for this evaluation. We also compare our solutions to METIS 4.0 for partitioning graphs in k balanced partitions and show the applicability of our algorithms in the mobile offloading scenario. Finally Section 6 concludes this paper.

2. Related work

2.1. Graph partitioning

Graph partitioning is a fundamental problem in many domains of computer science, such as VLSI design [9], parallel processing [10] and load balancing [11]. The graph partitioning problem tackles the problem of dividing a graph in k equal sets while minimizing the edges between the sets. When $k = 2$ this is also referred to as the min-cut bipartitioning problem. Finding a good solution for this problem is known to be NP-Hard [12]. In the following we give a brief overview of the state-of-the-art and recent advances in graph partitioning. For a more detailed survey of graph partitioning techniques we refer to [13,14].

A first class of algorithms are the so called move-based approaches, which try to iteratively improve the partition by vertex moves or swaps between the parts, such as the Kernighan–Lin (KL) algorithm [15]. By choosing moves that introduce a cost reduction of the graph cut this algorithm converges to a local optimum. Fiduccia and Mattheyses introduced a number of optimizations to the KL algorithm which led to a linear time algorithm for graph partitioning [16]. These move-based algorithms can be combined with stochastic methods such as simulated annealing [17], particle swarm optimization [18] or ant colony optimization [19] in order to escape from local optima. The biggest disadvantage of iterative improvement methods is that their performance deteriorates as the graphs get larger.

In order to partition large graphs the multilevel approach has become widely adopted [20,21,17–19,22]. The main idea is to iteratively coarsen the initial graph by merging vertices according to a matching until a small graph with a similar structure remains. This graph can then be partitioned with a spectral method [23,20] or a greedy graph growing algorithm [24]. Next the graph is again iteratively uncoarsened and a local improvement heuristic such as the KL algorithm is applied at each level. The multilevel scheme is also used in state-of-the-art graph partitioning libraries such as METIS [24], SCOTCH [25] and JOSTLE [26].

Recent work in graph partitioning explores methods based on diffusion [11] or maximum flow [27]. Also the combination of known techniques can result in new heuristics. Chardaire et al. use a PROBE (Population Reinforced Optimization Based Exploration) heuristic, combining greedy algorithms, genetic algorithms and KL refinement [28]. Loureiro and Amaral propose a greedy graph growing heuristic combined with a local refinement algorithm [29]. Martin uses a genetic algorithm improved with spectral methods and KL refinement [30].

All these methods partition the graph in a predefined number of parts of equal sizes. In the context of cloud computing, not all machines have equal capacity and also the number of machines that has to be used is not predefined, thus these algorithms cannot be readily used. Therefore, our algorithms take a number of possible machines with different capacities as input with the only constraints that no machine can exceed its maximum capacity and every component has to be deployed on exactly one machine. However, we can still use the ideas from the original graph partitioning problem to calculate a good deployment.

2.2. *Task allocation on the grid*

In the context of grid computing the graph partitioning problem is used for the allocation of parallel tasks on heterogenous infrastructure. Many proposed algorithms such as MiniMax [31], VHEM [32], QM [33] PaGrid [34], and MinEX [35] use the multi level paradigm in combination with a refinement algorithm, while others such as PART [36] use simulated annealing.

These algorithms all focus on parallel applications based on mesh models, such as fluid dynamics, where a big task is split up in smaller parallel tasks that are each executed on one of the processors in the grid. The goal is then to execute all these tasks as fast as possible, thus minimizing the execution time of the slowest node. In our problem the execution time metric makes less sense, since we focus on applications on the cloud, where components generate load as long as the end user runs the application. Also the structure of the task graph for the grid is based on the input mesh, while in our case the graph of cloud applications is based on the software components and their interactions, which is structured differently and of a more modest size.

3. Problem statement

Let $G = (V, E)$ be an undirected graph where $V = \{v_1, v_2, \dots, v_N\}$ is a set of N vertices and E is a set of edges connecting the vertices. The vertices represent units of deployment in a distributed software system and the edges represent communication overhead between those units. Each vertex v_i is assigned a cost w_i that indicates the amount of resources (i.e. CPU power) this component needs. $C = (c_{ij})$ is the adjacency matrix of G, i.e. if there exists an edge between v_i and v_j then c_{ij} equals the weight of this edge, otherwise $c_{ij} = 0$. The edge weights represent the cost of communication (i.e. bandwidth) between different software components.

The infrastructure is also modelled as an undirected graph $S = (K, L)$, where K is a set of available machines and L the links between the different machines. Each machine has a maximum capacity of M_m and each link is assigned a cost to use this link. From this graph a matrix $B = (b_{mn})$ can be deduced where element b_{mn} represents the cost to exchange data between machine m and machine n.

The goal now is to assign each vertex v_i to one of the machines so that the total data exchanged between the machines weighted by B is minimized. More formally, let the decision variables X_{im} represent the graph cut. The value of X_{im} is equal to 1 if component i is deployed on machine m, and 0 otherwise. We want to minimize the sum of the edge weights of the edges between nodes deployed on different machines, thus we introduce variable h_{ij}, which is equal to 1 if components i and j are deployed on different machines, 0 otherwise. Then the objective function to minimize becomes the weight of the graph cut (GC):

$$\mathrm{GC} = \sum_{i,j} h_{ij} \times c_{ij} \times b_{P(i)P(j)}. \tag{1}$$

With the function $P(j)$ returning the machine on which vertex j is deployed. Variables h_{ij} can be expressed in terms of X_{im} as follows:

$$h_{ij} = 1 - \sum_{m} X_{im} \times X_{jm}. \tag{2}$$

Two more constraints are needed to fully describe our problem.

$$\forall k : \sum_{i} w_i \times X_{im} \leq M_m \tag{3}$$

$$\forall i : \sum_{m} X_{im} = 1. \tag{4}$$

Eq. (3) states that the total amount of resources used by nodes on a machine cannot exceed the maximum capacity of that machine. Eq. (4) makes sure that every node is deployed on exactly one machine.

