This is a pre-print of a contribution published in Information Systems Architecture and Technology: Proceedings of 39th International Conference on Information Systems Architecture and Technology – ISAT 2018, edts: Leszek Borzemski; Jerzy Świątek; Zofia Wilimowska, published by Springer. The definitive authenticated version is available online via http://dx.doi.org/10.1007/978-3-319-99981-4_15

From Sequential to Parallel Implementation of NLP using the Actor Model

Michał Zielonka[0000-0003-3577-4244] , Jarosław Kuchta[0000-0003-3413-6830] , Paweł Czarnul[0000-0002-4918-9196]

Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology
Narutowicza 11/12, 80-233 Gdańsk, Poland

Abstract. The article focuses on presenting methods allowing easy parallelization of an existing, sequential Natural Language Processing (NLP) application within a multi-core system. The actor-based solution implemented with the Akka framework has been applied and compared to an application based on Task Parallel Library (TPL) and to the original sequential application. Architectures, data and control flows are described along with execution times for an application analyzing an online dictionary of foreign words and phrases.

Keywords: Parallelization, Actor Model, Akka, TPL, NLP.

1 Introduction

Nowadays, we deal with multi-core computers every day. Even a personal computer is usually equipped with a two- or quad-core processor with hyperthreading. Despite this, as our experience in cooperation with CI TASK[1] has shown, scientists who are not IT professionals, often write sequential applications and do not fully use the capabilities of multi-core hardware. They run their sequential applications in a multiprocessor environment and are uncomfortably surprised that they do not get acceleration of calculations. Proper parallelization is usually done by rewriting the program code from scratch or designing it to be parallel upfront especially for a multi-node cluster environment, so that the most time-consuming parts can be executed simultaneously by many processors. Unfortunately, such parallelization is generally very complicated and time-consuming.

[1] Centre of Informatics - Tricity Academic Supercomputer & networK

To avoid rewriting the code of our NLP application from scratch, we can use an alternative approach within a single computational node. One solution proposed by Microsoft is TPL (Task Parallel Library) [1]. If we define several tasks in the program that are loosely coupled, we can attach these tasks to separate threads. TPL helps us to define tasks, run them asynchronously, and return the results. Having a computer with e.g. a quad-core processor with hyperthreading, we can easily use eight logical processors in our application. Unfortunately, the TPL solution has its limitations. Firstly, it is useful in a pipeline or divide-and-conquer model of parallel application, when there is no need for inter-task communication. TPL fails with more complex communication due to lack of support for sending messages between tasks. The second problem is that tasks must be run explicitly, one per thread. Finally, TPL supports multicore calculations, so parallelization is limited to multicore processors.

All these problems are solved by the Akka [2] toolkit implementing the actor model [3]. In this model, application parallelization is based on the concept of actors – classes that perform specialized tasks. Objects that are instances of actor classes send each other data through messages that are placed in queues. Each queue is associated with an actor class that creates one or more instances depending on the load recognized by the queue length. After the data has been processed from the message, the actor's instance sends a message with the results to the queue of another actor's class.

Therefore, we decided to examine parallelization of sequential applications using the aforementioned example, and to compare the performance of parallel applications using both TPL and Akka solutions.

2 Related Work

Parallelized NLP is considered in the literature. For instance, a parallel parser with the work stealing approach is analyzed in [4]. Fine-grained parallelism is needed to obtain high speedup for this purpose according to [5] and on GPUs [6]. Our goal in the paper was parallelization of sequential elements of our application in a way that does not penetrate too much in the current architecture of the code and make it a scalable solution within a single multi-core node. There are many APIs to parallelize sequential applications [7]. OpenMP [8] is one of the popular APIs for this purpose. This is a well-known approach suitable for applications running on shared memory systems. OpenACC [9], similarly to OpenMP, allows extending sequential codes with directives and parallelization using GPUs. OpenCL [10] is an API allowing writing parallel programs for both multicore CPUs and GPUs with multiple threads running NDRange kernels. CUDA [11], focuses on NVIDIA GPUs Programming productivity, performance, and energy consumption when using OpenCL, OpenACC, OpenMP, and CUDA are discussed in [12]. Message Passing Interface (MPI) [13] allows to parallelize sequential applications for cluster environments and provides API for interprocess communication. For applications written in the .Net environment, we can often observe the use of Task Parallel Library to add parallelism and concurrency to the solution [1]. This approach will also be discussed and compared to the actor-model described later with the Akka toolkit [2].

