Distributed Optimization on Super Computers: Case Study on Software Architecture Optimization Framework

Ramin Etemaadi
Leiden Institute of Advanced Computer Science
Faculty of Science, Leiden University
Leiden, the Netherlands
etemaadi@liacs.nl

Michel R.V. Chaudron
Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
chaudron@chalmers.se

ABSTRACT

Nowadays advanced software systems need to satisfy large number of quality attributes at the same time. It is a very complex optimization problem which software architects must address. Evolutionary algorithms can help architects to find optimal solutions which meet these conflicting quality attributes. However, these metaheuristic approaches in multiobjective problems especially for high dimensions mostly take so long time to be executed. One of the best solutions to speed up this process is distributing execution of evolutionary algorithm on multiple nodes of a super computer or on the cloud.

This paper presents the results of distributed execution of evolutionary algorithm for multiobjective optimization of software architecture. We report implementation of two different ways for distributed execution of evolutionary algorithm: (1) Actor-based approach, (2) MapReduce approach. The case study in this experiment is an industrial software system which is derived from a real world automotive embedded system.

The results of this computationally-intensive experiment on a super computer give us 81.27% parallelization efficiency for distribution among 5 nodes.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.11 [**Software Engineering**]: Software Architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Distributed Execution; MapReduce Paradigm; Actor-based Distribution; Metaheuristic Optimization; Evolutionary Multiobjective Optimization Algorithms (EMOA); Software Architecture Design Optimization; Component-Based Software Engineering (CBSE)

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2598394.2605686.

1. INTRODUCTION

The architecture has deep impact on non-functional properties of a system such as performance, safety, reliability, security, energy consumption and cost. Due to the complexity of today's software systems, designing a system which meets all its quality requirements becomes increasingly complex. Hence, system architects have to employ optimization techniques to be able to explore more design possibilities and to find optimal architectural solutions. Metaheuristic approaches frame the challenge of designing architectures as an optimization problem and iteratively try to improve a candidate solution with regard to the given quality attributes. Evolutionary Algorithms (EA), as a well-known metaheuristic approach, is a common optimization technique for solving system architectural problems. However, evolutionary algorithms are generally slow especially when evaluation function is computationally-intensive and slow. It takes so long for an evolutionary algorithm to find near optimal solutions, therefore techniques are needed to speed up this process. Distributing the optimization process is one of the best solutions to achieve this goal.

This paper shows the results of distributed execution of evolutionary algorithm for multiobjective optimization of software architecture. For distributed execution of evolutionary algorithm, we have implemented two approaches: (1) Actor-based approach, (2) MapReduce approach. To execute our distributed implementation on a super computer, we have used DAS-4 (The Distributed ASCI Supercomputer 4) [16] to setup a computationally-intensive experiment. Our distributed implementation have been applied to a case study derived from a real world automotive system.

The paper is organized as following: Firstly, Section 2 describes our optimization framework, and its detailed modeling, optimization and evaluation parts. Then, Section 3 discusses our two proposed approaches for distributed execution. We have applied our approach on a case study, which is described in Section 4. Finally, the paper concludes in Section 5.

2. AQOSA FRAMEWORK

In this section, we present the AQOSA framework which our new degrees of freedom is implemented on. First, we start with general process of AQOSA framework in Section 2.1. Section 2.2 describes AQOSA modeling. Section 2.3 describes the optimization process in detail. Then, the evaluator evaluates each architecture for various quality attributes. Section 2.4 describes the used evaluation techniques.

2.1 Process

AQOSA is a framework which uses genetic algorithm optimization approach for automated software architecture design. The framework supports analysis and optimization of multiple quality attributes including response time, processor utilization, bus utilization, safety and cost. Figure 1 shows the architecture of the AQOSA framework. It uses an architectural Intermediate Representation (IR) model for describing the architectural design problem. The tool takes the following as input: i) an initial functional part of the system (i.e. components that provide the needed functionality and their interactions with other components), ii) a set of typical usage scenarios (includes trigger to create workloads), iii) an objective function (implying which architecture properties should be optimized), iv) a repository that contains a set of specifications of hardware and software components.

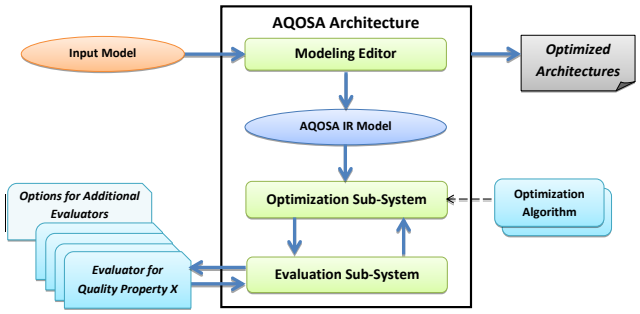

Figure 1: AQOSA overall architecture.

