Check for updates

Received: 2 August 2022 | Revised: 12 December 2022 | Accepted: 22 December 2022

DOI: 10.1002/cpe.7614

SPECIAL ISSUE PAPER

WILEY

Approximate execution and grouping of critical sections for performance-accuracy tradeoff

Zuhal Altuntaş | Sanem Arslan | Betul Boz

Computer Engineering Department, Marmara University, Istanbul, Turkey

Correspondence
Zuhal Altuntaş, Computer Engineering Department, Marmara University, Istanbul, Turkey.
Email: zuhal.altuntas@marmara.edu.tr

Summary

Approximate computing enhances the performance and energy efficiency of applications, while still achieving acceptable accuracy. Some of the multithreaded applications can tolerate the accuracy loss when critical sections are approximately executed, which in turn will eliminate the synchronization overhead of these applications and increase their performance. In this study, our objective is to explore the behavior of the critical sections and selectively skip the ones yielding performance improvements with an acceptable accuracy loss. We have observed the behavior of 62 critical sections of 4 selected applications. We have grouped them depending on their effects on program execution and skipped or approximated them accordingly. Our experimental study indicates that skipping 76% of the critical sections offers 2.5× performance gain with 16% accuracy loss for Raytrace whereas 1.4× performance improvement with 17% accuracy loss is obtained for Radiosity on average when 36% of the critical sections are skipped. For Water_NSquared the performance gain is 1.1× with 9% accuracy loss on average with 79% of critical sections skipped. For the Ocean_CP application we see a performance improvement of 1.6× with accuracy loss 1% when we skip 38% of critical sections.

KEYWORDS

accuracy, approximate computing, critical section, parallel application, performance

1 | INTRODUCTION

As predicted by Moore's law, the capabilities of modern computer systems have increased significantly as a result of the scalability of transistors. Thus, a high number of processors can be placed on a single chip to increase parallelism.

However, this poses performance and energy efficiency issues since application programmers tend to run highly resource-intensive application threads with strict correctness requirements. Approximate computing, which trades off accuracy for performance gain, has attracted the attention of researchers for this challenge in recent years.[1,2] Some applications may naturally be tolerant to inaccuracy. As an example, video encoding/decoding or image processing applications can tolerate the quality loss in compression or in the output for performance improvement. Some machine learning algorithms make decisions based on probabilistic models that do not require 100% accuracy. Furthermore, a number of scientific applications (such as iterative solvers) can converge to a correct solution with more iterations in case of a problem or they can provide approximate results.

Researchers have focused on approximate computing at different levels, such as skipping some of the tasks[3] or loop iterations,[4] reducing precision,[5] omitting some of the memory accesses,[6] using faulty or inexact hardware.[7,8] Some of these techniques rely on additional hardware usage for approximation, which may be a limitation in terms of cost. On the other hand, a set of them is implemented at the software-level by skipping some of the program statements. An important problem in the latter one is deciding which parts of the software should be approximated. Such

 © 2023 John Wiley & Sons, Ltd.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

pre-processing may increase the execution time of the approach although it provides performance gain in the approximation phase. Therefore, focusing on the program regions, which leads to performance bottleneck, might be a promising approach for approximation.

In multi-threaded programs, a set of threads can work on different regions of an application and they may require to communicate over shared memory. The *mutual exclusion* property ensures that each thread enters the code regions which update the shared data sequentially and those regions are called *critical sections*.[9] Thus, the execution of parallel threads is serialized in these regions, which may lead to a performance bottleneck. In that sense, critical sections can be a promising candidate for the approximation to provide performance gain.[10] There are a set of studies in the literature that skips a subset of critical sections while ensuring accuracy at an acceptable level.[11-15] However, these approaches require inserting a considerable amount of code into the application to determine when those critical sections should be skipped, and this might increase the execution time or energy consumption of the applications.

In this work, we propose to selectively skip a set of critical sections to eliminate the synchronization overhead of multi-threaded applications. Although the critical sections can be vital for some programs (and we cannot think of the elimination of all of them), we observe that a number of applications can produce acceptable results by skipping some of them. We monitor 2.5× performance improvement with 16% accuracy loss, 1.4× performance improvement with 17% accuracy loss, 1.1× performance improvement with 9% accuracy loss and 1.6× performance improvement with 1% accuracy loss for Raytrace, Radiosity, Water_NSquared and Ocean_CP, respectively. We obtain these results when we skip a set of critical sections, which do not cause application failure with a crash or deadlock, or cause significant performance degradation. In our study, we partition the critical sections into three groups; (i) the ones that can be totally skipped, (ii) the ones that can be partially skipped, and (iii) the ones that must be exactly executed. By skipping a critical section, we mean not executing that particular section rather than just skipping the *lock* statements.

In the first group, we experimentally observe that skipping such critical sections does not yield an output quality loss of more than 5%. As an example, critical sections that update a global variable by simple mathematical operations such as increment are good candidates for this group. In the second group, skipping all of them may lead to marginal corruption in the output, therefore, they can be partially skipped. We do not execute the critical sections for a set of randomly chosen threads for this group. As an example, the critical sections that distribute the overall job among threads are in this group. In the last group, they have to be executed exactly since skipping them may lead to a program crash due to an error such as a segmentation fault. As an example, the critical sections that allocate memory for necessary calculations are in this group.

To group critical sections, we need to profile the applications a priori. The candidate critical sections are then executed at random rates between 50% and 99%. In our experiments, we start with the lowest execution rate (i.e., 50%). If the quality loss in the output is at an acceptable level (e.g., if at least 70% of the output is similar to the original output), we did not change the execution rate. Otherwise, we increase this rate to allow the critical sections to be executed by more threads or more times. We use Raytrace, Radiosity, Water_NSquared and Ocean_CP applications from SPLASH-2 benchmark suite[16] to evaluate the effectiveness of our approach. All of the applications contain diverse and a high number of critical sections. Specifically, there are 12 critical sections (executed 149,770 times with 64 threads) for Raytrace and 37 critical sections (executed 2,502,480 times with 32 threads) for Radiosity, 9 critical sections (executed 8,392,944 times with 64 threads) for Water_NSquared and 4 critical sections (executed 24,860 times with 64 threads) for Ocean_CP in total. Our experimental evaluation validates our approach by presenting 2.5× performance gain with 16% accuracy loss on average by skipping 76% of critical sections for Raytrace, 1.4× performance gain with 17% accuracy loss on average by skipping 36% of critical sections for Radiosity. Performance gain for the Water_NSquared application is 1.1× where the accuracy loss on average is 9% when we skip 79% of critical sections. Performance gain for Ocean_CP application is 1.6× where the accuracy loss on average is 1% when we skip 38% of critical sections. Our approximation approach requires a minimal code change in application source code with at most 1% update.

This article is an extension of the work done in Reference 17. The extensions are the experimental results (both profiling step and accuracy and performance results) for both Water_NSquared and Ocean_CP applications, and Appendix which contains example critical sections from all three classes of critical sections.

The article is organized as follows: Section 2 explains the works in literature; Section 3 describes our methodology and how it is applied; Section 4 gives the results and discusses them; and Section 5 concludes our article.

