Highly Scalable Web Service Composition using Binary Tree-based Parallelization

Patrick Hennig, Wolf-Tilo Balke
L3S Research Center
Hannover, Germany
{hennig, balke}@L3S.de

***Abstract*—Data intensive applications, e.g. in life sciences, pose new efficiency challenges to the service composition problem. Since today computing power is mainly increased by multiplication of CPU cores, algorithms have to be redesigned to benefit from this evolution. In this paper we present a framework for parallelizing service composition algorithms investigating how to partition the composition problem into multiple parallel threads. But in contrast to intuition, the straightforward parallelization techniques do not lead to superior performance as our baseline evaluation reveals. To harness the full power of multi-core architectures, we propose two novel approaches to evenly distribute the workload in a sophisticated fashion. In fact, our extensive experiments on practical life science data resulted in an impressive speedup of over 300% using only 4 cores. Moreover, we show that our techniques can also benefit from all advanced pruning heuristics used in sequential algorithms.**

Keywords–web services; service compostion; parallelization

I. INTRODUCTION

The composition of individual services to build flexible workflows with reusable components is at the heart of the web service paradigm. *Static* approaches like orchestration and choreography are already part of the current standards. Orchestrations use a central coordinator to invoke available sub-processes and orchestration languages like BPEL [1] are already used in many systems. In choreographies there is no central coordinator, but complex tasks are defined via a conversation specification using choreography languages such as WS-CDL [2].

For scenarios needing flexible interactions between a set of independent providers, however, *dynamic* compositions are needed: services have to be invoked on-the-fly to form a (usually not predefined) workflow to reach some goal, or facilitate some task. But here research is still challenged with the enormous size of the planning space. Moreover, many rather technical problems in the implementation have to be solved, such as service discovery, selection, interoperability, reliability, or QoS constraints.

The traditional field of application for service compositions consists of (usually rather limited) *business* and *e-commerce* scenarios such as secure payments, travel planning, or e-shopping. But recently also fields with more special requirements like e.g., real-time constraints for digital item adaptation in *multimedia services* (see [3,4]) successfully applied the dynamic composition paradigm to create complex workflows.

Currently, a new challenge is posed by *data-intensive services* used in the *life sciences* either for exchanging and processing experimental data or even as a tool for modeling (and understanding) complex systems such as metabolic networks in biology (see e.g. [5,6]). Here, due to the enormous number of existing services and their applicability in many different processes, *scalability problems* are raised to new heights: the number of services to be efficiently composed ranges in the area of several thousand in each workflow. Of course, finding all potential compositions is a NP-hard combinatorial problem with exponential time and resource complexity. But recent work in [7] proposes a novel binary-tree-based representation technique promising reasonable efficiency for composition algorithms.

In this paper we focus on the question of how to harness the power of multi-core architectures to build composition frameworks providing the actual scalability needed for life science applications. Building on binary trees for modeling web service compositions, we show that the web service community needs sophisticated algorithms beyond straightforward parallelization to create efficient solutions. In fact, our experiments show that simple parallelization even leads to inferior performance over efficient serial composition algorithms. Using the running example of finding workflows in metabolic networks for the area of systems biology (so-called pathways), we propose two novel parallelization techniques. Our optimizations are specifically tailored to the needs of the service composition problem, and their basic paradigms of course can also be exploited in general application scenarios. Our experiments show that using state-of-the-art quad core machines composition times can be more than halved using our base algorithms and can even be further improved by sophisticated pruning heuristics.

The paper is organized as follows: in section 2, we give a brief overview of related work. In section 3, the serial baseline algorithm is presented. In section 4, we introduce our sophisticated parallelization approaches, while section 5 covers advanced pruning heuristics. Finally, section 6 closes this paper with a short conclusion and outlook.

II. RELATED WORK IN SERVICE COMPOSITION

The general problem of automatic service composition has been around for some time and gained additional attention with the research efforts invested in the Semantic

Web and service-oriented architectures. Formally current methods can be roughly classified into three categories: logic-, rule- and AI planning-based approaches. For a good overview of existing methods see [8].

However, when talking about the respective performance and benefits approaches from different categories cannot be compared easily. [9] proposes a rough categorization to enable service composition method comparison using the categories *service connectivity*, *non-functional properties*, *composition correctness*, *automatic composition* and *composition scalability*. In particular, scalability is identified as the most important challenge in the area of web service compositions. Since we are interested in parallelizing the composition problem, we will focus on the respective measure in our experiments.