Then, AQOSA iterates through the following steps:

1. Generate a new set of candidate architecture solutions: To this end, AQOSA uses a representation of the architecture where it knows which are the degrees of freedom in the design and how to generate alternative architecture instances.
2. Evaluate the new set of candidate architecture solutions for multiple quality properties: This works by generating analysis models from the architecture model using model transformations and then analyzing these models.
3. Select a set of (so far) Pareto-optimal solutions.
4. Iterate to step 1 until some stopping criterion holds. This can be a maximum number of generations or a criterion on the objective function.

2.2 Architecture Modeling

Because AQOSA is designed to optimize architectures in a wide range of domains, it aims to be independent from specific modeling languages. Hence, it uses its own internal architecture representation, AQOSA intermediate representation (AQOSA-IR). AQOSA architecture modeling is based on an Eclipse EMF [5] model. AQOSA-IR model integrates multiple quality modeling perspectives (such as performance, safety, etc.) for the architectural level optimization purpose.

AQOSA-IR consists of four major parts: Assembly, Scenarios, Repository and Objectives.

- Assembly: It includes software components and their assembly for delivering system functionalities. Every component provides some services, and interaction between different components are defined by flows and actions within flows.
- Scenarios: It defines expected scenarios for the system. So, the architect can define best-case, worse-case or normal-case for the system. It stores real-time constrains of the system such as expected completion time and deadline.
- Repository: It stores various possible choices for processors, buses and component implementations. Based on this repository, the AQOSA framework is able to change assigned processor for each node, or assigned bus instance for each bus line, etc. It contains required specifications of each possible hardware or software.
- Objectives: It defines the objectives which the framework optimizes.

A sample of AQOSA-IR model is presented in [8].

2.3 Architecture Optimization

The AQOSA optimizer tries to optimize software architecture with respect to potentially contradicting quality attributes based on evolutionary algorithms. In general, various global optimization techniques have been used in handling complex engineering problems. Younis et al. [18] compared variety of optimization methods and revealed the pros and cons of these global optimization methods. For AQOSA framework, we have chosen Genetic Algorithms (GA) for optimization method.

In the following, first we describe compatible evolutionary algorithms with AQOSA framework. After that, we discuss implemented degrees of freedom for our architecture optimizer. And then, we detail how we represent software architecture in the form of a genotype. This shows how we would be able to generate various architectures.

2.3.1 Evolutionary Algorithms

Because of conflicts between different quality attributes in the software architecture, AQOSA uses Evolutionary Multi-Objective Algorithms (EMOA) to improve the architecture. It has been implemented based on the Opt4J optimization framework [14]. The system designer can choose one of the following GA algorithms for his design problem:

- NSGA-II (non-dominated sorting based multi-objective evolutionary algorithm): It is one of the most widely used EMOA techniques and has been proposed by Deb [4].
- SPEA2 (an improved version of Strength Pareto Evolutionary Algorithm): It has been suggested by Zitzler and Thiele [19], and it is also widely used.
- SMS-EMOA (S-Metric Selection Evolutionary Multi-Objective Algorithm): It has been proposed by Emmerich, Beume and Naujoks [6]. It is a representative of the class of hypervolume-based EMOA, which recently gained popularity in the EMOA field.

A comparison of EMOA algorithms for software architecture domain is discussed in our earlier work [13].

2.3.2 Degrees of Freedom

When an architect finalizes an architectural design for a system, generally there are still some ways in which the solution can be varied without changing the functionality. We call them *Degrees of Freedom*(DoF). The component-based paradigm that underlies our approach, allows us to recompose components in different topologies and wrappers. However, we only consider variations of architectural designs that do not modify the interfaces used in the architecture in order to guarantee that our optimization process does not change the functionality of the system.

In the following, we describe the degrees of freedom supported by AQOSA framework:

Number of hardware nodes .
If an architectural model contains `n` software components, then these can be deployed on a number of hardware nodes, ranging between a minimum of `1` and a maximum of `K·n` hardware nodes (for some natural number `K > 0`), because the number of nodes in an architecture is finite. Adding more hardware nodes may provide more processing capacity and therefore give better performance. Vice versa, removing hardware nodes may reduce the total cost of the system.

