Parallel Collision Detection with OpenMP

To cite this article: Ting-Hsuan Chien et al 2018 J. Phys.: Conf. Ser. 1069 012180

View the article online for updates and enhancements.

Related content

- Parameters analysis of a porous medium model for treatment with hyperthermia using OpenMP
  Ruy Freitas Reis, Felipe dos Santos Loureiro and Marcelo Lobosco

- Servo-controlling structure of five-axis CNC system for real-time NURBS interpolating
  Liangji Chen, Guangsong Guo and Huiying Li

- Computational parallel for shallow water-sediment concentration coupled model
  IB P A Prandhana and P H Gunawan
Parallel Collision Detection with OpenMP

Ting-Hsuan Chien, Jhe-Wei Lin and Rong-Guey Chang
Department of Computer Science and Information Engineering, National Chung Cheng University, Chia-Yi, Taiwan, 62102 {cth103p, jhe106p, rgchang}@cs.ccu.edu.tw

Abstract. Collision detection is a very important issue for 5-axis machining. In this paper, we parallelize the collision detection of 5-axis machining with OpenMP to improve its performance. First, we apply profiling tools to analyse collision detection to obtain the performance spectrum. Then we apply some techniques to exploit parallelism, reduce the number of instructions, and parallelize the critical parts with OpenMP to improve speedup. The results show that the proposed approach can outperform the sequential version by 17 times.

1. Introduction
Collision detection is not only an important issue but also a necessity in many applications, such as computer games, physical simulations, model engraving, virtual reality, etc. The most important goal of collision detection is to determine if the two objects intersect. This paper will discuss how to accelerate the collision detection for five-axis machining. Previous work worked on the parallelism for collision detection with GPU (Graphic Processing Unit). For example, (Wen-Chi et al.) used GPU computing to accelerate real-time collision detection on Computer Numerically Controlled (CNC) machine tools [1]. (JC et al.) used GPU operation to determine whether virtual objects will collide based on the g-code move path [2]. A five-axis machine tool is not just expensive but also costly to maintain. Additionally, the large-scale five-axis machine tool is usually more than 200,000 US dollars. Although GPU is very suitable for solving problems with higher data parallelism, however, GPU is too expensive for many applications. As a result, we aim at applying more popular and cheaper multicores to parallelize collision detection in this paper.

OpenMP is flexible and it does not change the architecture of a program. We can see that it is used in many parallel technologies by many researchers. For example, (Seyong et al., 2009) presents a compiler framework for automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications [3]. (Rabenseifner et al., 2009) describe potentials and challenges of the dominant programming models on hierarchically structured hardware: Pure MPI, pure OpenMP and hybrid MPI+OpenMP in several flavours [4]. (Ming-Tse, 2013) used OpenMP parallel operation matrix multiplication, relationship between test matrix size and parallelization efficiency [5]. (Bradley and Gaster, 2008) describe an alternative approach to programming SIMD machines using an extended subset of OpenMP allowing us to model naturally the programming of arbitrary sized SIMD arrays while retaining the semantic completeness of both OpenMP and the parent languages [6].

The main goal for this paper is to propose the way that applies OpenMP to parallelize a CNC application, where we first calculate whether two objects collide, then uses OpenMP to synchronize the two parts, and finally, use SIMD Worksharing to construct vectorization for improvement. Besides, we will discuss the parallel differences between GPU and CPU in this paper.

The rest of this paper is organized as follows. Section 2 presents the background and Section 3 shows how to analyse our program architecture by each tool. Section 4 presents analyse the collision
detection and how do we modify the program. The results are shown in Section 5. Finally, we conclude this paper in Section 6.

2. Background

2.1. The Concept of Multi-Core Architectures
The evolution of multi-core processors arises from the development of parallel technology; Figure 1 shows the traditional program execution mode. The assigned event is divided into several small tasks, but only one instruction is executed by the CPU each time. Figure 2 shows the parallel concept of a multicore. It divides the assigned tasks into a series of independent small tasks. Each task is sent to the corresponding CPU/Processor, and multiple instructions can be executed at the same time.

