INTRODUCTION TO PARALLEL PROGRAMMING (CS575)(PAPER PROJECT)

LOAD-BALANCED PIPELINE PARALLELISM

Lin, Si Thu
ID-933-957-884 | OREGON STATE UNIVERSITY

1.What is the general theme of the paper you read? What does the title mean?

Decoupled Software Pipelining

Although there are some works on pipeline parallelism like “Decoupled Software Pipelining”, they still have weakness in executing applications in the data parallel fashion. Lack of data balancing is one of the biggest issues for them. It can be inferred from the title of the paper that the paper is mainly about the data balancing among the threads. In this paper, the LBPP (load-balanced pipeline parallelism) is often compared to the Traditional Pipeline Parallelism (TPP).

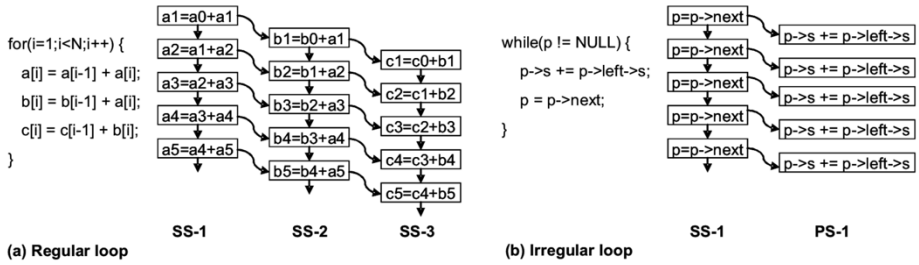

Figure 1: Examples of loop level pipeline parallelism for regular and irregular loops. The arcs show the dependency relationships.

A loop can be regular loop in which the loop counts can be known like the one in the Figure 1 or irregular one in which the number of loops cannot be known. In both TPP and LBPP, the stages of the loop are categorized into two group, the parallel stage and sequential stages. If a stage requires the data from the previous iterations, the stage is called “sequential stage” and the dependency is called “Cross-iteration dependency”. If a stage needs the data from stages from the same iteration, the stage is “parallel stage” and the dependency is “Intra-iteration dependency”. In TPP, each stage is assigned to different core for regular loops like in the Figure 1(a) regardless of whether it is sequential or parallel one. As a drawback, it can cause that the compiler needs to know the number of available cores. As for the LBPP, the number of available cores does not matter and the number of cores which is used for the parallelism can even be changed during the runtime.

In Figure 1(b), the first instruction is the parallel stage and the second one is the sequential stage. What is different from the regular loop is that the second instruction is assigned to one core and the first one is assigned to as many cores as necessary since the first one can be run in parallel. I believe that many parallel jobs cannot be run, and many parallelisms cannot happen since the parallel stage (PS-1) need the data from the sequential stage (SS-1). However, using many threads for the parallel stage will make the execution time of the parallel stage considerably less obvious and the sequential stage execution time will dominate the total execution time. In TPP, all stages cannot be run at the same time and synchronization is needed because of the intra-iteration dependencies. However, we do not need to worry about the cross-iteration dependencies because one iteration starts only after another and the data from the previous iteration is ready to be used in the same thread when the next iteration starts.

Load-balanced Pipeline Parallelism

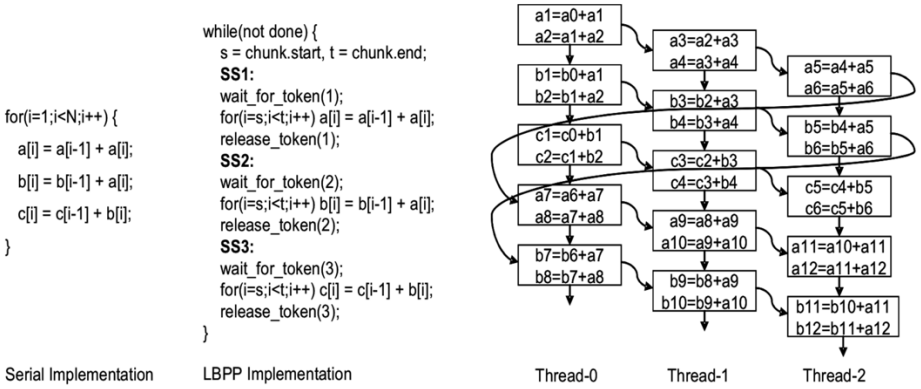

Figure 2: Implementation and execution of LBPP for a regular loop with a chunk size of 2 iterations that uses three threads.

Stages of a loop iteration are distributed between cores in TPP, but a whole iteration is run on the corresponding core and the iterations are distributed between cores in LBPP. LBPP works by chunks which is very important in this technique. One chunk represents a stage and contain more than one iteration. So, there will be three kinds of chunk if there are three stages in the loop. All different chunks must have the same number of iterations. Chunks are placed in the same order in the threads as the stages are placed in the loop. In the Figure 2, there are three stages in the loop. Therefore, there are three kinds of chunks all of which involve the same number of iterations, which is two. "LBPP maps the chunks to threads in round robin order."