2 | RELATED WORK

Approximate computing has been implemented at different levels of hardware, architecture or software.[2] Approximate computing at the hardware level can be achieved by using voltage scaling or using inexact hardware. Kulkarni et al. propose a new multiplier architecture using 2×2 inaccurate building blocks and represent multiplications in fewer bits.[7] Saadat et al. also propose approximate multipliers and discusses a minimally based approximate integer multiplier using approximate log function.[8] Osorio et al. use truncated multipliers for approximate multiplication.[18] Our proposed work differs from these works as it does not require any special hardware; hence, it is easier to implement.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

At the architectural level, the use of neural networks and memory access skipping are examined. Zhang et al. find less important neurons in a neural network and approximate them.[19] Yazdanbakhsh et al. propose an implementation that predicts the value of miss operations, and if they can be safely approximated.[20] Karakoy et al. propose an approach to skip data accesses such that performance benefits are maximized and error is bounded by user-defined level.[21] Brand et al. propose "anytime" floating point instructions that compute only a predefined number of most significant bits.[22] The number of computed bits is embedded into the instruction and the resulting floating number still has the same number of precision bits.

Software level approximation techniques include loop perforation, memoization, precision scaling and data access approximations. Sidiroglou et al. identify loops that are suitable for perforation and find all possible tuning combinations.[4] Alvarez et al. use memoization to map similar inputs to similar outputs and apply this technique to multimedia applications.[6] Rubio et al. decrease the precision of floating-point values for performance gain within a reasonable error rate.[5] Kim et al. use a subset of data to decrease the number of memory accesses.[23]

Approximate computing can be applied to multiple levels in an application simultaneously as well. The study in Reference 24 proposes a system that combines existing approximation techniques at different levels. Their study automatically tunes approximation based on the accuracy requirements of the application.

Approximate computing is studied in parallel programs at the software level to reduce synchronization overheads as well. Renganarayana et al. propose an approach that identifies an acceptability criterion, then relaxes the synchronization points.[15] If the result is not acceptable by the criterion, they compute the exact version to ensure the result is of the specified quality. Khatamifard et al. directly execute critical sections without locking, and then check if there were any conflicts.[14] In the case of conflicts, they execute a recovery mechanism if the error is significant. There are studies that try to wait for lock acquisition for a specified time.[12,13] If they cannot acquire the lock within a given time, they will not wait anymore and execute the critical section. The specified times are changed according to the error rate of the computations. Both these studies use relaxed synchronization which means they loosen the synchronization requirements of parallel programs. On the other hand, Akram et al. propose three approximate lock techniques that decide to wait for the lock or skip the critical section altogether depending on different conditions; namely, the number of threads waiting for the lock, the wait time of the current thread and the predetermined rate.[11] Our work is different from theirs in the sense that they examine the critical sections locked-based, which means they approximate the acquisition of all instances of a lock while our work is critical section based. We decide to approximate a critical section only after considering its effect on the output solely without looking at the other critical sections that are protected by the same lock. Also, our work does require little to no knowledge about the application or the system in use.

3 | METHODOLOGY

Our study aims to reduce the time spent on critical sections as synchronization overhead. Therefore, we choose a set of programs from the SPLASH-2 benchmark suite which contains parallel applications with critical sections.[16] Among the applications presented in the suite, we choose the ones with the high number of critical sections and with diverse variety. Out of 12 critical sections of the Raytrace application, there are critical sections that deal with pointers and global indices necessary for the program executions. Out of 37 critical sections of the Radiosity application, there are critical sections that, besides using pointers and updating indices, make calculations for energy and RGB values for the resulting image. Out of 9 critical sections of the Water_NSquared application, there are critical sections that assign jobs to threads and calculate potentials. Out of four critical sections of the Ocean_CP application, there are critical sections that update global movements and error.

Raytrace and Radiosity programs are used in computer graphics for image rendering purposes. The Raytrace application uses a tracing algorithm to convert a three-dimensional scene to a two-dimensional image.[25] The Raytrace algorithm is widely used in game graphics. The GPU manufacturer NVIDIA has manufactured a graphics card with cores only for ray tracing.[26] The Radiosity application finds the light distribution of surfaces with defused light reflections.[27] Since they are both used in real-time applications, it is important that they are executed in a short time and accuracy up to some point can be traded for performance improvement.

Water_NSquared finds the forces and potentials of water molecules when they are in liquid state[25] using Gear's method.[28] Ocean_CP examines currents in large-scale ocean movements.[29] Both of these applications are scientific applications that are suitable for approximate computing as they use iterations to converge to a solution.

It is crucial to decide which critical sections should be approximated for selected applications. We profile the critical sections manually based on how important they are for the program execution and output accuracy. According to their effect on the program, we execute them exactly or approximately; or we skip them altogether. To group the critical sections appropriately for our study, we use the following steps.

For each critical section, we first skip it and execute the program without it to examine the program behavior. Here, by skipping a critical section, we mean eschewing the critical section altogether, rather than skipping only the lock statements protecting the critical section. There are three ways skipping a critical section can affect program output:

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1. Program may fail. This indicates that the skipped critical section is vital for the program termination. Therefore, we conclude that the program cannot be executed without this critical section successfully. This type of critical sections are marked as inapproximable. Inapproximable critical sections are executed exactly without any approximation techniques in test runs.
2. Program may terminate normally with minimal to no output degradation. If the accuracy degradation of the program output is within an acceptable range after skipping the critical section, it indicates that this critical section is not important. This type of critical sections are marked as unimportant. In test runs, we do not approximate this type of critical sections. Instead, unimportant critical sections are totally skipped.
3. Program may terminate normally with significant output degradation. This last case is an indication that the critical section is important but not crucial. This means that the program termination does not rely on this group of critical sections but the output highly depends on them. This type of critical sections were marked as candidate critical sections and examined in more detail.

After this step, candidate critical sections are executed at different rates changing between 50% and 99% to see if any approximation rate gives a satisfactory result. If all execution rates still show serious degradation in output accuracy, then this critical section is classified as an inapproximable critical section to be executed exactly in test runs. If, however, one of the rates applied gives adequate results, then these critical sections are marked as approximable and this approximation rate is applied at test runs. Listing 1 shows how to execute the critical section according to the given rate. It requires minimal code changes for the implementation of rate-based execution. This method does not execute critical sections at the same time, hence mutual exclusion is still ensured. The overall selection strategy for approximating critical sections is given in Figure 1. With this conservative selection strategy, we make sure that the accuracy degradation of the final approximation is tolerable and there is no need for additional code to compensate for quality loss.

```
1 if (rand() $>$ rate) {
2     pthread_mutex_lock(&lock);
3         // Critical section to be executed
4     pthread_mutex_unlock(&lock);
5 }
```

Listing 1: Code change to execute critical sections according to given rate

After we decide on the critical section groups, we skip unimportant critical sections, execute approximable critical sections at the predetermined rate and execute inapproximable critical sections exactly to see if it causes any failures. For one case in the Raytrace application, we faced a crash caused by an approximated critical section. For these type of cases, we add minimal failure avoidance code that does not affect the output quality before the critical section is executed.

Figure 2 compares the original output of the Raytrace application to the approximated version with all unimportant critical sections skipped and approximable critical section executed at the rate of 90%. The difference between the images is not significant and within an accepted limit (that is less than 20%).

4 | EXPERIMENTAL STUDY

4.1 | Experimental setup

The experiments are run on a 2 socket 10 core Intel Xeon 2.50 Gz system with 2 threads per core. The memory on the system is 64 GiB. L1 cache is 640 KiB, L2 cache is 2560 KiB and L3 cache is 25 MiB.

All of the applications we use from the SPLASH2 benchmark suite are executed with their native inputs. The applications are compiled using the makefiles provided in the SPLASH2 benchmark without any modifications.

Raytrace is a widely used image processing application (e.g., NVIDIA has manufactured graphic cards with special cores for ray tracing[26]). It has a total of 12 critical sections and the time spent in each of them is shown in Table 1. Critical sections 2 and 8 are the most time-consuming ones. When we run the program with its default options, the critical sections 1, 3, 5, and 6 are not executed. These critical sections are not shown in the table. We execute the program with different numbers of threads from 1 to 64. The critical sections operate to accomplish many different tasks such as memory operations and mathematical increment operations on global indices.