Number of connections between hardware nodes .
If `n` hardware nodes are chosen for the deployment of components, then the maximum number of connections between hardware nodes can be calculated by $Max_c(n) = \frac{n(n-1)}{2}$. This maximum represents a case that all of the nodes are connected 1-by-1 together by a dedicated connection. As a minimum we can assume a single central bus which connects all of the nodes. This DoF has significant impact on performance and cost of the system.

Network topology.
By definition, network topology is the layout of interconnections between hardware nodes. Figure 2 shows some possible topologies for connecting a network with 4 nodes. Even with the same number of nodes and the same number of connections, network topologies might be different. The impact and importance of this DoF is discussed in authors' previous work [7]. Obviously, not all possible topologies are valid and therefore AQOSA performs a validation process before evaluation.

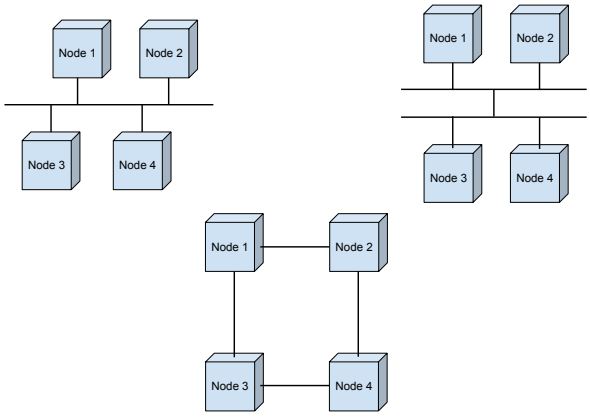

Figure 2: Three possible topologies for a 4-node network.

Software on hardware allocation .
Given a hardware network topology, allocation of software components on hardware nodes is another degree of freedom. This degree of freedom also has large effect on the processors utilization and performance of the whole system.

Software components replacement .
Different components (e.g. developed by different vendors) that implement the same functionality are considered different architectural alternatives. Our assumption is that one component can replace another if and only if they implement the same functionality. If not, the replacement violates the system functionality which is not accepted.

Processors replacement .
This degree of freedom entails that each hardware node can be replaced by another hardware node (processor) in the repository. These processors may be different in processing clock rate, energy consumption, failure probability range, and cost. It has large effect on all quality attributes.

Buses replacement .
This degree of freedom is similar to the hardware components replacement, but is aimed at replacement of buses. They may be different in communication bandwidth, communication delay, failure probability range, and cost as well.

2.3.3 Architecture Genotype

Inspired by the biological concept of genotypes, in genetic programming and evolutionary algorithms, the genotype is the genetic representation of an *individual* which in this case is an architecture design. AQOSA framework used a genotype which consists of a set of 5 genomes.

The deployment genome encodes that each software component is deployed on which hardware node. The framework calculates the number of nodes based on deployment of software components and then stores it in the nodes genome. The nodes genome encodes which hardware specification is chosen for each node in the system. The specification for each hardware node includes processing clock rate, range of failure probability, and cost. The communication genome, like the nodes genome, encodes which hardware specification is chosen for each communication line. Their specification includes bandwidth, transmit delay, and cost. Topology is represented by two Boolean matrices which we call it connection genomes. These matrices show the connections between buses to nodes and amongst buses themselves. Each cell in the matrix can have value True or False, where True means this particular node and particular line are connected (or that these particular lines are connected).

Genotype Validation.
Evolutionary algorithms in AQOSA can apply various genetic operators such as *Copy*, *Mutate* or *Crossover* to the genotype. However there is no guarantee that valid architectures will be generated as offspring. Therefore, it is necessary to validate the offspring. In this process the optimizer performs sanity checks for each genotype in the generated offspring. For example, if the model defines that *component3* should communicate with *component5* and then the generated offspring deploys *component3* on *node2* and *component5* on *node4*, this process should check whether there are any communication paths between *node2* and *node4*.

2.4 Architecture Evaluation

The AQOSA evaluation sub-system gets an evaluation model (e.g. Queuing Networks and Fault Tree) which is transformed from a combination of the AQOSA IR and a genotype (which is described in Section 2.3.3) for a particular quality attribute (e.g. Response Time or Safety). It feeds evaluation models to each evaluator and returns the results to the optimization sub-system.

It should be pointed out that our framework is aimed at optimizing architectures. So, from optimization point of view, the framework is targeted for supporting relative comparisons between architectural solutions. For this purpose, AQOSA uses analysis tools for particular quality attributes as plug-ins. We assume that these analysis tools are developed and validated by domain experts. However, to examine the accuracy of AQOSA evaluation implementation, we have compared our tool results with results reported in relevant publications. In case of performance attributes, our QN implementation can achieve the same results as the results published in [17]. In case of FTA, we compared our results with the results published in [10].