4. Algorithms description

4.1. *Integer linear programming (ILP)*

The problem defined in Section 3 can be seen as an Integer Linear Programming (ILP) problem, thus an ILP solver (IBM ILOG CPLEX [37]) can be used to determine the optimal solution for this problem. The amount of time and resources needed to solve this problem grows exponentially with the size of the graph, so heuristics are needed to find a good solution faster. We can still use the ILP solution for smaller graphs to benchmark our algorithms that exhibit better scaling behaviour, possibly at the expense of optimality.

4.2. *Multilevel graph partitioning (MLKL)*

In a first heuristic we use a multilevel refinement strategy as first proposed by Hendrickson and Leland [20]. The idea is to coarsen down the graph by merging connected vertices until a small graph is obtained. Then this graph is partitioned and uncoarsened again, while optimizing the partition in each uncoarsening step. Thus, the partitioning consists of three phases:

Coarsening. The graph G is coarsened in a sequence of smaller graphs $G_1, G_2, \dots, G_m$ such that $|V_1| > |V_2| > \cdots > |V_m|$.

Initial partitioning. A partition P is computed on the coarsest level of the graph.

Uncoarsening and refinement. Partition P of graph G_m is uncoarsened back through all the intermediate graphs. In each uncoarsening step a refinement algorithm is executed in order to find a better partition.

4.2.1. *Coarsening*

During coarsening, from graph G_i a graph with fewer vertices G_{i+1} is created by collapsing edges and combining the vertices connected by those edges. When an edge is collapsed the two vertices connected by the edge are reduced into one whose weight is the sum of the weights of both vertices. When both vertices have an edge to a third node, these two edges are collapsed in one edge with a summed edge weight. Every iteration of coarsening, a matching – a set of edges without common vertices – is created and the matched vertices are combined. In order to find a small edge cut, it is beneficial to collapse heavy weighted edges, since these are unlikely to be in the best cut.

Our coarsening algorithm uses the *heavy-edge matching* (HEM) algorithm which is a widely used coarsening scheme proposed by Karypis and Kumar [24]. The vertices of the graph are visited in a random order and each vertex u is matched with an unmatched neighbour v such that the weight of the edge (u, v) is the maximum over all valid incident edges of u. In order to be able to map the vertices of the coarsest graph on the different machines we added one more constraint to only match two vertices when the sum of their vertex weights is smaller than the size of the smallest part.

4.2.2. Initial partitioning

After coarsening the problem consists of choosing a feasible deployment of the software components on the infrastructure at hand. We assume that there are enough machines and resources available to find such a deployment. This problem reduces to a simple bin packing problem that can be solved with a first-fit algorithm.

The vertices are ordered by descending vertex weight and the machines are ordered by descending maximum capacity. For each vertex the list of machines is iterated and it is assigned to the first one that has enough capacity left, then the capacity of that machine is diminished with the vertex weight of that vertex.

Algorithm 1 Calculate initial partition

```
$vertices : OrderedList$
$machines : OrderedList$
**for all** $vertex\ v_i \in vertices$ **do**
  **for all** $machine\ m \in machines$ **do**
    **if** $w_i \leq M_m$ **then**
      $P(i) \leftarrow m$
      $M_m \leftarrow M_m - w_i$
      break
    **end if**
  **end for**
**end for**
```

4.2.3. Uncoarsening and refinement

In the final step the graph is gradually uncoarsened again and a KL-like algorithm is applied to improve the initial partition found in the previous step. The proposed algorithm is based upon the refinement algorithm of Hendrickson and Leland [20]. The fundamental idea of the algorithm is the concept of the gain associated with moving a vertex to a different part. The gain reflects the net change in cut size that would result from switching a vertex from one machine to another. More formally, the gain introduced by moving vertex i from set m to set n, the corresponding gain g_{mni} can be expressed as

$$\begin{aligned} g_{mni} = & \sum_{j \in V \& P(j)=n} c_{ij} \times b_{mn} - \sum_{j \in V \& P(j)=m} c_{ij} \times b_{mn} \\ & + \sum_{j \in V \& P(j)=p, p \neq m, p \neq n} c_{ij} \times (b_{mp} - b_{np}) \end{aligned} \tag{5}$$

where again the function $P(j)$ returns the current part of vertex j. The base refinement algorithm is presented in pseudocode as Algorithm 2, and is composed of two loops. Each pass of the outer loop will try moving vertices around to find a better cut. This outer loop ends after a pass that did not improve the best partition so far, implying that the algorithm has reached a local optimum. To avoid the algorithm getting stuck in an infinite loop a vertex can only be moved once during a single pass of the outer loop. The inner loop will iteratively select a vertex to move, i.e. the vertex having the largest gain, subject to some additional rules. Notice that this also could be a move with a negative gain. This gives the algorithm the possibility to escape to some extent from getting stuck in a local optimum. When a vertex is moved the gains of all its neighbours should be updated. This process completes when no more suitable move is found.