Radiosity is also an image-processing application. It has a total of 37 critical sections. Table 2 shows the time spent in each critical section for the application. As seen from the table, critical sections 22 and 24 are the most time-consuming ones. When the program is executed with its default options, the critical sections 8, 9, 11, 17, 19, 20, 21, and 23 are not executed. These critical sections are not shown in the table. We run the application

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

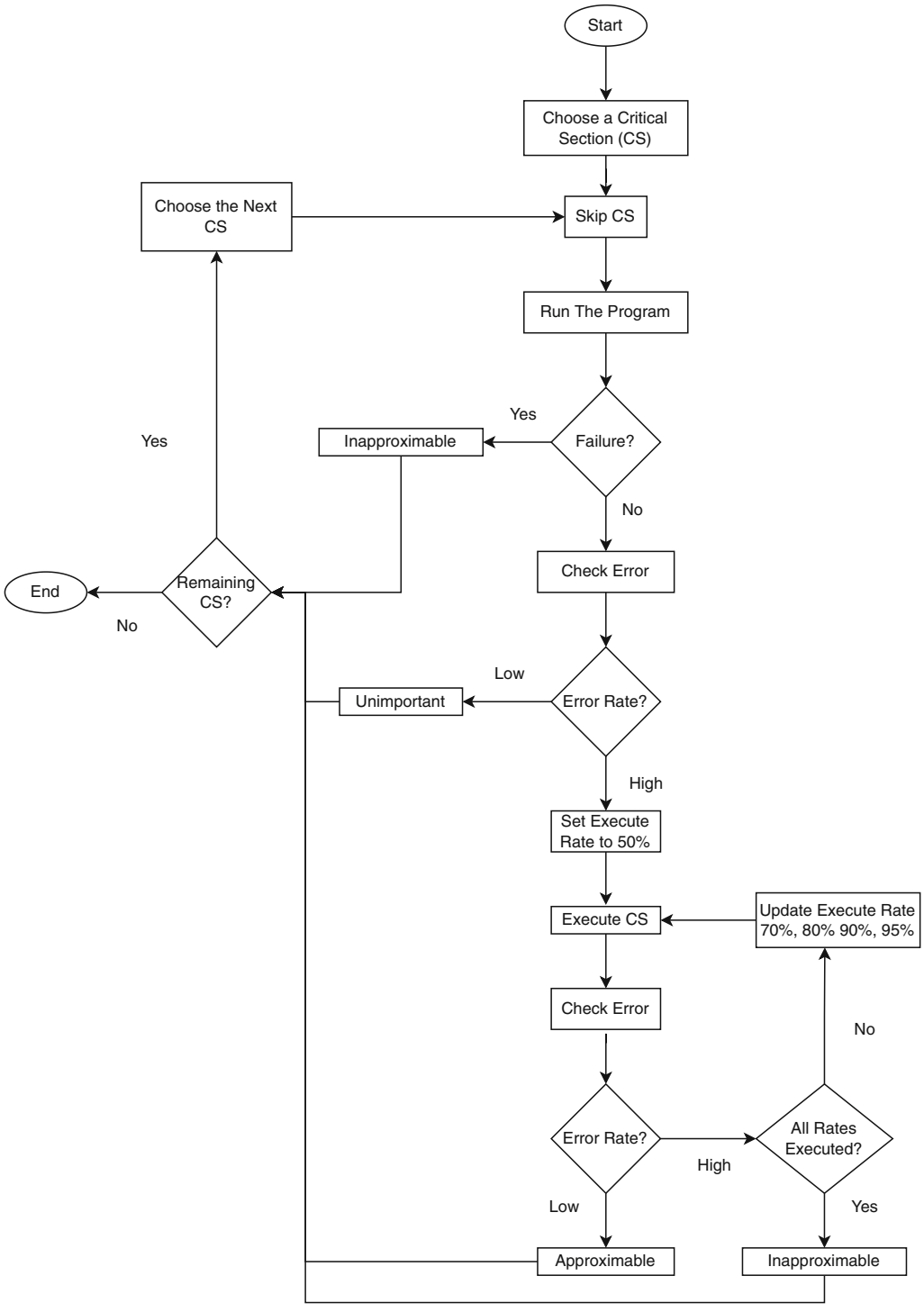

FIGURE 1 Flow chart to group critical sections.

with different numbers of threads from 1 to 32. When we executed the application with 64 threads, it did not terminate. The critical sections protect tasks such as global variable incrementation, memory operations, and RGB calculation.

The Water_NSquared application computes the forces and potentials in a system with water molecules in N^2 time complexity. It has nine critical sections in total. How much time is spent on these critical sections is shown in Table 3. Critical sections 2, 3, and 4 are the ones that consume the most time. All of the critical sections are executed with the default options. The application is executed with 1 to 64 threads. Tasks such as distributing the jobs to different threads and updating global potentials are protected by these critical sections.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

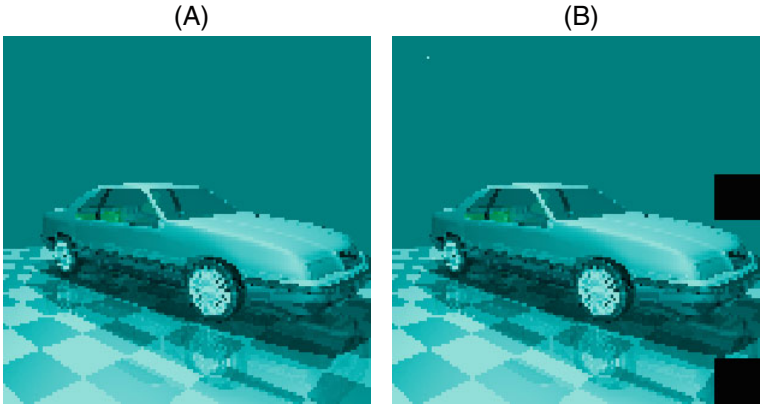

FIGURE 2 Raytrace output results. (A) Original (B) approximated with rate 90%

TABLE 1 Critical section execution times in seconds versus number of threads for the Raytrace application.

CS \ # of threads	1	2	4	8	16	32	64
Critical Section 2	0.045045	0.044946	0.044148	0.045839	0.048992	0.049654	0.051768
Critical Section 4	0.001184	0.003386	0.009472	0.021401	0.034236	0.039257	0.029624
Critical Section 7[a]	0.000001	0.000003	0.000012	0.000029	0.000119	0.000162	0.000375
Critical Section 8	0.043295	0.046181	0.062294	0.151837	0.310622	0.465998	0.855143
Critical Section 9	0.012892	0.013926	0.018651	0.045553	0.093984	0.137614	0.245827
Critical Section 10	0.000687	0.000725	0.000924	0.002419	0.004463	0.007591	0.012793
Critical Section 11	0.012557	0.014516	0.021480	0.048858	0.090619	0.145662	0.233294
Critical Section 12	0.000850	0.001525	0.003422	0.006906	0.011003	0.013494	0.018350

[a]Approximable critical section.

Ocean_CP application computes ocean movements on large scale. It has a total of four critical sections. Table 4 shows how much time is spent in each one of these critical sections. Critical section 1 is the most time-consuming one. All of the critical sections are executed with the default options. The application is executed with 1 to 64 threads. The code segments in critical sections carry out tasks such as updating global error and the movements of all the ocean. These applications are chosen as good candidates to work with for our proposed work since all of them contain many critical sections with highly different characteristics.