3 Original Sequential Application

The original sequential application is an NLP (Natural Language Processing) application for analyzing an online dictionary of foreign words and phrases[2]. The dictionary consists of over 14,000 HTML files. Each file consists of one or several entries. There are over 30,000 entries in this dictionary.

In order to avoid frequent Internet access, the files have been downloaded once and stored locally. After processing by the HTML parser, the files have been divided into entries, which have been saved in the relational database for further analysis. Both the results of the entire analysis and the results of individual stages are recorded in the same database.

The NLP component of the application became the subject of our interest. The NLP algorithm used is well known [14] and can be described as a processing pipeline [15] (Fig.1). First, the text is divided into paragraphs and sentences (corresponding to our dictionary entries). This has already been done while storing entries to the database. Then entries go to the process of tokenization, during which token sequences are generated. Tokens correspond to words and punctuation marks. Usually, in the next stage, tokens are subject to the process of parts of speech (POS) recognition, and then – parsing. In our application, the POS tagging step has been omitted, since dictionary entries have their own formal syntax. Instead, the correction step was added, during which syntax errors made by the authors of the dictionary are removed.

However, not all syntax errors can be identified and removed at this early stage. Some of them are revealed later – at the stage of tokenization, and some others – at the stage of parsing. Therefore, the process requires relapses. Errors detected at the parsing stage are returned to the tokenization step, which has to produce a different sequence of tokens. Sometimes it is required to return to the error correction stage.

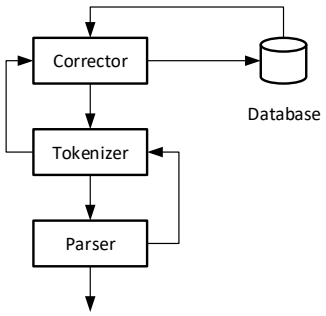

Fig. 1. Dataflow for NLP in the original sequential application

The algorithm of the original application has been implemented in C# using the Visual Studio IDE and the local MS SQL Server database.

[2] https://www.slownik-online.pl/

4 Parallelizing the Sequential Application

The most-seemingly natural way of parallelizing the application for a parallel environment would be to process all dictionary entries in parallel independently – in separate processes run on separate cores and machines. However, we parallelized our application to run in a single node environment available for an average scientist – such as a laptop with an Intel i7 processor – where the number of threads possible for parallel processing is rather small when using typical multi-core CPUs.

4.1 TPL Approach

In the first approach, we used the solution based on the *Task* class, which represents the operation performed asynchronously (although not necessarily in parallel). Each task is associated with a separate thread, which is placed in the *thread pool*. The thread pool is equipped with a core load balancing mechanism that matches the number of threads to run to the processor's capabilities.

TPL does not support synchronization between threads, so it is suitable rather for pipelining and the divide-and-conquer model. In both models, it is enough to take the result after the asynchronous operation has ended.

Since our application includes feedback loops, we introduced an additional revision step to simplify the algorithm. in which we grouped backward tokenization and backward correction operations. We defined four tasks (correction, tokenization, parsing, revision) in the processing pipeline (as shown in Fig.2). We put buffer variables between the tasks to pass the results of one task to be processed by the second task.

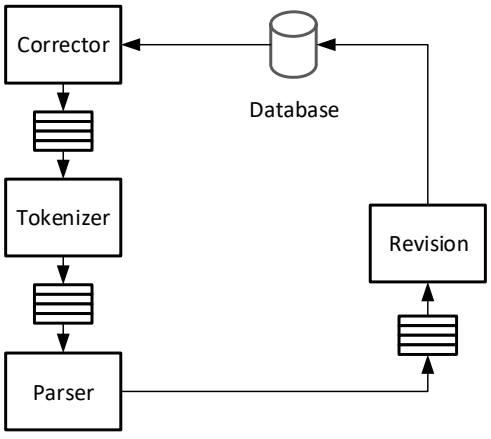

Fig. 2. Pipeline processing using TPL

We implemented the TPL algorithm using a common TaskFactory class instance with a StartNew() operation. Each StartNew() invokes a specific operation (Corrector, Tokenizer, Parser, Revision). These operations take the results of the previous stages

which are stored in buffers. The whole algorithm was implemented without using "async" or "await" keyword, as we used a Task.WaitAll() operation to suspend the main thread until all the tasks finalize their processing (see below code sample).

```
var taskFactory = new TaskFactory();
var correctorStage = taskFactory.StartNew(
  ()=>Corrector(fileEntries.Count,correctorBuffer));
var tokenizerStage = taskFactory.StartNew(
  ()=>Tokenizer(correctorBuffer,tokenizerBuffer));
var parserStage= taskFactory.StartNew(
  ()=>Parser(tokenizerBuffer,parserBuffer));
var revisionStage= taskFactory.StartNew(
  ()=>Revision(parserBuffer));
Task.WaitAll(correctorStage, tokenizerStage,
             parserStage, revisionStage);
```