In the following, we describe the supported quality attributes by the AQOSA framework.

Response Time.

Response time refers to a time interval during which the response to an event must be executed. The time interval defines a response window given by a starting time and an ending time. These can either be specified as absolute times (time of day, for example) or offsets from an event which occurred at some specified time [2]. The ending time is also known as a deadline. AQOSA measures response time for each event in the system as offsets from the request time, and then scores them based on predefined deadlines. More technically, within the QN simulation process, it generates scenario events as the queuing network inputs and records the response time of outputs for the events.

Processor Utilization.

Processor utilization is the percentage of time which a resource is busy. AQOSA measures this percentage for each processor in the architecture separately. To do so, it translates the architecture to the queuing network. After the simulation of the network, this quality attribute will be retrieved from the network statistics. Regarding the final measure for this metric, AQOSA can be configured to return either average, minimum, or maximum of processors utilizations in the network.

Bus Utilization.

Like processor utilization, bus utilization is the percentage of bandwidth that a bus uses. Similar to above, AQOSA measures this metric for each bus in the architecture separately. Again, this metric can be configured to return either average, minimum or maximum of buses utilizations.

System Safety.

Förster [11] claims: *"Software does not fail randomly but will invariably fail again in the same way under the same conditions. While for mass-produced hardware parts it is possible to assign a failure probability, for software a similar assumption does not seem entirely realistic"*. Accepting this hypothesis, we assume for each component that the output fails if either the input fails or the hosting hardware crashes. So, AQOSA analyzes the corresponding fault tree for each system output and calculates the failure probability of each output based on its fault tree dependability on various inputs and related hardware nodes.

System Cost.

The cost quality attribute is important from a market point of view. Fortunately, it is easily calculated by adding the cost of used software components, hardware nodes, and communication lines.

3. PARALLEL EXECUTION

In the following, we discuss two approaches for parallel execution of evolutionary algorithms which we implemented in our framework:

3.1 Actor-based Distribution (Akka Framework [1])

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined by Carl Hewitt [12] but have been popularized by the Erlang language.

Akka is a framework which helps developers in writing correct concurrent, fault-tolerant and scalable applications. Akka by using the Actor Model raises the abstraction level and provides a better platform to build correct concurrent and scalable applications. Actors also provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

Actors give developers:

1. simple and high-level abstractions for concurrency and parallelism,
2. asynchronous, non-blocking and highly performant event-driven programming model,
3. very lightweight event-driven processes.

Fault tolerance through supervisor hierarchies with "let-it-crash" semantics. Excellent for writing highly fault-tolerant systems that never stop, systems that self-heal. Supervisor hierarchies can span over multiple JVMs to provide truly fault-tolerant systems.

3.2 MapReduce Paradigm (Hadoop Framework [15])

MapReduce was firstly introduced by Dean et al. in [3]. MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. MapReduce programs are written in a particular style influenced by functional programming constructs, specifically idioms for processing lists of data. This requires dividing the workload across a large number of machines. This model would not scale to large clusters if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale. Users

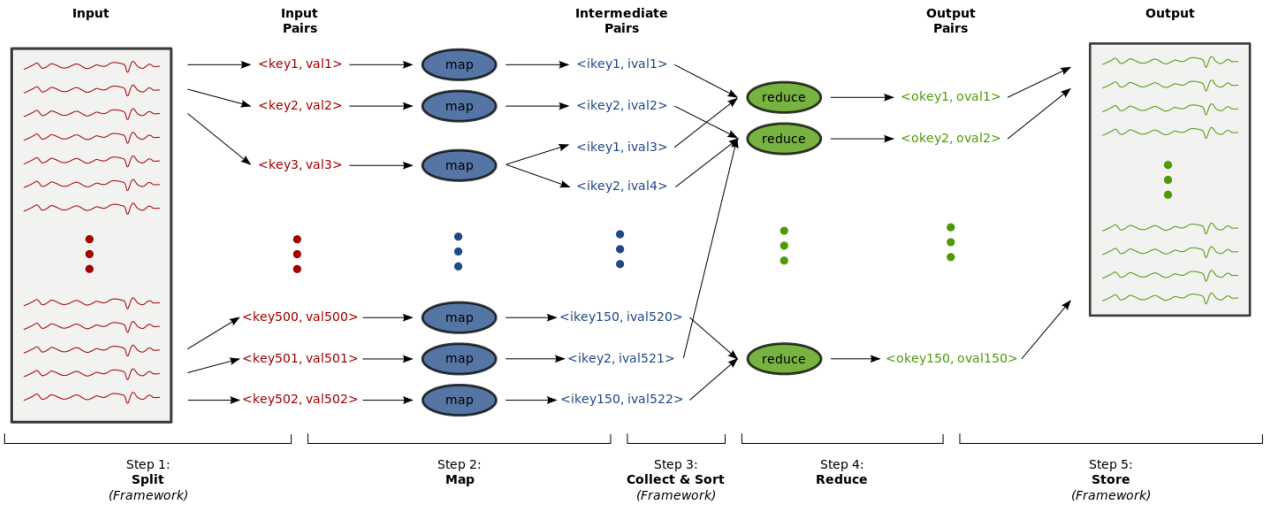

Figure 3: A visualization of Map and Reduce processes.

specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines.

The Apache Hadoop software library is a framework that implemented MapReduce programming model. This framework allows for the distributed processing of large data sets across clusters of computers using MapReduce paradigm. Conceptually, MapReduce programs transform lists of input data elements into lists of output data elements. A MapReduce program will do this twice, using two different list processing idioms: *map*, and *reduce*. The Hadoop MapReduce framework takes these concepts and uses them to process large volumes of information. Figure 3 shows a visualization of this process.

Mapping List.

The first phase of a MapReduce program is called mapping. A list of data elements are provided, one at a time, to a function called the Mapper, which transforms each element individually to an output data element. As an example of the utility of map: Suppose you had a function toUpper(str) which returns an uppercase version of the input string. You could use this function with map to turn a list of strings into a list of uppercase strings. Note that we are not modifying the input string, we are returning a new string that will form part of a new output list.

Reducing List.

Reducing lets you aggregate values together. A reducer function receives an iterator of input values from an input list. It then combines these values together, returning a single output value. Reducing is often used to produce "summary" data, turning a large volume of data into a smaller summary of itself. For example, "+" can be used as a reducing function, to return the sum of a list of input values.

4. CASE STUDY

4.1 Implemented Actor-based Distribution

Figure 4 depicts scheme of our implemented Akka actors. We used 5 nodes, 1 master node and 4 worker nodes. On each worker node, Akka framework was set to be initialized which run 4 actors on ports 2551, 2552, 2553 and 2554 respectively. Therefore, 16 actors in total were set to be initialized. They were responsible for evaluating an individual based on requested software quality attributes, such as response time, processor utilization, bus utilization, safety and cost. On the master node, Akka framework was set to start an actor called 'Evaluator Balancer' (as depicted in Figure 4). This actor is responsible for distributing evaluation jobs to other 16 worker actors. It used a round-robin balancing for the for distribution of assigned jobs. AQOSA framework also started on master node and it was calling balancer actor whenever it wanted to evaluate an individual.

This experiment has been run on DAS-4 [16] super computer that every node is a powerful computer with a 8-core (each core runs 2.67GHz and 12MB cache) processor and 48GB memory.

4.2 Automotive Subsystem

To examine the efficiency of actor-based distributed implementation of our software architecture optimization framework, we applied it to a real case study from automotive industry. The case study was conducted at Saab Automobile AB and has been reported in the previous authors' work [9]. The system represents the Saab 9-5 Instrument Cluster Module ECU (Electronic Control Unit, a node in a network) and the surrounding sub-systems. It consists of 18 components as depicted in Figure 5. The Instrument Cluster Module is responsible for 8 concurrent user functions. Hence, for providing these functionalities, it should be able to response 6 sporadic tasks and 4 periodic tasks concurrently. Details of these tasks have been reported in [9] and [8].

