

**Institute of Technology, Carlow**

**B.Sc. (Honour) in Software Development**

**CW228**

**Design**

**Thread Pool Benchmarking**

**Name: Bowen Nong**

**Login ID: C00163585**

**Supervisor: Joseph Kehoe**

**Date: January 11th 2013**

Contents

[1. Introduction 2](#_Toc345666798)

[2. Thread Pool 2](#_Toc345666799)

[2.1. Use Cases 2](#_Toc345666800)

[2.1.1. Use Cases Diagram 2](#_Toc345666801)

[2.1.2. Brief 3](#_Toc345666802)

[2.2. API 3](#_Toc345666803)

[2.2.1. Operator() 3](#_Toc345666804)

[2.2.2. Enqueue() 4](#_Toc345666805)

[2.2.3. TerminalAll() 5](#_Toc345666806)

[2.3. Domain Model 5](#_Toc345666807)

[2.4. System Sequence Diagram 6](#_Toc345666808)

[2.4.1. Create Thread 6](#_Toc345666809)

[2.4.2. Setup/Start 6](#_Toc345666810)

[2.4.3. Terminate Thread 7](#_Toc345666811)

[2.5. Tests 7](#_Toc345666812)

[3. Benchmarking in OpenMP 8](#_Toc345666813)

[3.1. OpenMP 8](#_Toc345666814)

[3.1.1. How Does It Schedule 8](#_Toc345666815)

[3.1.2. Simple Codes 9](#_Toc345666816)

[3.2. Benchmark Code 9](#_Toc345666817)

[3.3. Tests 11](#_Toc345666818)

[3.3.1. Flocking 11](#_Toc345666819)

[3.3.2. Swarming 13](#_Toc345666820)

[3.3.3. A\* Search 15](#_Toc345666821)

[3.3.4. Performance VS Number of cores 15](#_Toc345666822)

[3.3.5. Performance VS Size of flock 15](#_Toc345666823)

[4. Conclusions 15](#_Toc345666824)

[5. Reference 16](#_Toc345666825)

# Introduction

The threads were used to increase the speed of running programs. They allow the computer to work seen like doing job on the same time. Actuary, the CPU just allocate the work time for each thread. When the threads are created or killed by system, the CPU will give the sources and time. If each task needs to create a thread for working, it will waste lots of sources and time. However, on the other way, we created a group of threads for jobs. Then we do not kill the thread. When a thread finishes a task, it will “sleep” in the queue. According to the scheduling algorithms of threads, the thread goes out of queue for the new task. It will re-use the threads and save the time and sources. So the “box” which we use to create, kills and manages the group of threads is thread pool.

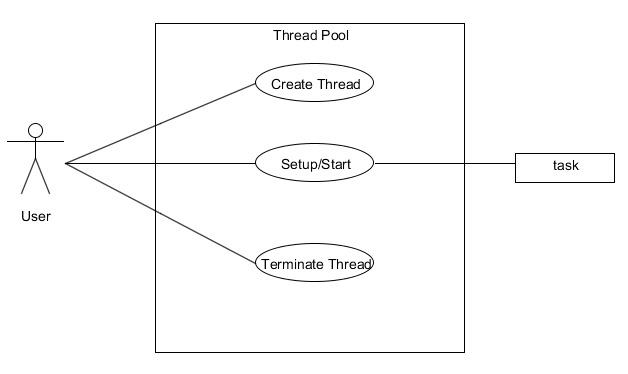
Thread pool is managed by the system, so that programmers do not need to spend time taking or cart thread management. And they can concentrate on application tasks. The threads are stored by priority.

According to this project, the thread pool schedule algorithms will be known, and which schedule algorithms are suitable in games. The timing metrics will be used in the project.

# Thread Pool

## Use Cases

### Use Cases Diagram



### Brief

**Name:** Create Thread

**Actor:** User

**Description:** This case begins with a number of threads are created to perform a number of tasks, which are usually organized in a queue. The number of threads used is a parameter that can be tuned to provide the best performance. And there are many more tasks than threads.

**Main Scenario:**

1. The thread pool creates a group of threads in the thread pool which like a queue.
2. Each task is stored in the other queue.