![Figure 1. Traditional execution mode](image1)

![Figure 2. Parallel concept](image2)
2.2. **OpenMP**

OpenMP is a flexible API (Application Programming Interface) that was released by the OpenMP Architecture Review Board in October 1997. OpenMP supports a wide spectrum of console systems, including Windows, GNU/Linux, Mac OS X, etc. It implements Multi-Threading parallelism with the shared memory architecture. OpenMP supports programming languages such as C, C++, and Fortran. The supported compilers include open-source GCC, Sun Studio, intel Compiler, and Open64 compilers.

Why uses OpenMP? The rationale is that the computer architecture belongs to multi-core CPU computers. In addition to OpenMP, Pthread is another choice. Pthreads are low-level APIs and have fine-grained thread management. In addition, OpenMP has many friendly high-level APIs for programmer’s developers to use. Since OpenMP is much more efficient than Pthreads for development and structure maintenance, we choose OpenMP to parallelize collision detection in this paper. The comparison between OpenMP and Pthreads is shown in Table 1.

|                        | OpenMP                        | POSIX Thread                  |
|------------------------|-------------------------------|-------------------------------|
| Manage by coarse-grained| Manage by fine-grained        |                               |
| Flexible architecture   | Modify the architecture       |                               |
| Simple development      | Hard development              |                               |
| High –level API         | Low-level API                 |                               |

2.3. **Collision Detection**

The main goal of collision detection in the paper is to put two objects in a three-dimensional space and determine whether the two objects will intersect each other. In fact, collision detection is still a challenge. Especially in the three-dimensional space, the solution to collision detection needs complex structures and algorithms. Most non-instant pre-simulations applied to CAD/CAM (Computer Aided Design/Manufacturing) are used to solve complex collision detections. The aim of this research is to design a CNC machine system to detect parallel collision detection.

2.4. **Collision Detection with GPU**

J. C. Chang purposed a method that using GPU to design a CNC machine system to detect whether each axis or tool intersects with another axis and thereby solve the problem of collision detection effectively. Their work focused on the simulation of the objects of a CNC machine system. In this paper, NVIDIA CUDA (Compute Unified Device Architecture) is used for parallelization. CUDA is a parallel computing platform and programming model introduced by NVIDIA in 2006. Prior to the development of CUDA, OpenCL or DirectX technology is used to write parallel programs on GPU. It was very difficult for programmers to design and write parallel applications.

Implementing collision avoidance is divided into two parts:

- Pre-processing;
- CNC G-code simulation;

The results of this paper pointed out that the use of GPU can handle the calculation of the triangular mesh. It shows that the worst time of collision detection is 486.16us.

3. **The Performance Analysis of Collision Detection**

In this paper, before using OpenMP to accelerate Collision Detection, we should first understand the architecture, characteristics, and rules of the program. Therefore, we use program analysis tools to help us understand more accurately and quickly. Using program architecture, we can find out Collision Detection HotSpot. In this section we will introduce Gprof, Gprof2Dot and Perf separately.
3.1. Gprof
Gprof is GNU profiler tool, which runs on linux, AIX, Sun and other operating systems and performs performance analysis of C, C++, etc. In addition, it can display the program's running time, the number of function calls, the time cost of the function. We use Gprof to help us understand the run time of the Collision Detection program, the number of function calls, the time cost of the function, so that we can easily see where the program needs to be improved. In other papers, (Varley, 1993) explains the necessary work and analyses the limitations of Gprof as an analytical tool without time constraints [7].

This step use Gprof to analyses the results of the program. The analysis includes the execution time and number of calls for each function. The analysis results are shown in Table 2. As a result, we can know the ratio of tri_tri_intersect() time, the time it takes for the function to be called, and the number of times it is called.

| Time(%) | Cumulative seconds | Self seconds | Call times | name                              |
|---------|--------------------|--------------|------------|-----------------------------------|
| 41.71   | 11.28              | 11.28        | 806557696  | tri_tri_intersect()               |
| 25.09   | 18.07              | 6.79         | 7088       | Object::isCollisionOccurred()     |
| 8.58    | 20.39              | 2.32         | 0633293262 | Object::getTriangles()            |
| 7.65    | 22.46              | 2.07         | 806557696  | mollerTriToTriDetect()            |
| 6.23    | 24.14              | 1.69         | 814036423  | Object::getNumOfTriangles()       |
| 5.10    | 25.52              | 1.38         | 2530494747 | std::fabs()                       |
| 2.83    | 26.29              | 0.77         | 806557696  | isTwoTrianglesCollisionOccurred() |

3.2. Gprof2Dot
Gprof2Dot is a python script that converts data into a relation graph. The input can be like Linux perf, Valgrind's callgrind tool, VTune Amplifier XE, gprof, and so on. It can delete nodes below a certain standard, use an exploratory way to record the transfer time between functions, and clearly mark the Hotspot of the program with the color difference.

