International Journal of Computer Applications Technology and Research
Volume 4– Issue 9, 668 - 672, 2015, ISSN: 2319–8656

Improved Strategy for Distributed Processing and Network Application Development

Eke B. O.
Department of Computer Science,
University of Port Harcourt
Port Harcourt, Nigeria

Onuodu F. E.
Department of Computer Science,
University of Port Harcourt
Port Harcourt, Nigeria

Abstract: The complexity of software development abstraction and the new development in multi-core computers have shifted the burden of distributed software performance from network and chip designers to software architectures and developers. We need to look at software development strategies that will integrate parallelization of code, concurrency factors, multithreading, distributed resources allocation and distributed processing. In this paper, a new software development strategy that integrates these factors is further experimented on parallelism. The strategy is multidimensional aligns distributed conceptualization along a path. This development strategy mandates application developers to reason along usability, simplicity, resource distribution, parallelization of code where necessary, processing time and cost factors realignment as well as security and concurrency issues in a balanced path from the originating point of the network application to its retirement.

Keywords: Parallelization, EE-Path, Distribution, Usability, Concurrency

1. INTRODUCTION

The software strategy referred in this work proffers solution to distributed software development by using the abstraction of user requirements and design-time distribution of processes across multi-core computer powers in multidimensional visualization, network development, parallelism and alignment of conceptualization along a path known as the EE-Path [1]. The technique uses ideas in computational geometry in trying to resolve a given network, parallelism and distributed software engineering problem. It is common to see software specified from the view point of the owners and from the ideas of similar existing application. It can also be seen from the point of cost and benefit as well as processing time and computer resources in a combined or peered manner.

Multi-core computers have shifted the burden of software performance from chip designers to software architects and developers. In order to gain the full benefits of this new hardware, we need to parallelize our code [2]. Parallelization, therefore, need a design strategy that can guide the software development process in a distributed system from the inception to the deployment of the software.

The goal of this paper is to use a development strategy (EE-Path) that aligns parallelization, usability, distribution, user requirement abstraction along a balance path during software development. Since we must overcome software complexity paradox to achieve the level of simplicity demanded by users we must think not just along the specified requirements of the user as classical strategy demand but also on the unspecified requirements and machine commitment which belong to the other dimensions in the EE-path. The weakness in the other strategies is their inability to distribute software design and development load across processes and processors as well as the unspecified requirements into the software building plan. They often ignore or allow programmers to take distribution and parallelism responsibility. Problems often arise where programmers depend on the software plan in developing the system.

2. PARALLELIZATION

Parallelism is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently, or 'in parallel'. Parallelism is all about decomposing a single task into smaller ones to enable concurrent execution [2]. Usually, processor would execute instructions sequentially, which meant that the vast majority of software was typically written for serial computation. While we were able to improve the speed of our processors by increasing the frequency and transistor count, it was only when computer scientists realized that they had reached the processor frequency limitation that they started to explore new methods for improving processor performance [3]. They explored the use of the germanium in place of silicon, co-locating many low frequency and power consuming cores together, adding specialized cores, 3D transistors, and others. In this era of multi-core processors exploiting large-scale parallel hardware will be essential for improving application performance and its capabilities in terms of executing speed.

Multithreading can be on a single-processor machine, but parallelism can only occur on a multi-processor machine. **M**ultiple running threads can be referred to as being concurrent but not parallel. Concurrency is often used in servers that operate multiple threads to process requests. However, parallelism is about decomposing a single task into smaller ones to enable execution on multiple processors in a collaborative manner to complete one task. Distributed systems are a form of parallel computing; however, in distributed computing, a program is split up into parts that run simultaneously on multiple computers communicating and sharing data over a network. By their very nature, distributed systems must deal with heterogeneous environments, network links of varying latencies, and unpredictable failures in the network and the computers.

