Accredited Ranking SINTA 2
Decree of the Director General of Higher Education, Research, and Technology, No. 158/E/KPT/2021
Validity period from Volume 5 Number 2 of 2021 to Volume 10 Number 1 of 2026

Published online on: **http://jurnal.iaii.or.id**

JURNAL RESTI
(Rekayasa Sistem dan Teknologi Informasi)
Vol. 6 No. 2 (2022) 207 – 212 ISSN Media Electronic: 2580-0760

Critical Section Overhead Reduction for OpenMP Program by Nesting a Serial Loop to Increase Task Granularity of Parallel Loop

Adnan[1], Intan Sari Areni[2], Zulkifli Tahir[3]
[1,3]Department Teknik Informatika, Fakultas Teknik, Universitas Hasanuddin
[2]Department Teknik Elektro, Fakultas Teknik, Universitas Hasanuddin
[1]adnan@unhas.ac.id*, [2]intan@unhas.ac.id, [3]zulkifli@unhas.ac.id

Abstract

This paper presents a simple method to reduce performance loss due to a parallel program's massive critical sections of parallel numerical integration. The method is to transform a fine grain parallel loop into a coarse grain parallel loop which is nesting a sequential loop. The coarse grain parallel loop is by nesting a loop block to make task granularities coarser than that naive one. In addition to the overhead reduction, the method makes the parallel work fraction significantly larger than the serial fraction. As a result, nesting a serial loop within a parallel loop improves the parallel program's performance. Compared to the naïve method, which does not scale performance of parallel program of numerical integration, the nesting serial loop method scales a parallel program up to 3.26 times fold relative to its sequential program on quad-core processor. This result shows that the proposed method makes parallel program much faster compared to the naïve method.

Keywords: critical section, overhead reduction, OpenMP, task granularity, Parallel loop

1. Introduction

Parallel programming is a method to increase processor utilization in a parallel computer due to almost all computers with multicore integrates multiple individual processors and cache memory [1]. Ranging from small laptops to big servers in data centers has more than one processor. However, without parallel programming, only one processor can be utilized. The rest will be useless and have low resource utilization.

Parallel programming is not so easy to do. A programmer may deal with some problems in this regard among race conditions, or it could be performance lost due to high overhead, high rate of a cache miss, or load imbalance.

In this paper, our research concern is large overhead due to low parallel works to critical section ratio. We assume a rectangular integration method in the pi value estimation program. Although the program is simple, many other programs assume the same pattern. At least part of many of those programs has the same pattern. The pattern is a parallel R-W operation to a shared variable within a massively parallel loop. To avoid race condition due to parallel R/W operations, the naïve approach is to involve critical section. However, critical section contributes to large overhead in small granularity of work of parallel loop. In other words, the problem is the fine grain computation results in low performance of parallel program.

A detailed study that analyzes the effect of task granularity on the performance of parallel java programs is shown in [2]. The study takes advantage of a novel profiler that measures the granularity of every executed task [3].

There is an efficient method to overcome the problem in parallel R/W operations, such as the method of in OpenMP reduction clause [4], Cilk++ hyper objects [5], or MPI reduce [6]. All of the methods are to reduce overhead by eliminating critical sections.

The most efficient parallel method so far is lazy task creation [7]. Lazy task creation is a technique to increase task granularity. Its overhead is almost the same as the overhead of a function call. Cilk and StackThreads /MP [8] adopt the lazy task creation method.

Another method to increase task granularity in the work-stealing method is the extended work-stealing stealing strategies [9] and Dynamic multiple items work-stealing strategy [10] . Those methods increase the task granularity of stolen tasks and make processors

Accepted: 02-02-2022 | Received in revised: 29-03-2022 | Published: 20-04-2022

busy at works. These researchs improve the performance of the StackThread/MP fine grain task parallel library such that it outperforms the Cilk in the UTS benchmark and has comparable performances in other benchmarks.

Controlling the granularity of parallel task is an interesting idea. One research of this is an algorithm to control task granularity [11]. However, currently we argue that controlling the task granularity manually is better than the automatic one for some reasons.