Since the whole iterations of a stage (a chunk) is executed only after another chunk, the intra-iteration dependencies are automatically satisfied. In other words, all stages of an iteration are executed on the same thread unlike TPP, which cause that the data does not need to be transferred to other threads for the intra-iteration dependencies. The data transfer between threads for the cross-iteration dependencies is also reduced by using chunks. It does not require to be transferred until the chunk boundaries are not reached. So, the larger the chunk size is, the less data need to be transferred between threads. The executing order of the sequential stages are controlled by the token so that they are executed one after another. The token will be sent to another thread only at the end of a chunk as well. Therefore, a large chunk can reduce the synchronization overhead by making the token required to be sent less often. The irregular loop is also managed in the similar way. The difference is that the local buffers are used for the irregular loop for handing over the data to the next chunks. Although there are advantages of big chunk sizes, there are also some disadvantages. If the chunk size is so big that it does not fit in the available space of the private cache, the shared data between the stages in the thread will be removed, which can result in having to fetch that data again from the L2 cache or memory later. So, there is a trade-off for the chunk size.

Locality

Considering the facts given above, the LBPP have better locality than TPP since the former need to do data transfer between threads only for sequential stages. Assuming 3000 iteration, chunk size of 500, 8-byte values and no false sharing for the example from the Figure 1, TPP will send 8(Byte value) *3000(iteration count) *2(two parallel stages in every iteration) =48000 bytes for the parallel stages. In the same condition, LBPP will transfer only 144 bytes. Here is the explanation. There are three sequential stages in the loop. So, the total sequential stages to be

executed in 3*3000(iteration count) =9000. Since data for the cross-iteration dependencies will be moved only when the chunk boundaries are crossed, the times the data needs to be transferred is 9000(total sequential stages count)/500(chunk size) =18. Therefore, it is 18*8(Byte value) =144 bytes. Hence, LBPP requires less cache-cache transfers which are expensive.

Thread number Independence

The code produced by the compiler will vary depending on the number of available cores in TPP. In LBPP, the code from the compiler is the same for every number of cores used for execution since each whole iteration is run on corresponding core in round robin order. Even when the limitation for the parallelism improvement is reached, increasing the cores for execution can improve the performance in that there will be more private memory space provided by more cores, which is helpful in avoiding cache miss which needs to access lower level cache and memory in the cache hierarchy.

Load Balancing

In LBPP, the same parallel stages from different iterations are executed on separate threads in favor of better load balancing, which can highly increase the parallelism. However, those are run on the same thread since each stage is assigned to each thread in TPP, which make the parallelism of TPP not as good as that of LBPP. There is one advantage of TPP. When enough threads are used to make the execution time of the parallel stages rarely obvious. The execution time of sequential stages will determine the overall performance, which is also said by Amdahl's Law. In this situation, the traditional pipeline parallelism (TPP) like DSWP will have less synchronization cost because the same sequential stages of all iterations are executed on the same thread and the data does not need to be transferred to other threads for the cross-iteration dependencies. Nevertheless, the data is required to be moved to other threads in LBPP. Fortunately, using large chunk size can amortize that by minimizing the times of data transferred to other threads.

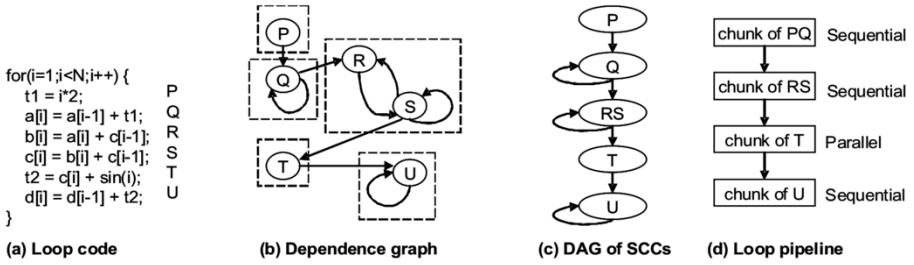

Figure 4: Different steps of constructing the pipeline for a loop.

LBPP IMPLEMANTATION

There are four steps in implementing LBPP. They are DAG construction, pipeline design, adding synchronization and chunking.

DAG construction

The main purpose of this graph is to divide the stages into two groups, sequential stages and parallel stages. This step itself contains two procedures. The one is creating dependence graph. The other is changing the created graph to DAG graph in which each node represents a chunk. DAG graph main dependencies between stages.