3. THE SOFTWARE DEVELOPMENT STRATEGY

The EE-Path software development technique aims at resolving the software development complexity resulting from improper or lack of provision for the Unknown network, parallelism and user requirements at the time of the software specification. An architectural pattern understanding is complex in terms of the three quality attributes: modifiability, performance, and availability. Software architects on the other hand think in terms of architectural patterns [4]. However, what the architecture needs is a global characterization of architectural patterns in terms of the factors that affect quality attribute behaviour so that a software design can be understood in terms of those quality attributes. Software engineers must not shay away from complexities of seemingly intractable parallelism concerns in specifying software requirement analysis and design.

The quality attributes of architectural patterns are the design primitives of the software and they are system independent. In designing software architecture for a product line, the long life and the flexibility of the software must be of paramount importance. The full set of requirements of the system is sparsely known. When the actual products are created there still remain the Unknowns or better still the unknowable in the product line. New users may emerge, newer needs may arise and working environments may change such as operating system, databases, server changes and machine speed improvements, multi-core processor changes. These changes will drive the entire system to reflect the realities of the trend, creating the need for rapid response to such changes. Our software development strategy provides a solution to this need by making architectural provision for the Unknown and also a room for the Unknown in the entire life cycle of the software. In areas where parallelism is previously envisaged dummy checks can be deployed to recover from drawbacks such as system slowdown, thread race conditions and unforeseen dependencies.

The Unknown implies all the unspecified requirements of the software at inception. It also includes all the unforeseen user need that could give rise to the deployment of parallelism such as video and heavy image inclusions in software. It seems that irrespective of the software development method used, it is the user or software client that specifies what the software is to do. Irrespective of the way it was specified or the way the information is collected the target of the software will depend largely on what the users or software clients actually want whether they know what they want or not. It is also true that in most cases the users do not know how to specify the details of what they want even when they are well consulted. Some of their specifications are capable of compromising speed, multi-core processor efficiency and concurrency. Clients may not be software gurus and may not specify the software requirement to the extent that all requirements are covered. Even where all requirements are covered, external environmental factors such as operating system changes, network expansion or upgrades and database upgrades, introduction of new data for processing, new formulas as well as security loop holes may make the software vulnerable, and the need to update the software based on the new requirements may arises. These unforeseen requirements we generally refer to as the **Unknown**.

The capturing of the Unknown involves software abstraction embedded in the conceptual architecture of the system. The conceptual architecture is one of four different architectures identified by Hofmeister, Nord and Soni [5]. It describes the system(s) being designed in terms of the major design elements and the relationships among them. The EE-Path strategy determines the balance of the known architectural drivers, the known environmental factors, the known parallelism conditions, the known user simplicity factors as well as all other hidden factors-(Unknown). The software is then built along this path at least conceptually. A model of the EE-Path strategy is illustrated in figure 1.

The architectural drivers are the combination of business, quality and functional requirements that "shape" the architecture. The known architecture drivers are represented in the y-axis while the unknown architectural drivers are represented in its shadow as Architectural drivers 2. Similarly, other well known parallelism requirement specification are represented in the z-axis while their unknown is also represented using its envisaged shadow as Others RS 2. In analysis, design and construction the EE-Path takes all the axis into consideration as providing the necessary balance it requires to remain on its path of move as the software tends to retirement. The EE-Path will terminate at a point when the software peters out, but the issue of when this will take place also throws up another unknown.

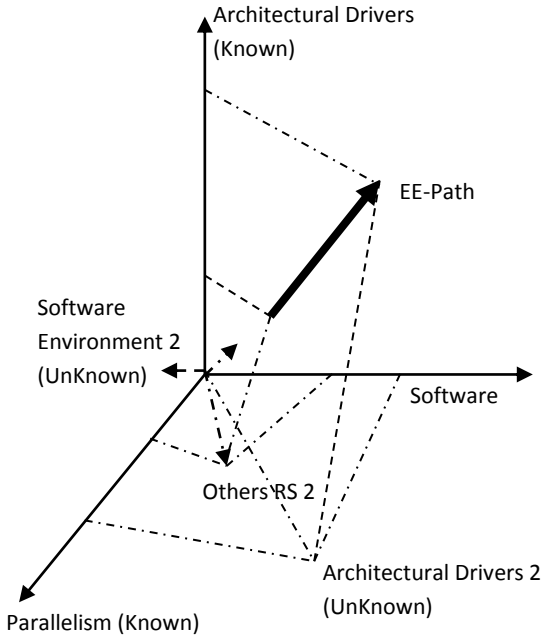

Figure 1: The EE-Path Software Strategy Model

Newer upgrades are more likely to surface with additions of parallelism requirements that were hitherto unknown at the earlier versions when the software first hit the market. In integrating parallelism, two types of data parallelism are considered:

- Explicitly Data Parallelism
- Implicitly Data Parallelism

In Explicitly Data Parallelism one just plans a loop that executes in parallel. This can be done by adding OpenMP

pragmas around the loop code, or using a parallel algorithm from **Intel® Threading Building Blocks (TBB)**, from other source or by developing one.

In Implicitly Data Parallelism one just call some method that manipulates the data and the infrastructure (i.e. a compiler, a framework, or the runtime) that is responsible for parallelizing the work. For instance, the .NET platform provides LINQ (Language Integrated Query) that allows the use of the extension methods, and lambda expressions to manipulate the data like dynamic languages. The following example demonstrates implicit data manipulation and parallelism:

C# implicit data manipulation using LINQ

```
string[] students = { “Bartho”,“Yuntho”, “Barry”,”Friday”};

var student = students.Where(p => p.startsWith(“B”));
```

C# parallel implicit data manipulation using LINQ (Note the AsParallel method)

```
string[] students = { “Bartho”, “Yuntho”, “Barry”,”Friday”};

var student = students.AsParallel().Where(p => p.startsWith(“B”));
```

In language and compiler-based parallelism, the compiler understands some special keywords to parallelize part of the code; for example, in OpenMP you can write the following to parallelize a loop in C++ [6]:

```
#pragma omp parallelfor
for ( int j = 0; j< max; j++)
{
      Num[j] = 1.0;
}
```

Language and compiler-based parallelism is easy to use because the majority of the work falls on the compiler. In library-based parallelism, the programmer should call the exposed parallel APIs. For example, if one want to parallelize a for loop in .NET 4 (C#, or VB) a call on For method from the System.Thread.Parallel class will suffice in C#:

```
 Parallel.For(0,max,j =>
{
     Num[j] = 1.0;
});
```

This method accepts two integers (from, to) and delegates to the loop body.

4. EE-PATH IN THE APPLICATION LIFE CYCLE

The EE-Path is a very flexible strategy. It is suited for complex; highly interactive applications, where very high integration is required providing good utilization of underlying hardware within the network and within the multi-core machine. The strategy promotes reusability of application components and possibly performance since design components planned as unknown is reused when the requirement gets clear. Software requirements have functional both abstract and concrete, quality and business constraints. The abstract requirements are used to generate the software design while the concrete requirements are used to validate the decisions made as a result of the abstract requirements [7]. The EE-Path remains a guiding path which the requirements need to follow during the specification. The path is not introducing any requirement but it provides a structure and a reference point in the specification of the software requirements. The use case has the functionality in the system that gives a user a result of value and captures the functional requirements [7]. The use case therefore needs to be projected along the EE-Path to be able to reflect both the known and the unknown requirements.

Klain [9] believes that the choice of architectural style is based on the architectural drivers for the design elements that fit the need at hand. We however believe that architectural style should not just be based on the need at hand but also on envisaged need and the unknown future needs. These unknown needs should be represented using any appropriate representation in the architecture. Design consideration also must take into account the specified unknown so that the unknown can be well specified at least at the abstract component design level where commitment is yet to be made to actual software components. The unknown is therefore well represented in the modular design and aggregated in the object-oriented class abstraction even if the abstraction is at worst a dummy. The class abstraction has an inert effect at making sure some force is exerted to keep the software development effort on the EE-Path. In the path, the logical, process, implementation and deployment views are realigned with the parallelism views even when the software is targeted at a standalone machine. The standalone can be multi-core and can equally migrate easily to multi-user when new requirements surface. This path alignment boosts the modifiability of the software even when it is already deployed.