The transformation of a single sequential loop into a pair of the nested sequential loop is known as loop blocking or tiling [12]. These practices are originally to take benefit from cache hierarchy. Our research aims to have parallel slackness and optimal task granularity and may benefit from the cache hierarchy. Because the multicore processor incorporates multiple processors and multiple levels of cache memory, our method is a simultaneous practice to exploit both parallelism and locality in a multicore processor at once.

Our approach is quite different from them, as our method reduces the overheads by transforming a single parallel loop into a parallel loop and nested sequential loop pair. In this research, the specific purpose of the loop transformation is to increase the grain size of parallel works. In addition, this method to increase task granularity is quite different from Cilk language and StackThreads/MP which are based on Lazy task creation. All of the previous research mentioned above are more appropriate for recursive task structures. While the method in this study is suitable for iterative task structures.

2. Research Methods

The rectangle rule of the numerical integration program is chosen as a base of two benchmark programs in the first step. The first program is implemented with the naïve method, and the second is the improved version. We evaluate the sequential execution time TS of the benchmark programs and then evaluate parallel execution.

2.1. Rectangular Rule of Numerical Integration

In this paper, we propose a simple method that makes tasks granularity large enough to reduce the amount of synchronization. As a case study, we consider a parallel program commonly used in teaching. This program performs numerical integration for the estimated value of pi.

As the first step of the research we evaluate the serial implementation of the rectangular integration for pi number estimation. The implementation is based on equation 1. In the equation, x is the midpoint of each rectangle i.

$$pi = \sum_{i=0}^{n} \frac{4.0}{(1+x^2)} \Delta x \quad (1)$$

From the serial implementation, we obtain serial execution time data T_S which is used as a baseline. The baseline will be used to evaluate parallel work overheads and speedup. In further discussions, we refer to the T_S while mentioning works. We execute a time command in the MSYS2 command prompt to obtain the serial execution time (real part). We obtain the real part, which is the wall time, from the time command in MSYS2.

Serial algorithm listing is shown as the following:

Serial Algorithm

```
Input: N
Output: pi
BEGIN
 NUMBER = 2000000000;
 NUMBER i
 NUMBER step = 1.0/N
 NUMBER x = 0.0
 NUMBER sum = 0.0
 FOR i = 0 to N STEP 1 DO
     X = (i + 0.5) * step;
     Sum = sum + 4.0/(1 + x^2);
 END FOR
     pi = sum * step;
 END
```

The straightforward implementation of OpenMP is used for the naïve parallel version of the test program. The OpenMP implementation means that the program is multithreaded. In the multithreaded program, we can assign the number of threads executing parallel regions. Parallel region refers to the structured block of code which follows the pragma omp parallel in an OpenMP program.

Equivalent parallel algorithm listing is shown as in the parallel Algorithm 1.

Naïve Parallel Algorithm 1 (pi_wcs)

```
Input: N
Output: pi
BEGIN
 NUMBER N = 2000000000;
 NUMBER i
 NUMBER dx = 1.0/N
 NUMBER x = 0.0
 NUMBER sum = 0.0
 PARALLEL FOR i = 0 to N STEP 1 DO PRIVATE(x)
     X = (i + 0.5) * dx;
     CRITICAL SECTION DO
       Sum = sum + 4.0/(1 + x^2);
     CRITICAL SECTION END
 END PARALLEL FOR
 pi = sum * dx;
 END
```

The naïve parallel algorithm, which presented as in parallel algorithm 1, looks very similar to that of the serial one. We add a few lines of OpenMP directives and clauses to transform the serial program to be parallel. Behind the scenes, the compiler translates the OpenMP directives and following structured blocks into multithreaded codes. The compiler links additional works from the OpenMP library to our code. We refer

DOI: https://doi.org/10.29207/resti.v6i2.3848

to these additional works while mentioning overhead. As a result, the for-loop that follows #pragma omp parallel is shared by multiple threads and executed simultaneously.

The naïve version of the parallel program for the rectangular rule of the pi program performs worse than its serial version. Work overhead, which is additional works in a parallel program, is significantly larger than the granularity of parallel works. Fine grain parallel works always result in significant overhead. The work overhead *wo* is quantitatively defined in equation 2. In equation 2, $T_P(1)$ is the parallel execution time with the processors equal to 1. According to equation 2, we calculate the work overhead by subtracting the work T_S from the $T_P(1)$.

$$T_p(1) = T_S + wo \quad (2)$$

Parallel work overhead is similar to the work overhead, but we scale the parallel execution time by the number of processors.