The accuracy metric used for the Raytrace application is the ratio of absolute pixel difference to the number of pixels in the images. A lower value indicates a better result. For Radiosity, the used accuracy metric is the relative distance between pixel values as in the literature.[11] For Water_NSquared, the accuracy metric used is also the relative distance, this time between potential levels. The accuracy metric for Ocean_CP is again the relative distance. For the last three applications, a higher metric value indicates a better performance. Performance improvement is measured as the ratio between the original run and the approximated runs. Speedup is measured as how faster the parallel execution is compared to the serial execution for both original and approximated versions.

4.2 | Experimental results

4.2.1 | Profiling results

The first step of the experimental study is to profile the critical sections and group them according to their importance. For this reason, the applications are executed by skipping one critical section at a time for all critical sections. If the program crashes when a critical section is skipped, we say that this critical section is inapproximable. Otherwise, we execute the program a total of 10 times for each number of threads to average the change

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Critical section execution times in seconds versus number of threads for the Radiosity application.

# of threads / CS	1	2	4	8	16	32
Critical Section 1	0.000001	0.000004	0.000015	0.000069	0.000127	0.000537
Critical Section 2	0.000064	0.000250	0.000345	0.000719	0.001641	0.003992
Critical Section 3	0.010674	0.024512	0.058722	0.124334	0.299792	0.707814
Critical Section 4	0.010635	0.024549	0.058297	0.124959	0.307932	0.744267
Critical Section 5	0.010657	0.024685	0.058320	0.123868	0.298856	0.706640
Critical Section 6	0.033402	0.076232	0.177677	0.339106	0.761737	1.689041
Critical Section 7	0.033618	0.105489	0.261425	0.537885	0.956197	1.421788
Critical Section 10	0.075466	0.161918	0.379606	0.842540	1.877551	3.152100
Critical Section 12	0.093818	0.206731	0.457730	0.927692	1.681830	2.110523
Critical Section 13	0.078340	0.168708	0.390999	0.873103	1.968089	3.344759
Critical Section 14	0.025733	0.052590	0.118453	0.244959	0.419468	0.529476
Critical Section 15	0.077056	0.164282	0.380200	0.850784	1.912940	3.259048
Critical Section 16	0.002160	0.004644	0.011043	0.024798	0.056904	0.092056
Critical Section 18	0.014374	0.030417	0.068276	0.139384	0.251641	0.327092
Critical Section 22[a]	0.196353	0.402623	0.897441	1.833476	2.611008	3.101151
Critical Section 24	0.201203	0.405126	0.882623	1.816337	2.852601	3.715161
Critical Section 25	0.036220	0.078281	0.171345	0.346971	0.580247	0.738571
Critical Section 26	0.000459	0.001257	0.002087	0.004647	0.009868	0.030503
Critical Section 27	0.014071	0.030913	0.069894	0.139199	0.254843	0.344121
Critical Section 28	0.001512	0.003170	0.007357	0.013704	0.024121	0.033109
Critical Section 29	0.001894	0.004019	0.007692	0.011179	0.019594	0.141368
Critical Section 30	0.000373	0.000849	0.001541	0.002333	0.006525	0.057149
Critical Section 31	0.000006	0.000027	0.000190	0.049004	0.125452	0.244293
Critical Section 32	0.000000	0.000004	0.000098	0.050021	0.529724	4.006006
Critical Section 33	0.022984	0.206957	0.562401	1.160687	2.390667	5.503219
Critical Section 34	0.023372	0.271154	0.762190	1.500753	5.461560	30.280519
Critical Section 35	0.001471	0.013028	0.047520	0.104417	0.390729	1.966255
Critical Section 36	0.000001	0.000001	0.000003	0.000006	0.000006	0.000015
Critical Section 37	0.000007	0.000098	0.000118	0.000317	0.000962	0.003502

[a]Approximable critical section.

TABLE 3 Critical section execution times in seconds versus number of threads for the Water_NSquared application.

# of threads / CS	1	2	4	8	16	32	64
Critical Section 1	0.000015	0.000031	0.000074	0.000176	0.000667	0.001494	0.129872
Critical Section 2	0.191800	0.082121	0.259654	0.834006	3.462481	20.382125	178.801414
Critical Section 3	0.000000	0.079074	0.275228	0.569573	1.838460	18.022711	125.043838
Critical Section 4	0.000000	0.322576	0.968337	3.818108	15.084575	41.039845	384.406547
Critical Section 5	0.000012	0.000024	0.000053	0.001959	0.018967	0.027042	0.045967
Critical Section 6	0.000023	0.000061	0.000174	0.007465	0.003063	0.027326	0.061977
Critical Section 7[a]	0.000010	0.000008	0.000015	0.000029	0.000071	0.000227	0.001537
Critical Section 8	0.000006	0.000006	0.000091	0.000006	0.000023	0.000017	0.000444
Critical Section 9	0.000002	0.000003	0.000006	0.000015	0.000054	0.000341	0.000493

[a]Approximable critical section.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 Critical section execution times in seconds versus number of threads for Ocean_CP application.

CS \ # of threads	1	2	4	8	16	32	64
Critical Section 1	0.000618	0.004945	0.046277	0.109647	0.394229	1.689796	3.385408
Critical Section 2	0.000003	0.000005	0.000017	0.000019	0.000037	0.000035	0.000080
Critical Section 3[a]	0.000001	0.000006	0.000010	0.000028	0.000086	0.000243	0.012759
Critical Section 4	0.000011	0.000028	0.008236	0.003041	0.001160	0.033633	0.046279

[a]Approximable critical section.

in the output for each of the execution rates until we find a satisfactory result. Therefore, we can see the effects of the execution rate for each thread number.

The Raytrace application crashed when we skipped critical sections 2 and 12 in Table 1. This means, these critical sections are vital for the program and should be classified as inapproximable. They are executed exactly in the test runs. The application output degrades highly in accuracy when critical section 7 is skipped. However, it finishes execution normally and does not crash. We executed this critical section at rates 50%, 70%, and 80% at first; however, the accuracy was less than 70% for these runs. The application output was at a satisfactory level only after we run the critical section at a rate of 90%. With this rate, the accuracy was over 70%. Hence critical section 7 is classified as the only approximable critical section for the Raytrace application. When we skipped other critical sections, it had no to minimal effect on the output accuracy. This means all other critical sections had to be classified as unimportant. We then skip all unimportant critical sections at once to see how they affected the output.

The inapproximable critical sections perform jobs such as allocating memory for tree nodes. Consequently, skipping these critical sections causes segmentation fault. The only critical section that is approximable for the Raytrace application initializes the thread identification numbers. After this initialization, work is distributed among each thread. Skipping this critical section at some rate means that some portion of threads will not get the jobs they are supposed to execute, hence it will yield a lower accuracy. Most of the unimportant critical sections give identification numbers to rays, which does not affect the output accuracy when skipped.

Skipping critical sections 2, 7, 12, 13, 14, 18, 24, 26, 27, 28, 30, 33, 34, 35, and 36 in Table 2 caused the Radiosity application to crash. The application terminates with no output produced when we skip the critical sections 1, 16, 29, 31, 32, and 37. Therefore, the program cannot be executed without these critical sections. They are classified to be inapproximable critical sections and executed exactly in test runs. When we skip the critical sections 10, 15, and 22, the program produces an output, however, the output is highly corrupted. We examine these critical sections in more detail. They are tested with different execution rates as shown in Figure 1. Even after we execute critical sections 10 and 15 at a rate of 99%, the output is still not acceptable. Consequently, they are classified as inapproximable. For this application, critical section 22 is the only approximable critical section with an execution rate of 95%. Lower execution rates did not result in acceptable outputs. When we skip the remaining critical sections, the output results are affected minimally. All other critical sections are classified as unimportant critical sections. Therefore, we skip these critical sections for the test run as well.