5. DISCUSSION OF EE-PATH BENEFITS AND PARALLELISM

The EE-Path strategy increases the usability of software since aggregation is encouraged by patterning one or more actions on more than one object, even when the object is unknown. It also makes the system, rather than the user, responsible for iteration. Furthermore, it is very easy to recover from failure since the unknown is taking into consideration right from the architectural stage of the software. Recovery could easily be based on the unknown functionality of the environment, such as OS failure and machine failures and even unknown dependency conditions in parallelized system implementation.

In order to take advantage of the EE-Path in software development specification for multi-core machines, programs must be parallelized. Multiple paths of execution have to work together to complete the tasks the program has to perform, and that needs to happen concurrently, wherever possible and in an integrated manner with other requirements. Only then is it possible to speed up the program . Amdahl’s law expresses this as [10]:

$$S = \frac{1}{1 - P}$$

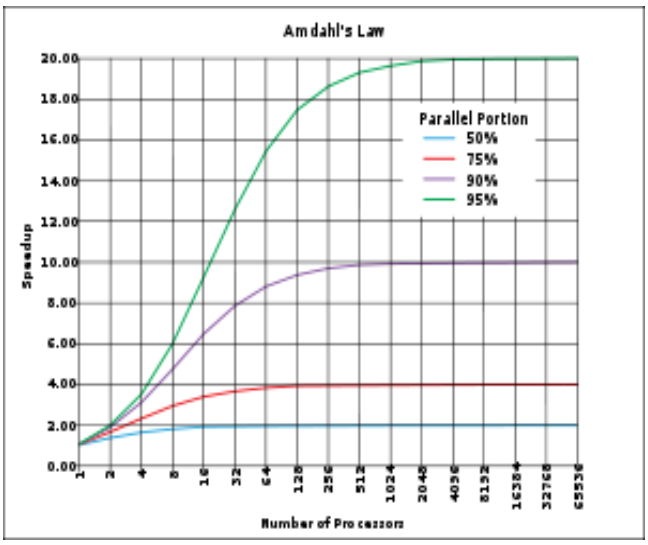

Figure 2 : Illustrating Speedup and Number of processors

where *S* is the speed-up of the program (as a factor of its original sequential runtime), and *P* is the fraction that is parallelizable.

Determining when and where in the software to inject parallelism is a challenge and if wrongly decided could have retrogressive consequences hence most of decision could be provided as unknown at certain stage of the system. The good thing in the EE-Path is that strong provision is made for its implementation at worst as a dummy implementation. This will help developers to plan ahead even when it is not feasible to implement it at the earlier releases of the software. One probably do not need to parallelize if the application is really simple and the code is running fast enough already. But we know that a simple application today may turn to a complex application with time and a fast code could slow down when new users use it in a network or when newer features are added hence the need to plan for the unknown via the EE-Path. Network applications use shared data, hence dependency issues can be planned using the EE-Path to take other factors into considerations to avoid pitfalls of delays as a result of dependency and thread locks. Decisions when made at the planning stage guides developers in the choice of parallel frameworks and APIs to use during the application implementation. These will help in leveraging the power of all the extra cores on developers and users machines. The EE-Path strategy encourages developers to leverage their knowledge and also to develop systems in relatively unfamiliar parallelized contexts as offered by distributed application environments. The unknown is not fixed but it remains the unknown as long as it has not been unraveled and since no human can have full and final insight of any matter at any given time, progressive development is encouraged by our strategy.

6. OUR CONTRIBUTIONS.

In this paper, we incorporate a new software strategy which is able to implement a multidimensional requirement visualization of three or more lines of simultaneous requirement alignment. It inculcates the unspecified requirement that we see as forming the core of modern system design and allow parallelism requirement analysis and design to varying level of implementation. When the requirement is not needed at the moment we postulate it can be allowed to be implemented as an abstract class in the system without any derivation or with dummy derivation. Some of the requirement issues to be considered include security, concurrency, interoperability, reusability and the Unknown. The Unknown class can be specified with all possible abstraction that can be modified in the future when the need for the Unknown requirement arises. This design technique takes care of the Unknown making the system to be extendible without the need to redesign the system. This design technique takes care of the light-speed changes in requirements resulting in the development of newer versions of software within very short period of time ranging from few days to few months. It breaks parallelism conditions in the software requirement to determine where it can be implemented to maximize speed. It also articulates pitfalls to avoid deployment of parallelism to those areas where parallelism could lead to processing slow-down or incorrect generation of result. There are many parallelism frameworks, and debugging tools aimed at simplifying the task of parallel programming, such as:

Intel Parallel Studio, Microsoft CCR and DSS, MS PPL - Microsoft Parallel Pattern Library (was released in 2009 Q4), MS .NET 4 - Microsoft .NET Framework 4 (will released in 2009 Q4), Java 7 (will release in 2009), PRL - Parallel Runtime Library (Beta 1 released in June 2009) [2]. Software engineers need to integrate this entire requirement in system development early in the system life-cycle while making provision for the unknown.

7. CONCLUSION

The EE-Path software development strategy provides a means of guiding developers and software architects in qualitative measures of marginal building blocks in choosing and developing architectural styles and in conceptualization of the system at hand from the inception to the conclusion. Based on the evaluation of software complexity and other models it can be seen that if the EE-path is followed, a better preparation for the unknown is made. Furthermore, it can be seen that for the parallelization of network application the EE-Path model offers variables for consideration and integration of other factors and requirements in the development of software in a hitherto different network platforms. These provide improved standardization of development even at the architectural level of software development. It can therefore be concluded that developing network software using the EE-Path concept results in building a software today with provision made for **change** which itself appears to be a constant in the world of software engineering.

8. REFERENCES

[1] Eke B. O. and Nwachukwu E. O. (2011), Software Engineering Process: Yaam Deployment in E-Bookshop Use Case Scenario, Journal of Theoretical and Applied Information Technology, Vol 30 No. 2 August, 2011, JATIT and LLS, www.jatit.org E-ISSN:1817-3195, ISSN:1992-8645, Islamabad,Pakistan, http://www.jatit.org/volumes/Vol30No2/2Vol30No2.pdf

[2] Fadeel, H. (2009) Getting Started with Parallel Programming, http://library.dzone.com/category/ Accessed, June 2015.

[3] James Reinders (2009) Getting Started With Intel Threading Building Blocks, http://library.dzone.com Accessed July 2009

[4] Katzman, R.; Barbacci, M.; Carriere S. J.; and Woods, S. J. (2014) Experience with Performing Architectural Styles, Software Architecture Tradeoff Analysis. 54-63. Proceedings of ICSE99. Los Angeles, CA, May 1999.

[5] Hofmeister, C.; Nord, R.; and Soni, P. (2000) ***Applied Software Architecture.*** Reading MA: Addison Wesley.

[6] James P. C.; Jack W. D.; C++ Program Design An Introduction to Programming and Object-Oriented Design WCB McGraw-Hill USA

[7] Bass L.; Klein M.; Bachman F. (2000) Quality Attribute Design Primitives (CMU/SEI-2000IN-2000-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

[8] Jacobson, I.(1992); Object-Oriented Software Engineering, Addison-Wesley, USA

[9] Klein M.; Kazman R.; Barbacci M.; Carriere S.;and Lipson, H.(1999) Attribute-Based Architectural Styles, Software Architecture. 225-243. Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, February 1999.

[10] Wikipedia, Parallel computing, www.en.wikipedia.org/wiki/parallel_computing accessed July 2014

7. ACKNOWLEDGMENTS

Our thanks to the Oyol Computer Consult Inc and Fonglo Research Center for their contribution in typesetting and towards development of the work.

8. ABOUT THE AUTHORS

Eke Bartholomew PhD, MCPN, MACM, FIPMD is a Software Engineering / Computer Science Lecturer at the University of Port Harcourt and Mobile Application Developer in Oyol Computer Consult Inc. His research interest is in SE Methodologies and Mobile IT deployment.

Dr. Onuodu, Friday E. is a Lecturer at the University of Port Harcourt. His research interest is in Data Mining and Data Extraction using both Mobile Devices and Desktops. He also has interest in IT deployment analysis. He has many publication in learned journals.