Algorithm 2 KL Refinement

```
$Bestpartition \leftarrow Currentpartition$
Compute initial gains
Compute $\forall p : Free(p)$ the amount of free space on part $p$
**repeat**
  $MovedList \leftarrow EmptyList$
  **repeat**
    **repeat**
      Select vertex move, not in $MovedList$ with highest gain $g_{mni}$
      **if** $\exists p : Free(p) < 0$ **then**
        **if** $Free(m) < 0$ and $Free(n) - w_i > Free(m)$ **then**
          $MoveFound \leftarrow True$
        **end if**
      **else**
        $MoveFound \leftarrow True$
      **end if**
    **until** MoveFound
    **if** MoveFound **then**
      Perform move and add move to $MovedList$
      Update gains of neighbors of moved vertex
      **if** $Currentpartition < Bestpartition$ **then**
        $Bestpartition \leftarrow Currentpartition$
        **if** No possible moves with positive gain **then**
          Break
        **end if**
      **end if**
    **end if**
  **until** No more moves possible
**until** No better partition found
**return** $BestPartition$
```

When a better partition is found, the inner loop will only accept moves with positive gain and then start another pass. The rationale for this is that when the algorithm finds a better partition, it will try to descend as deep as possible to get the local optimum. Then, a new iteration is started, and the moves that were in *MovedList* in the previous iteration can again be selected to come closer to the optimum.

Instead of creating balanced partitions, we need to make sure that no machine gets overloaded. A naive rule in this respect would be to forbid vertex moves that would violate the capacity condition of the target. However, this rule could cause to a move to local optima, from which there is no escape (see Fig. 3).

Therefore we do allow a move to a part that exceeds the maximum part weight, provided that there is no other part with excessive weight. If there is a part s with too much weight, then we choose the vertex with the highest gain that moves away from m and reduces the amount of overload. This way situations as in Fig. 3 will also converge to an optimum ($m0 = \{n0, n2\}, m1 = \{n1, n3, n4\}$).

When two possible moves have equal gain, we prefer moves that do not exceed the target parts boundary. If that does not differentiate the possible moves, we prefer those to a part that is not empty. When still no differentiation is made between two moves, ties are broken randomly.

Instead of continuing until all vertices are moved, one can stop the algorithm earlier to save time, for example when the graph cut of the current partition deviates by more than a certain cutoff threshold of the best solution found at that moment. Proper choices for this threshold can reduce the execution time while not

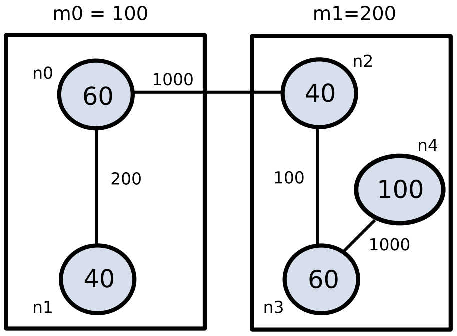


Fig. 3. This partition cannot be optimized when no moves exceeding the target parts maximum boundary are allowed.

sacrificing much solution quality. If a threshold value of 0 is chosen, the algorithm will not accept any moves with negative gain, and the algorithm will behave like a steepest descent algorithm ending in the nearest local minimum.

4.3. *Simulated annealing (SA)*

A second approach followed for solving the k-partitioning problem is based on simulated annealing (SA), a combinatorial optimization technique introduced by Kirkpatrick et al. [38] and independently by Cerny [39], inspired by the cooling process of metal. To use the SA technique in the context of the k-partitioning problem, we use it as a refinement technique after bin packing, inspired by Johnson et al., who showed the effectiveness of SA for the graph bipartitioning problem [40].

The SA algorithm moves from one solution to a neighbour solution by moving a vertex from one part to another. A move will be accepted with probability $\exp(g/T)$, in which g is the gain of a move, as introduced in Section 4.2, and T a temperature parameter that is lowered gradually over time. The algorithm in pseudocode is shown as Algorithm 3. In order to find the optimum in situations such as described in Fig. 3, vertex i is moved with positive gain to a part that has not enough capacity left with probability $\exp(\frac{Free(p)-w_i}{T})$, where $Free(p)$ is the amount of free space on part p. By making this also dependent on the temperature this assures that towards the end of the algorithm it will converge to a valid solution without causing overoccupied parts.

The performance of SA is very dependent on the choice of the different annealing parameters: the initial temperature T_1, the cooling schedule, the epoch length L, and the stopping condition [40]. Park and Kim later proposed a better set of annealing parameters that led to better results [41], on which we have based our parameter values.

4.3.1. *Initial temperature T_1*

Kirkpatrick et al. [38] propose a value of T_1 high enough to make the initial probability of accepting transitions to be close to 1. However, a too high initial temperature may lead to an unnecessary long computation time, as pointed out in [40,41], where it is suggested that the initial temperature is chosen such that the fraction of accepted uphill transitions at the initial temperature is equal to a parameter P_1, called the initial acceptance probability. The initial temperature is then calculated as $T_1 = \frac{\overline{\Delta}}{P_1}$, where $\overline{\Delta}$ is calculated with the initial negative gains only.

4.3.2. *Cooling function*

The cooling function will gradually decrease the temperature until the algorithm reaches its stopping condition. We adopt a simple geometric cooling function, in which the temperature at the kth epoch is given by $T_k = \alpha \times T_{k-1}$, where $\alpha (0 < \alpha < 1)$ is a parameter called the cooling ratio. Johnson et al. [40] showed that other cooling functions such as linear or logarithmic functions do not affect the performance.