4.2 Approach Using the Actor Model and Akka Framework

The actor model was created to facilitate synchronization between the separate threads. Programmers writing parallel applications in C# usually use the classic *lock* instruction to describe the critical section. In many situations this is the best and easiest way. Unfortunately for large and complex systems, maintaining such code is very difficult, laborious and susceptible to deadlocks.

In the actor model we do not need to use any locks as the code in each actor class is executed only by one thread. Critical sections are therefore replaced by asynchronous messages. If one thread wants to read or change the state of other thread, it sends an asynchronous message. The most difficult part of the multi-threading is to modify the shared state. Using actor model, there is no need for such modification as a shared state does not exist. Each actor works independently of each other. If one actor's data is needed by another, the data is sent as non-modifiable asynchronous message.

The actor itself is an object that has an inbox, state and behavior. It communicates with other actors within the system using asynchronous, non-blocking messages (Fig.3). Different types of messages are queued in the inbox (for remote actors the data types must be serializable) and they are processed in series.

Akka is an open-source, free toolkit implementing an actor model. It was originally written in Scala on Java Virtual Machine, but a .NET implementation is also available.

Using Akka in our application, we created appropriate actor classes for our tasks. Actors among themselves submit indexes of data records ready to be processed. Below we can see the sample implementation of one of the actors.

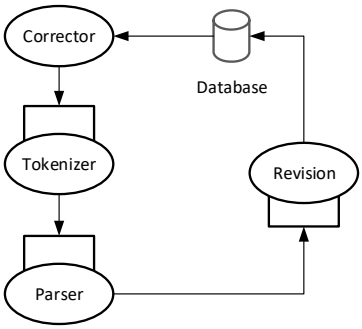

Fig. 3. Using Akka actors model

```
public class CorrectorActor: ReceiveActor
{
  public  IActorRef Tokenizer;
  public  CorrectorActor(IActorReftokenizer)
  {
    Tokenizer  =  tokenizer
    Receive<FileEntryIndex>(index =>
    {
      Corrector.TryCorrectEntry(fileEntry[index]);
      Tokenizer.Tell(index);
    });
  }
}
```

FileEntryIndex is a class representing the message which the actor receives. If someone sends a specific message to this actor, it starts his work. Corrector is an object derived from the sequential version of the application and contains the logic responsible for dictionary entry correction. The delivered message contains an index of this entry. After appropriate actions, finally a new message is sent to the next module. Thanks to this, the actor responsible for Tokenizer stage will receive information about work for itself and Corrector will handle the next message at that time. In this simple way we create actors responsible for each module. Then we move on to our main methods and initialize the Akka system and actors.

```
ActorSystem = ActorSystem.Create();
IActorRefcorrectorActor=ActorSystem.
  ActorOf(props,"correctorActor");
for(var i=0; i<fileEntry.Count; i++)
{
  correctorActor.Tell(i);
}
```

Here we send the data record index to the actor instead of immediately referring to the method responsible for the stage (as we did in the sequential version).

In addition, we are able to define how many instances of each actor we want to create. This means that we can, for example, create several instances of actors responsible for the first stage, the second and so on.

At this point, the system responsible for the actors will prepare as many Corrector actor instances as we define in the numberOfActors variable. The user also has the option to apply the algorithm to load-balance which will be used by the system. For load-balance, we used the round-robin algorithm shown in the code below.

```
var props = Props.Create(
  ()=>new CorrectorActor(tokenizerActor))
  .WithRouter(newRoundRobinPool(numberOfActors));
var correctorActor = ActorSystem.ActorOf
  (props, "correctorActor");
```

Akka guarantees that operations inside an actor algorithm are executed sequentially, so transferring the logic of each module of the existing sequential version of the application does not pose any major problem.