For the Radiosity application, most of the critical sections in the inapproximable group perform jobs related to tree nodes such as creating a node and node edges; or jobs such as creating and deleting a task. As these jobs are important for the Radiosity algorithm, they cannot be approximated. The only approximable critical section deals with interactions between nodes, which degrades the output as it affects total Radiosity, but does not cause a crash when skipped. The unimportant critical sections mostly perform jobs as normalizing RGB values, which have minimal effect on output.

The Water_NSquared application crashes only when critical section 9 in Table 3 is skipped. Therefore, this is the only inapproximable critical section for this program and we execute it exactly for the test runs. When critical section 7 is skipped, the application does not crash, but its output deteriorates highly. We execute this critical section with rates 50%, 70%, and 80%, but none of these rates give satisfactory results. Then we execute this critical section with a 90% execution rate and the accuracy is over 90% with this rate. Hence, critical section 7 is assigned to be an approximable critical section for the Water_NSquared application. Skipping all other critical sections has minimal to no effect on the output, hence they are assigned to be unimportant.

The only inapproximable critical section of the Water_NSquared application distributes the jobs to different threads. Skipping this critical section results in no threads getting any work. The approximable critical section updates the potential levels. Therefore, skipping some of the executions of this critical section lowers the accuracy. The unimportant critical sections execute jobs such as marking some molecules according to the time they are processed, which does not affect the potentials significantly.

Ocean_CP application crashed when we skipped critical sections 1 and 2 in Table 4. Consequently, these two critical sections are classified as inapproximable critical sections. Skipping critical section 3 does not yield an abnormal termination but it lowers the accuracy greatly. Therefore, we executed this critical section with 50% accuracy. This execution rate increased the accuracy substantially. Critical section 3 is the only approximable critical section for the Ocean_CP application. The output is not affected by skipping critical section 4. This critical section is the only unimportant critical section of this application and it is skipped at test runs as well.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The inapproximable critical sections perform tasks such as partitioning jobs to threads. Therefore, skipping them means no job will be done. The only approximable critical section updates a global variable that is related to the movements. Skipping this critical section at some rate will lower the accuracy at some point. The only unimportant critical section also updates a global variable. But this global variable does not directly affect the result.

4.2.2 | Accuracy and performance results

After we profile the critical sections, we execute inapproximable critical sections exactly, skip unimportant critical sections and execute approximable critical sections at the rates associated with them. We apply all of the decided skipping and approximation at the same time for the accuracy and performance results. We execute all of the programs 10 times and present the average results in this section.

We present the accuracy and performance results for the Raytrace application for both original and approximate runs in Figure 3. The figure shows that the accuracy is over 70% for all executions. It is consistently over 85% when we execute the program with more than 4 threads. The

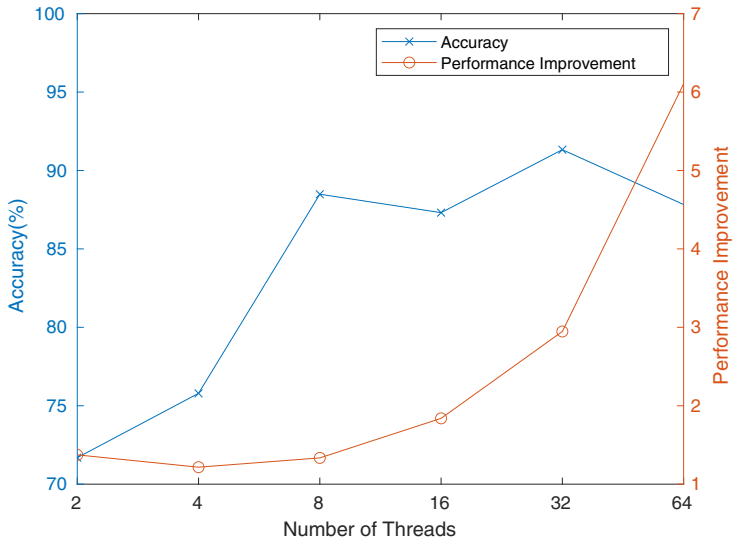

FIGURE 3 Raytrace performance and accuracy results for rate = 90%.

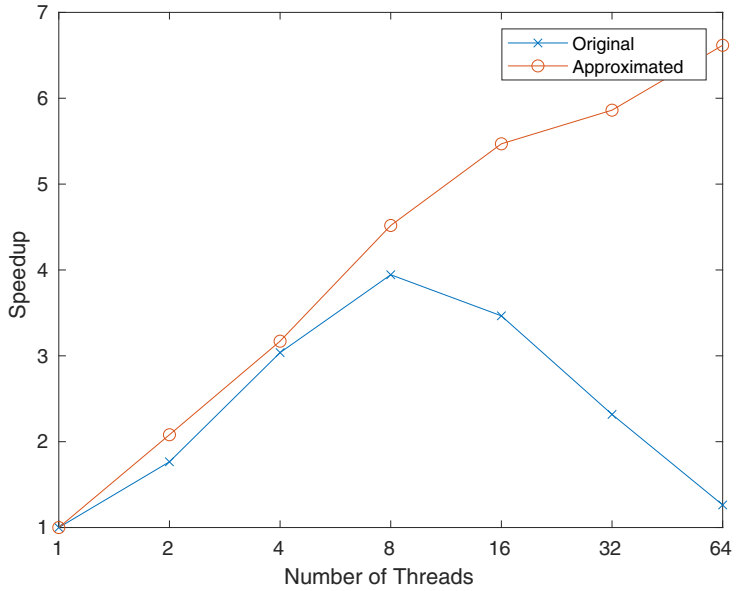

FIGURE 4 Raytrace speedup curve for original and approximated for rate = 90%.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

performance improvement is around 1.3× for 2, 4, and 8 threads, around 1.8× for 16 threads, around 3× for 32 threads, and around 6× for 64 threads. Critical section 7 is where the threads are assigned different tasks. This means that skipping some of the executions of it has an effect of skipping some of the tasks that are normally executed. This results in high performance gain as we can see in the experiments.

The speedup curve of the Raytrace application for its original and approximated versions as the number of threads increases is given in Figure 4. The figure shows that the performance gain for the application becomes more prominent as the number of threads increases. While the synchronization overhead becomes more significant with more number of threads in the original runs, the approximated version continues the performance gain. The reason might be skipping 76% of all critical sections for the approximated version; hence, the synchronization overhead drops significantly.

Figure 5 shows the accuracy and performance results of the Radiosity application for both approximated and original versions. The accuracy is between 82% and 84% for all executions and the performance improvement is between 1.3× and 1.45× for executions with different numbers of threads.

The speedup curve of the Radiosity application for its original and approximated versions as the number of threads increases is given in Figure 6. The figure indicates that the speedups for both the original and approximated versions are parallel. As we skip only 36% of all critical section executions, approximated version behaves very similarly to the original version. When we execute the application with more than 8 threads, the speedup

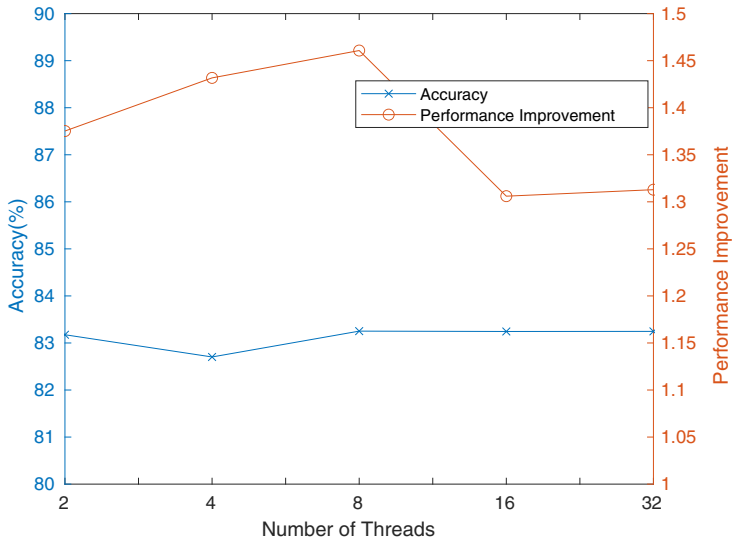

FIGURE 5 Radiosity performance and accuracy results for rate = 95%.