**Name:** Setup/Start

**Actor:** User & Task

**Description:** This case begins after a group of threads are created in a queue. The results from the tasks being executed might also be placed in a queue. When a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can be terminated, or sleep until there are new tasks available.

**Main Scenario:**

1. The thread runs the task.
2. The thread completed the task.
3. The thread input queue to wait next task.

**Name:** Terminate Thread

**Actor:** User

**Description:** This case begins after the number of tasks has been completed. The thread pool will delete the queue of threads and free the memory.

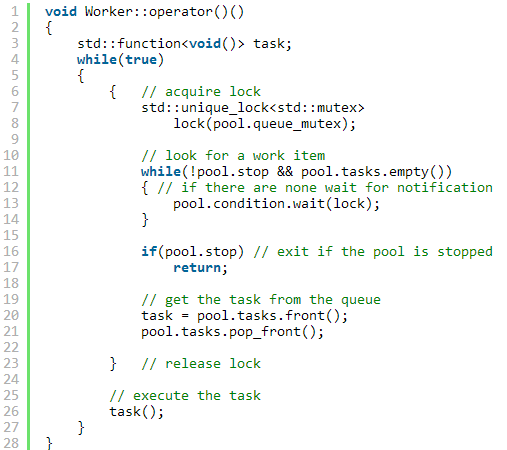
**Main Scenario:**

1. The thread pool finishes the tasks and kills the threads.

## API

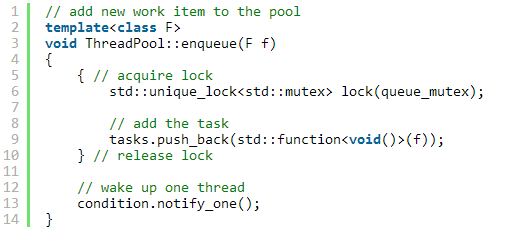
### Operator()

The function operator() will take the threads working for tasks. It first locks the pool, then takes the threads for tasks and operates the task. If the pool is stopped, the operator also stopped.

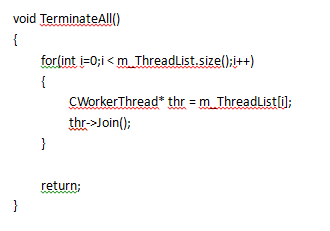


### Enqueue()

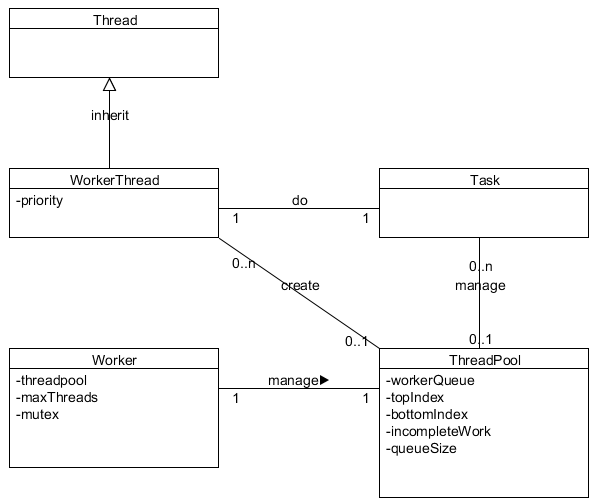
The enqueue() function just locks the queue, adds a task to it and wakes up one thread in case any thread was suspended. And it is the threads running interface.