Many approaches to solve the actual service composition problem have been proposed. Common formal approaches use finite state machines [10], algebra (e.g. π-calculus [11]) and Petri nets [12] to implement service composition. Still, a major problem of automatic composition is the 'semantic gap' between concepts that people understand and data that computers are able to interpret. Here approaches like the service ontology OWL-S [13] can help to ease the problem and thus recent work often decouples the semantic interoperability problem from pure and efficient composition techniques. Since our focus is on efficiency improvement, for the rest of this paper we will also abstract from semantic ambiguities and assume that all services are well-described by their characteristics like pre- and postconditions, etc.

Recently a number of graph- or tree-based methods have been proposed to specifically speed up the composition process in terms of finding possible solutions, see e.g. [7,14,15]. The underlying rationale is that calculations needed for compositions can be described as graphs in which vertices represent computations and edges reflect data dependencies. Thus, the approaches purely focus on most efficient ways to find the possible compositions generally for a restricted tree depth to avoid cycles. We will build our parallelization framework on these approaches and in particular use the approach in [7] as a serial baseline algorithm for comparison.

Currently multi-core computers promise massive performance gains for parallelized algorithms, which in turn has sparked new interest in the complex domain of problem partitioning. For a basic introduction see [16] for the design and analysis of parallel algorithms, including trees and graphs. A good overview on shared-memory multiprocessor architecture principles, concurrency (e.g. concurrent stacks), monitors and blocking synchronization is given in [17].

Also in the web service composition community first approaches to parallelization have already been designed. In [18] a choreographer is modeled that selects necessary components of a complex composition goal state and executes the planning for the individual components in parallel. Similarly, compositions in [19] are built over portions of a pre-computed table of composition possibilities. However, both approaches safely restrict parallelization to portions of the composition graph that can be executed in parallel. As we will show in our experiments, the actual synchronization needed for such simple parallelization over the entire composition graph severely decreases the performance (often even beyond that of highly optimized serial algorithms).

In contrast our parallelization approaches reflect the *structure* of the composition problem. Following the model of tree-based computation graphs given in [7], it divides the calculation among multiple processors by partitioning the vertices. For the actual partitioning of graphs for parallel processing an introduction is given in [20]. As we will see, exploiting both the graph structure and the special characteristics of the composition problem leads to our approaches' superior performance.

III. Parallelizable Service Composition

Web services can be flexibly composed in service chains to perform even complex tasks. A composition request R abstractly defines the required input and output parameters R.in and R.out. Analogously, WS.in and WS.out describe the input and output parameters of a web service WS in the service repository. In the following, WS can also be a set of web services. In that case, WS.in and WS.out are the union of the input and output parameters of the single web services in WS. A web service WS_x is able to satisfy the preconditions of another web service WS_y if the condition $WS_x.out \supseteq WS_y.in$ is met. Then, it is equivalent to say "the output parameters of WS_x match the input parameters of WS_y", or just "WS_x matches WS_y".

Web service composition frameworks need a representation model for compositions. Recently, binary trees have shaped up as the most efficient method [7]. Building on these tree structures, we designed our framework for parallelization. To be self-contained, we will shortly revisit their characteristics. The main idea is to create a binary tree for each composition request and store all information of the current composition process in this data structure. The composition tree has the following properties:

- nodes are web services or web service sets,
- links can only exist if inputs and outputs of both linked nodes fit/match,
- the root node reflects the output of a requested web service
- a branch is a set of serially linked nodes
- a left branch reflects a disjunction of all nodes (except the topmost) on this branch,
- a right branch reflects a conjunction of all nodes (except the topmost) on this branch.

Fig. 1 depicts the principle of a binary composition tree. WS_1 and WS_2 are sets of web services, WS_{11}, WS_{12}, and WS_{21} are individual web services. This small example shows that R.out can be satisfied by either WS_{11} together with WS_{12} or only by WS_{21}. In a logical representation this can be expressed as $R.out = WS_{11} \wedge WS_{12} \vee WS_{21}$.

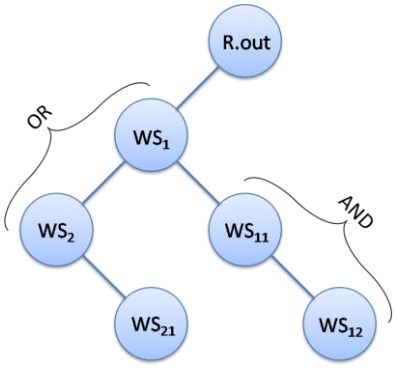

Figure 1. Binary composition tree

Basically, our composition framework performs the following workflow steps:

1. import service repository into database
2. build inverse index for all web service outputs
3. create binary composition tree
4. extract solution trees from composition tree