Our research is aimed to reduce the work overhead wo. A method to reduce the overhead is to make task granularities are large enough such that the ratio B of computation to critical section is significantly large. In the naïve implementation, this ratio is equal to 1. We propose this method because the most significant overhead is due to the time for starting and finishing the critical section. Making coarse the task granularity is usually adopted to reduce the degree of parallelism (N). Each iteration is a parallel task in the naïve parallel program (Parallel Algorithm 1). Each parallel task contributes to work overhead and also synchronization overhead. Therefore, we reduce O(N) to O(N/B) overhead. It means that the less parallelism is, the less overhead to manage parallel tasks is. This method is appropriate for OpenMP because the OpenMP is not a language to implement lazy task creation to increase task granularities.

Parallel algorithm 2 shows the method to reduce the overhead. The parallel algorithm 2 that optimizes the program cost by nesting a sequential loop within a parallel loop. In the parallel algorithm 2 the critical number section is now reduced to N/B. In the algorithm, each thread allocates additional memory for localsum and ii variables from its stack as a private memory.

2.2. Experimental Setup and Configuration

We conducted some experiments to measure serial execution time and parallel execution time for some implementations. Based on the serial program execution time and parallel program execution time, we analyze work overhead and speed up, which shows how faster the parallel program is related to its serial version. For the parallel program, we set the number of threads to 1, 2, 4, and 8 (SMT mode). In addition, we scale the parameter task grain size B to analyze its effect on the work overhead *wo*.

```
Parallel Algorithm 2 (pi_nl)
Input: N, B
Output: pi
BEGIN
NUMBER N = 2000000000;
NUMBER pi = 0.0;
NUMBER i;
NUMBER dx = 1.0/N;
NUMBER x = 0.0;
NUMBER sum = 0.0

PARALLEL FOR i = 0 TO N STEP B DO PRIVATE(x)
    PRIVATE NUMBER ii;
    PRIVATE NUMBER localsum = 0.0;
    FOR ii = i TO i+B STEP 1 DO
        IF ii < N
            X = (ii + 0.5) * dx;
           localsum = localsum + 4/(1+x^2);
        ELSE BREAK //inner loop
    END FOR
    CRITICAL SECTION BEGIN
        sum = sum + localsum;
    CRITICAL SECTION END
END PARALLEL FOR
pi = sum * dx;
END
```

Table 1 shows the specifications of the hardware in this research. The hardware is a laptop equipped with 8th generation Core i7 Quad Core processor.

Tabel 1. Hardware Specification

Hardware	Specification
CPUs	Gen 8th Core i7 4 Cores, 2 Threads/Core 3 GHz, HyperThreading Enabled
RAM	DDR4 16 GB
Storage	SSD 512 GB SATA

Table 2 shows the specifications of the hardware. All versions of test case programs are compiled with compiler optimization switch -O2. This compiler option ensures that the compiler generates an optimized serial program and does so for all the parallel versions.

Tabel 2. Software Specification

Software	Specification
Operating System	Windows 10 64 bits Home Edition
Software distribution and development Platform	MSYS2 For Windows 64bits GCC OpenMP Library

3. Results and Discussions

This section presents the results of our experiments. Our experiments measure serial programs execution time and also parallel programs execution times.

The first parameter to be discussed is accuracy. Tthis parameter is the most important parameter before parallel program performance. The rapid execution time of a parallel program is useless if the program is not accurate. In this case, accuracy is the similarity between

DOI: https://doi.org/10.29207/resti.v6i2.3848

the estimated values of pi between serial and parallel programs. The accuracy result is shown in Table 3.

Tabel 3. The Result of Accuracy

Number of threads	Pi number estimation		Accuracy (%)
	Serial program	Parallel program	
1	3.141592653589	3.141592653589	100
2		3.141592653589	100
4		3.141592653589	100
8		3.141592653589	100

From the results of the experiments, we make work overhead and speedup analysis.