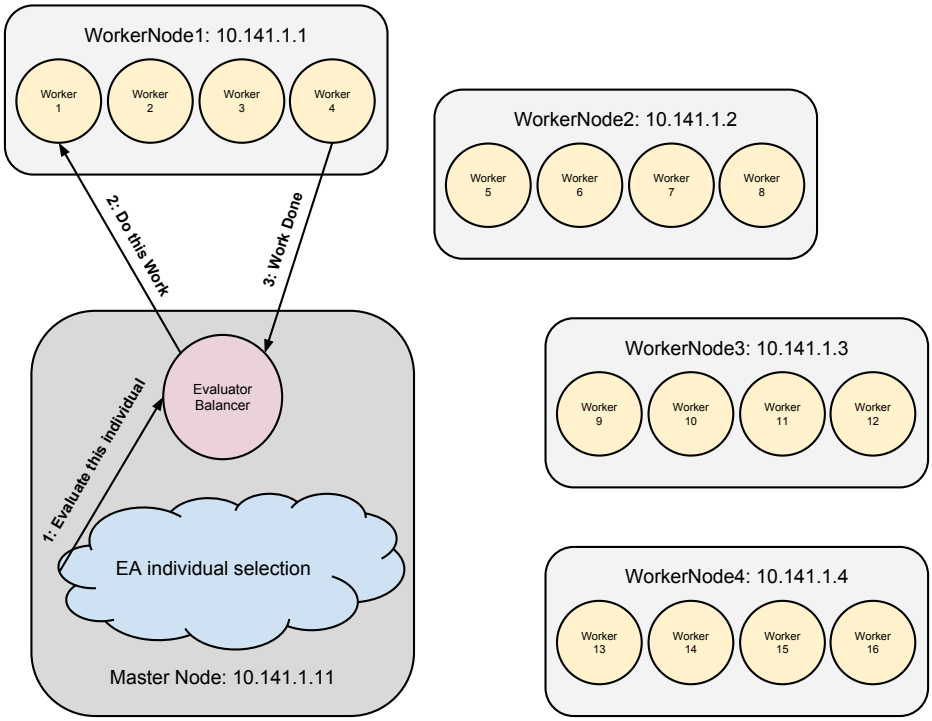

Figure 4: AQOSA implementation of actor-based distribution scheme.

For generating new architectural solutions, the repository of hardware components contains these elements:

- 28 Processors: ranging over 14 various processing speeds from 66MHz to 500MHz; Each has two levels of failure rate. A processor is more expensive if it has less chance of failure and vice versa.
- 4 Buses: with bandwidths of 10, 33, 125, and 500 kbps, and latencies of 50, 16, 8, and 2 ms. A bus is more expensive if it supports higher bandwidth.

As an estimate of the size of the design space in this case study, consider the following reasoning: Assume we omit architecture topology changing and fix an architecture with six processors and three bus lines for their interconnections (exactly like the current realization in the industry). For these constraints there are $28^6 \cdot 4^3$ different possibilities, which is more than 30 billion architectures. When also considering variations in the architecture topologies, this number would be even considerably higher.

After defining the above hardware options, AQOSA was run 30 times based on NSGA-II algorithm with these adopted parameter settings: initial population size(α) = 256, parent population size (μ) = 64, number of offspring(λ) = 64, archive size = 32, number of generations = 60, crossover rate set to 0.95, and all quality attributes are aimed to be minimized.

4.3 Results

Table 1 shows the execution times (in milliseconds) of 30 runs of the experiment. The first column is the execution number. The second column is the execution times of actor-based distributed implementation. As described in Section 4.1, it was distributed on 1 master node and 4 worker nodes. The third column shows the execution times of running the same design problem on single node.

In parallel computing speedup parameter has been defined by the following formula:

$$S_p = \frac{T_1}{T_p}. \tag{1}$$

where p is the number of processors, T_1 is the execution time of the sequential algorithm, and T_p is the execution time of the parallel algorithm with p processors. Therefore, in our experiment speedup for the average of 30 times run is equal to:

$$S_5 = \frac{684,116}{168,353} = 4.0635 \tag{2}$$

Moreover, efficiency of parallel algorithm has been defined by this formula:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}. \tag{3}$$

This means when running an algorithm with linear speedup, doubling the number of processors doubles the speed. However, this case is ideal situation and it is considered ideal scalability.

To calculate the efficiency of our implemented actor-based distributed algorithm, we apply the aforementioned formula, and therefore:

$$E_5 = \frac{S_5}{5} = \frac{4.0635}{5} = 0.8127 \tag{4}$$

In other words, our implemented actor-based distributed algorithm in case of this experiment on a real-world case study shows 81.27% efficiency which is acceptable metric.