Using Gprof2Dot to understand the architecture of the entire Collision program. As shown in Figure 3, we can clearly understand the architecture of the Collision Detection program and we can also know the relationship between each function call and the called. One can even know which part is called the most often, or which function performs the heavier weight; and from this figure we can see that the main, exec Code, test Collision, and Object::is Collision Occurred functions make up the total ratio. Above 96%, in addition, we can know that is Two Triangles Collision Occurred () is called 806557696 times by the Object::is Collision Occurred () call, which is the maximum number of called times.
Figure 3. The analysis result of Gprof2Dot

3.3. Perf

Compared to gprof, Perf has the advantage of being tightly integrated with the Linux kernel and can benefit from the new features first introduced into the core. The basic principle of perf is to sample the target and record whether or not the detected event occurred and the number of occurrences under a specific condition. For example, according to the tick interrupt sampling, the sampling point is triggered within the tick interrupt, and the process context is judged at the sampling point.

4. Parallelizing Collision Detection

In this work, OpenMP is used to implement Collision Detection with Multi-Threading to achieve parallel effect, and then shorten the execution time of Collision Detection. The method flow of this paper is subdivided into two steps to illustrate: Firstly, the use of the results of the analysis of the collision detection programs to modify the program to make OpenMP easier to parallelize; Secondly, explanation of how to use OpenMP to accelerate conflict detection SIMD vectorization technology.

4.1. Preliminary Parallization

We use OpenMP to achieve Collision Detection with Multi-Threading parallel computing results. From the analysis results, it is evident that the Collision Detection consumes the most computational resources with the for loop, because it is necessary to determine whether the two objects have intersections, and a large number of triangular meshes are needed to calculate each other to determine whether the two objects would collide. In the process of parallelization, the first is to determine whether the parallelized data are independent of each other. If the data is not independent of each other, then parallelization cannot be performed. Otherwise, the data access error will occur.

In addition, we refer to the Loop-Unrolling proposed by (Murthy et al., 2010) to reduce the failure of Branch Prediction and reduce the number of instruction sets to reduce the overhead of the for loop [8]. In event of Multi-Threading parallel, the operating system will switch data from one Thread to another Thread, causing Context-Switching and extend execution time. In order to solve this problem, the schedule in OpenMP Directive for is used to help Thread arrange execution for loop. There are four schedule arrangements, namely: dynamic, static, guided, and runtime, detail is shown in Table 3.
### Table 3. Schedule method

| Schedule method | Explanation |
|-----------------|-------------|
| Dynamic         | The allocation method is dynamic allocation. When a Thread executes a chunk, the next chunk executes. |
| Static          | The number of iterations of the for loop is cut into chunks according to the chunk_size, and the chunks are assigned to the Thread on average. |
| Guided          | Cuts the chunks in a decremental manner. The first group has a larger number of for loops and gradually descends to chunk_size in an exponential manner. |
| Runtime         | The parallelization of the for loop is not statically determined at compile time, but deferred until the program execution dynamically determines the method to be used based on the environment variable. |

#### 4.2. Improvement with SIMD

Finally, we use the part of the SIMD vectorization loop for OpenMP Directive to vectorized each Thread execution block, as shown in Figure 4. Note that the chunk size should be set to a multiple of the SIMD length, otherwise the remainder will not be executed.

