МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

**Звіт**

З лабораторної роботи № 4 з дисципліни

«Моделювання систем»

«**Оцінка точності та складності алгоритму імітації**»

| **Виконав(ла)** | *ІП-13 Бабіч Денис* |  |  |
| --- | --- | --- | --- |
|  | (шифр, прізвище, ім'я, по батькові) |  |  |

| **Перевірив(ла)** | *Дифучин А. Ю.* |  |  |
| --- | --- | --- | --- |
|  | (посада, прізвище, ім'я, по батькові) |  |  |

Київ 2024

# ОСНОВНА ЧАСТИНА

**Мета роботи**: Побудувати імітаційні моделі системи з використанням формалізму моделі масового обслуговування.

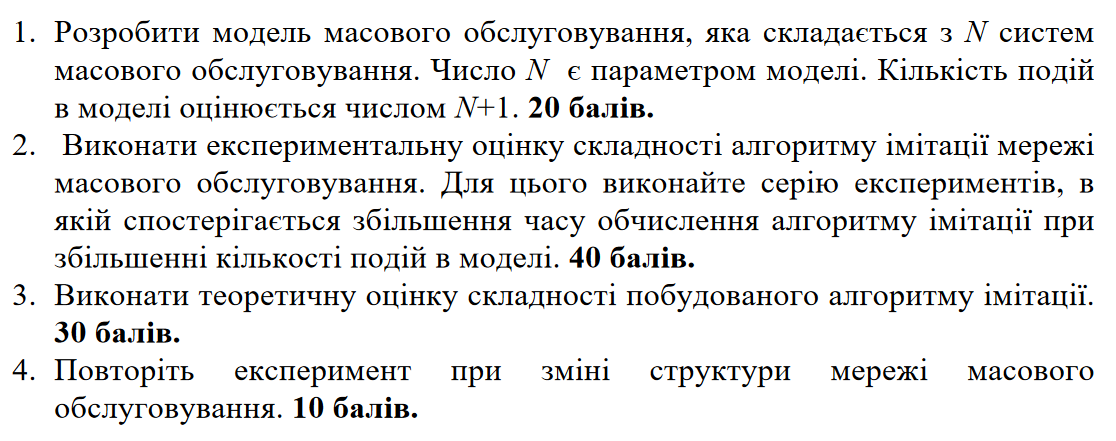


Рисунок 1.1 – Завдання лабораторного практикуму

**Виконання завдання**:

Було виконано 10 прогонів бенчмарку для кількості подій від 100 до 1000, з кроком 100 та з використанням експоненційного генератор з математичним очікуванням 1.