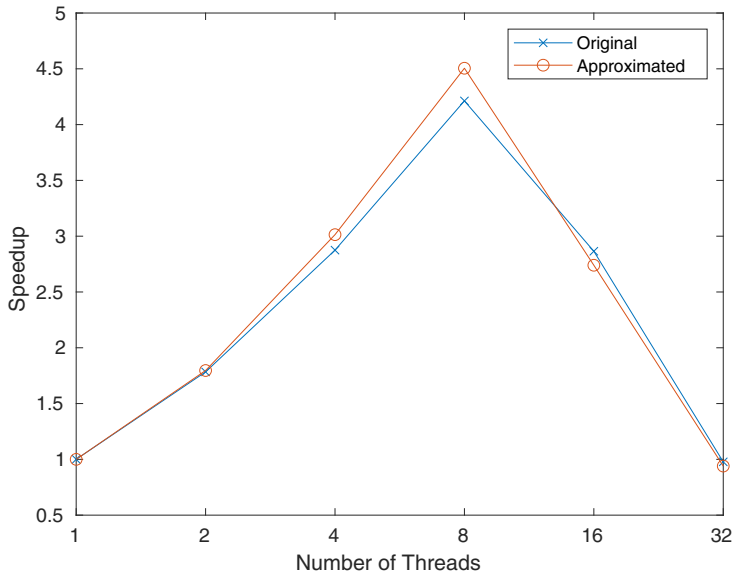

FIGURE 6 Radiosity speedup curve for original and approximated for rate = 95%.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

decreases as a result of higher synchronization overhead caused by many threads. Executing the program with 32 threads performs slightly worse than the one with 1 thread. Execution with 8 threads gives the best performance and accuracy result for Radiosity.

Figure 7 shows the accuracy and performance results of the Water_NSquared application for both approximated and original versions. The accuracy is consistently higher than 90% for all executions. The performance improvement peaks at 1.25 when the program is executed with 4 threads. On average, it has a performance improvement of 1.1×.

The speedup curve of the Water_NSquared application for its original and approximated versions as the number of threads increases is given in Figure 8. The figure shows that the original and approximated versions of the programs behave very similarly in terms of speedup. For this program, synchronization overhead is not very significant. This may explain why skipping a high portion of the critical sections does not increase the performance as much as the Raytrace and the Radiosity applications.

Figure 9 shows the accuracy and performance results of Ocean_CP application for both approximated and original versions. The accuracy of the approximated version is very close to the original version for all executions at 99%. The performance improvement is 1.3 for both 2 and 4 threads.

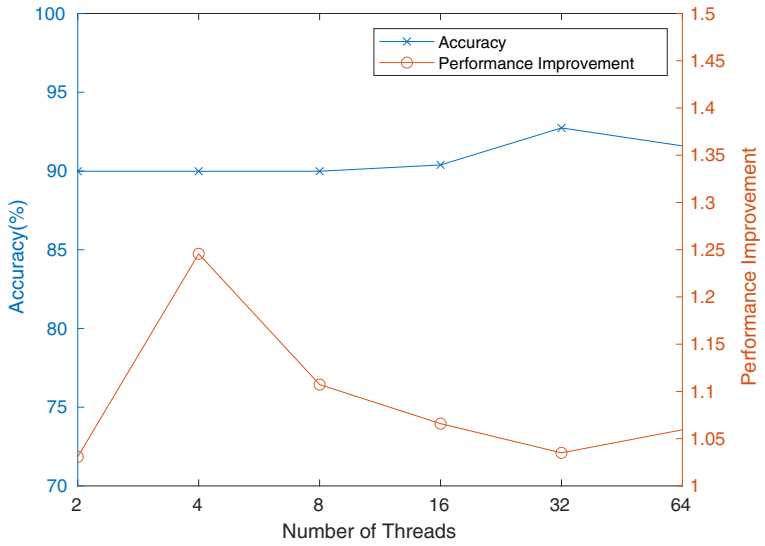

FIGURE 7 Water_NSquared performance and accuracy results for rate = 95%.

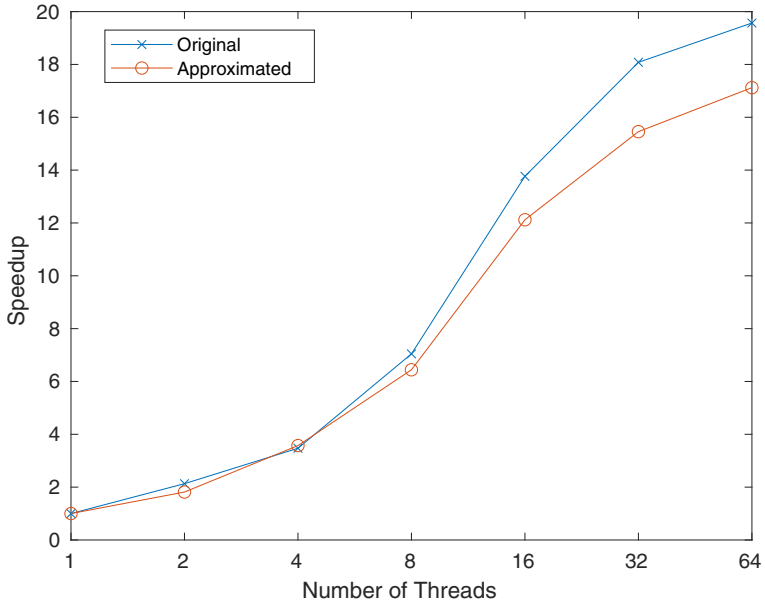

FIGURE 8 Water_NSquared speedup curve for original and approximated for rate = 90%.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

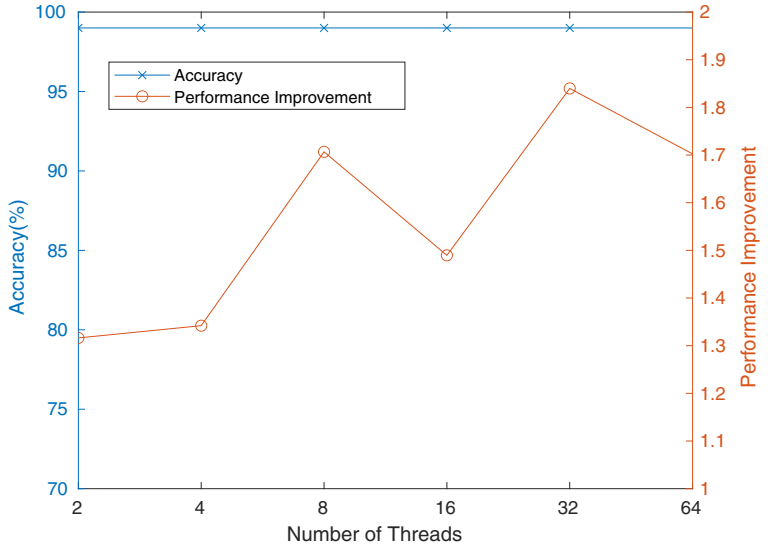

FIGURE 9 Ocean_CP performance and accuracy results for rate = 50%.

