The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

TBB-Based Parallel Programming

Lecture 3. Parallelizing Complex Loops

Nizhni Novgorod

2014

Lecture 3. Parallelizing Complex Loops

Objectives

The purpose of this lecture is to study the TBB tools that enable parallel implementation of complex algorithms such as those for pipelined computing, loops where the number of iterations is not known in advance etc.

Abstract

The lecture describes the tbb::parallel_reduce template function that enables implementation of parallelized simple loops as illustrated by the scalar vector multiplication problem. It also describes structures that enable implementation of parallel programs involving loops where the number of iterations is not known in advance, sorting and pipelined computing.

Guidelines

The tbb::parallel_reduce function is intended for parallelizing computations represented as the for loop with reduction (a typical problem whose parallel implementation will benefit from reduction is scalar multiplication of vectors).

The tbb::parallel_reduce operating procedure is similar to that of tbb::parallel_for. Let us note that tbb::parallel_reduce, just like tbb::parallel_for, splits range as long as their size exceeds grainsize. However, tbb::parallel_reduce does not make copies of the input Functor for each split (except for the case below) but operates links to Functors.

As opposed to tbb::parallel_for, tbb::parallel_reduce includes an additional computation stage, i. e. reduction and, depending on how computations are distributed among threads, implements one of two schemes.

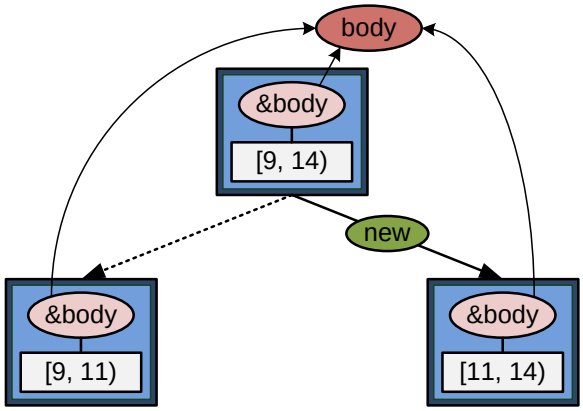

Fig. 1. Single-thread computations by tbb::parallel_reduce

The first scheme of tbb::parallel_reduce operation is implemented when the following computation grain is processed by the same thread as the previous one. In this scheme, only one Functor is necessary and existing. For example, let thread 0 create a computational grain sized [9, 14) at its next iteration. Let the same thread process this grain (Fig. 1). As the iterative space

size exceeds 2, it is split and the pointer to the Functor is copied. This scheme is always implemented in case of single-thread computation.

If the next computational grain is processed by a thread different from the originating one, the second scheme consisting in generation of a new Functor by the splitting constructor, is followed (see Fig. 2). Let us note that, as opposite to Range, Functor splitting constructor does not actually split the Functor or its fields. In most cases, it operates in the same way as the copying constructor. Here, the use of splitting constructor enables a more complex behaviour of the Functor developers at the moment of transferring the Functor to another thread.

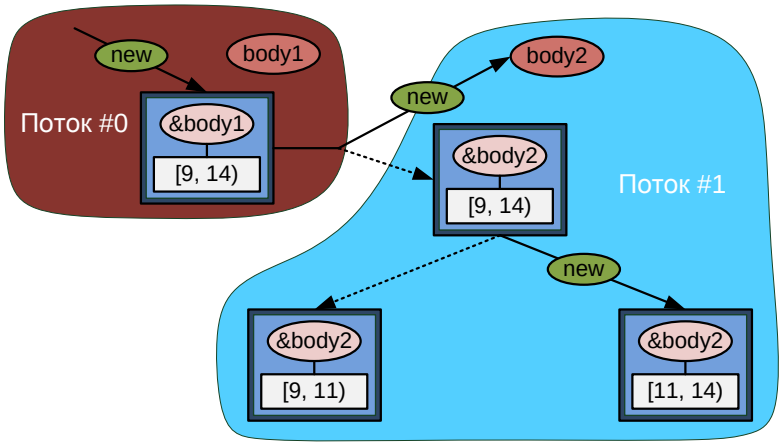

Fig. 2. Computations by tbb::parallel_reduce in case of thread change

For example, let thread 0 create two computational grains, [5, 9) and [9, 14). After this, thread 0 will continue computations for grain [5, 9) (this part is not shown in Fig. 2) and thread 1 will process grain [9, 14). To avoid data race, thread 1 will not use the link to the existing Functor body1, but will create a new Functor body2 using the splitting constructor and process it, plugging the respective link into the grain instead of the initial Functor, body1. What happens next is similar to the first scheme.

If all computations are single-threaded, reduction is not required as there is only one Functor copy which performs all computations on its own. If at any time the next computational grain is created by one thread while computations are performed by another one, a new Functor is created which means that the new Functor should be reduced by the old one. The reducing method is void join(Body& rhs). Its input is the pointer to the Functor which performed part of computations. The data it computed must be taken into account by the current Functor (this) in order to obtain the final result. The Functor transferred via the link is destroyed automatically when reduction is competed (the join function is called).

The TBB library contains the tbb::parallel_sort template function to sort the sequence. This function enables parallel sorting of the intrinsic data types of C++ and all classes where the swap and operator() methods are implemented. The latter must compare the two elements.

The TBB library contains the tbb::parallel_do template function for parallel processing of elements located in a certain input data stream. Elements can be added to the data stream in the course of computation

The TBB library contains the tbb::pipeline class enabling pipelined computing. This type of computation involves performing a sequence of stages for one and the same element. If at any single stage different elements can be processed in parallel, this class will help perform such computations. The tbb::pipeline class processes elements set by a data stream. Processing is based on filters to be applied to each element. Filters may be either sequential of parallel (parallel type).

Recommendations for Students

The information is mainly sourced from the official TBB web page https://www.threadingbuildingblocks.org/. The site features numerous documents and examples. A free library version for non-commercial use is also downloadable.

Andrews (2000) is a recommended introduction into parallel programming.

Quinn (2004) is also recommended as a description of typical problems of parallel programming.

Practice

1. Implement a parallel application to find the sum of vector elements using parallel_reduce.
2. Using tbb::sort, sort the strings in the dictionary order.
3. Using tbb::pipeline, implement a program that strops every number in a text file. Save the resulting text in a new file.

Test questions

1. What will you used to parallelize the for loop with the fixed number of iterations and reduction?
    a. parallel_for
    b. (+) parallel_reduce
    c. parallel_scan
    d. parallel_pipeline
    e. task
2. In case of Functor implementation for parallel_reduce:
    a. One must not call operator() directly for the Functor object.
    b. (+) The result of Functor operation must be saved in the Functor fields.
    c. The join() method performs reduction; its input is the iterative space.
3. tbb::parallel_reduce:

a. Has one type of call that has two input values, Range, Functor.

b. Has several types of call and can be called without indicating parameters (all parameters are default ones).

c. (+) Has several types of call; computation range and Functor has to be indicated for each of them

4. Computation process of parallel_reduce

a. (+) Is not deterministic (the thread responsible for a specific part of computations will be known only on the execution stage)

b. Is deterministic (the thread responsible for a specific part of computations will be known on the compilation stage)

5. What is reduction?

a. (+) Collection of computed results from all threads to obtain the overall result.

b. Summation of all results obtained by threads in the course of computations.

c. Sequential execution of threads.

6. Computation scheduling in parallel_reduce:

a. Is static and predetermined by the TBB library developers.

b. (+) Is determined by the Range used.

c. Is dynamic; the computation grain size is determined by the grainsize parameter.

7. operator() of the Functor sent to parallel_reduce:

a. (+) Accepts the Range as an input parameter

b. Has to be constant.

c. Accepts two inputs - Range and computation grain size.

8. tbb::sort enables sorting:

a. Only intrinsic data types of C++.

b. (+) Intrinsic data types of C++ and all classes where comparison and inversion methods are implemented.

9. tbb::pipeline:

a. Makes it possible to perform only those pipeline computations where each stage is to be performed by one thread only.

b. Makes it possible to perform only those pipeline computations where each stage is to be performed by an arbitrary number of threads.

c. (+) Makes it possible to perform only those pipeline computations where each stage can be performed either by one thread, or by arbitrary number of threads.

10. tbb::parallel_for:

a. (+) Enables parallel processing of elements in the input data thread. Elements can be added to the data stream in the course of computation

b. Enables parallel processing of elements in the input data thread. Elements cannot be added to the data stream in the course of computation

References

1. Intel® Threading Building Blocks Home Page: https://www.threadingbuildingblocks.org/

2. Intel® Threading Building Blocks Reference Manual: https://software.intel.com/en-us/node/506130

3. Intel® Threading Building Blocks User Guide: https://software.intel.com/en-us/node/506045

4. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley.

5. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.