An Intel Cilk Plus Based Task Tree Executor Architecture

MIROSLAV POPOVIC, ILIJA BASICEVIC
Faculty of Technical Sciences
University of Novi Sad
Trg Dositeja Obradovica 6, 21000 Novi Sad
SERBIA
miroslav.popovic@rt-rk.com    http://www.ftn.uns.ac.rs

Abstract: - The goal of this paper is to optimize system performance of a SOA-based critical infrastructure by modifying the architecture of the service component that is responsible for complex calculations on large-scale graph models, under soft real-time restrictions. On a request, this service component transforms a system model into a task tree, which then gets executed by the runtime library that is referred to as the Task Tree Executor (TTE). The strategy that was used to optimize the system performance was to introduce finer grained parallelism, thus better multicore CPU utilization. The result is a novel TTE architecture that executes TTE tasks as Intel Cilk Plus tasks rather than OS threads, which was the case for the previous TTE architecture. The experimental measurements of time needed for TTE reliability estimation, based on statistical usage tests, show that the novel TTE architecture provides the speedup of around 11x, on average, over the previous one. Although this paper deals with a particular SOA service component, it may serve as a case study demonstrating applicability of our strategy on a broader class of SOA-based systems.

Key-Words: - SOA, Architecture, Task trees, Parallel programming, Intel Cilk Plus, Statistical usage testing

1 Introduction

Proper parallel data processing is one of the key issues to be resolved with when designing software solutions for management of critical infrastructures, like oil, gas, and electricity distribution systems. These systems normally provide continuous system supervision and control, based on data acquisition and processing. They are expected to fulfill high availability, reliability, and security standards, and to address numerous design aspects related to various operational activities. A solution for all these requirements is typically found in a form of a SOA based system comprising a complex suite of service components, e.g. see [1-2].

The most crucial for design is a service component that provides requested calculations using various models of the system, which commonly take a form of a graph or a tree. The examples of these calculations include load flow calculation, network state estimation, evaluating performance indices, etc. Design of this kind of service components is awkward because all these calculations are nontrivial and have to be performed on large-scale graph models in soft real-time. Moreover, designers are forced to reuse legacy software, because of its enormous size (measured typically in millions of lines of code) and time-to-market pressure.

We have used two approaches in the past to design and develop such service components for the electricity distribution systems. The first approach [3-4] requires transforming network models into Task Tree Executor (TTE) task trees and refactoring legacy code by introducing callback functions that are executed by TTE tasks. The second approach [5] requires repackaging legacy code as libraries and executing them within Calculation Engine (CE) processes. Overall, the second approach requires less development effort and is more robust, whereas the first approach provides more fine grained parallelism.

The goal of this paper was to modify the previous TTE architecture based on threads [3] into a novel TTE architecture based on Intel Cilk Plus (Cilk), thus provide even finer grained parallelism. Essentially, the TTE tasks that were executed as threads now become more lightweight Cilk tasks (aka strands). The advantages of this approach are: (i) Cilk provides better multicore CPU utilization than local OS, (ii) Cilk provides almost infinite number of tasks, whereas local OS provides rather limited number of threads within a process, and (iii) Cilk provides excellent load balancing functionality within the novel TTE architecture.

The content of this paper is organized as follows. The related work is briefly covered in the subsection 1.1. The previous TTE architecture based on threads
is described in Section 2, whereas the novel TTE architecture based on Cilk is described in Section 3. The statistical usage testing method and the results of the experimental evaluation are presented in Sections 4 and 5, respectively. Final conclusions are given in Section 6.

1.1 Related Work

Cilk appeared as a simple extension of C that provides constructs for parallel control and synchronization. These constructs impose very little overheads – the typical cost of spawning a parallel thread is only between 2 and 6 times the cost of the C function call on a variety of contemporary machines [6]. Once spawned, these parallel threads are scheduled very efficiently on a shared memory multiprocessor (SMP), by the Cilk scheduler, which uses the work stealing scheduling method. In this method processors needing work steal computational threads from other processors. Analysis made by Blumofe and Leiserson [7] show that the expected time to execute a well-structured computation on P processors using their work-stealing scheduler is \( T_1/P + O(T_\infty) \), where \( T_1 \) is the minimum serial execution time of the multithreaded computation and \( T_\infty \) is the minimum execution time with an infinite number of processors.

To aid parallel programming, Cilk provided a tool called the Nondeterminator-2, which finds nondeterministic bugs called data races. The Nondeterminator-2 uses two algorithms, namely All-Sets and Brelly, to find data races. The former is exact but can sometimes have poor performance, whereas the latter imposes a locking discipline on the programmer but is guaranteed to run in nearly linear time [6].