Pipeline Design

In this step, the stages from the DAG graph are combined into one if that can benefit. In the figure 4(c), i*2 calculation, which is not as expensive as load operations, is done inside P. If the node P is treated as a separate stage, a store operation will be needed in P and a load operation will be needed in Q. Making P and Q a single stage will reduce the synchronization overhead by removing these store and load operation, which also does not disturb the dependency.

Chunking

In the regular loop, each thread independently decides on the chunk size. In an irregular loop with a single loop, one thread allows another thread to know that its loop terminates, and hand over the data. With an irregular loop with nested loops, the chunk size is decided on based on the inner loop. The chunking can be done by an iteration count, execution time or memory footprint.

Adding Synchronization

The synchronization is added to every sequential stage.

2.Who are the authors? Where are they from? What positions do they hold? Can you find out something about their backgrounds?

There are three authors for this paper. They are Md Kamruzzaman, Steven Swanson and Dean M. Tullsen.

Md Kamruzzaman

He got the B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology, M.Sc. in Computer Science from University of California, San Diego and Ph.D. in Computer Science from University of California, San Diego. He also worked on many research projects which are Software Data Spreading, Inter-core Prefetching, Underclocked Software Prefetching, Coalition Threading and Load Balanced Pipeline Parallelism. Therefore, he has many experience and knowledge about the parallelism and threads. He also took intern at Qualcomm Research, San Diego and worked as a lecturer at Bangladesh University of Engineering and Technology and American International University-Bangladesh, and as a teaching assistant at University of California, San Diego.

Steven Swanson

Steven Swanson is an associate professor in the Department of Computer Science and Engineering at the University of California, San Diego and the director of the Non-volatile Systems Laboratory. He received his B.S. and M.S degrees from UCLA and his PhD from the University of Washington. He also published eleven papers. He leads the Non-Volatile Systems Laboratory. He also co-leads projects to develop low-power co-processors for irregular, mobile applications (e.g., Android Apps) and to devise software techniques for using multiple processors to speed up single-threaded computations, and GreenDroid project.

Dean M. Tullsen.

He is a professor and chair of the Department of Computer Science and Engineering at the University of California San Diego. He is also the author of the book whose title is “Multithreading Architecture”. He has already published 122 papers so far. Besides, he worked as the Computer Architect at the AT&T Computer Systems/AT&T Bell Labs. He also taught in the Computer Science Department of Shantou University in China for one year.

3.What experiments did the paper present?

Three benchmarks are run on two different machines for the LBPP and TPP.

The mb_load is for the testing load balancing. The mb_ubal tests handling the unbalanced stages and mb_local tests the locality.

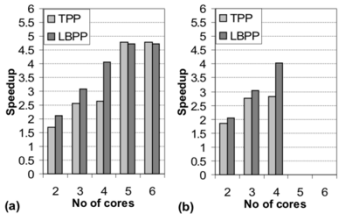

Figure 7: Scalability of *mb_load* – (a) Phenom (b) Nehalem

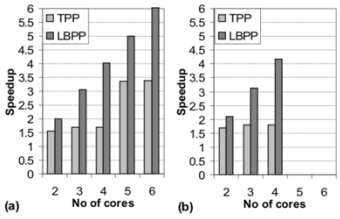

Figure 8: Scalability of *mb_ubal* – (a) Phenom (b) Nehalem

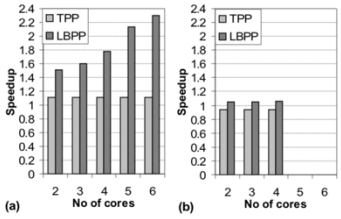

Figure 9: Scalability of *mb_local* – (a) Phenom (b) Nehalem

According to the results, LBPP outperform TPP basically.

4.What conclusions did the paper draw from them?

LBPP maintains locality, have better load-balance and make the compiler able to do compilation without knowing the number of available threads. Unlike TPP, synchronization between threads for sequential stages is required only when the next chunk is started in a thread. “Because all threads execute all stages, it is naturally load-balanced”. LBPP has the performance which is almost 50% as high as that of TPP, especially for low thread counts. Also, it provides less energy consumption by reducing the run-time and expensive cache-cache transfer.

5.What insight did you get from the paper that you didn’t already know?

In addition to CS575, I happen to take the COMPUTER ARCHITECTURE class in which parallelism has been learned in this term. After reading this paper, I believe more that the parallelism plays greatly important role in improving performance in the future. As being learned in the CS575 and COMPUTER ARCHITECTURE class, to increase the performance, we do not necessarily need to improve a processor. Instead, we can run multiple cores and threads parallelly to increase throughput. Now, I attain another knowledge that we can enhance the performance just by making the mechanism that exploit pipeline parallelism better without using additional threads.

6.Did you see any flaws or short-sightedness in the paper’s method or conclusions?

No, I cannot see any flaw.

7.If you were these researchers, what would you do next in this line of this research?

If I were one of these researchers, I would try to find a way to minimize the data transferring between threads because of the cross-iteration dependencies.