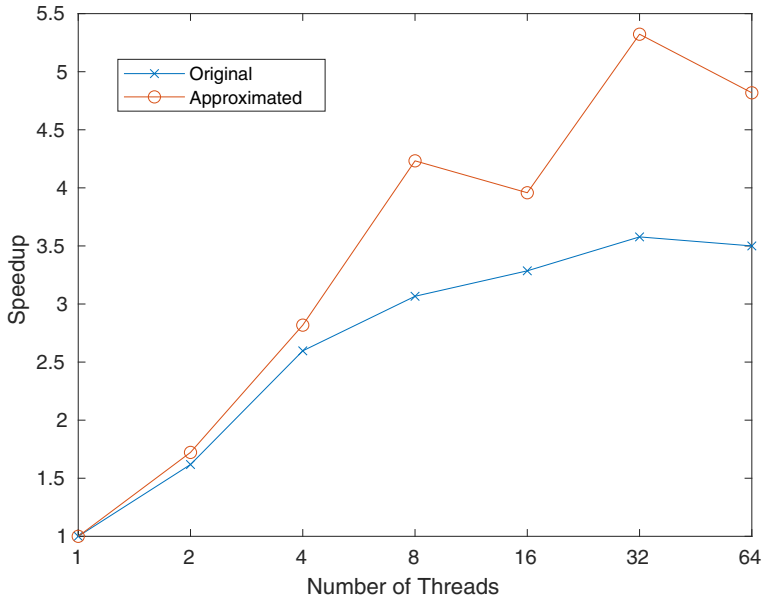

FIGURE 10 Ocean_CP speedup curve for original and approximated for rate = 90%.

When we execute the program with 8 and 16 threads, it becomes 1.7× and 1.5×, respectively. The performance improvement peaks at 1.8× when the program is executed with 32 threads. It is again 1.7× for 64 threads.

The speedup curve of the Water_NSquared application for its original and approximated versions as the number of threads increases is given in Figure 10. The approximated version outperforms the original version for this application. However, the synchronization overhead is still not as much as the Raytrace and the Radiosity application for Ocean_CP application for the original run of the program.

4.3 | Discussion

We apply our proposed work to applications with critical sections of many different characteristics. One observation is that critical sections with different characteristics are classified in different groups. If a critical section deals with memory allocations and pointer arithmetic they are usually grouped as inapproximable as skipping them might yield to segmentation fault. The critical sections where mathematical operations are calculated

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

are good candidates for approximation and skipping as they would not cause a crash, instead only resulting in accuracy loss. We see that critical sections which distribute jobs to different threads are never classified as unimportant as classifying them as such would result in threads not being able to accomplish any task. There are examples of critical sections for each class of critical sections in Appendix.

Another observation is that skipping a higher number of critical sections makes performance gain more prominent for applications with higher synchronization overheads as Raytrace speedup is higher compared to Radiosity. We show that our proposed work performs better with applications where synchronization is a significant overhead as this work aims to mitigate the said problem. This can be seen from the fact that Raytrace performance gain is higher compared to Water_NSquared even though they both have a high percentage of skipped critical sections.

Raytrace results presented in Figure 3 with 90% execution rate has accuracy less than 80% even though the 80% threshold is used generally in the literature.[11] We did not execute the program with a higher rate after we observed that the application performs well when it is executed with a higher number of threads. The accuracy is over 85% for more than 4 threads. Nevertheless, a user may increase the execution rate if a higher accuracy result is important for their needs.

Our work requires a profiling step for each application it is applied to. This is the main limitation of our work. We skip critical sections one by one and examine the program behavior. We execute the program 10 times in this manner for all number of threads to see the average effect of skipping the critical section to the output. We repeat this process for each rate until the execution rate produces an acceptable output for each of the critical sections. Even though this step requires no knowledge about the application in question, the users still need to execute the programs as they skip critical sections one by one to assess their importance and to group the critical sections accordingly.

As a future work, we plan to address this problem with an adaptive rate according to the user's accuracy needs so that the program dynamically updates the execution rate of the approximated critical sections. We aim to implement a tool that will automatically skip the critical sections one by one and execute the program to classify the critical sections. Then, for candidate critical sections, it will increase the execution rate and output an execution rate for all approximable critical sections according to the needed accuracy level.

5 | CONCLUSION

Our proposed work aims to alleviate synchronization overhead by skipping and approximating a subset of critical sections in parallel applications. It examines critical sections and groups them according to their importance for the program execution to decide on which critical sections to skip and approximate. Our work requires minimal code change and minimum application knowledge, hence it is easy to apply. We performed our experiments using Raytrace, Radiosity, Water_NSquared, and Ocean_CP applications from the SPLASH2 benchmark. Out of a total of 62 critical sections, we skipped 30 of them and approximated 4 of them. Performance improvement is 2.5× on average for the Raytrace application with an average accuracy loss of 16%. For the Radiosity application, the performance gain is 1.4× on average and the accuracy loss is 17%. For the Water_NSquared application, the approximated application runs 1.1× faster on average compared to the original version with only a 9% accuracy loss. For the Ocean_CP application, the approximated application runs 1.6× faster on average compared to the original version with only a 1% accuracy loss. Therefore, we observe a great potential for parallel applications by skipping a subset of critical sections. This potential can be exploited for a wide range of applications such as image processing, video decoding, and machine learning depending on probabilistic methods and scientific applications. The common trait of such applications is that they do not need exact calculations to be successful. In order to apply our work to such suitable applications, we plan to propose more intelligent techniques to automate our skipping approach as a future work based on the performance metric and accuracy rate given by the user.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in SPLASH2 at http://parsec.cs.princeton.edu/download/3.0/parsec-3.0.tar.gz, Reference number 16.

ORCID

Zuhal Altuntaş https://orcid.org/0000-0001-5935-678X
Sanem Arslan https://orcid.org/0000-0003-3019-7070