5 Performance Tests

This chapter presents performance results of selected solutions tested in two environments that are described in the next section. It is worth noting that we measured only include the duration of calculations. The time used for reading and writing operations to the database has been omitted.

5.1 Test Environment

Tests have been carried out on two different machines. The first is a typical laptop with a quad-core processor. Its parameters were as follows:

- Operating system: Windows 10 Pro 64
- Processor: Intel Core i7-6820HQ (2.70 GHz, 8MB L3 cache, 4 cores)
- Memory: 16 GB DDR4-2133
- Frameworks: .NET Core 2.0, Akka 1.3.2.0

In order to verify the scalability of the prepared solutions, the applications were also launched on a cluster node with many cores. Calculations were carried out at the TASK Academic Computer Center in Gdansk. The node parameters are as follows:

- Operating system: Ubuntu 16.04 64
- Processors: Intel Xeon Processor E5 v3 @ 2,3 GHz, 12-core (Haswell)
- Memory: 256 GB RAM DDR4
- Frameworks: .NET Core 2.0, Akka 1.3.2.0

5.2 Test Results

Figure 4 presents a comparison of application execution times using various numbers of actors to the versions using TPL and the sequential approach.

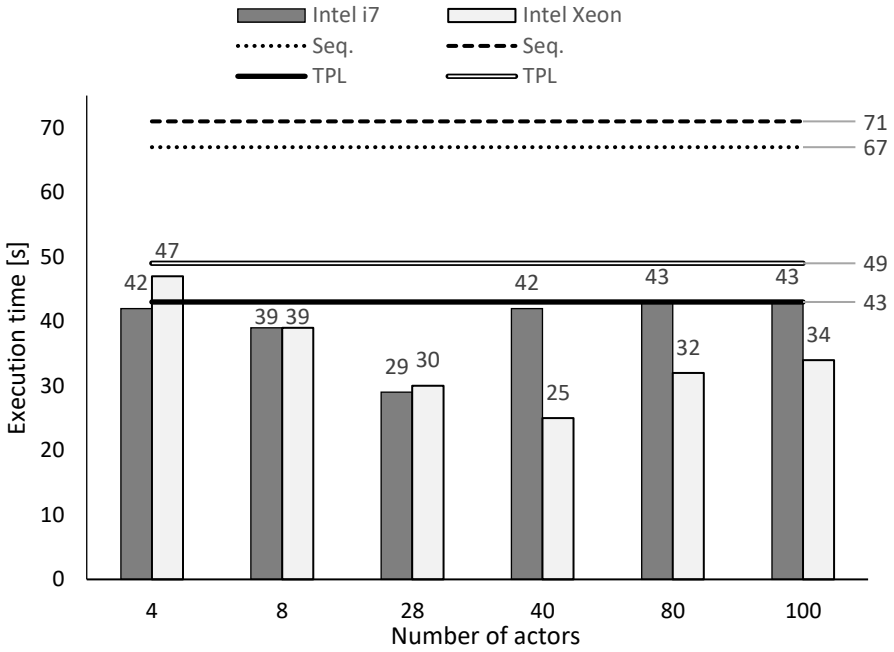

Fig. 4. Performance comparison

As we can see from the results, the best time for a quad-core machine is 29 seconds when using 28 actors. From the results on the cluster node, we see that for the sequential version and using fewer threads or actors, the results are slightly worse than on the first machine. This is due to the fact that the Intel Xeon E5 is a processor with a lower clock frequency the Intel Core i7.

With the use of higher number of cores (for 40 instances of actors), the application was executed in 25 seconds. This observation shows that the parallel approach is faster than sequential roughly 2.3x for Intel i7 and 2.8x for Intel Xeon. When performing parallelization, the code architecture was not interfered with, but only new modules were added using the existing sequential code elements. This makes the results satisfactory considering that we could easily convert a sequential application to its parallel version.

In Figure 5, we can observe the CPU (Intel Core i7) load at a given time. In total, there are 8 logical processors (4 physical cores and 4 virtual cores due to Hyper Threading Technology). Akka used all available cores as presented in the aforementioned figure.

The level of processor utilization was also measured on the cluster node. Table 1 shows the average load of each logical processor during application execution.