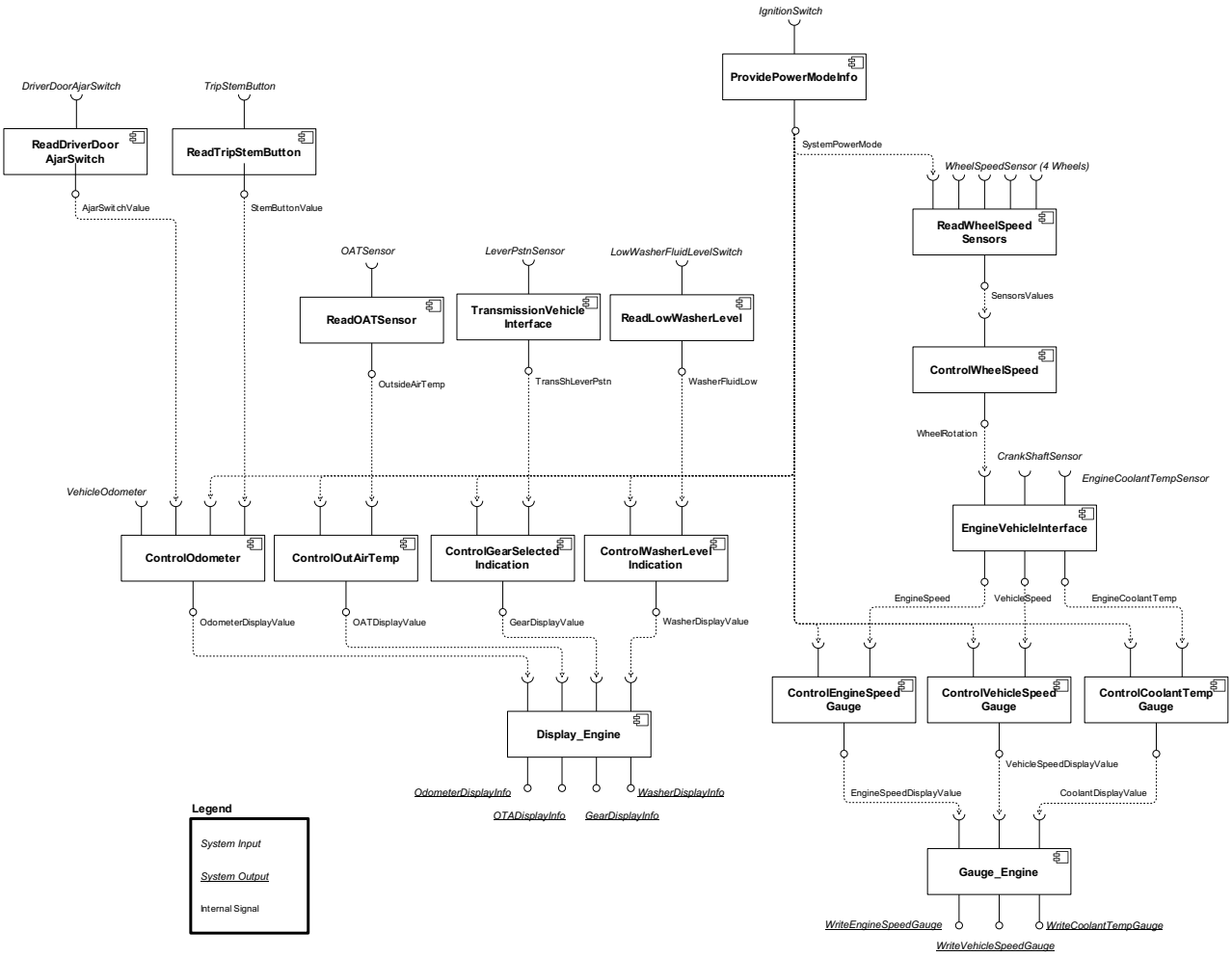

Figure 5: Component diagram of Automotive Instrument Cluster system.

4.4 Actor-based vs. MapReduce Approach

To compare two different distribution approaches which are described in Section 3, we set another experiment to run both approaches for the same case study. The case study is the same real-world automotive system which is already discussed in previous section.

In this experiment, we employed cluster of 16 nodes from DAS-4. 8 nodes for running actor-based distribution and 8 nodes for running MapReduce distribution. In this experiment, every node had a dual Intel® Xeon® Processor E5-2620 (15M Cache, 2.00 GHz) as processor and 64GB of memory. Both implementations only distribute the task for evaluation of an individual solution.

We started both implementations at the same time. The results show by the time 30 executions were completed using actor-based approach, MapReduce program managed to finish only 10 executions. Therefore, we can conclude that actor-based approach is faster for evolutionary algorithms, at least for this case study.

One explanation for this behaviour can be the number of nodes in the cluster. MapReduce approach probably needs large-size cluster of nodes to achieve its performance. Unfortunately, we did not have access to Hadoop cluster larger than 8 nodes to test this hypothesis.

5. CONCLUSION

In this paper we presented the results of a software architecture optimization experiment which calculates the efficiency of parallel execution for evolutionary algorithm. We defined this experiment based on a real world case study and we applied it for a software architecture optimization problem with five objectives. We showed that parallel execution of evolutionary algorithm for software architecture optimization can improve execution time significantly with acceptable efficiency in multiobjective optimization context.

