Automatic Generation of Optimized Workflow for Distributed Computations on Large-Scale Matrices

Farida Sabry, Abdelkarim Erradi, Mohamed Nassar, and Qutaibah M. Malluhi

KINDI Center for Computing Research
Qatar University
Doha, Qatar
{faridasabry,erradi,mohamad.nassar,qmalluhi}@qu.edu.qa

Abstract. Efficient evaluation of distributed computation on large-scale data is prominent in modern scientific computation; especially analysis of big data, image processing and data mining applications. This problem is particularly challenging in distributed environments such as campus clusters, grids or clouds on which the basic computation routines are offered as web/cloud services. In this paper, we propose a locality-aware workflow-based solution for evaluation of large-scale matrix expressions in a distributed environment. Our solution is based on automatic generation of BPEL workflows in order to coordinate long running, asynchronous and parallel invocation of services. We optimize the input expression in order to maximize parallel execution of independent operations while reducing the matrix transfer cost to a minimum. Our approach frees the end-user of the system from the burden of writing and debugging lengthy BPEL workflows. We evaluated our solution on realistic mathematical expressions executed on large-scale matrices distributed on multiple clouds.

Keywords: location-aware optimization, distributed computations, BPEL workflows, large-scale matrices.

1 Introduction

Cloud computing offers an attractive alternative to easily and quickly acquire IT services such as storage and computation services. Its adoption continues to grow as companies opt for flexibility, cost savings, performance and scalability. Cloud services such as Elastic MapReduce offer an attractive platform for outsourcing the storage and computations on large scale data because of their optimized algorithmic implementations and access to on-demand large-scale resources. We focus particularly on matrix algebra computations since they are used in many scientific domains; including but not limited to analysis of big data, image processing, computer graphics, information retrieval and data mining applications. The inputs are typically large-scale matrices and performing math operations (e.g. multiply, inverse, transpose, add/subtract, dot product…) on them could be long-running. In this paper, we consider the scenario where several cloud services are offering matrix storage and basic matrix operations with different service characteristics. Based on availability, quality of service (QoS), reliability, security and data locality, the optimal decomposition, task scheduling and task assignment of a mathematical expression vary. We propose automated workflow generation and execution in order to optimize the response time of expression evaluation, given the available services and their characteristics, as well the data locality. Our solution improves the productivity of the users by releasing them from the tedious task of manually and timely generating and editing the workflows depending on the input expressions.

The composition could be written using a business workflow language such as BPEL (Business Process Execution Language) [2] or YAWL (Yet Another Workflow Language) [3]. Scientific workflow tools like Taverna [4], Kepler [5], and Pegasus [6] can do similar task; some of them adopt BPEL whereas others use their own language. We choose BPEL because it is a standard XML based language for specifying a Web Services composition. It is also used by some scientific workflow systems. A BPEL process is composed of activities that can be combined through structured operators that specify the control and data flow that govern the ordering of these activities. BPEL constructs include messaging activities (e.g. invoke, receive, reply), sequential execution, conditional branching, structured loops, concurrency constructs (e.g., parallel execution, event-action constructs, correlation sets), exception handling (try-catch blocks). A BPEL engine is responsible for managing the process instances lifecycle, such as process instance creation, termination, and executing according to the process definition. The engine is also responsible for binding the partners to specific Web Services. Many BPEL engines are available as Open Source, such as Apache ODE [7], and commercial engines such as IBM WebSphere Choreographer [8].

Even though workflows can be used to automatically manage the execution of the expression computation, the system is not convenient if the end-users (e.g. researchers, developers) have to manually create a workflow and properly assign the tasks upon the addition of a new expression. Moreover the optimal execution is dependent on the data locality of input matrices and the QoS characteristics of the available matrix computations services. Our system automates the optimization and the generation of a BPEL workflow for the input expression. The resulting workflow is deployed to a BPEL workflow engine for execution.

The rest of the paper is organized as follows. Section 2 overviews related work. Section 3 gives an overview of the proposed method and section 4 presents the details of the transformation from a mathematical expression to a BPEL workflow and the optimization process. Section 5 highlights implementation details. Finally, we conclude and discuss future work in section 6.

2 Related Work

Service composition is closely related to workflow [9]; automatic workflow generation can be considered a subtask from automated web service composition. The latter term is considered more general as it includes an extra step of the automatic service discovery and selection from the set of available services. According to a survey of automated web services composition [10], this can be done using workflow techniques or AI planning. The workflow techniques can be further classified as either static or dynamic [9]. The static techniques mean that the requester should build an abstract process model before the composition planning starts. Only the selection and binding to atomic web services is done automatically. On the other hand, the dynamic composition both creates process model and selects atomic services automatically. This requires the requester to specify several constraints, including the dependency of atomic services, the user's preference and so on. An example for a static workflow generation approach was implemented in ASTRO project [11].