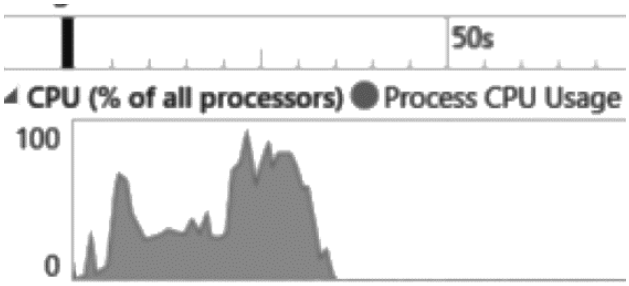

Fig. 5. CPU usage on Intel Core i7

CPU index	avg usage [%]	CPU index	avg usage [%]
0	86.4	12	73.4
1	85.6	13	73.1
2	73.4	14	50.6
3	73.1	15	49
4	72.3	16	50.5
5	68.5	17	49.1
6	71.4	18	44.2
7	70.5	19	42.5
8	71.1	20	45.5
9	67.4	21	38.4
10	68.2	22	32.7
11	68.5	23	34.6

Table 1. Intel Xeon CPU usage performance

6 Summary and Future Work

In the paper we showed that it is possible to parallelize an existing NLP application in an easy way using the actor model. The results showed that the obtained speedup is roughly 2.3 to 2.8 times using the actor based approach within a multi-core machine.

Programming with actor model separates the programmer from using threads. This simplifies writing efficient programs using the available processor cores. This is an important issue, because the computing power of new computers relies mainly on the number of cores. In the next step we plan to profile execution at a finer level to investigate bottlenecks. This will allow to model overheads and scalability of the actor model and the Akka implementation into our modeling and simulation environment called MERPSYS and perform performance tests on various CPUs from the MERPSYS database [16].

References

1. Leijen, D., Schulte, W., Burckhardt S.: The design of a task parallel library. SIGPLAN Not., vol. 44, no. 10, pp. 227– 242, Oct. (2009). doi: 10.1145/1639949.1640106
2. de Castilho, R.E., Gurevych, I.: A broad-coverage collection of portable NLP components for building shareable analysis pipelines. Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT (2014)
3. Wyatt, D.: Akka concurrency. Artima Incorporation (2013)
4. van Lohuizen, M.P.: Parallel processing of natural language parsers. In Parallel Computing: Fundamentals and Applications (200)
5. Van Lohuizen, M.P.: Effective Exploitation of Parallelism in NLP (1999)
6. Lai, C. Y.: Efficient Parallelization of Natural Language Applications using GPUs. vol. Technical Report No. UCB/EECS-2012-54. University of California at Berkeley, Electrical Engineering and Computer Sciences (2012)
7. Czarnul, P.: Parallel Programming for Modern High Performance Computing Systems. CRC Press (2018). ISBN 9781138305953
8. Chandra, R., Dagum, L., Kohr, D., Maydan, D., Menon, R., & McDonald, J.: Parallel programming in OpenMP. Morgan kaufmann (2001). ISBN 1-55860-671-8, 9781558606715
9. Wienke, S., Springer, P., Terboven, C., & an Mey, D. (2012, August). OpenACC—first experiences with real-world applications. In European Conference on Parallel Processing (pp. 859-870). Springer, Berlin, Heidelberg (2012). doi: 10.1007/978-3-64232820-6_85
10. Stone, J. E., Gohara, D., & Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(3), 66-73 (2010). doi: 10.1109/MCSE.2010.69
11. Nickolls, J., Buck, I., Garland, M., & Skadron, K.: Scalable parallel programming with CUDA. In ACM SIGGRAPH 2008 classes (p. 16). ACM. (2008, August). doi: 10.1145/1365490.1365500
12. Memeti, S., Li, L., Pllana, S., Kołodziej, J., & Kessler, C.: Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing,. ACM. (2017, July). doi: 10.1145/3110355.3110356
13. Gropp, W. D., Gropp, W., Lusk, E., & Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface (Vol. 1). MIT press. (1999). ISBN 0262527391, 9780262527392
14. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P.: Natural language processing (almost) from scratch. Journal of Machine Learning Research, 12(Aug), 2493-2537 (2011)
15. de Castilho, R. E., & Gurevych, I.: A broad-coverage collection of portable NLP components for building shareable analysis pipelines. In Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT pp. 1-11 (2014)
16. Czarnul, P., Kuchta, J., Matuszek, M., Proficz, J., Rościszewski, P., Wójcik, M., & Szymański, J.: MERPSYS: an environment for simulation of parallel application execution on large scale HPC systems. Simulation Modelling Practice and Theory, 77, 124-140. (2017)