Adaptation of Legacy Fortran Applications to Cloud Computing

Eugene Tulika[1], Anatoliy Doroshenko[1], and Kostiantyn Zhereb[1]

[1] Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
eugene.tulika@gmail.com, doroshenkoanatoliy2@gmail.com,
zhereb@gmail.com

Abstract. We propose an approach to the semi-automatic transformation of legacy Fortran applications for execution on cloud computing platforms. An architecture is proposed based on web-services choreography, which allows unlimited scalability of the system and reduces overhead on message passing. The approach is tested on an example program from the quantum chemistry field.

Keywords. virtualization, cloud computing, scalable parallelism, web-services choreography.

Key Terms. ConcurrentComputation, DataGrid, HighPerformanceComputing, FormalMethod, SoftwareSystem

1 Introduction

Fortran language exists since the 1950s and during this time positioned itself as one of the best tools for scientific research. Language has huge support from the software industry, new libraries and compilers are released that allow using modern technologies and standards. Examples are: compiler from Portland Group [1] for Fortran supporting GPGPU and CUDA; High Performance Fortran Forum [2] creates compilers and standards for high performance Fortran; OpenMP and MPI for parallel and distributed computation allow to use Fortran for modern cluster computing; Intel popularizes Fortran [3] in order to support proprietary Intel compilers and programs; Coarray Fortran language fork became a standard in Fortran 2003 and allows to support computation on distributed arrays [4]; there are many Fortran libraries [5] for numerical analysis. Besides, conservative policy for backward compatibility makes code written on old standards compatible with the new compilers. All this makes Fortran an attractive platform for scientific research. However, Fortran suffers from outdated standards that make it hard to write an efficient code for computations with distributed memory. Another problem is a great amount of legacy code which was written without taking into account distributed architectures.

Cloud computing became popular because of the need to reduce the cost of computations. The difference between cloud computing and other parallel approaches is usage of scalability (ability to support more resources) vs. performance (running fast

ICTERI 2016, Kyiv, Ukraine, June 21-24, 2016

on given resources). Scalable systems can contain components which individually perform poorly. Cloud computing is focused on reducing the cost and time/performance balance using cheap components, and therefore allows using more resources.

This paper describes work in progress aimed at parallelizing Fortran scientific applications for the cloud platform. We propose a methodology for semi-automatic parallelization of legacy applications. The paper also describes an architecture for running distributed applications on the cloud.

Related parallelization approaches for porting legacy applications to cloud platform have been described, such as Pydron [6] and Bio-Cirrus [7]. The contribution of this paper is 1) using rewriting rules technique to automate transformation steps and 2) using service choreography instead of orchestration to reduce overhead.

2 Description of the Studied Application and Problem Statement

In this paper, we discuss our approach of transforming legacy scientific applications based on an example Fortran application from the quantum chemistry field. Some properties we discuss are specific to this application, such as time distribution of subroutines (Section 2) or the structure of control flow graph (Section 3). However, our approach can be applied in the more general case.

The example application calculates the geometry of electron orbitals [8]. The application was initially optimized for single-core performance, without any parallelism. The sequential processing time has quadratic increase depending on the input size.

The first step of our approach requires identifying the most promising subroutines for parallelization. The example program consists of data input, computation subroutines, logging intermediate results into the file and data output. Profiling the application before optimization shows the time distribution for the main computational steps:

- Data input from files and initialization – approximately 1% of time;
- Subroutine `hcore` calculates integrals for every atom – 60% of time;
- Subroutine `iterc` – optimization of the geometry of molecule, 30% of the time.

Therefore, the main focus of parallelization would be spent in subroutines `hcore` and `iterc`. Time performance of algorithms in both subroutines has a linear dependency on the input size. Each of two subroutines is applied independently for every atom in the input, which allows parallelization. However, `iterc` depends on the results of `hcore`, because the calculation of the Fock operator requires integrals computed in `hcore`.

The generic question of Fortran parallelization for distributed architecture is widely studied [16], [17]. As a part of this paper, Asynchronous Network technique with the usage of web-service choreography is applied to the parallel algorithms. This provides a robust framework for the development of applications which can be distributed across cloud infrastructure without additional efforts, while usage of MPI for the highly scalable applications requires manual control over data placement and inter-process communication.