While steps 1 and 2 can be preprocessed, the remaining steps are performed after a service request was posed to the composition engine. Initially, a file repository is parsed and written into a database. Every file represents a web service described by OWL-S [13]. Further importing methods are possible, e.g. online repositories or other data formats. An artificial node *R.in* is added to the repository to allow all possible compositions (up to a certain depth) to be created. Next, an inverse index is built on the web services output parameters. Basically, this is a mapping of all output parameters existing in the repository to the services providing them. Once the query is posed and a service discovery does not lead to an appropriate service, a service composition is necessary. Now as a third step, the binary composition tree is built. The binary tree composition algorithm will be described in more detail later on. Finally, for some maximum tree depth, all solutions are extracted from the binary composition tree and visualized using the DOT markup language and Graphviz [21].

The essential algorithms of our composition framework are presented in the following. The main algorithm *compose* takes the imported service repository, the computed inverse index, a web service request and a maximum composition tree depth and calls algorithms to compute preconditions and build the binary tree. Preconditions are a list of sets of web services that fulfill another web service's inputs. After initializing some variables, a service discovery is done at line 8. If no service WS is found that matches the user request R so that $R.in \supseteq WS.in \wedge R.out \subseteq WS.out$ is true, the root node is pushed on a stack. This node was created before and it contains a set of web services, in this case, just one service with requested output parameters as input. The following while-loop takes nodes from the stack until it is empty. Within that loop, for every node it is checked, if its interior web service can be satisfied by the requested input parameters. In that case, the node is marked to be satisfied and we are done with that node and get the next one from the stack. If not, the preconditions are computed by calling *getPreconditions*. Then, *processPreconditions* processes the computed preconditions to create the binary tree. Finally, the root node is returned after working off the whole stack (which grows in *processPreconditions*), which contains the complete binary tree using links to child nodes.

Algorithm "compose"

```
Input webservices, outputIndex, requestedWS,
      maxDepth
Output rootNode
1  preconditions = new ArrayList();
2  rootWS = new Webservice();
3  rootWS.setInputs(requestedWS.getOutputs());
4  rootSet = new HashSet();
5  rootSet.add(rootWS);
6  rootNode = new BinaryNode(rootSet);
7  stack = new Stack();
8  "check if requested WS exists in webservices"
9  stack.push(rootNode);
10 while (!stack.isEmpty()) {
11   node = stack.pop();
12   ws = "webservice in node"
13   if (ws can be satisfied by req. input) {
14     node.setSatisfied(true);
15   } else {
16     preconditions = getPreconditions(ws,
                              outputIndex);
17     processPreconditions(preconditions, node,
                              stack, maxDepth);
18   }
19 }
20 return rootNode;
```

Algorithm "processPreconditions"

```
Input preconditions, rightNode, stack, maxDepth
1  if (preconditions == null) {
2    return;
3  }
4  for (precondition : preconditions) {
5    leftNode = new BinaryNode(precondition);
6    rightNode.insertLeft(leftNode);
7    if (leftNode.getDepth() > maxDepth) {
8      leftNode.remove();
9      return;
10   } else {
11     spreadNode(leftNode, stack, maxDepth);
12   }
13 }
```

This algorithm *processPreconditions* takes a list of preconditions, a binary node, a stack and a maximal depth parameter and builds or deletes branches under the given node. For every precondition a new node is created and attached as a left child node of the given node. If there is already a left child node, *insertLeft* attaches the new node as the left child of that left child node recursively. Additionally, it sets the new node's depth as parent node depth + 1. The algorithm terminates if the maximum tree depth is reached. Next, we call *spreadNode* for the new node to create a right branch using its interior web service set.

Algorithm "spreadNode"

```
Input node, stack, maxDepth
1  wsSet = node.getElement();
2  if (node.getDepth()+wsSet.size()>maxDepth) {
3    return;
4  }
5  for (ws : wsSet) {
6    singleSet = new HashSet();
7    singleSet.add(ws);
8    singleNode = new BinaryNode(singleSet);
9    node.insertRight(singleNode);
10   stack.push(singleNode);
11 }
12 return;
```

The functionality of *spreadNode* is partly similar to *processPreconditions*. A right branch is created from the given web service set by splitting it up into single nodes and attaching them as right child nodes to the given node. After getting the web service set from the given node in the first step, it is ensured that the right branch to be created will not exceed the tree depth limit. Next, for all services in the set, we create a new node, insert it as right child of the given node and push the new node to the stack to be processed later. Again, the node is inserted as a right child of the right child node recursively, and its depth is set to parent node depth + 1.