REFERENCES

1. Mittal S. A survey of techniques for approximate computing. *ACM Comput Surv*. 2016;48(4):1-33. doi:10.1145/2893356
2. Rodrigues G, Kastensmidt FL, Bosio A. Survey on approximate computing and its intrinsic fault tolerance. *Electronics*. 2020;9(4):557.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3. Goiri I, Bianchini R, Nagarakatte S, Nguyen TD. ApproxHadoop: bringing approximations to MapReduce frameworks. Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery ASPLOS '15; 2015:383-397; New York, NY.
4. Sidiroglou-Douskos S , Misailovic S , Hoffmann H , Rinard M . Managing performance vs. Accuracy trade-offs with loop perforation. Proceedings of the ESEC/FSE '11. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering; 2011:124-134; Association for Computing Machinery, New York, NY .
5. Rubio-González C , Cuong N , Hong DN , et al. Precimonious: tuning assistant for floating-point precision. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis SC '13; 2013:1-12 .
6. Alvarez C , Corbal J , Valero M . Fuzzy memoization for floating-point multimedia applications . *IEEE Trans Comput* . 2005 ; 54 (7): 922 - 927 . doi: 10.1109/TC.2005.119
7. Kulkarni P , Gupta P , Ercegovac M . Trading accuracy for power with an underdesigned multiplier architecture. Proceedings of the 2011 24th Internatioal Conference on VLSI Design; 2011:346-351 .
8. Saadat H , Bokhari H , Parameswaran S . Minimally biased multipliers for approximate integer and floating-point multiplication . *IEEE Trans Comput-Aided DesIntegr Circ Syst* . 2018 ; 37 (11): 2623 - 2635 .
9. Lamport L . A new solution of Dijkstra's concurrent programming problem . *Commun ACM* . 1974 ; 17 (8): 453 - 455 . doi: 10.1145/361082.361093
10. Chen G , Stenstrom P . Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis SC '12; 2012:1-11 .
11. Akram R , Alam MMU , Muzahid A . Approximate lock: trading off accuracy for performance by skipping critical sections. Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE); 2016:253-263 .
12. Islam B , Shezan FH , Shahriyar R . High performance approximate computing by adaptive relaxed synchronization. Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS); 2016:1204-1210 .
13. Sattar NS , Aqila T , Shahriyar R . Towards concurrent data structure development with relaxed synchronization. Proceedings of the 2016 9th International Conference on Electrical and Computer Engineering (ICECE); 2016:267-270 .
14. Khatamifard SK , Akturk I , Karpuzcu UR . On approximate speculative lock elision . *IEEE Trans Multi-Scale Comput Syst* . 2018 ; 4 (2): 141 - 151 . doi: 10.1109/TMSCS.2017.2773488
15. Renganarayana L , Srinivasan V , Nair R , Prener D . Programming with relaxed synchronization . *RACES '12* . Association for Computing Machinery ; 2012 : 41 - 50 .
16. Woo SC , Ohara M , Torrie E , Singh JP , Gupta A . The SPLASH-2 programs: characterization and methodological considerations. Proceedings 22nd Annual International Symposium on Computer Architecture; 1995:24-36 .
17. Altuntaş Z , Arslan S , Boz B . Approximate execution of critical sections for performance accuracy tradeoff. Proceedings of the BAŞARIM 2022 - 7th High-Performance Computing Conference; 2022 .
18. Osorio RR , Rodriguez G . Truncated SIMD multiplier architecture for approximate computing in low-power programmable processors . *IEEE Access* . 2019 ; 7 : 56353 - 56366 .
19. Zhang Q , Wang T , Tian Y , Yuan F , Xu Q . ApproxANN: an approximate computing framework for artificial neural network. Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE); 2015:701-706 .
20. Yazdanbakhsh A , Pekhimenko G , Thwaites B , Esmaeilzadeh H , Mutlu O , Mowry TC . RFVP: rollback-free value prediction with safe-to-approximate loads . *ACM Trans Arch Code Optim (TACO)* . 2016 ; 12 (4): 1 - 26 .
21. Karakoy M , Kislal O , Tang X , Kandemir MT , Arunachalam M . Architecture-aware approximate computing . *Proc ACM Meas Anal Comput Syst* . 2019 ; 3 (2):1-24. doi: 10.1145/3341617.3326153
22. Brand M , Witterauf M , Bosio A , Teich J . Anytime floating-point addition and multiplication-concepts and implementations. Proceedings of the 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP); 2020:157-164 .
23. Kim Y , Venkataramani S , Chandrachoodan N , Raghunathan A . Data subsetting: a data-centric approach to approximate computing. Proceedings of the 2019 Design, Automation Test in Europe Conference Exhibition (DATE); 2019:576-581 .
24. Sharif H , Zhao Y , Kotsifakou M , et al. ApproxTuner: a compiler and runtime system for adaptive approximations. Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '21; 2021:262-277; Association for Computing Machinery, New York, NY .
25. Pal Singh J, Gupta A, Levoy M. Parallel visualization algorithms: performance and architectural implications. *Computer*. 1994;27(7):45-55. doi:10.1109/2.299410
26. Stream like a boss with NVIDIA RTX graphics card . https://www.nvidia.com/en-me/geforce/rtx/ .
27. Hanrahan P, Salzman D, Aupperle L. A rapid hierarchical radiosity algorithm. Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques. Association for Computing Machinery SIGGRAPH '91; 1991:197-206; New York, NY .
28. Gear CW. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice Hall PTR; 1971.
29. Singh JP, Hennessy JL. Finding and exploiting parallelism in an ocean simulation program: Experience, results, and implications. *J Parallel Distrib Comput*. 1992;15(1):27-48 .

How to cite this article: Altuntaş Z, Arslan S, Boz B. Approximate execution and grouping of critical sections for performance-accuracy tradeoff. *Concurrency Computat Pract Exper*. 2023;e7614. doi: 10.1002/cpe.7614

APPENDIX A. EXAMPLE CRITICAL SECTIONS

The appendix includes example critical sections for each critical section class.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

A.1 Unimportant critical section

Unimportant critical section 25 of the Radiosity application is shown in Listing 2. It is a critical section that does not affect program output highly. In the Radiosity algorithm, the environment is divided into small pieces called *elements*. Radiosity is calculated according to the interactions between elements. The critical section in Listing 2 frees an interaction instance. If we skip it, the interaction among the elements is not added to the free list. This does not cause significant degradation in the output, as freed interactions do not affect output.

```
pthread_mutex_lock(&(global->free_interaction_lock));

interaction->next = global->free_interaction;
global->free_interaction = interaction;
global->n_free_interactions++;

pthread_mutex_unlock(&(global->free_interaction_lock));
```

Listing 2: Unimportant Critical Section Example: Critical Section 25 of the Radiosity Application

A.2 Approximable critical section

Critical Section 7 of the Water_NSquared application, which is classified as an approximable critical section, is shown in Listing 3. The critical section in Listing 3 calculates the potential energies for the Water_NSquared application. Approximating this critical section affects the output depending on the approximation level but does not cause failure.

```
pthread_mutex_lock(&(gl->PotengSumLock));

*POTA = *POTA + LPOTA;
*POTR = *POTR + LPOTR;
*PTRF = *PTRF + LPTRF;

pthread_mutex_unlock(&(gl->PotengSumLock));
```

Listing 3: Approximable Critical Section Example: Critical Section 7 of the Water_NSquared Application

A.3 Inapproximable critical section

Critical Section 2 of the Raytrace application, which is classified as an inapproximable critical section, is shown in Listing 4. It is an inapproximable critical section. The critical section in Listing 4 allocates tree nodes of a given size for the Raytrace application. It first tries to find an appropriate free space in the free list. Later, if the free space is greater than the space needed, it adds excess space to the free list again. Since this critical section allocates memory, skipping it causes segmentation fault, as the program attempts to access a memory space that is not allocated.

```
pthread_mutex_lock(&(gm->memlock));

prev = NULL;
curr = gm->freelist;
size = ROUND_UP(size);

while (curr && curr->size < size)       {
        if (curr->cksm != CKSM)         {
                fprintf(stderr, "GlobalMalloc: Invalid checksum in node.\n");
                exit(1);
        }
        if (curr->free != TRUE)         {
                fprintf(stderr, "GlobalMalloc: Node in free list not marked as free.\n");
                exit(1);
        }
        prev = curr;
        curr = curr->next;
```

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
}
if (!curr)      {
        fprintf(stderr, "%s: %s cannot allocate global memory .\n", ProgName , msg );
        exit(-1);
}
if (curr->size - size > THRESHOLD)      {
        next        = NODE_ADD(curr, nodesize + size);
        next->size  = curr->size - nodesize - size;
        next->next  = curr->next;
        next->free  = TRUE;
        next->cksm  = CKSM;
        curr->size  = size;
} else
        next = curr->next;
if (!prev)
        gm->freelist = next;
else
        prev->next   = next;

pthread_mutex_unlock(&(gm->memlock));
```

Listing 4: Inapproximable Critical Section Example: Critical Section 2 of the Raytrace Application

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7614 by Marmara University, Wiley Online Library on [29/01/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License