3 Equivalent Code Transformation for Scalable Parallelization

On the next step, we try to find and parallelize independent loops that allow unlimited scalability. Structurally, the application can be modeled with the following graph:

$$I \text{ -> } A([a_{1..n}]) \text{ -> } B([b_{1..n}]) \text{ -> } O \quad (1)$$

Graph vertices *I*, *A*, *B*, *O* stand for the sequential steps of computation: *I* – data input and allocation of the memory, *A* – calculation of the integrals for every atom (`hcore`), *B* – calculation of the Fock operator for every atom (`iterc`), *O* – final steps and data output. Graph edges correspond to the control flow between parts of the program, $a_{1..n}$ and $b_{1..n}$ stand for input data of subroutines. The goal of this step is to transform program to the parallel form:

$$I \text{ -> } A_{1..n}([a_{1..n}]) \text{ -> } B_{1..n}([b_{1..n}]) \text{ -> } O \quad (2)$$

We transform sequential loop *A*, into *n* parallel processes $A_{1..n}([a_{1..n}])$. Each process A_i will perform calculations on a single segment of data a_i. The same transformation will be applied to *B* as well. After manual analysis of the data dependencies, it was identified that there are no dependencies between iterations of the loop in code fragment *A*, and processes $A_{1..n}$ can be invoked in parallel, same applicable to the processes $B_{1..n}$. However, *B* had a dependency on the results of *A* and in order to address this dependency barrier synchronization will be used between parallel processes A_i and B_i.

After selection of the model for parallel computation, we should ensure that code fragments invoked in parallel do not have any side effects. This is done by replacing subroutines with pure functions (introduced in Fortran 95 standard). The following changes in code are needed:

1. Create FUNCTION instead of SUBROUTINE
2. Remove IMPLICIT statements – all local variables of the function should be explicitly declared.
3. Remove COMMON BLOCKs (global variables) – all corresponding global variables should be passed as an inputs/outputs of the function, and reads/writes to global variables performed by the calling code.
4. Remove read and write operations – all such operations should be invoked before and after the call to the pure function.

4 Automatic Code Transformation Using TermWare

TermWare [9] is a tool for automatic code transformation which can be applied to the task of transformation to pure functions. TermWare allows describing transformations in a declarative way which simplifies its development and makes them reusable. In the earlier work [10] TermWare was used to build high-level algebraic models of Fortran code and perform transformations on them. This paper uses a similar approach but focuses on the transition of the subroutines to the pure functions.

As an example of transformation we use a set of rules to transform IMPLICIT statements to explicit declarations of variables:

```
1. _MarkPure(Subroutine($name,$params,$return,$body))
   -> Function($name,$params,$return,_MkImp($body))
2. _MkImp([$x:$y]) -> [_MkImp($x):_MkImp($y)]
3. _MkImp(NIL) -> NIL
4. _MkImp(Declare($var,$type,$val) ->
   Declare($var,$type,$val) [check($var, $type)]
5. [_MkImp(Assign($var,$expr)): $y] [isUnchecked
   ($var)] -> [Declare_MARK($var, $type) : [Assign
   ($var,$expr) : $y ]] [inferType ($var,$type)]
6. [$x:[Declare_MARK($var,$type) : $y]] ->
   [Declare_MARK($var,$type):[$x:$y]]
7. Function ($name,$params,$return,
   [Declare_ MARK($var,$type):$y]) -> Function
   ($name,$params,$return,[Declare($var,$type):$y])
```

Rule 1 triggers transformation, marking the body of the function with the marker term `_MkImp`. Rules 2 and 3 walk through the body of the function and expand the `_MkImp` marker to all operations. Rule 4 memorizes the variables which have explicit declaration using the method `check($var, $type)` from the facts DB. Rule 5 finds variables without explicit declaration using method `isUnchecked($var)`. For these variables it determines the type with the method `inferType($var,$type)` and adds declaration marked with `Declare_MARK($var,$type)`. To determine the variable type, the method checks the variable name against IMPLICIT statements in Fortran code, as well as default convention that declares variables starting with `'i'-'n'` as INTEGER and all others as REAL. Rule 6 moves this declaration to the beginning of the function, and rule 7 removes the mark. As a result, term `Declare($var,$type)` is generated, which later is transformed to the declaration of the variable in the code.