Algorithm "getPreconditions"

```
Input ws, outputIndex
Output preconditionList
1  reqOutputIndex = new ArrayList();
2  for (entry : ws.getInputs()) {
3    inputSet = outputIndex.get(entry);
4    if (inputSet != null) {
5      reqOutputIndex.add(inputSet);
6    } else {
7      return null;
8    }
9  }
10 preconditionList = new ArrayList();
11 if (reqOutputIndex.size() == 1) {
12   preconditionList = convertSetToSetList(
                  reqOutputIndex.get(0));
13 }
14 for (i=0; i<reqOutputIndex.size()-1; i++) {
15   if (i == 0) {
16     preconditionList = combineSetListWithSet(
               convertSetToSetList(
                 reqOutputIndex.get(0)),
                 reqOutputIndex.get(i + 1));
17   } else {
18     preconditionList = combineSetListWithSet(
               combinationList,
               reqOutputIndex.get(i + 1));
19   }
20 }
21 return preconditionList;
```

The *getPrecondition* method takes a web service's input parameters and the inverted output index and returns a list of all preconditions by combining web services using *combineSetListWithSet*. This method combines a set list with a set by combining every set in the list with the given set. Maximum efficiency is reached by preventing the combination of useless sets at an early stage. *deleteAllSupersets* deletes all supersets of a given set list (including equal sets). *isSubsetInList* checks if a subset of a given set exists in a given set list. Fig. 2 exemplifies a generated binary composition tree for a single request. Here, gray nodes are the preconditions of the white nodes, and red nodes can be satisfied by the given input parameters.

Algorithm "combineSetListWithSet"

```
Input setList, webserviceSet
Output combinationList
1  reqOutputIndex = new ArrayList();
2  for (set : setList) {
3    for (ws : webserviceSet) {
4      combinedSet = new TreeSet();
5      combinedSet.addAll(set);
6      if (!combinedSet.contains(ws)) {
7        combinedSet.add(ws);
8        if (!isSubsetInList(
                 combinationList, combinedSet)) {
9          combinationList.add(combinedSet);
10       }
11     } else {
12       combinationList = deleteAllSupersets(
                  combinationList, combinedSet);
13       if (!isSubsetInList(
                 combinationList, combinedSet)) {
14         combinationList.add(combinedSet);
15       }
16       break;
17     }
18   }
19 }
20 return combinationList;
```

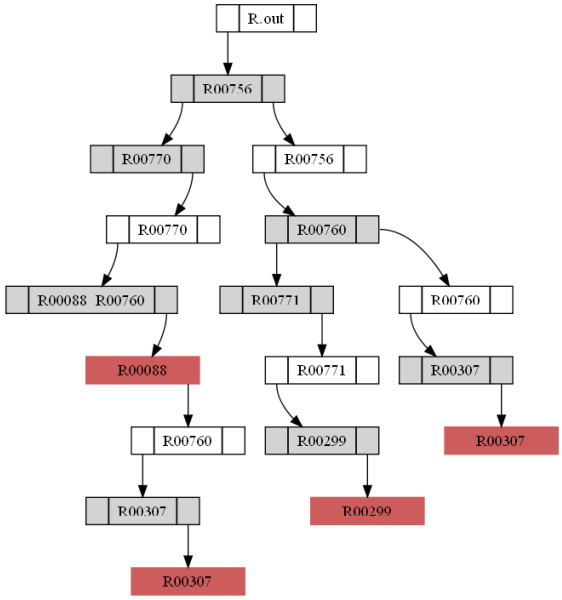

Figure 2. Example binary composition tree

Naïve Parallelization. Our first parallelization approach takes advantage of the usage of a stack in the serial algorithm. It is obvious to let different threads pick and process different nodes from the stack to get a very even distribution of work. The main thread takes nodes from the

shared stack and starts a thread for each one to process it. If the maximum number of free threads is reached, the main thread falls asleep and waits to be notified by a finished thread. Each time a thread has finished its work, it wakes up the main thread and waits to get another item from the stack. This way, the shared stack will be worked off until no more items are put on by the sub threads and the stack is empty. Fig. 3 depicts this principle.

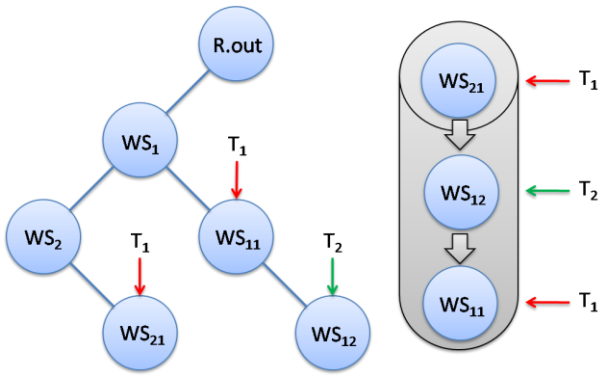

Figure 3. Naïve parallelization

Evaluation Setup. For all our experiments we use an Intel Core 2 Quad CPU Q9450 with 2.67 GHz per core. Furthermore this machine has 4 GB main memory. The operating system was Microsoft Windows 7 with 64 bit architecture. All algorithms have been implemented in Java version 1.6. We randomly generated composition requests and report the average time of several runs. Our web service repository includes 7816 different web services, created from biochemical reactions. We imported these reactions from the KEGG database [22].