Algorithm 3 Simulated Annealing

```
$Bestpartition \leftarrow Currentpartition$
$Temperature \leftarrow StartTemperatureT_1$
repeat
    $counter \leftarrow 0$
    repeat
        $counter++$
        Calculate gain $g$ to move vertex $i$ to part $p$
        if $g \geq 0$ then
            if $Free(p) \geq w_i$ then
                Perform move
            else
                Perform move with probability $e^{\frac{Free(p)-w_i}{Temperature}}$
            end if
        else
            if $Free(p) \geq w_i$ then
                Perform move with probability $e^{\frac{g}{Temperature}}$
            end if
        end if
        if $Currentpartition < Bestpartition$ then
            $Bestpartition \leftarrow Currentpartition$
        end if
    until $counter \geq L$ (epoch length)
    $Temperature \leftarrow \alpha.Temperature$
until stopcondition
return $BestPartition$
```

Table 1
Simulated annealing parameter values used in our SA refinement algorithm.

Parameter	Value
P_1	0.627
α	0.908
s	50
$F_{\min}$	0.02
I	5

4.3.3. *Epoch length L*

The epoch length L is the number of moves tried at each temperature level. Johnson et al. state that the epoch length should be chosen proportional with the neighbourhood size. In our case each vertex can be moved to one of the other machines, thus the epoch length becomes $L = s \times N \times (K - 1)$ with N the number of vertices and K the number of machines used. Experiments showed 50 to be a good value for s.

4.3.4. *Stopping condition*

The stopping condition decides when the algorithm reaches the frozen state and further search for a solution should be stopped. Trivial stopping conditions terminate the algorithm when the epoch count reaches a predefined maximum, or when the temperature drops below a pre-selected final temperature. We use Johnson's stopping rule [40], which increments a counter by one at the end of an epoch and the fraction of accepted moves is less than a predefined limit $F_{\min}$ and the counter is reset to 0 when a better solution is found. The SA is terminated when the counter reaches a predetermined limit I.

The actual values for the different parameters we use are shown in Table 1, which are based on the ones stated in [40,41] and lead to good results for our problem.

Because of the randomness in Simulated Annealing, different runs can lead to different solutions. The quality of the solution can be improved by executing the SA algorithm multiple times and

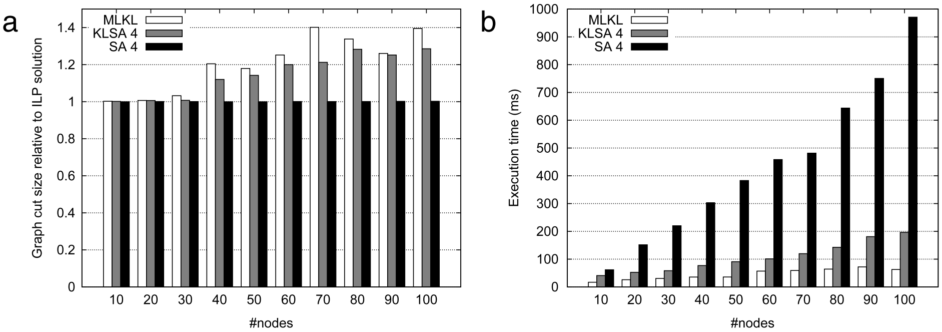


Fig. 4. The left shows the median of the resulting graph cut size of the MLKL, SA and KLSA relative to the optimal ILP solution. Simulated annealing reaches the optimum in more than 50% of the cases, while MLKL performs worse. KLSA is a bit better than MLKL but cannot achieve the quality of SA. The right shows the execution time of the different algorithms. Better solution quality comes at a price: KLSA and SA use respectively 2–10 times more execution time than MLKL.

taking the best solution. This comes of course at the cost of more computation time.

4.4. Hybrid algorithm (KLSA)

Because of the random factor of SA, it will typically explore a larger solution space than KL-based refinement, and hence arrive at better solutions at the cost of more computation time. To combine the best of both approaches, a hybrid approach is pursued in this section. The hybrid algorithm very much resembles the multilevel algorithm: only at the coarsest level the partition is first refined with a few runs of SA. At the coarsest level a better solution can be found with SA while the extra computation cost remains relatively low. In further uncoarsening iterations KL refinement is used again, since this is faster. Moreover, due to the characteristics of coarsening and uncoarsening, the globally optimal solution is expected to be situated in the neighbourhood of the coarsest graph, and thus the local optimum where KL gets stuck in the end is more likely also the global optimum. Again, the SA refinement at the coarsest level can be performed multiple times in order to get a better solution.

5. Evaluation results

To evaluate our algorithms we generated test graphs with different node sizes. Graphs were generated with the Eppstein power law generator [42] and the weights of the nodes and edges are assigned according an exponential distribution with parameter λ equal to 0.1 and 0.005 respectively. Although the actual graph structure of an application will depend on the software design, we chose these graph configurations as most design principles aim to reduce the dependencies between different software components thus resulting in graphs with few nodes with many neighbours. The resulting node weights are restricted between 1 and 100, which could intuitively represent the percentage CPU time needed for this component on a single processor core.