As an example of rules application, consider a simple procedure of square matrix multiplication (Table 1).

Table 1. Initial and transformed source code for removing IMPLICIT declarations

Initial code	Transformed code
`SUBROUTINE MATR_MULT(N,A,B,C)`	*`FUNCTION`* `MATR_MULT(N,A,B)`
`INTEGER,INTENT(IN) :: N`	`INTEGER,INTENT(IN) :: N`
`REAL*8,INTENT(IN)::A(N,N),B(N,N)`	`REAL*8,INTENT(IN)::A(N,N),B(N,N)`
`REAL*8,INTENT(OUT) :: C(N,N)`	`REAL*8 :: MATR_MULT(N,N)`
	`INTEGER :: I, J, K`
	`REAL*8 :: S`
`DO I=1,N`	`DO I=1,N`
`  DO J=1,N`	`  DO J=1,N`
`    S = 0.0D+00`	`    S = 0.0D+00`
`    DO K=1,N`	`    DO K=1,N`
`      S=S+A(I,K)*B(K,J)`	`      S=S+A(I,K)*B(K,J)`
`    END DO`	`    END DO`
`    C(I,J)=S`	`    MATR_MULT(I,J)=S`
`  END DO`	`  END DO`

```
END DO                                  END DO
```

In the initial code, some variables are used with no declaration. After transformation, all the variables are declared. Also, note that the syntax for SUBROUTINE is different from the FUNCTION. But such changes should not be described as additional rules. A simple substitution of the term “Subroutine” with the term “Function” is sufficient. During the code generation phase, all necessary changes are added automatically, which is one of the advantages of the TermWare and high-level algebraic models [10].

5 Transition to Distributed Application Executed on Cloud

Previously discussed transformation steps are not specific to any given parallel platform. Starting from this section, we discuss additional steps needed for the cloud platform. In order to transform application to distributed architecture, its source code has to be transformed to support network calls. Functions A_i and B_i are converted into web-services – separate programs with HTTP interface which can be invoked remotely. The body of the program is transformed into transaction script which invokes remote web-services and aggregates results. In order to use HTTP calls from the Fortran, *libcurl* library is used with the C interface [11]. Transformations of the functions to the separate web-services is done with the Java wrapper. Data for invocation of the remote services is composed in the transaction script. The script collects all input data of the program and sends messages to the remote services.

Cloud platform operation system, such as CloudStack, provides APIs to do scaling – provisioning of the nodes with the predefined configuration on demand. This capability is used by transaction script: after reading the input data and extracting the number of atoms, it makes a call to API of the cloud operating system and requires a startup of the necessary amount of nodes of needed type. In the simplest case, processing of N atoms will require $2N$ nodes, one for each A_i and B_i. However, the number of nodes, time they are actively working and the size of the used memory affects the cost of computation. For optimization of the cost the optimal parameters of configuration should be chosen, so that cost is minimized. In [12] a method of performance optimization of the service-oriented program was proposed based on load estimation. A similar approach can be used for minimization of the cost.

Approach when transaction script calls remote web-services in a service-oriented architecture is called Orchestration. Usage of Orchestration has disadvantages. Paper [13] shows that usage of separate transaction script increases the amount of calls between processes in most patterns of message passing in distributed systems, which increases overhead on data processing.

6 Transition to Choreography

Our goal is to reduce message passing overhead and eliminate a single point of failure represented by transaction script, by using choreography. From the perspective of distributed systems modeling, Choreography could be represented as an asynchronous

network [18]. The asynchronous network consists of the set of processes that communicate with each other. Communication can include: direct message exchange between nodes; broadcast when a node sends a message to each node including itself; multicast when message is sent to the subset of nodes. During the transition to choreography, transaction script is eliminated and its responsibilities are distributed between services. Unlike Orchestration or MPI where transaction script is waiting for the results of web-services execution, services in Choreography don't know about each other and send results of execution to the communication channel. During the execution, the service takes into account the state of the process and type of inbound message in order to determine its position relatively to other services. Having understood its position and taking into account communication protocol, service decides on what type of message should be sent back to the channel when the result is ready. The protocol has the following format: "If current process role is A_i and message M_1 is received as input, procedure F should be invoked and message M_2 should be passed to the processes $A_{2..m}$". This format is identical to the description of finite state machine.