![Parallel and vectorised](image)

**Figure 4. Parallel and vectorised**

#### 5. Result

This chapter describes the experimental environment used, demonstrate the use of OpenMP in details, as well as explore and use the difference between the GPU. In the experiment process, the program is edited, compiled, and executed by connecting piety to the server. The specifications of the server are shown in Table 4.

| Table 4. Experiment environment |
|---------------------------------|
| System                          | Ubuntu 16.04.2 LTS (Xenial Xerus) |
| Linux Kernel                    | 4.4.0-83-generic                  |
| CPU                             | IntelXeon CPU E5-2620@2.0GHz + 12core |
| Random Access Memory            | 16 GB                             |
| GPU                             | Low-level API                     |

We implement Collision Detection on the simulated machine to detect whether there is intersection between the tool and the two triangular grids of A and C. According to the analysis results, the most important factor affecting the efficiency is the number of triangular grids. The number of triangle grids used is shown in Table 5.
### Table 5. Triangular meshes in each axis

| Axis      | Number of triangle meshes |
|-----------|----------------------------|
| X         | 1582                       |
| Y         | 9840                       |
| Z         | 5082                       |
| A         | 7470                       |
| C         | 2037                       |
| Cutlery(tool) | 44                     |

Comparing the execution time of the tool with the middle of the A-axis, the side of the A-axis, and the C-axis is compared between the pre-parallel and the post-parallel. As shown in Figure 5, we can see the middle of the tool and the A-axis, the side of the A-axis, and the C-axis. The calculation time of the cost is greatly shortened after parallelization.

![Parallel block execution time](image)

**Figure 5.** Parallel block execution time

Figure 6 shows the results of Collision Detection execution time based on the CPU is 12 core, so we tested 12, 16, 24, 32, 36 Threads sequentially, we can find that the higher the number of Thread spent the shortest time, in addition, we used parallel GPU to compare with Chang JC parallel Collision Detection to discuss the difference between using CPU and GPU parallelization respectively.
6. Conclusion
Owing to new technological advances, Computer Numerically Controlled (CNC) becoming more prevalent. However, the cost of CNC is also relatively expensive. Therefore, the development of Collision Detection has helped the five-axis machine to avoid Collisions between the axes. Nevertheless, due to the huge number of triangular meshes on each axis of the simulation machine, the computation time is very long.

In addition, the GPU is expensive, and we want to use the CPU to accelerate the parallel Collision Detection to approach the same effect as to using the GPU. We use the OpenMP loop SIMD construct to parallelize and vectorize. We successfully used the OpenMP parallelization Collision Detection program, the demonstrated example shows that it improves 12 time of computation efficiency, comparing to that of the conventional Detection method.

7. References
[1] Wen-Chi Guo, "Parallelising anti-Collision Detection on GPU" 2013 Department of Computer Science and Information Engineering in National Chiao Tung University
[2] Chang JC., Chien TH., Chang RG. (2015) A Cyber Physical System with GPU for CNC Applications. In: Wang G., Zomaya A., Martinez G., Li K. (eds) Algorithms and Architectures for Parallel Processing. Lecture Notes in Computer Science, vol 9530. Springer, Cham
[3] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '09). ACM, New York, NY, USA, 101-110. DOI = http://dx.doi.org/10.1145/1504176.1504194
[4] R. Rabenseifner, G. Hager and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, Weimar, 2009, pp. 427 - 436. doi: 10.1109/PDP.2009.43
[5] Ming-Tse Hung, "Application of Particle Swarm Optimization using CPU Parallel Processing in Optimization Design of 2D Truss Structures" 2013 Department of Civil Engineering in National Chiao Tung University
[6] Bradley C., Gaster B.R. (2008) Exploiting Loop-Level Parallelism for SIMD Arrays Using OpenMP. In: Chapman B., Zheng W., Gao G.R., Sato M., Ayguadé E., Wang D. (eds) A

Figure 6. Collision Detection execution time.
Practical Programming Model for the Multi-Core Era. IWOMP 2007. Lecture Notes in Computer Science, vol 4935. Springer, Berlin, Heidelberg

[7] Varley Dominic A, "Practical experience of the limitations of gprof," Softw. Pract. Exper 23: 461-463. doi:10.1002/spe.4380230407

[8] G. S. Murthy, M. Ravishankar, M. M. Baskaran and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Atlanta, GA, 2010, pp. 1-11. doi: 10.1109/IPDPS.2010.5470423