These nodes have to be deployed on machines with randomly generated capacities of 100, 200, . . . , 800, which represents machines equipped with single, dual, . . . , eight core processors. The number of machines is chosen such that their total capacity is about 150% of the total node weight of the graph. For each graph size we generate 100 graphs and machine weights and calculate the best deployment with our different algorithms implemented in Java, which are evaluated according to the resulting graph cut size and processing time. All calculations were performed on Intel Xeon L5420 (2.5 GHz) quad core processors.

First we compare our algorithms with the theoretically optimal ILP solution for small graph sizes to show the effectiveness of our approach. For bigger graphs we compare our three heuristics on solution quality and execution time and look at the influence of the cutoff threshold for KL refinement and the number of executions for SA. To benchmark our algorithms to the state-of-the-art we compare the solution quality of our algorithms with METIS 4.0. Finally we also show the effectiveness of our algorithms in the offloading scenario where a mobile device distributes its load to the cloud.

5.1. Evaluation of MLKL, SA and KLSA

To evaluate how good deployments are found for cloud applications, we assume that the communication links between the different servers in the datacenter are equally weighted, and thus $\forall m, n : b_{mn} = 1$. Since the graphs represent interacting software components, we also assume that the graph size of interest is a few hundred nodes. For small graphs we can compare solutions to the optimal ILP solution, as shown in Fig. 4.

Both the SA and KLSA algorithm use the best solution of 4 parallel executions of simulated annealing. For small graphs simulated annealing performs best as it results in the optimal solution in more than 50% of the cases. The hybrid solution tends to give slightly better results than MLKL. However better solution quality comes at the cost of execution time: KLSA and SA use respectively 2–10 times the execution time of MLKL.

For larger graphs the ILP solver takes too long to find a solution so we compare the algorithms relative to the MLKL algorithm. We will also discuss the influence of different parameters: the cutoff threshold for the KL refinement and the number of parallel executions of the SA refinement.

Fig. 5 on the left shows the solution quality of the MLKL algorithm for values 0, 100, 500 and 1500 for the cutoff threshold. KL refinement is stopped when the graph cut of the current partition deviates by more than a certain cutoff threshold of the best solution found at that moment. A cutoff threshold of 0 means that no moves with negative gains are allowed and then the refinement behaves as a steepest descent algorithm. The higher the threshold, the more it approaches the original solution without threshold. As the graph size gets larger, less improvement is reached by allowing moves with negative gains. On the right the average execution time of the different algorithms is plotted. Using a cutoff threshold can lower the execution time at the cost of solution quality. Choosing a threshold high enough (1500 in our case) leads to a small profit in execution time without sacrificing much solution quality.

Fig. 6 shows the solution quality of the SA algorithm compared to the MLKL algorithm (left) and the execution time (right) for different number of execution runs. For smaller graphs SA finds better solutions than MLKL, but as the graph size gets larger MLKL becomes better, as due to the increase in search space SA will less likely find the random move that leads to a better solution. More execution runs of the SA algorithm leads to better solution quality at the cost of more execution time. Since the experiments were conducted on quad core machines there is a much bigger increase between 4 and 8 than between 1 and 4. As already seen for small

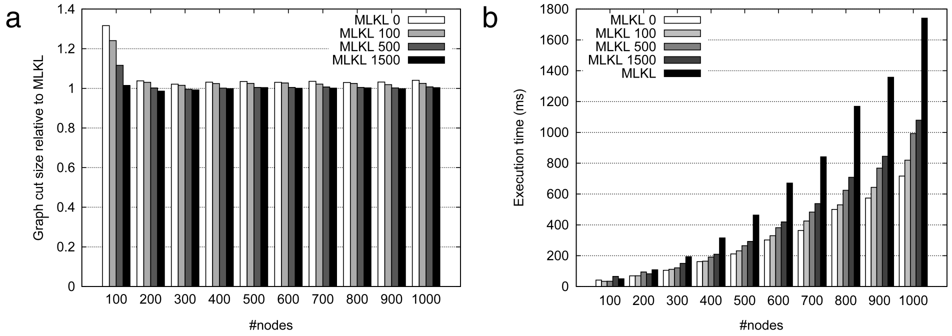


Fig. 5. The left graph shows the influence of a cutoff threshold on the solution quality relative to the MLKL algorithm without cutoff threshold, while the right shows the execution time. A lower cutoff threshold lowers execution time at the cost of a lower solution quality. The improvement of solution quality by allowing moves with negative gains is lowered as graph size gets larger.

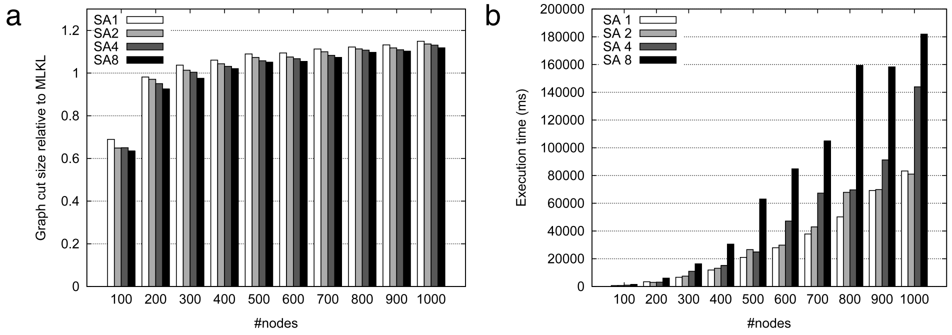


Fig. 6. The left graph shows the solution quality of SA relative to the MLKL algorithm, while the right shows the execution time. For smaller graphs SA results in better partitions, but uses much more execution time than MLKL.