The Cilk language has been developed, as an ANSI C extension, since 1994 at the MIT. A commercial version of Cilk, called Cilk++, that supports both C and C++, was developed by Cilk Arts, Inc. In 2009, Intel Corporation acquired Cilk Arts, the Cilk++ technology and the trademark. In 2010, Intel released a commercial implementation in its compilers under the name Intel Cilk Plus. In this paper we use the latter and refer to it as Cilk.

Kirkegaard and Aleen [8] also used Intel Cilk Plus to study the potential of individual optimizing techniques in terms of speedup. They applied 5 techniques, namely the array of structures style data layout, the nested computational iteration, the data dependent branches, the use of short C functions, and the use of built-in transcendental math functions, on the Google’s AOBench benchmark, to achieve the overall 16.47x speedup.

Luk et al. [9] use Intel Cilk Plus to demonstrate their synergetic approach to throughput computing. They use (i) cache-oblivious techniques to divide a large problem into smaller sub-problems that are mapped to different threads, (ii) compiler to exploit SIMD parallelism within each sub-problem, and (iii) auto-tuning to pick the best parameter values throughout the optimization process. Experimental results collected on a dual-socket quad-core Nehalem show that their approach achieves an average speedup of almost 20x over the best serial cases for an important set of computational kernels.

Agrawal, Leiserson, and Sukha [10] developed the Nabbit, a work-stealing library for execution of task graphs with arbitrary dependencies which is implemented as a library for the multithreaded programming language Cilk++. They evaluated the performance of Nabbit using a dynamic program representing the Smith-Waterman algorithm. Their results indicate that when task-graph nodes are mapped to reasonably sized blocks, Nabbit exhibits low overhead and scales as well as or better than other scheduling strategies. Interestingly enough, Nabbit is rather similar to our TTE architecture. The main difference is that Nabbit supports acyclic task graphs, whereas TTE supports task trees.

2 Thread Based TTE Architecture

TTE was designed with an idea that an application would partition a system model into slices, assign these slices to individual tasks, and transform the system model into the corresponding task tree, which would finally be executed by the TTE runtime library. Therefore, the TTE API [2] exports the following functions:

1. TS_CreateTaskGraph
2. TS_AddTask
3. TS_DeleteTask
4. TS_SetBottomUpProcFun
5. TS_SetTopDownProcFun
6. TS_ExecuteBottomUp
7. TS_ExecuteTopDown
8. TS_DestroyTaskGraph
9. TS_ExecuteBottomUpSequentially
10. TS_ExecuteTopDownSequentially

The API function TS_CreateTaskGraph creates the task tree. Its input parameters are the root task identification (ID), the pointer to the bottom-up processing function, the pointer to the top-down processing function, and the maximal number of local OS threads that will be used to execute the task tree in parallel. Both callback functions have the task ID as their parameter. The API function
**TS_AddTask** adds a new task to the task graph. Its parameters are the ID of the predecessor task and the ID of the new task. The API function **TS_DeleteTask** deletes the given task and all of its successors.

The API functions **TS_SetBottomUpProcFun** and **TS_SetTopDownProcFun** redefine the pointers to the bottom-up processing function and to the top-down processing function, respectively. The API functions **TS_ExecuteBottomUp** and **TS_ExecuteTopDown** execute the task tree bottom-up in parallel, and top-down in parallel, respectively. The API function **TS_DestroyTaskGraph** deletes the task tree.

The TTE architecture based on threads comprises two main components, the C module **TaskScheduler** and the class **Task**. The module **TaskScheduler** provides the TTE API. Internally, this module hides the pointers to the task tree root and to the callback functions as its static data. As the reaction to application calls of the TTE API functions, the module **TaskScheduler** builds the task tree by adding and deleting instances of the class **Task**.

The class **Task** has two fields that enable creating task trees, namely the pointer to the predecessor task and the list of the successor tasks in the task tree. Adding new task to a tree simply requires locating its predecessor task, setting new task’s predecessor field to the address of the predecessor task, and adding the address of the newly created task to the list of the successor tasks in the corresponding field of the predecessor task. Deleting a task from a task tree is more complex because deleting a given task means deleting itself and all its successors.

The API function **TS_ExecuteTopDown** starts task tree top-down execution by calling the class **Task** member function **executeTDinParallel** on the root task, which in turn recursively traverses the task tree from its top i.e. root task, towards all the task tree leafs. In each recursion, this function first calls the top-down callback function and then it starts new local OS threads for each of the current task’s successors by calling the function **CreateThread**. The simplified pseudo code of the function **executeTDinParallel** is the following:

```plaintext
executeTDinParallel(task) =
1   callback tdCallback(task.id)
2   for each successor in task.successors
3     CreateThread(executeTDinParallel, successor)
4   WaitForAllChildThreads()
```

The API function **TS_ExecuteBottomUp** is symmetrical to the function **TS_ExecuteTopDown**, and therefore is not further discussed here.