The results showed that for cases in which evaluation process is significant compare to updating Pareto front (selection process), efficiency of parallelization is considerable. However, for cases in which evaluation process is fast, parallelization would not help considerably. Comparing actor-based distribution and MapReduce distribution, in our case study, shows that actor-based distribution performs faster.

As the future work, it is interesting to extend the parallelization of execution to the selection process of evolutionary algorithm as well. In this way, in addition to evaluation process, selection algorithm would also execute in parallel which helps efficiency of parallelization even more. This should be implemented separately for each selection algorithm (such as NSGA-II, SPEA2, etc).

Run #	*Distributed (1 Master-Node + 4 Worker-Nodes)*	*Single-Node*
1	168,346	685,668
2	163,278	691,741
3	171,185	687,933
4	191,425	678,725
5	173,486	683,875
6	212,667	697,605
7	185,926	681,893
8	169,970	689,065
9	135,545	695,583
10	162,341	687,381
11	176,953	693,289
12	164,833	689,954
13	153,570	681,492
14	184,530	692,063
15	148,388	655,434
16	169,537	669,622
17	166,597	686,289
18	212,164	676,475
19	158,257	676,324
20	163,089	684,778
21	170,300	677,652
22	138,852	684,308
23	169,592	691,148
24	155,911	671,799
25	166,818	692,061
26	182,757	680,180
27	143,056	689,571
28	161,778	690,718
29	158,040	681,744
30	171,396	679,107
Average	168,353	684,116

Table 1: Execution time (in ms) of 30 runs of experiment

6. ACKNOWLEDGMENTS

This work has been supported by the Netherlands national project OMECA (Optimization of Modular Embedded Computer-vision Architectures).

This research has been executed on Dutch super computer DAS-4 (The Distributed ASCI Supercomputer 4). DAS-4 is a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imaging (ASCI).

7. REFERENCES

[1] Akka Framework: an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. http://akka.io/.

[2] M. Barbacci, M. H. Klein, T. A. Longstaff, and C. B. Weinstock. Quality Attributes. Technical Report CMU/SEI-95-TR-021, Carnegie Mellon, 1995.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[4] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[5] Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/.

[6] M. Emmerich, N. Beume, and B. Naujoks. An EMO Algorithm Using the Hypervolume Measure as Selection Criterion. In C. A. C. Coello, A. H. Aguirre, and E. Zitzler, editors, *EMO*, volume 3410 of *LNCS*, pages 62–76. Springer, 2005.

[7] R. Etemaadi and M. R. V. Chaudron. Varying topology of component-based system architectures using metaheuristic optimization. In V. Cortellessa, H. Muccini, and O. Demirörs, editors, *EUROMICRO-SEAA*, pages 63–70. IEEE Computer Society, 2012.

[8] R. Etemaadi, K. Lind, R. Heldal, and M. R. V. Chaudron. Details of an automotive sub-system: Saab instrument cluster module. *CoRR*, abs/1306.0555, 2013.

[9] R. Etemaadi, K. Lind, R. Heldal, and M. R. V. Chaudron. Quality-driven optimization of system architecture: Industrial case study on an automotive sub-system. *Journal of Systems and Software*, 86(10):2559–2573, 2013.

[10] M. Förster and D. Schneider. Flexible, Any-Time Fault Tree Analysis with Component Logic Models. In *ISSRE*, pages 51–60. IEEE Computer Society, 2010.

[11] M. Förster and M. Trapp. Fault Tree Analysis of Software-Controlled Component Systems Based on Second-Order Probabilities. In *ISSRE*, pages 146–154. IEEE Computer Society, 2009.

[12] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In N. J. Nilsson, editor, *IJCAI*, pages 235–245. William Kaufmann, 1973.

[13] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron. An Evolutionary Multiobjective Optimization Approach to Component-Based Software Architecture Design. In *IEEE CEC*, pages 432–439. IEEE, 2011.

[14] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J: a modular framework for meta-heuristic optimization. In N. Krasnogor and P. L. Lanzi, editors, *GECCO*, pages 1723–1730. ACM, 2011.

[15] The Apache™ Hadoop® project. http://hadoop.apache.org/.

[16] The Distributed ASCI Supercomputer 4. http://www.cs.vu.nl/das4/.

[17] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

[18] A. Younis and Z. Dong. Trends, features, and tests of common and recently introduced global optimization methods. *Engineering Optimization*, 42(8):691–718, 2010.

[19] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. Technical report, ETH Zurich, 2002.