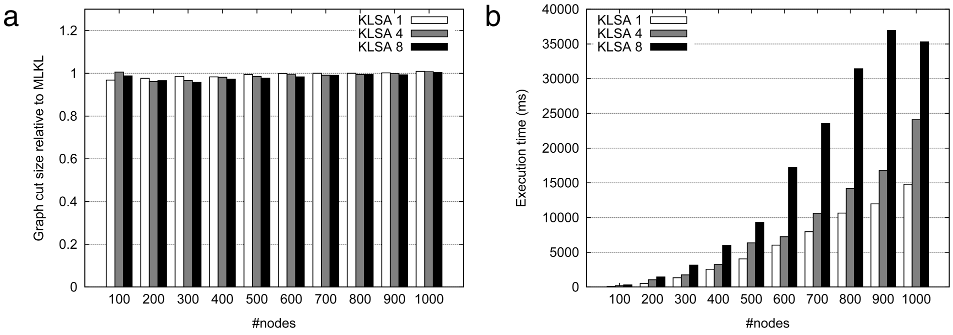


Fig. 7. The left graph shows the solution quality of KLSA relative to the MLKL algorithm, while the right shows the execution time. KLSA results in slightly better partitions than MLKL, but this improvement diminishes as the graph size gets larger. Regarding execution time it performs better than SA, but it is still slower than MLKL.

graphs in Fig. 4, the execution time of SA is much higher than that of MLKL.

The hybrid approach (KLSA) shown on Fig. 7 is a few percent better than MLKL, but this improvement becomes smaller as the graph size gets larger. Again this improvement comes at the cost of execution time. Since the expensive SA refinement is now only executed at the coarsest level, its execution time is lower than SA but still much higher than MLKL.

5.2. Comparison with METIS 4.0

A special case of our problem statement, where $\forall m, n : b_{mn} = 1$ and $\forall m : M_m = (\sum_i w_i)(1+\beta)/K$ represents the traditional graph partitioning problem that partitions the graph in K balanced partitions with imbalance factor β. For this problem we can compare our algorithm to well known graph partitioning software such as METIS [24].

Fig. 8 shows the solution quality of our MLKL algorithm relative to the solution of KMETIS to partition our graphs into 2, 4, 8, 16,

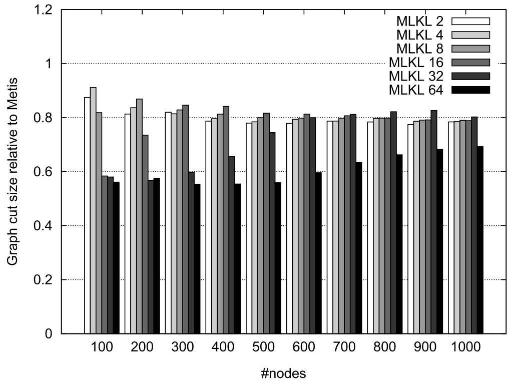


Fig. 8. Solution quality of the MLKL algorithm relative to METIS 4.0 to partition the graph in 2, 4, 8, 16, 32 or 64 balanced parts. Our algorithm results in 10%–40% lower graph cuts.

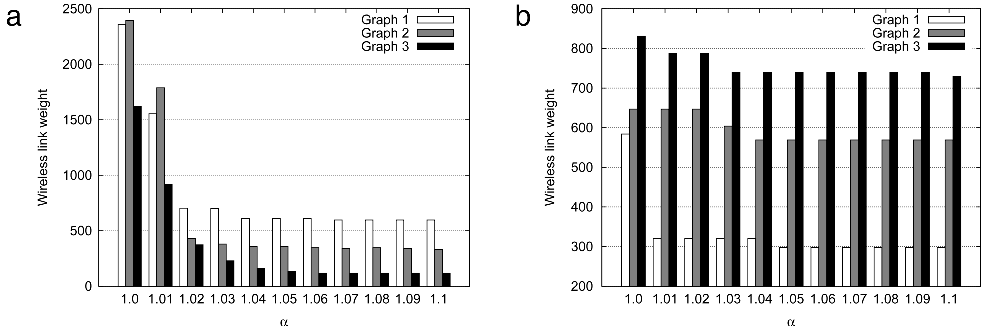


Fig. 9. By increasing the parameter α the bandwidth on the wireless link can be influenced. For fine grained graphs composed of many small components shown on the left the influence is more gradual than for the coarser graphs shown on the right.

32 and 64 balanced parts. Although both algorithms are based on the multilevel partitioning paradigm and KL-like refinement, our algorithm results in 10%–40% lower cut sizes. This is due to the fact that our algorithm allows more imbalance during the refinement, and a faster cutoff during refinement of METIS, as it is designed to scale to larger ($>$10,000 vertices) graphs.

5.3. *Offloading scenario*

Until now we always assumed that all application components are deployed in the cloud, where all servers are interconnected by the same links and thus $\forall m, n : b_{m,n} = 1$. However, in the context of mobile cloud computing where the cloud is used to offload parts of the application from the mobile device to the cloud, an important limiting factor will be the wireless link connecting the device to the internet. Depending on the technology and the signal strength the available bandwidth between the mobile device and the cloud – and thus the optimal deployment – will vary. In order to calculate an optimal deployment in this case, the b_{mn} parameters can be used to attach more importance to the wireless link and try to limit the amount of data exchanged via that link.