According to [11], one of the phases for the automatic composition of web services is the translation between the external and internal languages used by the service composition system. The external language is used by the service users to express what they can offer or what they want in a relatively easy manner. For example, BPMN (Business Process Modeling Notation) to BPEL translation is presented in [12] where the designer uses BPMN graphical notations to easily describe the process control flow and data flow and then it gets automatically translated to BPEL. This work can also be considered static in the sense that BPMN is describing the control/data flow as input. Similar work was proposed in [13] but using XPDL (XML Process Definition Language) which is a graph-structured language mainly used in internal process modeling. However, in this work the generated outputs are abstract BPEL processes that are not fully executable and deployable and they need some manual editing to be ready for deployment. Also in [12], it is stated that it cannot detect all pattern types and the code produced by this transformation lacks readability.

Our approach for automatic workflow generation presented in this paper is considered dynamic in the sense that the workflow steps and the process model that describes the control flow and data flow are not input by the requester but they are created automatically according to the parsing of the input expression. Additionally the atomic services used for computations are selected based on their functionality and QoS such as accuracy, reliability, performance and security. We assume that developers/researchers are using contract-based web service composition; and they are provided with the WSDLs representing the interfaces of the available services and their characteristics. Our proposed framework depends on the service-oriented architecture where large-scale mathematical computations are offered as services and this differs from other distributed execution engines like MapReduce [23] or DryadLINQ [24].

3 Overview of the Proposed Framework

We can think of the problem of mathematical expression to workflow transformation with analogy to the compilation process [15]. In software compilation, the compiler compiles a program into intermediate form, optimizes intermediate form and generates target code for the running architecture. In hardware compilation, the compiler compiles an HDL model into a sequencing graph, optimizes the sequencing graph and generates gate-level interconnection for a cell library [16].

In our framework of distributed mathematical expression evaluation using services on the web or on the clouds, the end-user (researcher/developer) enters a mathematical expression (e.g.$A * B + C * D$) following a specific grammar such as XPath grammar or JEP (Java Expression Parser) [14]. The expression is then compiled to an intermediate form of a parsed expression tree. This intermediate form is optimized and then the workflow is generated to coordinate the execution of services on the distributed environment. We focus on mathematical expressions but the framework can be extended to more generic computation models.

The main components of our proposed framework are depicted in Fig. 1. First, the developer/researcher inputs the expression and the resources' references corresponding to the aliases of the operands (i.e., the location where each operand is stored). A configuration file specifies additional parameters such as the registry address where the WSDLs of the services are stored. These WSDLs serve as the interface to the external cloud services to be invoked or composed in the generated BPEL process. A parser parses the input mathematical expression into an expression tree. An optimizer then transforms the tree to a more consolidate form based on data locality of operands and identifies independent operations that can be done in parallel. The optimizer also annotates the nodes of the tree based on their types (operands vs. operators). Then the translator traverses the tree and maps the tree parts to corresponding BPEL activities. Attributes of these activities like the partner link to the service to invoke, the values of the input variables to this service and their types are initialized according to the annotations set by the optimizer. The output of the translator is a BPEL process accompanied by a deployment descriptor so that it can be deployed to a BPEL engine for execution. In the next section we present formal definitions and explain in more details the different steps of the automation process.

4 From Expression to BPEL

Before we go through the automation steps in details, it is important to formally define the following key terms: computation services, operations, operands and expression trees.

Definition 1: [*Computation Services*] are defined as a set of services S = {s_1,s_2,…s_n}, n ≥ 1 where each s_i S is defined by [id, O_{s_i}, QoS_{s_i}] where id is the unique service identifier (e.g. the URL of the service) and O_{s_i} is a set of operations $\{o_{js_i}\}$ provided by s_i. Each $o_{js_i} \in O_{s_i}$ is further defined by its input, output and port type ($X_{o_{js_i}}$, $Y_{o_{js_i}}$, $PT_{o_{js_i}}$) where 1 ≤ j ≤ $|O_{s_i}|$. QoS_{s_i} is the set of quality of service parameters for each service s_i: $< P_i, D_i, r_i, a_i >$ where P_i is the set of execution price for all $o_{js_i} \in O_{s_i}$, D_i is the set of expected execution durations for all o_{js_i}, r_i is the reliability and a_i is the availability of the overall service.

In our framework the service definitions are obtained from a local registry by parsing the corresponding WSDL files.

Definition 2: [*Operators*] are the set of predefined tokens representing unary and binary operations on matrices such as addition, subtraction, multiplication, dot product, inverse of a matrix and transpose of a matrix: $O = \{ +,\ -,\ *,\ .,\ -,\ \wedge^{-1},\ ' \}$.

Definition 3: [*Operands*] are the set of input literals L used in the input mathematical expression, $L = \{ l_1, l_2, \ldots l_m\}$ where each l_k is an alias for a resource matrix M_k with metadata (location, nRows, nCols, datatype). The $(l_k\ , M_k)$ mapping tuples are stored to a hash map so-called LM.