3.1. Serial programs execution time

The serial execution time is displayed in Table 4. The results show that the proposed method improves the sequential execution time of the benchmark.

Table 4. Sequential Execution Time

Method	Sequential execution time
Single loop	2.581 secs
Nested loop	2.350 secs

3.2. Parallel execution time

Parallel program execution time is the program's execution time that is compiled with OpenMP directives and APIs. However, it is executed with one single thread. The results show in Table 5.

Table 5. Parallel Execution Time

Number of threads	pi_wcs	pi_nl
1	30.338 secs	2.567 secs
2	50.983 secs	1.321 secs
4	106.621 secs	0.829 secs
8	90.343 secs	0.791 secs

3.3. Overhead and Scalability analysis

This section presents work overhead and scalability analysis. The work overhead analysis is derived from the parallel program execution time using a single thread ($T_P(1)$) and serial program execution time T_S as denoted in equation 2.

Figure 1 compares work overhead in both the proposed and naïve parallel programs. The proposed method results in the numerical integration are better than the naïve one. It can be seen from the lower $T_P(1)$. Execution of fine grain parallel works with critical sections in a loop contributes to the large overhead.

Figure 2 shows overhead analysis when a single process single thread executes the parallel program with fine grain parallel works. In a parallel program with fine grain parallel works, there are a significant fraction of sequential codes and a relatively small fraction of parallel codes. According to Amdahl's Law [13], such a parallel program has a low speedup.

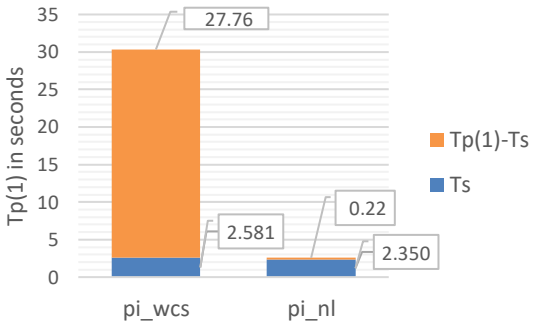

Figure 1. Overhead comparison of two methods

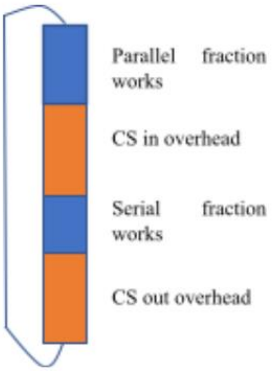

Figure 2. Overhead Illustration of Fine Grain Parallel Works in Single Thread Mode

Figure 3 shows overhead analysis when multiple threads execute a parallel program with fine grain parallel works. In this case, the overhead (wait time) due to the critical section increases as the number of threads increases.

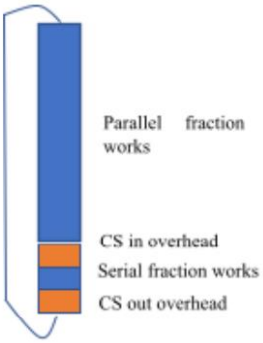

Figure 3. Overhead Illustration of Fine Grain Parallel Works in Multiple Threads Mode

As the grain size of parallel works increases, the overhead due to critical section decreases. In addition, this method has parallel fraction increases. Figure 4 depicts an improvement in parallel works fraction and lowers the overhead of the critical section. In the multiple threads scenario presented in Figure 5, making the granularity parallel works increase effectively to reduce overhead (wait time and critical sections).