To calculate the optimal deployment in the offloading scenario, we added one extra machine m_{device} with a small capacity of 25 to our randomly generated list of machines and locked one of the nodes to this machine. This represents, for example, a node that takes care of the input or the GUI and must be deployed on the mobile device. The connectivity weights between the machines are set to 1 between the server machines, but to $\alpha \geq 1$ for each link between a server and m_{device}. When increasing α the algorithm will calculate a new deployment and will try to lower the bandwidth on the wireless link.

Of course the granularity with which one can influence the bandwidth on the wireless link will depend on the design of the software and the granularity of the components. If the software is composed of many small components one will be able to gradually tune the bandwidth requirements, but when the software is composed of a few big components fewer possibilities are available. Fig. 9 shows the weight on the wireless link as a function of α for six graphs. On the left results are shown for three graphs composed of 500 small ($\lambda = 0.5$) components, while on the right the results for three coarser graphs of 200 bigger ($\lambda = 0.1$) components are plotted. The more fine grained graphs on the left can be tuned more gradually than the coarser graphs on the right.

6. Conclusions and future work

In this paper we presented algorithms for partitioning software graphs for deployment on the cloud. The best partition is calculated taking into account the infrastructure heterogenity. In contrast to deployment optimization for computational grids, we do not minimize the execution time of a set of mesh structured tasks, but we focus on minimizing the bandwidth between software components. A multilevel KL-based algorithm is presented as a fast partitioner which allows realtime deployment calculations. Simulated annealing improves the solution quality for small graph sizes, at the cost of computation capacity. A hybrid algorithm produces slightly better partitions than KL in an intermediate execution time. We compared our KL-based algorithm to METIS for a wide range of graphs and found 10%–40% better partitions. We also showed how our algorithm can be used to adapt the deployment to take into account wireless link quality in a mobile cloud computing use case.

Future work consists of integrating these algorithms in an actual software deployment framework to automatically distribute software components on a cloud infrastructure. The algorithms can also be adapted to optimize alternative objectives, for example energy consumption.

References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, M. Zaharia, Above the clouds: a Berkeley view of cloud computing, Tech. Rep., 2009.

[3] Noesis solutions, optimus. http://www.noesissolutions.com/Noesis/.

[4] T. Verbelen, T. Stevens, P. Simoens, F. De Turck, B. Dhoedt, Dynamic deployment and quality adaptation for mobile augmented reality applications, Journal of Systems and Software 84 (2011) 1871–1882.

[5] S. Ou, K. Yang, J. Zhang, An effective offloading middleware for pervasive services on mobile devices, Pervasive and Mobile Computing 3 (4) (2007) 362–385.

[6] M. Kristensen, Scavenger: transparent development of efficient cyber foraging applications, in: 2010 IEEE International Conference on Pervasive Computing and Communications, PerCom, 2010, pp. 217–226.

[7] T. Verbelen, P. Simoens, F. De Turck, B. Dhoedt, Cloudlets: bringing the cloud to the mobile user, in: Proc. of the 3rd ACM Workshop on Mobile Cloud Computing & Services, MCS'12, 2012.

[8] S. Han, S. Zhang, J. Cao, Y. Wen, Y. Zhang, A resource aware software partitioning algorithm based on mobility constraints in pervasive grid environments, Future Generation Computer Systems 24 (6) (2008) 512–529.

[9] C. Alpert, Recent directions in netlist partitioning: a survey, Integration, The VLSI Journal 19 (1–2) (1995) 1–81.

[10] B. Hendrickson, Graph partitioning models for parallel computing, Parallel Computing 26 (12) (2000) 1519–1534.

[11] H. Meyerhenke, B. Monien, S. Schamberger, Graph partitioning and disturbed diffusion, Parallel Computing 35 (10–11) (2009) 544–569.

[12] T.N. Bui, C. Jones, Finding good approximate vertex and edge partitions is NP-hard, Information Processing Letters 42 (3) (1992) 153–159.

[13] P.-O. Fjallstrom, Algorithms for graph partitioning: a survey, Computer and Information Science 3 (10) (1998).

[14] K. Schloegel, G. Karypis, V. Kumar, The Sourcebook of Parallel Computing, Morgan Kaufmann, 2003, pp. 491–541.

[15] B. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell System Technical Journal 49 (2) (1970) 291–307.

[16] C. Fiduccia, R. Mattheyses, A linear-time heuristic for improving network partitions, in: Papers on Twenty-Five Years of Electronic Design Automation, ACM, 1988, p. 247.

[17] L. Sun, M. Leng, An effective multi-level algorithm based on simulated annealing for bisecting graph, Lecture Notes in Computer Science 4679 (2007) 1.

[18] L. Sun, M. Leng, S. Yu, A new multi-level algorithm based on particle swarm optimization for bisecting graph, Lecture Notes in Computer Science 4632 (2007) 69.

[19] M. Leng, S. Yu, An effective multi-level algorithm based on ant colony optimization for bisecting graph, Lecture Notes in Computer Science 4426 (2007) 138.

[20] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, CDROM-Supercomputing'95, 1995, 28-es.

[21] Y. Saab, An effective multilevel algorithm for bisecting graphs and hypergraphs, IEEE Transactions on Computers 53 (6) (2004) 641–652.

[22] C. Aykanat, B. Cambazoglu, B. Ucar, Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices, Journal of Parallel and Distributed Computing 68 (5) (2007) 609–625.

[23] A. Pothen, H.D. Simon, K.-P. Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM Journal on Matrix Analysis and Applications 11 (3) (1990) 430–452.