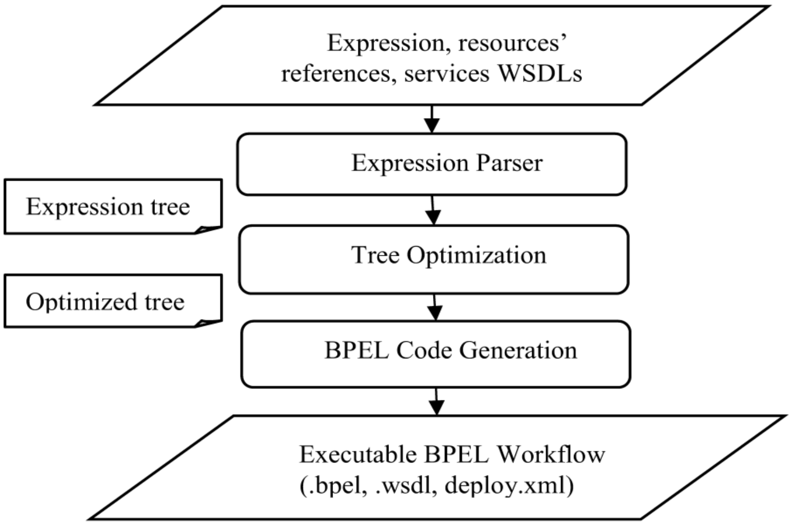

Expression, resources' references, services WSDLs
Expression Parser
Expression tree
Tree Optimization
Optimized tree
BPEL Code Generation
Executable BPEL Workflow (.bpel, .wsdl, deploy.xml)

Fig. 1. Mathematical expression to BPEL workflow generation

Definition 4: *[Expression Tree]* is the binary tree obtained from parsing the input string expression and is defined by $(root, N, C)$ where $N = \{n_1, n_2 \dots n_w\}$ is the set of tree nodes, $n_t \in \{O, L\}, root \in N \; and \; C = \{ (n_u, n_v), (n_u, n_c) \dots . \}$ represents the connections between the nodes, where (n_u, n_v) means n_u is a parent of n_v. The following conditions apply:

- $root$ is the only node with no parents.
- The leaf nodes must belong to L.
- Internal nodes belong to O, a hash map OS maps each operator node $n_t \in O$ to the service $s_k \in S$ offering this operation and being selected to do the operation according to data locality, concurrency considerations and QoS parameters.
- Each node has at most two direct children.
- Methods $left(n_t)$ and $right(n_t)$ get the left and right child of node n_t.

Given these definitions, we discuss next expression-to-BPEL translation steps in more details.

4.1 Expression Parser

Parsers have undergone significant progress and can now be automatically generated from a simple specification of the language (i.e., BNF grammar). This can be done using one of the existing parser generators like YACC, Bison or ANLTR. There are two main approaches to building parsers that are used in practice: top-down (also known as recursive descent or LL and its variant LL(*) [17] used by ANTLR) and bottom-up (aka shift-reduce, LR and its variant LALR used by YACC and Bison).

In our work, we use the open-source JEP which implements the Shunting-yard algorithm that is considered a bottom-up parser and is used to convert the human-readable infix notation to RPN (Reverse Polish Notation) that is optimized for expression evaluation. The output of this step is a left-deep parse tree, an example is shown in Fig. 2(a).

4.2 Tree Optimization

The goal of tree optimization is to maximize parallel execution of independent operations within the expression and minimize overall evaluation time. The time is mainly composed of two factors: the computation time for the operations run by the different services, and the data transfer time for matrix resources that need to be moved from one location to another in the distributed environment. As a first step, we assume that all the servers implement all the operations and have similar computation capabilities and quality of service characteristics. In this context, the tree execution cost is measured by the data transfer cost.

For an expression tree of $\boldsymbol{x}$ operator nodes and a set S of $\boldsymbol{n}$ available servers all implementing services for these operators, there are $\boldsymbol{n^x}$ possible execution plans to select services from S to execute the $\boldsymbol{x}$ operations. The order in which to invoke these $\boldsymbol{x}$ operations makes the search space even larger. Using exhaustive search to select the optimal plan in this space becomes practically impossible when the expression size increases. We refer to the query optimization problem in distributed databases that have similar conditions to get an optimal query execution plan [19, 20] where projection is done before join and joins of collocated tables are done first to decrease the data to be transferred, cost-optimization techniques are used to choose the optimal execution plan.