DOI: https://doi.org/10.29207/resti.v6i2.3848

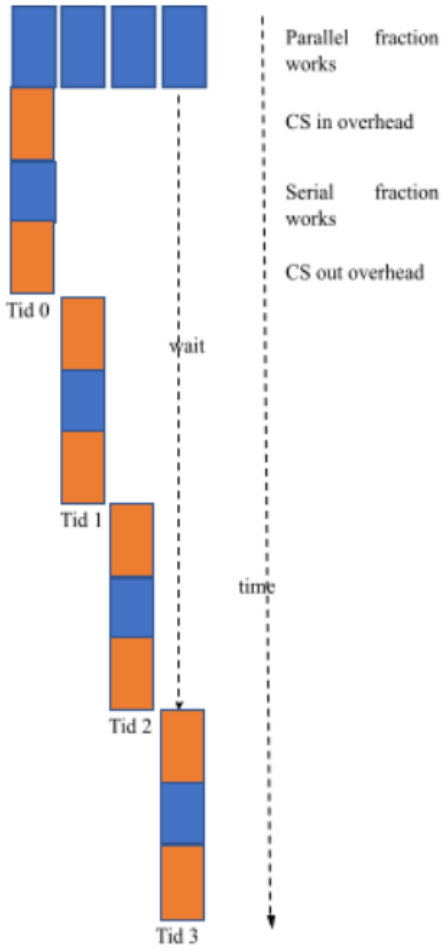

Figure 4. Overhead Reduction of Coarse Grain Parallel Works in Single Thread Mode

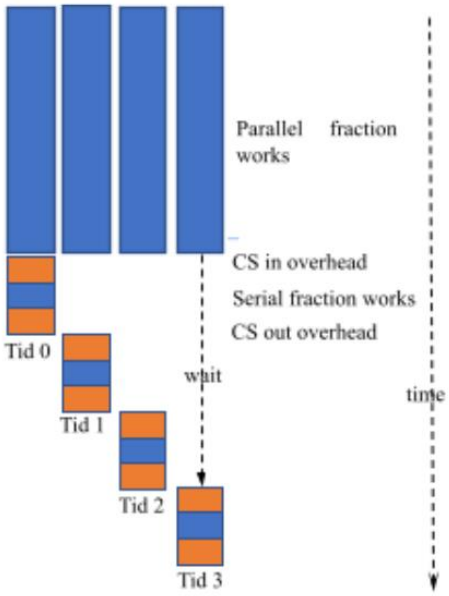

Figure 5. Overhead Reduction of Coarse Grain Parallel Works in Multiple Thread Mode

Figure 6 shows an analysis of how the overhead increases as the number of threads increases. However, the overhead rate of the proposed method is significantly lower than the naïve one.

Furthermore, we evaluate the parallel work overhead of the proposed method with the number of threads equal to one but with the grain size scaled from 1 to 106. Figure 7 presents work overhead analysis for the benchmark program with the proposed method (pi_nl). As the figure shows, the overhead decreases as the grain size increase. We can see that task granularity B of 105 iterations diminishes the work overhead. In general, increasing the grain size will lead to better efficiency. However, the grain size should not be too large to avoid either load imbalance or lack of parallelism.

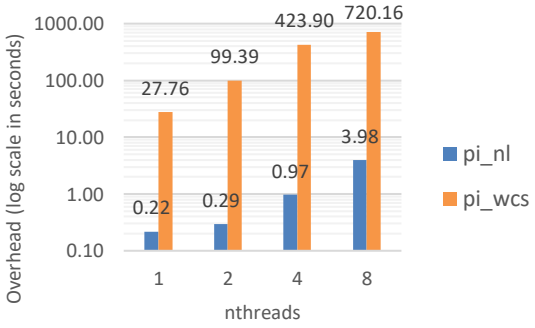

Figure 6. Work Overhead Rate to the Number of Threads

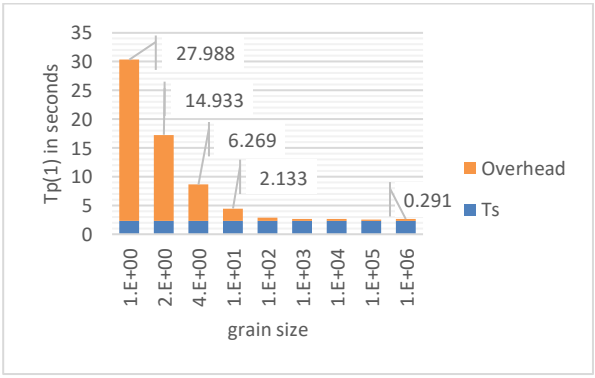

Figure 7. Overhead Rate to the Grain Size

A scalability analysis is depicted in Figure 8 that shows the proposed method obtains the parallel program of numerical integration scales with good performance. Although its speedup is low when the number of threads equals 8, this fact is known well. The cause is not the program, but the program is executed in SMT mode [14]. As presented in [15] , SMT mode may slightly improve performance in non-uniform workloads. However, in this research, the benchmark program has a uniform workload. In contrast with the naïve method, the method results in large overhead such that the numerical integration performs poorly and does not scale at all.