**3 Cilk Based TTE Architecture**

The main difference between the novel and the previous architecture, at the high-level architectural view, is that the novel architecture uses the Intel Cilk runtime library instead of the local OS (Windows/Linux) threads library. This evolutionary step was made by modifying the class **Task** such that the member functions **executeTDinParallel** and **executeBUinParallel** in the novel architecture simply delegate parallel task tree execution to new member functions **executeTDbyCilk** and **executeBUbyCilk**, respectively.

As such, this modification is transparent to the module **TaskScheduler**, thus the way it creates task tree remained untouched, as well as the way parallel top-down, and bottom-up, task tree execution takes place. Even more importantly, with this adaptation of the TTE architecture huge legacy applications (comprising millions of lines of code) may remain unchanged.

Thanks to Cilk’s expressiveness, the simplified pseudo code for the member functions **executeTDinParallel** and **executeBUinParallel** from the previous architecture, almost directly map to the pseudo code for new member functions **executeTDbyCilk** and **executeBUbyCilk**, respectively. Essentially, the call to the function **CreateThread** is replaced with the keyword **cilk_for** and the call to the function **WaitForAllChildThreads** is replaced with the keyword **cilk_sync**.

Once these mappings were introduced, synthesizing new **Task** member functions was rather straightforward. Consequently, the pseudo code of the function **executeTDbyCilk** is the following:

```plaintext
executeTDbyCilk(task) =
1   callback tdCallback(task.id)
2   for each scsr in task.successors
3     cilk_spawn scsr.executeTDbyCilk(scsr)
4   cilk_sync
```

In the pseudo code above the name **scsr** stands for the **successor task**. Similarly, the pseudo code of the function **executeBUbyCilk** is symmetrical and not further discussed here.

**4 Statistical Usage Testing**

We used the method published in [11] for statistical usage testing and operational reliability estimation of both TTE architectures. For the sake of completeness of this paper, we provide a brief overview of the method [11] in this section. We
A task \(\tau\) is a callback function that executes as a local OS thread. A task tree is an undirected radial (i.e. acyclic) graph of tasks \(TG\) whose nodes are tasks interconnected with links indicating predecessor-successor relations. A task tree comprises a set of \(k\) tasks \(TK = \{\tau_1, \tau_2, \ldots, \tau_k\}\), and a set of \((k-1)\) links \(L = \{l_1, l_2, \ldots, l_{(k-1)}\}\).

A task tree execution path, a path in a task tree or a trace, is a sequence of terminations of individual tasks \(\tau_1\tau_2\ldots\tau_k\) during the task tree execution. The length of this sequence is always equal to \(k\). A task forest is a series of task trees of the same complexity that is generated as a test suite. A test case is a single task tree execution described by the corresponding path.

Let \(r_t\) be a software product tree-reliability and \(r_p\) be a path-reliability. The product reliability \(r\) is obtained by multiplying the two:

\[
r = r_tr_p
\]

If \(r_t = r_p\), then:

\[
r = r^{1/2}
\]

Similarly, let \(M_t\) be a tree-confidence-level and \(M_p\) be a path-confidence level. The total confidence level \(M\) is the sum of the two:

\[
M = M_t + M_p
\]

If \(M_t = M_p\), then:

\[
M_t = M_p = M/2
\]

When given \(r\) and \(M\) we find the requested number of trees \(N_t\) and number of paths \(N_p\) for each tree as:

\[
N_t = N_p = \log_r^{1/2} (M/2)
\]

The total number of test cases \(N\) is obtained as a product of \(N_t\) and \(N_p\):

\[
N = N_tN_p = (\log_r^{1/2} (M/2))^2
\]

To generate a test suite with \(N\) test cases, we simply generate \(N_t\) task trees and execute them \(N_p\) times each. Therefore, the method of statistical testing and reliability estimation for applications based on task trees consists of the following steps:

1. Given the desired level of product reliability, calculate \(N_t\) and \(N_p\).
2. Generate \(N_t\) task trees.
3. Execute each task tree \(N_p\) times.
4. Check the coverage metrics report.
5. If the report shows poor coverage, return to step 2.
6. Report any unexpected behavior to the design and implementation team.