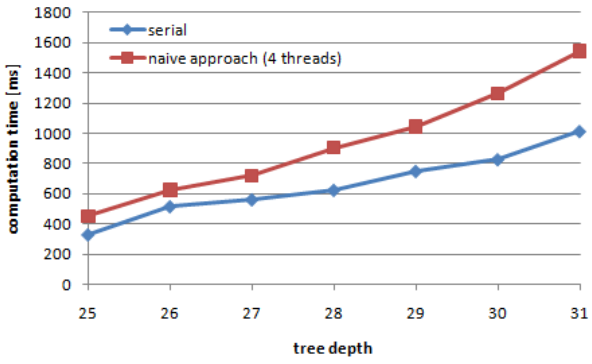

Figure 4. Results of naïve approach

Results. Fig. 4 shows the performance of the naïve parallelization against the state-of-the-art serial algorithm from [7]. Interestingly the results show that the serial algorithm is always faster. Thus, it is not at all trivial to parallelize an efficient sequential composition approach. The main reason is the parallelization and synchronization overhead, which we will minimize in the next section.

IV. Sophisticated Parallelization

Generally, to implement good parallelization, the problem has to be partitioned minimizing the amount of synchronization overhead. In particular, a lot of blocking synchronization has to be implemented for concurrent processing. We propose two sophisticated parallelization approaches to distribute the workload to multiple threads. Moreover, we always minimized parallelization overhead by reusing finished threads using Java's Executor service.

Thread-optimized parallelization. This approach is a modification of the naïve approach. The main difference is that before new threads are created, all nodes are checked by the main thread: if its preconditions are already satisfied by the given inputs, no new thread will be created. Additionally, it is checked if preconditions of the current node have been computed previously. In this case, we process the node within the main thread. Otherwise, a thread is created to process this node as in the naïve approach. This way we save overhead using a simple hash map to remember previously computed preconditions.

Sub-tree partitioning. The basic idea of this approach is to minimize parallelization overhead, specifically targeting concurrency problems. This is realized by dividing the binary composition tree into sub-trees in the topmost layer (see Fig. 5). Each sub-tree is then computed by a single dedicated thread, using its own stack. On one hand, each thread can work off its own stack and there is no need for synchronization or interaction between threads anymore. On the other hand, the distribution of work is less even, because sub-trees may differ in depth and complexity. Thus, if the binary composition tree is balanced, this approach can tap its full potential.

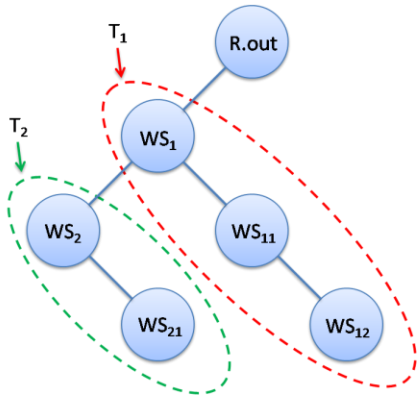

Figure 5. Sub-tree partitioning

Results. Again, we compared the performance of both our approaches to the serial approach, to see if we can now gain some performance speedup through parallelization. Fig. 6 shows the computation time of the serial algorithm and both sophisticated parallelization algorithms using 4 threads for varying composition tree depths.

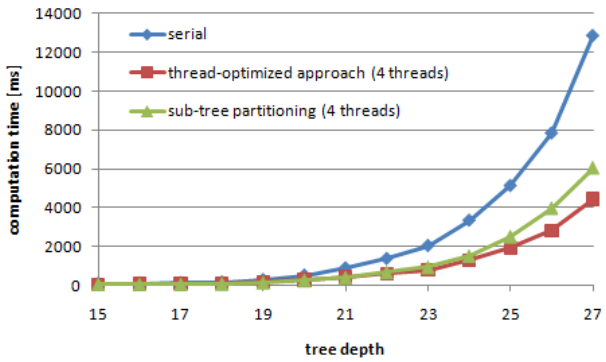

Figure 6. Results of sophisticated approaches

The results reveal a definite improvement through parallelization. The thread-optimized approach is about 3 times faster than the serial algorithm, while the sub-tree partitioning method still reaches more than a double speedup, both using 4 threads on a 4 core machine. The reason for the thread-optimized parallel algorithm being a bit faster is because the workload is distributed more evenly.

In the next experiment we investigated the correlation between computation time speedup and the number of used threads. The computed binary composition tree has a fixed depth of 30. Fig. 7 presents the behavior of both our approaches' computation times depending on the number of used threads, varying from one to eight. First, we can observe that the thread-optimized approach benefits most from increasing the number of threads due to better distribution of workload. If just one thread is used, the computation times are pretty similar for both parallel algorithms because no parallelization is done at all. With two to four threads, we can observe a definite speedup of 312%, respectively 211% as expected. From five to eight threads there is no significant additional speedup, because we just used 4 CPU cores (still, a slight speedup can be gained due to hyperthreading effects).

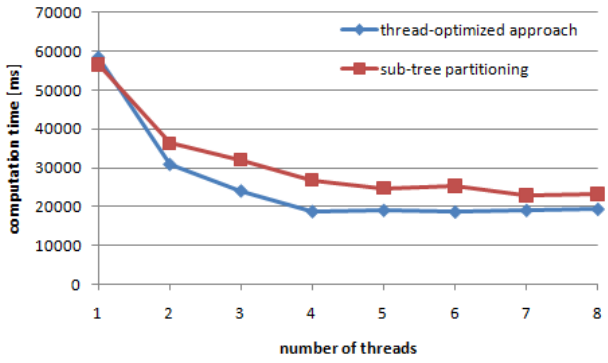

Figure 7. Correlation of speedup and number of threads

V. ADVANCED PRUNING HEURISTICS

Since today's most efficient sequential web service composition algorithms gain their efficiency by pruning optimizations, we also tested two advanced pruning heuristics in both our parallelization approaches.

Optimization 1: rememberPreconditions. To prevent previously computed preconditions to be computed again, in this heuristic we enhanced our framework by a lookup table of previously calculated results. Like in the thread-optimized parallelization, we globally reduce computation repetitions by remembering every result (satisfied and failed nodes) of precondition computations. If our table holds an entry for some specific node, it is directly processed. Otherwise, the algorithm proceeds as before. Since this lookups have to be done very frequently, we decided to use a hash map for the purpose of maximum performance: the hash map *formerPrecond* has to be declared and initialized in our baseline algorithm *compose* and line 16 has to be replaced by the following code.

```
1  if (formerPrecond.containsKey(ws) {
2    preconditions = formerPrecond.get(ws);
3  } else {
4    preconditions = getPreconditions(ws,
                       outputIndex);
5    formerPrecond.put(ws, preconditions);
6  }
```

Optimization 2: deleteBranches. If a given web service node fails, this algorithm recursively deletes the entire right branch the failed node is part of. To prevent inconsistencies while concurrently processing the same binary composition tree with multiple threads, we have to wrap the whole algorithm with a synchronized block.

Initially it is checked, whether the given node has a parent node. If so, we try to find the root of the right branch and delete all parent-links on the way up. Deleting the links is important for concurrent processing, because other threads might still be working on a deleted branch. But such threads are forced to immediately stop processing, if the backtracking does not lead to parent nodes anymore. Furthermore, for every step up on the branch, the current node may have to be removed from some position deep within the stack, because other threads could have modified the stack in the meantime.

After the right branch's parent node was found, we check if it has a left child node (again, after ensuring that there is still a parent node). If it has a left child, we re-link it with the parent node and vice versa, so existing sub-trees won't be lost. Otherwise, we check if its parent node is a left or right child. If it is a left child or has no parents at all, the left child is set null. If it's a right child, then it fails and the whole (conjunctive) right branch can also be deleted recursively. This way, all failing nodes and branches are pruned, and no obsolete nodes stay on the stack. This method has to be integrated in *processPrecondition* before line 2 with parameters *rightNode* and *stack* when no preconditions have been found and before line 8 with

parameters *leftNode* and *stack* if the new node exceeds the allowed tree depth limit. Due to the same reason, *deleteBranches* is called in *spreadNode* before line 3 with parameters *node* and *stack.*