Communication Protocol can replace the part of transaction script responsible for service invocation and results processing. Part of responsibilities related to data input is transferred to web-service itself. Code fragment I performing initial data processing is implemented by the new service I_1 which plays the role of starting point in the machine description. Invocation of tail fragment O which outputs the result should be implemented by a separate service O_1, which will be executed after synchronizing all the processes $B_{1..n}$ and will do the post-processing of the data and data output.

In order to follow the protocol, every web-service should have a controlling mechanism which executes state machine by the description of the protocol. Controlling mechanism is written in Java. Choreography execution is initiated by an initial event, e.g. the file with input data has been uploaded to the cloud data volume. Process I_1 in the waiting state receives this event from the file system and invokes Fortran code responsible for initial data processing. After the event is processed, process I_1 sends broadcast message with the results of invocation to all the processes of the asynchronous network. In the simplest case, this message is received by all the processes $A_{1..n}$, $B_{1..n}$, I_1, and O_1. The processes $B_{1..n}$, O_1, and I_1 ignore this message because B_i and O_1 processes do not have enough information yet to start, and process I_1 already finished its part of the work. It means that only processes $A_{1..n}$ start execution.

An important aspect of choreography is the implementation of barrier synchronization. In [15] global and local synchronizers are described that can be used for synchronization of the processes of the asynchronous network. If global synchronizer is used, then the separate process should be responsible for controlling the conditions of the barrier. The local synchronizer is controlling just its neighbors. In this case message "process reached barrier" is gradually distributed through the network which allows reducing the amount of interactions between services.

In order to achieve better resource utilization using Choreography, following considerations should be taken into account. If certain web-service, such as I_1, is expected to consume fewer resources than others, it should run on the smaller node (with less RAM and CPUs). If some web-services are never expected to run simultaneously, such as A_i and B_i, they should re-use the same node. Ideally, it should be the same service with a same controlling mechanism which can play different roles in the protocol. This will guarantee that minimal amount of nodes are started at any point of

process execution. Choreography allows reducing the number of messages passed between transaction script and web-services. It also allows better resource utilization having non-blocking requests between web-services. Besides, it provides integration framework with the generic rule sets defined which reduce the amount of code needed to be written to convert the application into distributed system.

7 Testing of the Approach

In order to verify the proposed approach, we used a simplified task with the same model of the data dependency – Gaussian elimination. Testing was performed on the Amazon cloud platform and compares two different configurations of the system. Both configurations have the same amount of processors – 8, and the same amount of RAM – 32Gb. The first configuration consists from one server AWS m4.2xlarge (26 ECUs, 8 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 32 GiB memory, EBS only) and is used to run the sequential program. Second configuration consists of four servers AWS m4.large (6.5 ECUs, 2 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 8 GiB memory, EBS only) and it is used to run the service-oriented application (with choreography). We have measured the time spent on the processing of the square matrices of different sizes. Comparison of execution time is in Fig. 1. For smaller matrix size, the sequential program runs faster because of overhead in a parallel program. However, for larger matrices the execution time of sequential program grows faster, and it becomes less efficient.

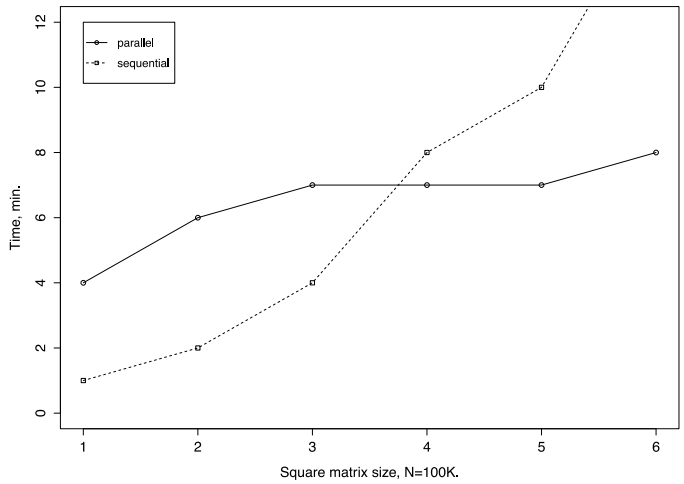

Fig. 1. Comparison of the execution time of sequential and service-oriented versions

8 Conclusion

The paper describes the work in progress of scaling legacy Fortran code using cloud platforms. Proposed architecture uses choreography of web-services which allows

unlimited scalability and reduces overhead on message passing. Scaling exercise is performed on application from quantum chemistry field for calculation of atoms orbitals. One of the main results of the paper is a methodology for adjustment of the legacy source code to the cloud infrastructure, including transition steps to distributed scalable architecture. Our future research directions include automating additional transformation steps using TermWare framework, applying our approach to different applications, as well as testing different cloud configurations to find the most efficient ways of parallelizing legacy applications.

References

1. PGI Compilers & Tools, http://www.pgroup.com/products/pvf.htm
2. High Performance, http://hpff.rice.edu
3. Fortran is more popular than ever; Intel makes it fast, https://software.intel.com/en-us/blogs/2011/09/24/fortran-is-more-popular-than-ever-intel-makes-it-fast
4. Coarrays in the next Fortran Standard, ftp://ftp.nag.co.uk/sc22wg5/N1751-N1800/N1787.pdf
5. Netlib Repository, http://netlib.org
6. Müller, S. C., Alonso, G., Amara, A., and Csillaghy, A.: Pydron: Semi-automatic parallelization for multi-core and the cloud. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 645-659.(2014)
7. Karlsson, T. J. M., et al.: Bio-Cirrus: a framework for running legacy bioinformatics applications with cloud computing resources. In Advances in Computational Intelligence, pp. 200-207. Springer Berlin Heidelberg. (2013)
8. Doroshenko, A., Khavryuchenko, V., Iegorov, V., Suslova, L.: Modeling for quantum chemistry computations (in Russian). Upravliayushchie sistemy i mashiny N 5, pp. 83-87. (2012)
9. Doroshenko A., Shevchenko R.: A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae 72 (1–3), pp. 95–108. (2006)
10. Tulika, E., Zhereb, K., Doroshenko, A.: Fortran Programs Parallelization Using Rewriting Rules Technique (in Ukrainian). Problems in Programming N 2-3, pp. 388-397. Kyiv (2012)
11. Libcurl – the multiprotocol file transfer library, http://curl.haxx.se/libcurl
12. Tulika, E.: Performance Optimization in SOA Using Load Estimation and Load Balancing (in Ukrainian). Problems in Programming, N 2-3, pp. 193-201. Kyiv (2010)
13. Barker, A., Weissman, J.B. and Van Hemert, J.I..: Reducing data transfer in service-oriented architectures: The circulate approach. Services Computing, IEEE Transactions on Services Computing, N 5(3), pp. 437-449. (2012)
14. Peltz, C.: Web services orchestration and choreography. Computer, (10), pp. 46-52. (2003)
15. Lynch, N.A.: Distributed algorithms. Morgan Kaufmann, San Francisco (1996)
16. Golub G., Ortega J.: Scientific Computing: An Introduction with Parallel Computing, 215-236. Academic Press (1989)
17. Schelter S., Boden C., Schenck M., Alexandrov A., Markl V.: Distributed matrix factorization with MapReduce using a series of broadcast-joins. In Proceedings of the 7th ACM conference on Recommender systems (RecSys '13). ACM, New York, NY, USA, pp. 281-284. (2013)
18. Barker A., Walton C., Robertson D.: Choreographing Web Services. IEEE Transactions on Services Computing, N 2(2), pp. 152-166. IEEE Computer Society. (2009).