We narrow down the search space using the matrix locality information, where we favor operations involving collocated matrices. The basic principle is that matrices that are co-located in storage must be close to each other in the tree whenever it is possible. To do so, we use properties of commutativity, associativity and distributivity of the different operators to identify chains of commutative operators (e.g., matrix addition) and chains of associative operators (e.g., matrix multiplication). We sort the commutative chains based on the data locations. In this way, collocated matrices would be close and put into parenthesis to be operands of the same operator. We also use matrix size as a tie break for associative chains (i.e., we prefer to put together into parenthesis the operands of which the multiplication leads to smaller-size matrices). This problem is the same as the matrix chain multiplication problem [18] and has a well-known dynamic programming solution which we modified its score to favor doing computations for collocated matrices first.

To simplify the explanation of the optimization procedure we consider as example the expression $\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C} * \boldsymbol{D} * \boldsymbol{E} + \boldsymbol{F} + \boldsymbol{G}$ and the size-location description shown in Table 1. The optimization of this expression is shown in Fig. 2. Fig. 2(a) represents the tree as output by the parser. In Fig. 2(b) we use the associative property of multiplication to do D*E first as matrix D and matrix E both belong to server S1 and must be given priority to decrease data transfer. Similarly the commutative property of addition is used to swap matrix B and matrix F. Indeed matrix A and matrix F both belong to server S1 and can be locally added without additional data transfer.

Note that within the same sub-tree, well known compiler optimization techniques for arithmetic expressions are used to optimize further the execution and identify independent sequences of operations that can be done in parallel. There are a lot of optimization techniques for arithmetic expressions, like tree-height reduction, factorization, expansion and common sub-expression elimination [16, 17]. For example if we assume all the matrices belong to the same location in Fig. 2(a), tree height reduction would recognize that the root node must be changed so the tree height would be 4 instead of 6.

After this step is done we apply the following two-phases-traversal algorithm:

1. The first phase: we identify independent sub-trees that can be run in parallel while traversing down the tree based on the two following conditions:
   - All the nodes of a sub-tree must be hosted by the same server
   - A sub-tree must contain as much nodes as possible. In other words, we expand a tree until no more nodes can be added.
   - Each sub-tree is annotated according to the hosting server where its operations would be invoked so that the generated workflow invokes the services for computations of the sub-trees in parallel e.g. Fig. 2(c).
2. The second phase: going up the tree we generate the main meta-tree representing the final computation steps with annotations added specifying the servers selected to do each operation. Again the goal is to reduce the data transfer volume. So we choose the server where most matrices are located. The metric can be merely the number of matrices but preferably we select the server hosting the maximum sum of the sizes of the operands.

We analyze the transfer cost in terms of the number of matrix elements which is practically reflected in the file size. In this simple example the transfer cost is reduced from 2100100 elements (if re-ordering and optimization algorithm were omitted) to 1001000 elements. This gain is computed given the sizes depicted in Table 1 and assuming dense matrices. Another factor affecting the selection of services and discussed extensively in literature is the QoS parameters. For example, QoS parameters and techniques used in [21] can be applied to choose services with least response time and price. Currently our prototype is based solely on data locality and data size but we intend to extend it to QoS optimization as well. The last step is transforming the optimized tree, along with the annotations of the selected transfer and computation services and obtaining the finally executable BPEL workflow as described next.

Table 1. Example of a distribution of sizes and locations

Matrix ID	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Size	1000 *1000	1000 *1000	1000 *1	1 *100	100 *1000	1000 *1000	1000 *1000
Location	S1	S2	S2	S1	S1	S1	S2

4.3 BPEL Code Generation

The translation task from the optimized expression tree to BPEL workflow is based on the mapping rules shown in Fig. 3. In the rules, o_u, o_v and o_l represent operator nodes and l_i, l_j represent operand nodes. The rules has for mission to map the expression tree parts to their equivalent BPEL constructs such as assign, invoke, receive, sequence, and flow. BPEL *Assign* activity is used to exchange values between incoming and outgoing message variables. *Invoke* activity is used to do the service invocation. *Receive* activity is to receive an input message or a callback message. *Sequence* activity is to group some activities to be done in sequence. *Flow* activity is used when different sequences are to be done in parallel. Attributes of these activities like the partner link to the service to invoke, values of input variables to the service and their types are initialized according to the annotations values of the nodes (operands and operators: $n_t \in \{O, L\}$).

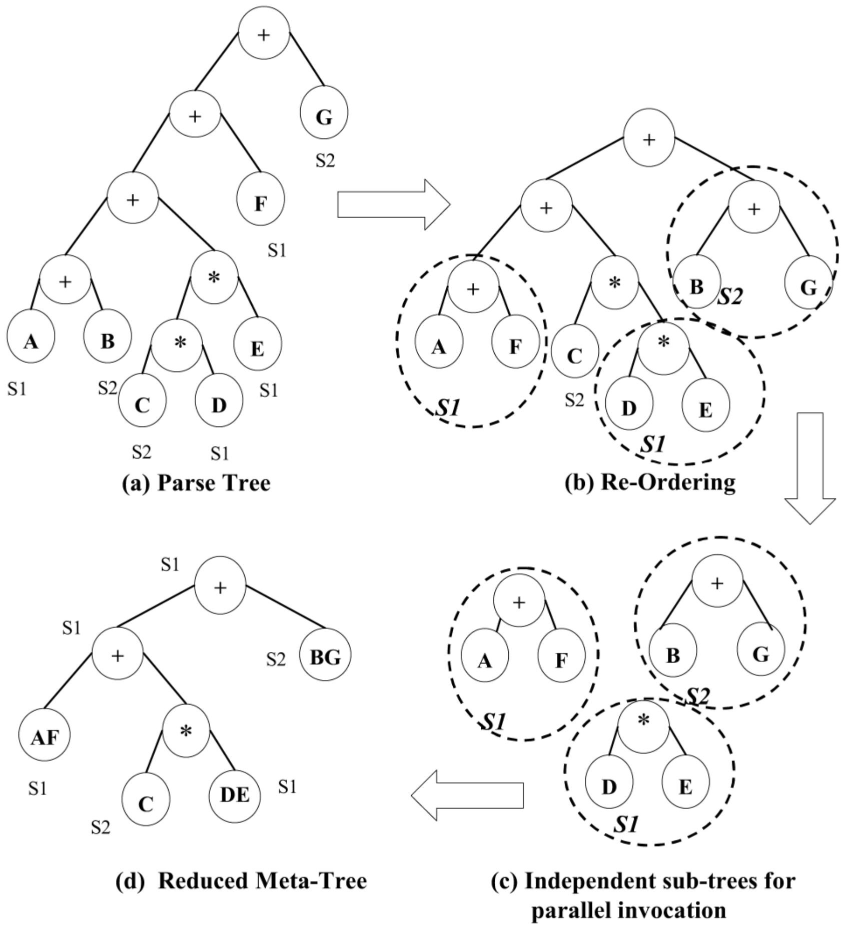


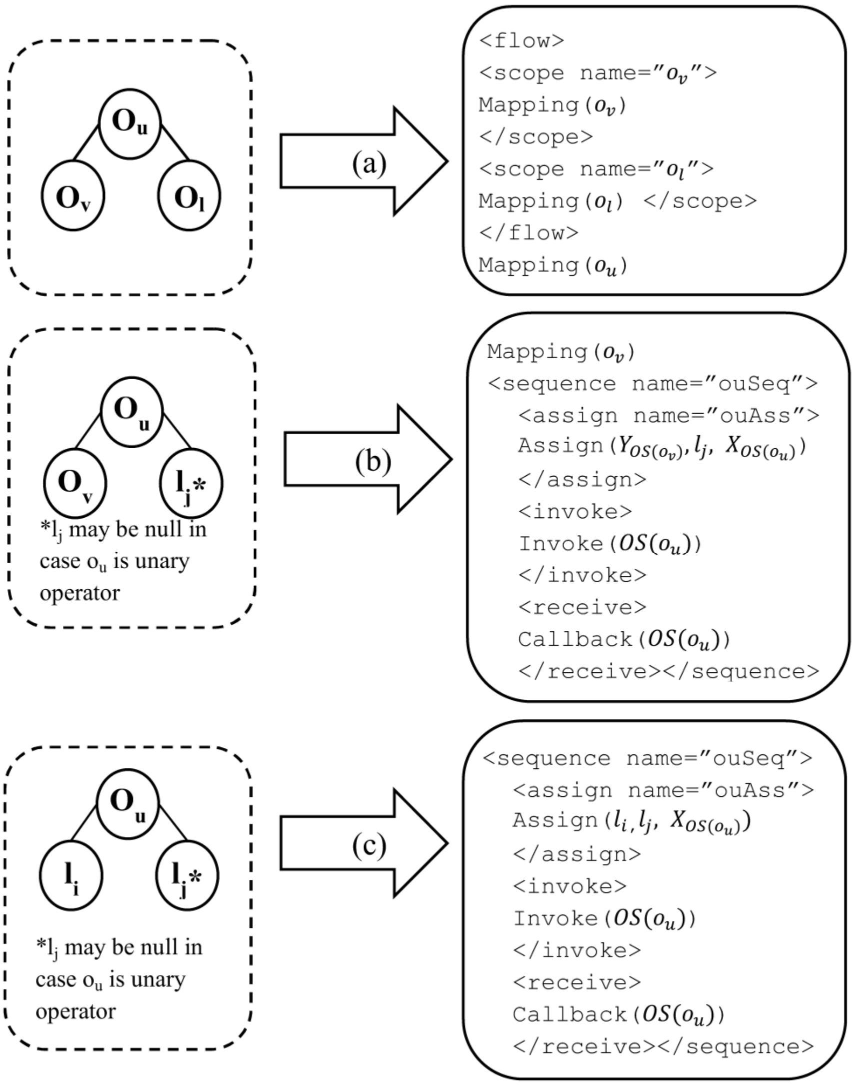
Fig. 2. Simple scenario example for tree optimization for A+B+C*D*E+F+G → (A+F) +C*(D*E)+(B+G)

The output of this transformation is a BPEL process saved to a ".bpel" file, a workflow interface description saved to a ".wsdl" file. This is because workflow itself is deployed as a web-service. A deployment descriptor saved to "deploy.xml" is also generated so that the workflow can be deployed to a BPEL engine for execution.

The translation algorithm of an expression tree T to executable BPEL code that includes the BPEL constructs to be used and the control flow is shown in Fig. 4. The algorithm is a post-order traversal for the expression tree T with the mapping rules shown in Fig. 3 applied. The rule case (c) in Fig. 3(c) is considered the base case used for the recursive traversal where the tree has an operator $o_u \in O$ as a parent and its two children are operands $\{l_i, l_j\} \in L$ or only left child l_i in case where o_u is a unary operator. In this case, the mapping is a *sequence* activity that includes (*assign, invoke, receive*). The BPEL *assign* activity is for assigning input values for the variable used in the invocation. The *invoke* activity and then the callback *receive* activity are to get the information about the intermediate result location. The attributes of these activities are determined from the computation services definition S and the OS mapping.$OS\ (o_u)$is the selected service for operation o_u. The $LM\ (l_k\,, M_k)$ mapping is used to get the metadata of the input matrices. Case (a) occurs when the two children are operators which mean that the services in these two paths can be executed in parallel. This corresponds to the BPEL *Flow* construct including two sequences for the mapping of the two children where each child has its own scope. Case (b) occurs when one of the children is an operatoro_i and the other is a literal l_j which means that the mapping of o_i and o_u will be a *Sequence* activity. A flow stack is maintained so that during traversal if case (a) is encountered a *Flow* activity is pushed into the stack and the two paths are executed in parallel. The activity is popped out once its left and right children return.

5 Implementation and Experimentation

We made the prototype for Mathematical Expression to BPEL (ME2BPEL) available at https://code.google.com/p/me2bpel/. The objective of the system is to generate a correct, optimized and executable BPEL workflow from the input mathematical expression and resources' references to aliases used in the expression. The inputs are WSDL files representing the interface to different web services on different servers and an expression to be evaluated with metadata about operands used in the expression provided. The whole system operation can be summarized as follows. First, the expression is being parsed using JEP API that uses shunting yard algorithm. Then we detect commutative chains and matrix multiplication chains by traversing the tree. Matrix multiplication chains and their order of execution are determined using the modified dynamic programming approach using data locality as well as matrices sizes. Sorting the commutative chain is done with respect to data locality and the expression tree structure is updated accordingly with annotating operator nodes for collocated operands. The rest of operator nodes are then annotated with the location to execute according to the minimum data transfer criterion. BPEL code generation is done according to the algorithm in Fig. 4. We modified the unified framework package [22] for generation and serialization of BPEL constructs. We used web services using MapReduce for matrix multiplication and addition operations that we used in [1] for testing. These input WSDLs are read and de-serialized using wsdl4j library.



```
<flow>
<scope name="$o_v$">
Mapping($o_v$)
</scope>
<scope name="$o_l$">
Mapping($o_l$) </scope>
</flow>
Mapping($o_u$)
```

```
Mapping($o_v$)
<sequence name="ouSeq">
  <assign name="ouAss">
  Assign($Y_{OS(o_v)}$, $l_j$, $X_{OS(o_u)}$)
  </assign>
  <invoke>
  Invoke($OS(o_u)$)
  </invoke>
  <receive>
  Callback($OS(o_u)$)
  </receive></sequence>
```

```
<sequence name="ouSeq">
  <assign name="ouAss">
  Assign($l_i$,$l_j$, $X_{OS(o_u)}$)
  </assign>
  <invoke>
  Invoke($OS(o_u)$)
  </invoke>
  <receive>
  Callback($OS(o_u)$)
  </receive></sequence>
```

Fig. 3. Mapping expression tree patterns to the corresponding BPEL constructs where Mapping(o_x) is a recursive function with case (c) as the base case

```
(1) **Input:** OptimizedExpressionTree T, S, OS=$(o_j, S_i)$ , $o_j = (X_j, Y_j, PT_j)$and LM=$(l_k, M_k)$ mapping
(2) **Output:** workflow.bpel, workflow.wsdl, deploy.xml
(3) **Begin**
(4) Workflow W, LastActivity A, FlowStack FS, currentFlow = 0
(5) Initialize W
(6) **Transform T:**
(7) **if** $left(T) \in L$ AND ($right(T) \in L$ OR $right(L)$ $is$ $null$) // case Fig. 3(c)
(8)         W←addMapping(rule in Fig. 3(c)) to currentFlow
(9) W←connectSequenceToLastFlow(FS)
(10)Update A
(11) LM←addIntermediateResult($Y_{T,OS(T)}$)
(12)     **Else if** $left(T) \in O$ AND ( $right(T) \in L$ OR $right(L) is null$ ) // case Fig. 3(b)
(13)**Transform** $left(T)$
(14)W←addMapping(rule in Fig. 3(b)) to currentFlow
(15)W←connectSequenceToLastActivity(A)
(16)Update A, Update FS
(17)LM←addIntermediateResult($Y_{T,OS(T)}$)
(18)     **Else if** $left(T) \in O$ AND $right(T) \in O$ //case Fig. 3(a)
(19)W←addFlow(), currentFlow= currentFlow+1
(20)W←connectFlowToLastActivity(A)
(21)Update A, Update FS
(22) **Transform** $left(T)$
(23)**Transform** $right(T)$
(24) Update FS
(25)W←addMapping(rule in Fig. 3(a)) to currentFlow
(26) W←connectSequenceToLastActivity(A)
(27) Update A, LM←addIntermediateResult($Y_{T,OS(T)}$), currentFlow = currentFlow -1
(28)**End Transform**
(29)Serialize W to get workflow.bpel
(30)Generate WSDL from W to get workflow.wsdl
(31)Generate Deployment Descriptor from W to get deploy.xml
```

Fig. 4. Translation algorithm of expression tree to BPEL workflow

The first experiment is to test for ten different expressions available on the project page as a sample dataset with different number of literals ranging from 4 to 10. The data locality optimization is not taken into consideration in this experiment and it is assumed that the data matrices are stored on the same server offering these web services. Results are shown in Fig. 5 with an average speed-up (Tsequential/Tworkflow) of 1.8. From the results it is clear that the optimized workflow achieve better results for expressions with larger number of literals and which have operations that can be done in parallel.

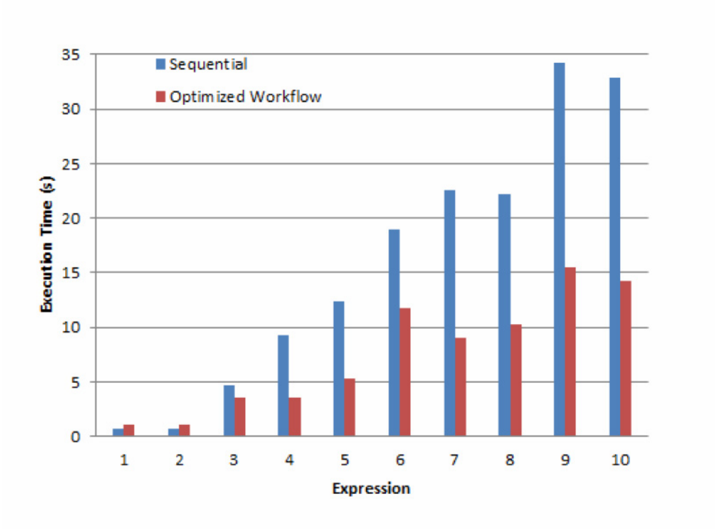


Fig. 5. Optimized workflow execution time vs. the sequential execution time for 10 different expressions

In the second experiment, we assume matrices are stored on different servers and according to the data locality optimization step; a service is chosen to execute a certain operation in an expression tree so that it minimizes the data transfer between servers. So we compare the time taken for data transfer being logged by the web services under test between the optimized workflow with web services selection according to data locality and random web services selection. Fig. 6 shows that for most of the expressions under test, the data transfer time is less when web services are selected according to data locality (expression 8 has all its data stored on the same server, that's why no data transfer time recorded). Some cases show no improvement; this depends on the heterogeneity of the distributed data.

6 Conclusion

Web and cloud-based services evaluating large-scale mathematical operations are typically long running and require the composition of multiple asynchronous computation services. We proposed an automated workflow generation solution in order to coordinate and optimize the execution of these services. We show how to automatically generate workflows for evaluating composed expressions while taking into account the storage location of input matrices and minimizing the data transfer between servers. Our solution optimizes the run-time execution of the services composition by maximizing parallel calls whenever possible. We aim by this contribution to increase the productivity of the system users (researchers or developers) and equipping them with a dynamic workflow generation tool, making the system accessible for non-expert workflow developers.

For future work, we aim to incorporate QoS-based service selection. This feature will allow selecting the most appropriate service among functionally-equivalent computation services having the same score according to data locality and size of input data but offering different QoS guarantees.

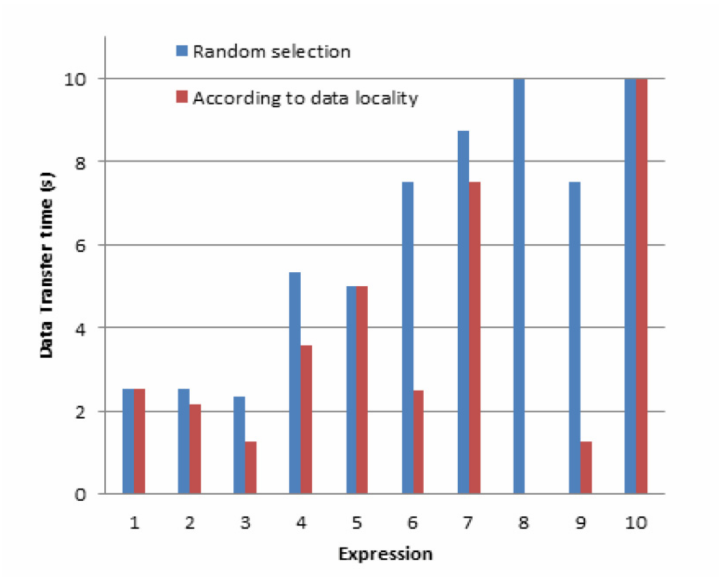


Fig. 6. Data transfer time taken by services selected according to data locality vs. random selection for different expressions

Acknowledgments. This publication was made possible by a grant from the Qatar National Research Fund; award number NPRP 09-622-1-090. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the Qatar National Research Fund.

References

1. Nassar, M., Erradi, A., Sabri, F., Malluhi, Q.: Secure Outsourcing of Matrix Operations as a Service. In: 6th IEEE International Conference on Cloud Computing, pp. 918–925. IEEE Press (2013)
2. Web Services Business Process Execution Language v2.0, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
3. Van der Aalst, W.M.P., ter Hofstede, A.: YAWL: Yet Another Workflow Language. Information Systems 30(4), 245–275 (2005)
4. Taverna Workflow Management System, `http://www.taverna.org.uk/`
5. Altintas, I., Berkley, C., Jaeger, E., Jones, M.: Ludascher. B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Scientific and Statistical Database Management International Conference, pp. 423–424 (2004)

6. Sonntag, M., Karastoyanova, D., Deelman, E.: BPEL4Pegasus: Combining Business and Scientific Workflows. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 728–729. Springer, Heidelberg (2010)
7. Apache ODE: `http://ode.apache.org/`
8. WebSphere Application Server Enterprise Process Choreographer, `http://www.ibm.com/developerworks/websphere/`
9. Dustdar, S., Schreiner, W.: A survey on web services composition. Journal of Web and Grid Services 1(1), 1–30 (2005)
10. Rao, J., Su, X.: A Survey of Automated Web Service Composition Methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
11. Trainotti, M., Pistore, M., Calabrese, G., Zacco, G., Lucchese, G., Barbon, F., Bertoli, P.G., Traverso, P.: ASTRO: Supporting Composition and Execution of Web Services. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 495–501. Springer, Heidelberg (2005)
12. Ouyang, C., Dumas, M., ter Hofstede, A.H.M., van der Aalst, W.M.P.: Pattern-based translation of BPMN process models to BPEL web services. International Journal of Web Services Research 5(1), 42–62 (2007)
13. Yuan, P., Jin, H., Yuan, S., Cao, W., Jiang, L.: WFTXB: A Tool for Translating Between XPDL and BPEL. In: 10th IEEE International Conference on High Performance Computing and Communications, pp. 647–652. IEEE Press (2008)
14. JEP (Java Expression Parser), `http://www.singularsys.com/jep`
15. Kastner, R., Hosangadi, A., Fallah, F.: Arithmetic Optimization Techniques for Hardware and Software Design. Cambridge University Press, Cambridge (2010)
16. Bacon, D., Graham, S., Sharp, O.: Compiler Transformations for High-Performance Computing. ACM Computing Surveys 26(4), 345–420 (1994)
17. Parr, T., Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator. In: Programming Language Design and Implementation Conference (PLDI), pp. 425–436 (2011)
18. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 3rd edn., pp. 370–377. MIT Press (2009)
19. Hameurlain, A.: Evolution of Query Optimization Methods: From Centralized Database Systems to Data Grid Systems. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2009. LNCS, vol. 5690, pp. 460–470. Springer, Heidelberg (2009)
20. Evrendilke, C., Dogac, A., Nural, S., Ozcan, F.: Multidatabase query optimization. Journal of Distributed and Parallel Databases 5(1), 77–114 (1997)
21. Zeng, L., Benatllah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions On Software Engineering 30(5), 311–327 (2004)
22. Unify framework package, Software Languages Lab, Vrije Universiteit Brussel, `http://soft.vub.ac.be/svn-gen/unify/src/org/unify_framework/`
23. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM - 50th anniversary issue 51(1), 107–113 (2008)
24. Yuan, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P.K., Currey, J.: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI 2008 Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, pp. 1–14 (2008)