Algorithm "deleteBranch"

```
Input node, stack
1  synchronized {
2  firstTime = true;
3  if (node.getParent() == null) {
4    return;
5  }
6  while (node.getParent() != null &&
           node.getParent().getLeft() != node) {
7    node = node.getParent();
8    node.getRight().deleteParent();
9    if (firstTime) {
10     firstTime = false;
11   } else if (stack != null) {
12     firstTime = false;
13     stack.remove(node.getRight());
14   }
15 }
16 if (node.getLeft() != null) {
17   node.getLeft().setParent(node.getParent());
18   node.getParent().setLeft(node.getLeft());
19 } else {
20   node = node.getParent();
21   if(node.isLeftChild()||node.hasNoParent()){
22     node.setLeft(null);
23   } else if (node.isRightChild()) {
24     deleteBranch(node, stack);
25   }
26 }
27 }
```

Results. We investigated our algorithms' computation time improvement, when both advanced pruning heuristics are applied. For both parallel methods we again used four threads, varied binary composition tree depth and measured computation time. Fig. 8 shows the results: both parallelization methods still outperform the serial algorithm, when applying the two advanced pruning heuristics to all algorithms. It's interesting to observe that the sub-tree partitioning approach benefits most from the new heuristics, now outperforming the serial and the thread-optimized parallel approach. The reason is that the overhead for workload balancing spent by the thread-optimized approach cannot be amortized after the pruning is applied.

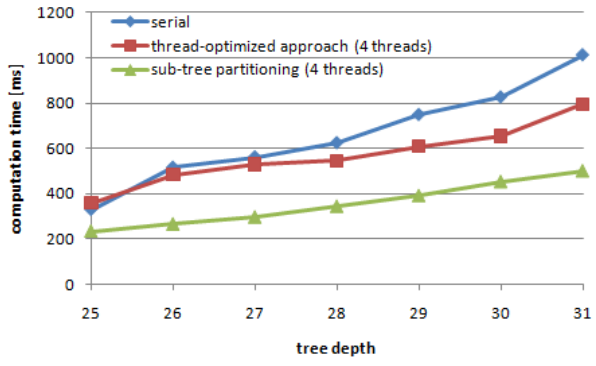

Figure 8. Advanced pruning heuristics

Moreover, we also investigated scalability when increasing the input set size, i.e. the number of web services for the composition. We present the respective results in Fig. 9, which shows a quite similar scalability behavior of all composition methods.

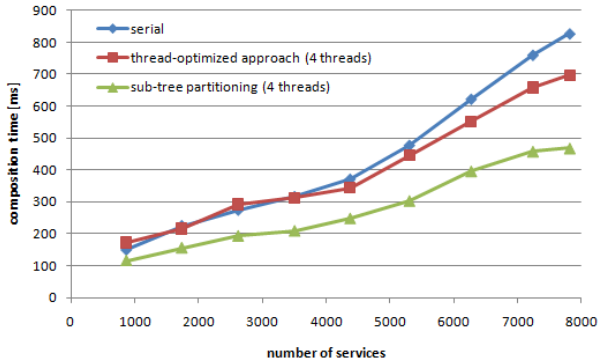

Figure 9. Varying the number of services

Since advanced pruning heuristics have a great impact on overall computation times, we had a closer look at their uncorrelated influence. We examined the computation times for all parallel approaches with and without optimizations with four threads and a fixed tree depth of 30. Fig. 10 shows the impact of both optimizations.

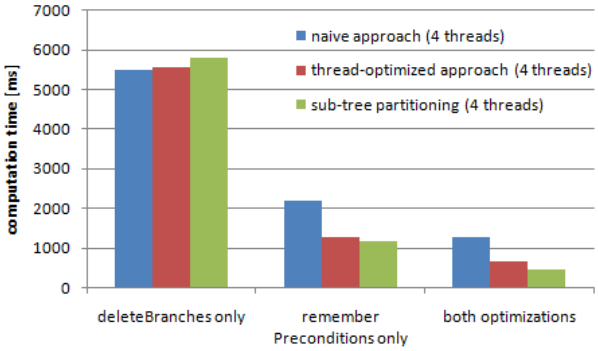

Figure 10. Impact of optimizations

As seen before, the sub-tree partitioning approach benefits most. In this case, we can save up to 98.31% computation time. Using the naïve and thread-optimized approach, we can save up to 93.34% and 96.5% when applying both advanced pruning heuristics to the algorithms. Obviously, the naïve approach profits least, if *rememberPreconditions* is applied, because the significantly higher synchronization overhead doesn't pay off anymore.

VI. Conclusion and Outlook

In this paper we presented a web service composition framework featuring two parallel service composition approaches based on binary trees. After showcasing that parallelization of state-of-the-art service composition algorithms is by far not trivial, we showed our algorithms to

outperform the most efficient sequential algorithm. While being up to three times faster on a four core platform, our experimental results revealed that a considerable performance and scalability improvement is possible through parallelization and is additionally enhanced by pruning heuristics. Thus, we can utilize the great potential of multi-core systems, being able to handle the general composition problem in extensive service repositories. Due to high scalability and a fast preprocessing in cases of repository changes, we conclude that it is thoroughly possible to easily apply our framework algorithms to dynamic real time composition systems with vast web service repositories.