DOI: https://doi.org/10.29207/resti.v6i2.3848

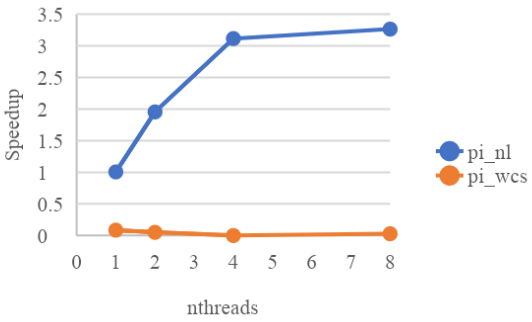

Figure 8. Speed Up of Parallel Numerical Integration Programs Using Naïve Method and Nested Loop Method

4. Conclusion

This research reduces the overhead of massive critical sections such that a parallel program has a better performance than the naïve method. The contributed of the method is performance improvement such that parallel program performs faster than the naïve parallel program. The method to improve the performance is to make the grain size of parallel works much larger than the number of critical section operations. This research shows the method proven to be useful to improve the performance of parallel programs such as parallel numerical integration using rectangular rule and other parallel programs with the same pattern.

References

[1] S. Najem N and S. Sami I, "Multi-core Processor : Conceppts And Implementations," *International Journal of Computer Science and Information Technology*, pp. 01-10, 2018.

[2] A. Rosà, E. Rosales and W. Binder, "Analysis and Optimization of Task Granularity on the Java Virtual Machine," *ACM Transaction on Programming Languages and Systems*, vol. 41, no. 5, p. 47, 2019.

[3] A. Rosà, E. Rosales and W. Binder, "Analyzing and Optimizing Task Granularity on the JVM," in *Association for Computing Machinery*, New York, 2018.

[4] J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *Proceedings of First European Workshop on OpenMP*, 1999.

[5] M. Frigo, P. Halpern, C. E. Leiserson, Lewin-Berlin and Stephen, "Reducers and other Cilk++ hyperobjects," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009.

[6] K. Hasanov and A. Lastovetsky, "Hierarchical redesign of classic MPI reduction algorithms," *The Journal of Supercomputing*, vol. 73, no. 2, pp. 713-725, 2017.

[7] E. Mohr, D. Kranz and R. Halstead, "Lazy task creation: a technique for increasing the granularity of parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 264-280, 1991.

[8] K. Taura, K. Tabata and A. Yonezawa, "StackThreads/MP: Integrating Futures into Calling Standards," *ACM SIGPLAN Notice*, vol. 34, no. 8, pp. 60-71, 1999.

[9] Adnan and M. Sato, "Efficient Work-Stealing Strategies for Fine-Grain Task Parallelism," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, Anchorage, 2011.

[10] Adnan and M. Sato, "Dynamic Multiple Work Stealing Strategy for Flexible Load Balancing," *IEICE Transactions on Information and Systems*, vol. E95.D, no. 6, pp. 1565-1576, 2012.

[11] A. Fonseca and B. Cabral, "Controlling the granularity of automatic parallel programs," *Journal of Computational Science*, vol. 17, no. 3, pp. 620-629, 2016.

[12] J. M. Cardoso, J. G. F. Coutinho and P. C. Diniz, "Chapter 5 - Source code transformations and optimizations," in *Embedded Computing for High Performance*, Boston, Morgan Kaufmann, 2017, pp. 137-183.

[13] J. L. "Gustafson, ""Amdahl's Law"," in *"Encyclopedia of Parallel Computing"*, "Springer US", 2011, pp. "53--60".

[14] C. Jung, D. Lim, J. Lee and S. Han, "Adaptive Execution Techniques for SMT Multiprocessor Architectures," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Chicago.

[15] Adnan, D. K. Oktahidayat and A. Achmad, "Performance Improvement with Non-Uniform Loads on SMT Processors," in *5th International Conference on Computing Engineering and Design (ICCED)*, 2019.

DOI: https://doi.org/10.29207/resti.v6i2.3848