### TerminalAll()

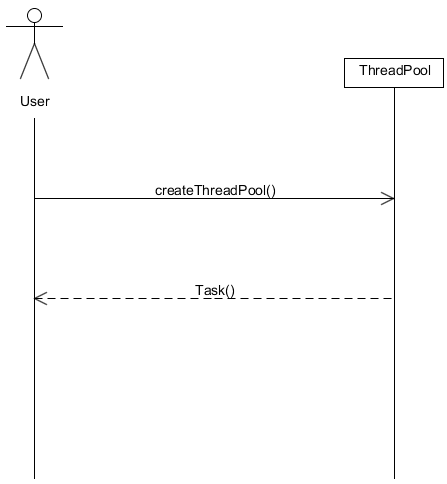


## Domain Model

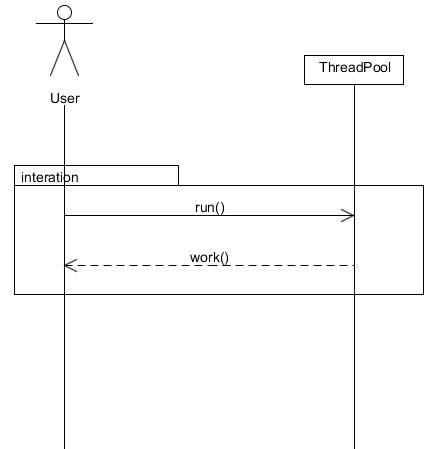


## System Sequence Diagram

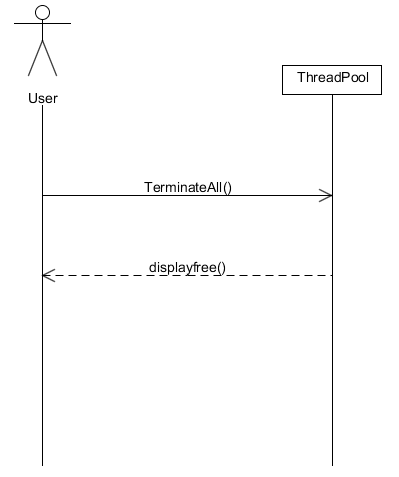
### Create Thread



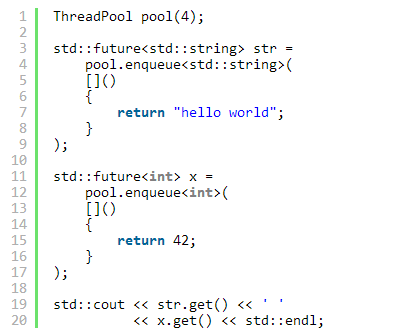
### Setup/Start



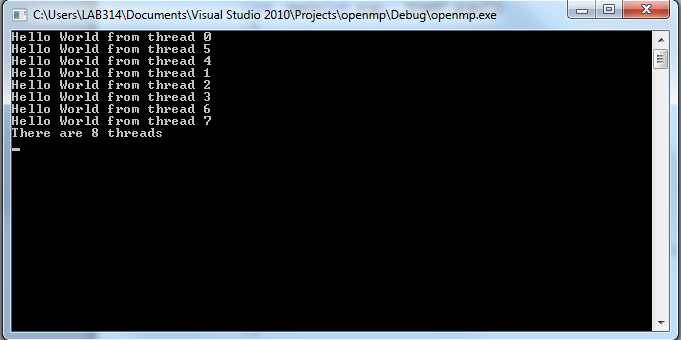
### Terminate Thread



## Tests

Figure 1

Use the “Hello world” program to test the thread pool. The screenshot is a simple example of thread pool testing from internet. If it is successful, the control will show like follow Figure 2.

Figure 2

# Benchmarking in OpenMP

## OpenMP

### How Does It Schedule

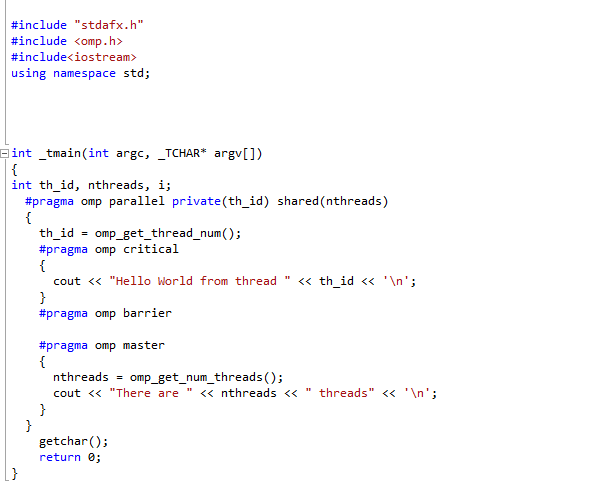
The OpenMP has three kinds of scheduling, which are dynamic, guided and static.

**Static:** all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter chunk will allocate chunk number of contiguous iterations to a particular thread.