5 Experimental Evaluation

Statistical usage testing and reliability estimation method described in the previous section was used both to test TTE architectures and to compare their performance. The measure of the performance that was used in the experiments was the time in seconds that was needed to execute all the \(N\) test cases from the given test suite. For the sake of completeness of this paper we provide the execution time measurements data for individual test suits for both TTE architectures, and to enable easier performance comparison between the two architectures we provide the relative speedup (RS) calculation results. The relative speedup \(RS\) is defined as follows:

\[
RS = T_p/T_n
\]

where \(T_p\) is the test suite execution time for the threads based TTE architecture and \(T_n\) is the test suite execution time for the Cilk based TTE architecture.

All the measurements were conducted on the dual-core symmetric multiprocessor, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, with Windows7 Professional® 64-bit OS. The measurements data and the calculation results are given the following three tables:

1. Table 1: \(T_p\) values and test verdicts.
2. Table 2: \(T_n\) values and test verdicts.
3. Table 3: calculated \(RS\) values.

The columns of Table 1 are organized as follows. The column No Tasks contains the number of TTE tasks used to construct task trees, the column No Trees shows the number of tasks that may be constructed by the given number of TTE tasks, the next three columns within the common column Duration (in seconds) show test suites execution times for the three distinctive values of desired reliability \(r\) (0.9, 0.95, and 0.99), and the last column Verdict shows the test verdict.

As could be seen from Table 1, test suite execution time increases with the number of TTE tasks as well as with the value of desired reliability \(r\). As the last column indicates, TTE based on threads successfully passed all the tests.
The columns of Table 2 are organized in the same way as the columns of Table 1. Similarly, as in Table 1, test suite execution time increases with both number of tasks and the value of given reliability r. Again, the latter causes faster growth of the test suite execution time than the former. The last column of Table 2 shows that TTE based on Cilk successfully passed all the tests. Besides all these similarities, the measured values of test suites execution times are significantly different. Apparently, it takes much less time for the Cilk based TTE to complete all the tests than it does for the thread based TTE. This fact is more obvious from Table 3.

Table 3 shows the values of relative speedup RS of test suite execution on the new TTE architecture and on the previous one, for various numbers of tasks and desired operational reliability r figures. The columns of Table 3 are organized similarly as the columns of Tables 1 and 2. The additional row shows the average RS calculated over different values of desired operational reliability r, whereas the last column shows the average RS evaluated over different number of tasks (instead of the test suit verdict like in Tables 1 and 2). The bottom-right cell of Table 3 shows an overall RS average when evaluated over all the RS values.

According to the expectations the relative speedup RS increase both with the number of tasks for a given operational reliability r, as well as with the desired operational reliability r for a given number of tasks. Evidently, RS grows much faster with the desired r than with the number of tasks, which appears quite natural, because the needed testing effort increases much more with operational reliability r than with the number of tasks. As a consequence of these trends, both the average RS, calculated per task, increases with the number of tasks, and the average RS, calculated per given operational reliability r, increases with the value of r.

The overall average relative speedup is 10.88 (bottom-right cell in Table 3), which is a good result for the dual-core target machine we used in the experiments. Of course, it would be interesting to see how this average speedup of around 11x changes with the number of available cores in the target platform, and we have a plan to conduct more experiments in that direction in the future.

But, even more important fact that may be seen by observing the values of average RS in the last row of Table 3, is that overall average RS of 11x is actually much limited by the RS value of around 3x for r=0.9. For the values of r greater than 0.9, average RS goes up to 20x (for r=0.99). So after analyzing this data, one becomes aware that the novel TTE architecture provides scalable performance relative to given operational reliability r. This fact becomes even more obvious by observing Figure 3, which
illustrates the average relative speedup $RS$ as a function of a given operational reliability $r$.

![Fig 1. RS as a function of r](image)

6 Conclusion
This paper shows an approach to apply parallel programming techniques based on Intel Cilk Plus language on a class of service components within SOA based industrial systems. Based on the evolutionary principle, commonly assumed by designers and architects of large-scale mission-critical infrastructures, the approach introduces the Intel Cilk Plus runtime library instead of the conventional local OS threads library with minimal adaptations of the legacy TTE architecture. By switching from OS threads to Intel Cilk Plus tasks, the approach eliminates the three notorious shortcomings of threads-based parallel software, namely the burden of low-level details, the load imbalance, and the oversubscription.

The final result of the approach is the novel TTE architecture based on Intel Cilk Plus runtime library that executes TTE tasks as Intel Cilk Plus tasks rather than local OS threads, which was the case for the previous TTE architecture. Essentially, the novel TTE architecture uses finer grained parallelism, which yields better multicore CPU utilization. The novel TTE architecture exhibits the average relative speedup of 11x over the previous TTE architecture. Moreover, the average relative speedup is scalable relative to the operational reliability $r$, and goes up to 20x for $r=0.99$.

Acknowledgement
This work has been partly supported by the Serbian Ministry of Education & Science, through grants III 44009 and TR 32031, 2011-2014.

References: