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 4. Task-Based Programming

Nizhni Novgorod

2014

Lecture 4. Task-Based Programming

Objectives

The purpose of this lecture is to study the task-based mechanisms that form the basis of all TBB-implemented algorithms. Moreover, tasks enable efficient parallelization of recursive algorithms.

Abstract

The lecture describes the task mechanism which is the most flexible one for parallel programming purposes. It reviews the methods for scheduling and synchronization of task execution. The queens problem illustrates a parallel implementation of recursive algorithms based on tasks.

Guidelines

Tasks are represented in TBB as the tbb::task class. This is a basic class for task implementation, i. e. it must be inherited by all user-defined tasks.

tbb::task contains the virtual method task::execute where computations are performed. This method is used to perform the required computations and return the pointer to the next task to be executed. If NULL is retuned, a new task is selected from the pool of tasks ready to be executed.

Each task has a number of related attributes:

- owner - a thread where the task belongs.
- parent - an attribute equal either to NULL or to the pointer to another task whose refcount field will be reduced by 1 upon completion of the current task. The value of this attribute is computed using the parent method.
- depth is the task depth in the task tree. The value of this attribute can be computed using the depth method and set using the set_depth method.
- refcount is the number of tasks with the current task in the parent field. The refcount value can be computed using the refcount method and set using the se_ref_count method.

Tasks must be created only using the new operator overloaded in the TBB library. Tasks are destroyed automatically by means of a virtual destructor. They can also be destroyed manually using the task::destroy method. In this case, the refcount field of the destroyed task must be equal to 0.

The task can be in one of five states at any specific time. The task state is changed when library methods are called or when specific actions take place (i. e. task::execute completion). The TBB library features task::state, the method which returns the current state of the task for which it was called.

For convenience in handling groups of tasks, the library provide the tbb::task_list class. This class is actually a task container.

The basic methods to manage task scheduling and synchronization are as follows:

- void task::set_ref_count(int count) – sets refcount equal to count.
- void task::wait_for_all() – waits for all child tasks to be completed. Refcount must be equal to the number of child tasks + 1.
- void task::spawn(task& child) – enqueues the task ready to be executed and returns control to the software code that called this method. Current and child tasks must belong to the thread that calls the spawn method. child.refcount value must be greater than zero. Before calling spawn, use task::set_ref_count to set the number of slave tasks for the parent task.

The library offers a set of methods enabling reuse of tasks for optimization purpose thus supporting reusability of the allocated resources and reduction of contingencies.

Among the advantages of tasks is that they help implement parallel versions of recursive computations with reasonable ease. This is illustrated by the queens problem.

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 algorithm of Fibonacci sequence computation using logical tasks, based on a recursive algorithm.
2. Implement a function for the problem of vector addition, that has the same functionality as parallel_for.
3. Implement a parallel quicksort version based on tasks.
4. Rewrite the queens problem solution using recycle_to_reexecute.

Test questions

1. After calling wait_for_all(), the following takes place:
   a. Refcount value must be set as equal to the number of child tasks plus 1.
   b. (+) Waiting for all child tasks to complete.
   c. Waiting for all tasks to complete.
2. What structure will you use to parallelize the recursive function?
   a. parallel_for

b. parallel_reduce
c. parallel_scan
d. parallel_ pipeline
e. (+) task

3. The software code below:
   MyTask &t = *new (allocate_child()) MyTask();
   a. Enables correct memory allocation to a task from any function/method.
   b. (+) Enables correct memory allocation to a task only from execute().
   c. Enables correct memory allocation to a task from main().
4. The execute() method of the task class:
   a. (+) Is virtual
   b. Is private.
   c. Returns void.
   d. (+) Returns task*.
5. The execute() method of an “empty” task:
   a. Must not be implemented.
   b. (+) Must return NULL.
   c. Must return this.
6. The new(task::allocate_root()) operator:
   a. Creates a child task for the current one. Has to be called from the execute() method.
   b. (+) Creates a root task.
7. The new(this.allocate_child()) operator:
   a. (+) Creates a child task for the current one. Has to be called from the execute() method.
   b. Creates a root task.
8. Memory must be allocated to tasks:
   a. Statically.
   b. Dynamically, using malloc.
   c. (+) Dynamically, using the intentionally overloaded new operator.
9. tbb::task_list:
   a. (+) Is a container for tasks.
   b. Is intended to store arbitrary objects in the form of a singly-linked list.
   c. Is intended to store arbitrary objects in the form of a doubly-linked list.
10. The task::recycle_as_continuation() method:
   a. (+) Must be called within the execute method body.
   b. (+) Enables task re-execution upon its completion.
   c. Proceeds to the beginning of the execute function.

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.