[24] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1) (1999) 359–392.

[25] F. Pellegrini, Scotch and libScotch 5.1 user's guide, LaBRI, Université Bordeaux I, August 2008. http://www.labri.fr/pelegrin/scotch/.

[26] C. Walshaw, M. Cross, JOSTLE: parallel multilevel graph-partitioning software an overview, in: Mesh Partitioning Techniques and Domain Decomposition Techniques, 2007, pp. 27–58.

[27] R. Khandekar, S. Rao, U. Vazirani, Graph partitioning using single commodity flows, Journal of the ACM 56 (4) (2009) 1–15.

[28] P. Chardaire, M. Barake, G.P. McKeown, A PROBE-based heuristic for graph partitioning, IEEE Transactions on Computers 56 (12) (2007) 1707–1720.

[29] R. Loureiro, A. Amaral, An efficient approach for large scale graph partitioning, Journal of Combinatorial Optimization 13 (4) (2007) 289–320.

[30] J.G. Martin, Spectral techniques for graph bisection in genetic algorithms, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation—GECCO'06, 2006, p. 1249.

[31] S. Kumar, S. Das, R. Biswas, Graph partitioning for parallel applications in heterogeneous grid environments, in: Proceedings of the 16th International Parallel and Distributed Processing Symposium, vol. 00, 2002, p. 167.

[32] B. Arafeh, K. Day, A. Touzene, A paradigm for allocating parallel application tasks to heterogeneous computing resources on the grid, in: Proceedings of the International Conference ParCo'05, 2005.

[33] P. Phinjaroenphan, S. Bevinakoppa, P. Zeephongsekul, A heuristic algorithm for mapping parallel applications on computational grids, Advances in Grid Computing 3470 (2005) 1086–1096.

[34] S. Huang, E. Aubanel, V.C. Bhavsar, PaGrid: a mesh partitioner for computational grids, Journal of Grid Computing 4 (1) (2006) 71–88.

[35] D.J. Harvey, S.K. Das, R. Biswas, Design and performance of a heterogeneous grid partitioner, Algorithmica 45 (3) (2006) 509–530.

[36] J. Chen, V. Taylor, Mesh partitioning for efficient use of distributed systems, IEEE Transactions on Parallel and Distributed Systems 13 (1) (2002) 67–78.

[37] IBM ILOG CPLEX. http://www.ilog.com/products/cplex/.

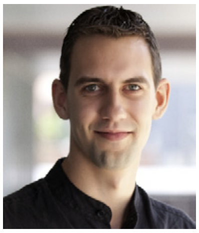
[38] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680. (New York, NY).

[39] V. Cerny, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, Journal of Optimization Theory and Applications 45 (1) (1985) 41–51.

[40] D.S. Johnson, C.R. Aragon, L.a. McGeoch, C. Schevon, Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning, Operations Research 37 (6) (1989) 865–892.

[41] M.-W. Park, Y.-D. Kim, A systematic procedure for setting parameters in simulated annealing algorithms, Computers & Operations Research 25 (3) (1998) 207–217.

[42] D. Eppstein, J. Wang, A steady state model for graph power laws, in: 2nd International Workshop on Web Dynamics, 2002.



Tim Verbelen received his M.Sc. degree in Computer Science from Ghent University, Belgium in June 2009. Since then he has been working as a Ph.D. researcher at the Department of Information Technology (INTEC) of the Faculty of Engineering at Ghent University. His main research interests include mobile computing, context-aware computing and adaptive software.

Tim Stevens received his M.Sc. degree in Computer Science from Ghent University, Belgium, in June 2001. In January 2009, he obtained his Ph.D. degree in Computer Science Engineering from the same university. Before joining the Department of Information Technology (INTEC) of Ghent University in August 2003, Tim was a database and system administrator for the Flemish public broadcasting company (VRT). At present, he is a postdoctoral researcher affiliated with INTEC.

His main research interests include future access network architectures, IPv6, Quality of Service (QoS) and traffic engineering in IP networks, anycast-based services, and cloud and grid computing.

Filip De Turck received his M.Sc. degree in Electronic Engineering from the Ghent University, Belgium, in June 1997. In May 2002, he obtained the Ph.D. degree in Electronic Engineering from the same university. During his Ph.D. research he was funded by the F.W.O.-V., the Fund for Scientific Research Flanders. From October 2002 until September 2008, he was a post-doctoral fellow of the F.W.O.-V. and part time professor, affiliated with the Department of Information Technology of the Ghent University. At the moment, he is a full-time professor affiliated with the Department of Information Technology of the Ghent University and the IBBT (Interdisciplinary Institute of Broadband Technology Flanders) in the area of telecommunication and software engineering. Filip De Turck is author or co-author of approximately 250 papers published in international journals or in the proceedings of international conferences. His main research interests include scalable software architectures for telecommunication network and service management, cloud computing, performance evaluation and design of new telecommunication and eHealth services.

Bart Dhoedt received a Masters degree in Electro-technical Engineering (1990) from Ghent University. His research, addressing the use of micro-optics to realize parallel free space optical interconnects, resulted in a Ph.D. degree in 1995. After a 2-year post-doc in opto-electronics, he became Professor at the Department of Information Technology.

Bart Dhoedt is responsible for various courses on algorithms, advanced programming, software development and distributed systems. His research interests include software engineering, distributed systems, mobile and ubiquitous computing, smart clients, middleware, cloud computing and autonomic systems. He is author or co-author of more than 300 publications in international journals or conference proceedings.