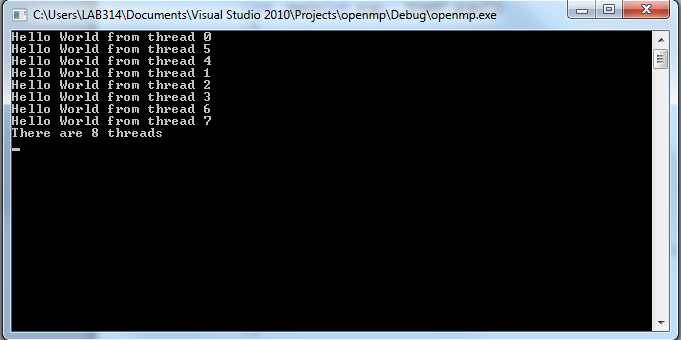
**Dynamic:** some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter chunk defines the number of contiguous iterations that are allocated to a thread at a time.

**Guided:** A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter chunk.

### Simple Codes



There are 8 threads output like this:



Example 3: it shows the simple “Hello World” code of OpenMP. [OpenMP]

## Benchmark Code

**FPS(Frames Per Second)**

This is the benchmark code design.

timeGetTime()  
It returns system time since windows started in milliseconds.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | //.h  #ifndef \_TIME\_H\_  #define \_TIME\_H\_  #pragma comment (lib,"Winmm.lib")  #define WIN32\_LEAN\_AND\_MEAN  #include <windows.h>  #include <mmsystem.h>  class cTime  {  public:  static cTime\* Instance();  static void Destroy();  void operator()();  UINT64 GetDTime(){return mDTime;}  private:  cTime();  ~cTime();  static cTime \*mInstance;  UINT64 mLastGameTime;  UINT64 mDTime;  };  #endif |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 | //.cpp  #include "Time.hpp"  cTime \*cTime::mInstance = 0;  cTime \*cTime::Instance()  {  if(mInstance)  return mInstance;  return mInstance = new cTime;  }  void cTime::Destroy()  {  if(mInstance)  delete mInstance;  }  cTime::cTime():  mDTime(0)  {  mLastGameTime = (UINT64)timeGetTime();  }  cTime::~cTime(){}  void cTime::operator()()  {  UINT64 ticks = (UINT64)timeGetTime();  mDTime = ticks - mLastGameTime;  mLastGameTime = ticks;  } |

Example usage

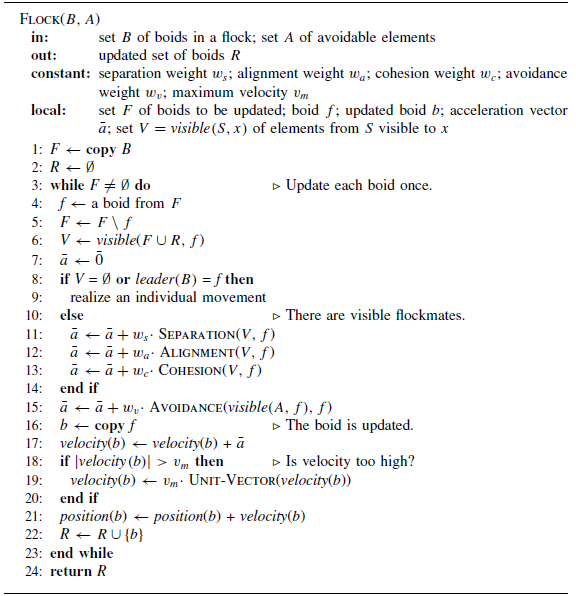
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | #include "time.h"  #include <iostream>  using namespace std;  void main()  {  cTime &TimeRef = \*cTime::Instance();  TimeRef();  // ... code to benchmark here ... //  TimeRef();  cout<<"Benchmark: "<<TimeRef.GetDTime();  } |

The codes will get the time of goal algorithms.