Our future work focuses on realizing semantic matchmaking within our service composition framework by implementing different metrics (e.g. cosine, extended Jacquard [23]) and ontologies. With these modifications, it will be possible to compute top-k results or stop execution after finding one valid solution. Finally, we will try to detect and prevent composition loops by more advanced techniques like computing hash values of composition paths instead of just restricting the binary tree depth.

REFERENCES

[1] "Web Services Business Process Execution Language (WSBPEL)," http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] "Web Services Choreography Description Language (WS-CDL)," http://www.w3.org/TR/ws-cdl-10/.

[3] X.G. Klara, X. Gu, K. Nahrstedt, R.N. Chang, and C. Ward, "QoS-Assured Service Composition in Managed Service Overlay Networks," Proc. of the 23rd Int. Conf. on Distributed Computing Systems (ICDCS'03), Providence, Rhode Island, 2003.

[4] S. Tönnies, B. Köhncke, P. Hennig, and W. Balke, "A Service Oriented Architecture for Personalized Rich Media Delivery," Proc. of the Int. Conf. on Services Computing (SCC'09), Bangalore, India, 2009.

[5] G. Zheng and A. Bouguettaya, "Discovering Pathways of Service Oriented Biological Processes," Proc. of the 9th Int. Conf. on Web Information Systems Engineering (WISE'08), Auckland, New Zealand, 2008.

[6] G. Zheng and A. Bouguettaya, "A Web Service Mining Framework," Proc. of the Int. Conf. on Web Services (ICWS'07), Salt Lake City, Utah, USA, 2007.

[7] A. Zhou, S. Huang, and X. Wang, "BITS: A Binary Tree Based Web Service Composition System," Int. Journal of Web Services Research, vol. 4, 2007.

[8] A. Marconi and M. Pistore, "Synthesis and Composition of Web Services," Formal Methods for Web Services, Springer, 2009.

[9] N. Milanovic and M. Miroslaw, "Current Solutions for Web Service Composition," IEEE Internet Computing, vol. 8, 2004.

[10] Ç.E. Gerede, R. Hull, O.H. Ibarra, and J. Su, "Automated composition of e-services: lookaheads," Proc. of the 2nd Int. Conf. on Service Oriented Computing (ICSOC'04), New York, NY, USA: ACM, 2004.

[11] S. Narayanan and S.A. McIlraith, "Simulation, Verification and Automated Composition of Web Services," Proc. of the 11th International World Wide Web Conference (WWW'02), Honululu, Hawaii: 2002.

[12] R. Hamadi and B. Benatallah, "A Petri Net-based Model for Web Service Composition," Proceedings of the 14th Australasian Database Conference (ADC'03), Adelaide, Australia, 2003.

[13] "OWL-S: Semantic Markup for Web Services," http://www.w3.org/Submission/OWL-S/.

[14] H. Tang, F. Zhong, and C. Yang, "A Tree-Based Method of Web Service Composition," Proc. of the Int. Conf. on Web Services (ICWS'08), Beijing, China, 2008.

[15] M.M. Shiaa, J.O. Fladmark, and B. Thiell, "An Incremental Graph-based Approach to Automatic Service Composition," Proc. of the Int. Conf. on Services Computing (SCC'08), Honolulu, HI, USA, 2008.

[16] J. JáJá, An Introduction to Parallel Algorithms, Redwood City, CA, USA: Addison Wesley, 1992.

[17] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.

[18] J. Pathak, S. Basu, R. Lutz, and V. Honavar, "Parallel Web Service Composition in MoSCoE: A Choreography-Based Approach," Proc. of the 4th European Conf. on Web Services (ECOWS'06), Zurich, Switzerland, 2006.

[19] P. Bartalos and M. Bielikova, "Semantic Web Service Composition Framework Based on Parallel Processing," Proc. of the 11th Conf. on Commerce and Enterprise Computing (CEC'09), Vienna, Austria, 2009.

[20] B. Hendrickson and T.G. Kolda, "Graph partitioning models for parallel computing," Parallel Computing, vol. 26, 2000.

[21] "Graphviz," http://www.graphviz.org/.

[22] "KEGG: Kyoto Encyclopedia of Genes and Genomes," http://www.genome.jp/kegg/.

[23] M. Klusch, B. Fries, and K. Sycara, "Automated semantic web service discovery with OWLS-MX," Proc. of the 5th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS'06), Hakodate, Japan: 2006.