## Tests

### Flocking

**Decision-making:** The Decision-making is the set of problems which are strategic, tactical and operational.



Example 18: It is an Algorithm example of decision-making. It is Flocking algorithm. [GameAlgorithm]

Description:

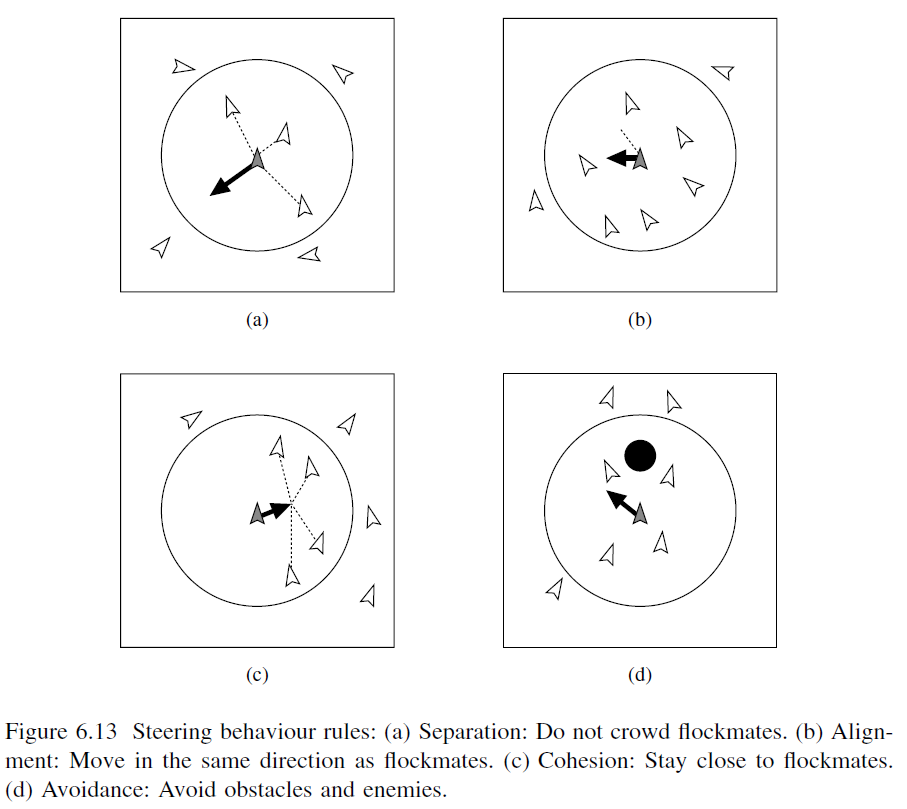
There are four behavior rules in this algorithm. They are following:

(i) *Separation*: Steer to avoid crowding local flockmates. A boid should maintain certain distance from the nearby boids to avoid collisions with them.

(ii) *Alignment*: Steer towards the average heading of local flockmates. A boid should move in the same direction as the nearby boids and match its velocity accordingly.

(iii) *Cohesion*: Steer to move towards the average position of local flockmates. A boid should stay close to the nearby flockmates.

(iv) *Avoidance*: Steer to avoid running into local obstacles or enemies. A boid should escape dangers when they occur.

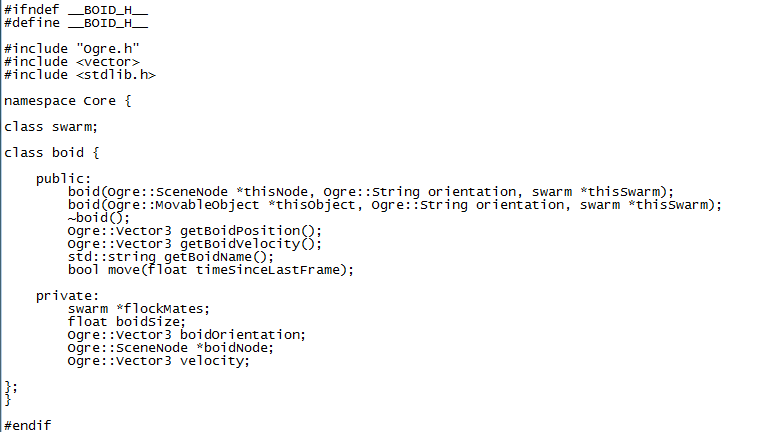


### Swarming

Swarm algorithms, which are based on flocking algorithms. It present another approach with multiple search traces. In swarm algorithms the members of the population ‘fly’ in the search space. Because of avoidance, they keep a minimum distance from each other and cover a larger area than a single search trace, and because they fly as a swarm, they tend to progress as a unit towards better solutions. As a way to escape local optima, the members can never slow down under a minimum velocity, which can allow them to fly past and free from local optimum, especially if it is crowded.

It implemented 2 classes: "Swarm" and "Boid". Swarm is used to hold pointers to all boids of a swarm but doesn't calculate much, movement happens in Boid. 

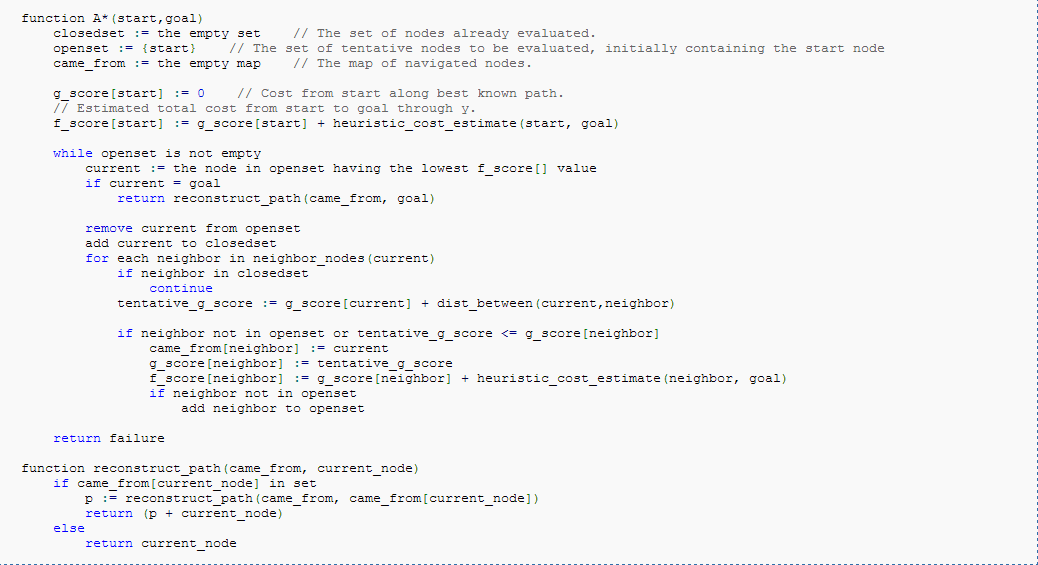
This is the “swarm.h”.



This is the “Boid.h”.

### A\* Search

A\* uses a [best-first search](http://en.wikipedia.org/wiki/Best-first_search) and finds a least-cost path from a given initial [node](http://en.wikipedia.org/wiki/Node_(graph_theory)) to one [goal node](http://en.wikipedia.org/wiki/Goal_node). Here is the algorithm displayed by pseudocode.



Here is a more detailed look at how A-Star operates. The algorithm maintains two sets, the OPEN list and the CLOSED list. The OPEN list keeps track of those nodes that need to be examined, while the CLOSED list keeps track of nodes that have already been examined. Initially, the OPEN list contains just the initial node, and the CLOSED list is empty. Each node n maintains the following:

g(n) = the cost of getting from the initial node to n.

h(n) = the estimate, according to the heuristic function, of the cost of getting from n to the goal node.

f(n) = g(n) + h(n); intuitively, this is the estimate of the best solution that goes through n.

### Performance VS Number of cores

It should performance the number of cores from 1 to n. And count the running time of program. The project should record those information and analysis them. Then giving the parsing diagram.

### Performance VS Size of flock

It should performance the size of flocking from 1 to n. And count the running time of program. The project should record those information and analysis them. Then giving the parsing diagram.

# Conclusions

A summary of thread pool benchmark, the project is simple designed. This project will use a thread pool that implements a number of these algorithms. It will need to develop a number of game algorithms benchmark programs that can be used to test these algorithms. The benchmarks will each represent a different type of concurrent problem. Finally, the algorithms will be tested with these benchmarks to compare the different algorithms performance. According to compare each using time, analysis which thread pool schedules are suitable.

# Reference

[C++11] <http://progsch.net/wordpress/?p=81>  
[Assess 24-10-2012]

[OpenMP] <http://msdn.microsoft.com/en-us/magazine/cc163717.aspx>  
[Assess 24-10-2012]

[GameAlgorithm] Jouni Smed, Harri Hakonen, “Algorithms and Networking for Computer Games”, *University of Turku, Finland, 2006*