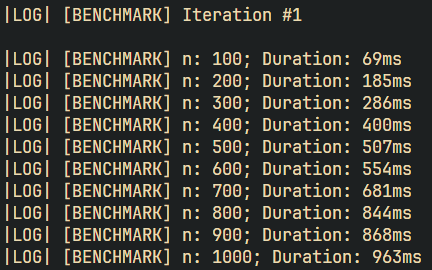


Рисунок 1.2 – Приклад виводу для кожної ітерації системи послідовних моделей

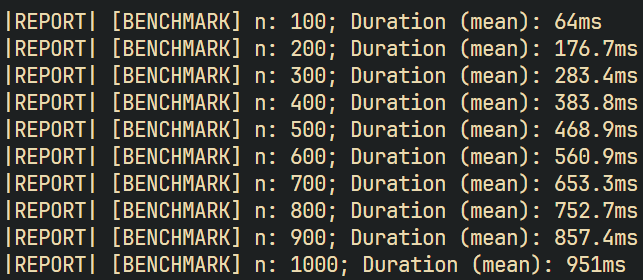


Рисунок 1.3 – Результат роботи бенчмарку системи послідовних моделей



Рисунок 1.4 – Метрики для систем з послідовних моделей

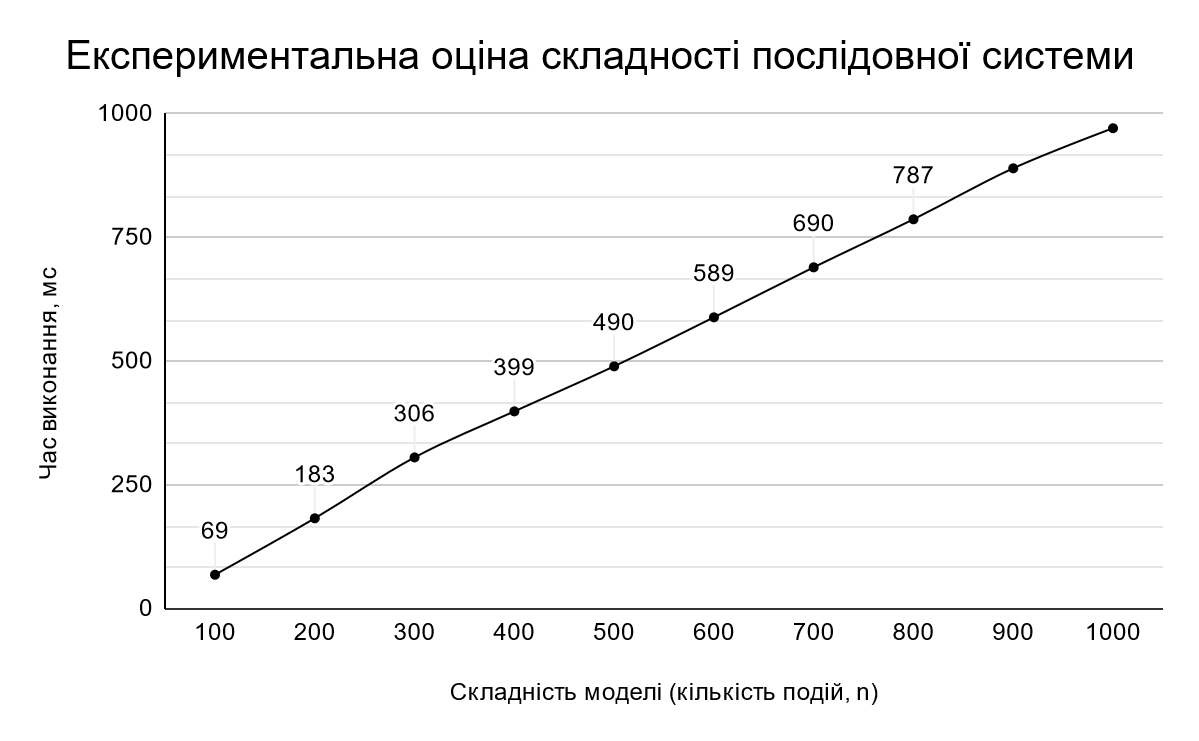


Рисунок 1.5 – Візуалізація метриків систем з послідовних моделей для 10 ітерацій

Де *v* – інтенсивність подій, *timeMod* – час моделювання, *k* – кількість елементарних операцій для обробки однієї події, що вираховується як кількість успішних операцій за одиницю часу ().

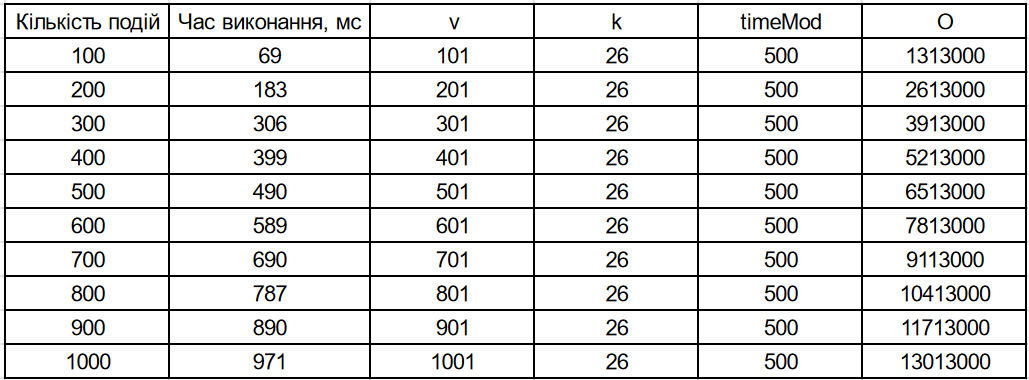


Рисунок 1.6 – Підраховані теоретичні значення складності системи

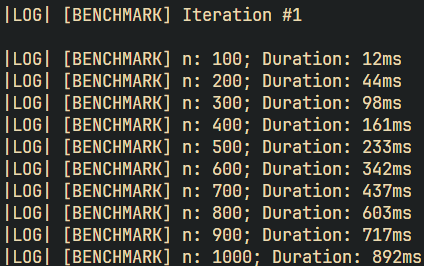


Рисунок 1.7 – Приклад виводу для кожної ітерації системи паралельних моделей

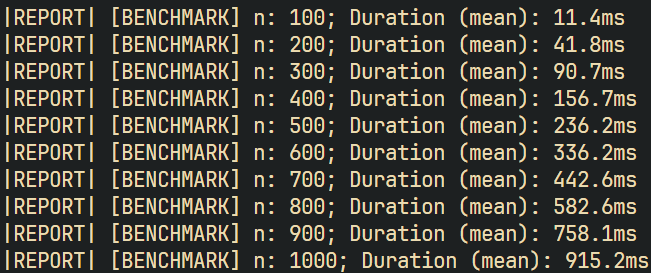


Рисунок 1.8 – Результат роботи бенчмарку системи паралельних моделей

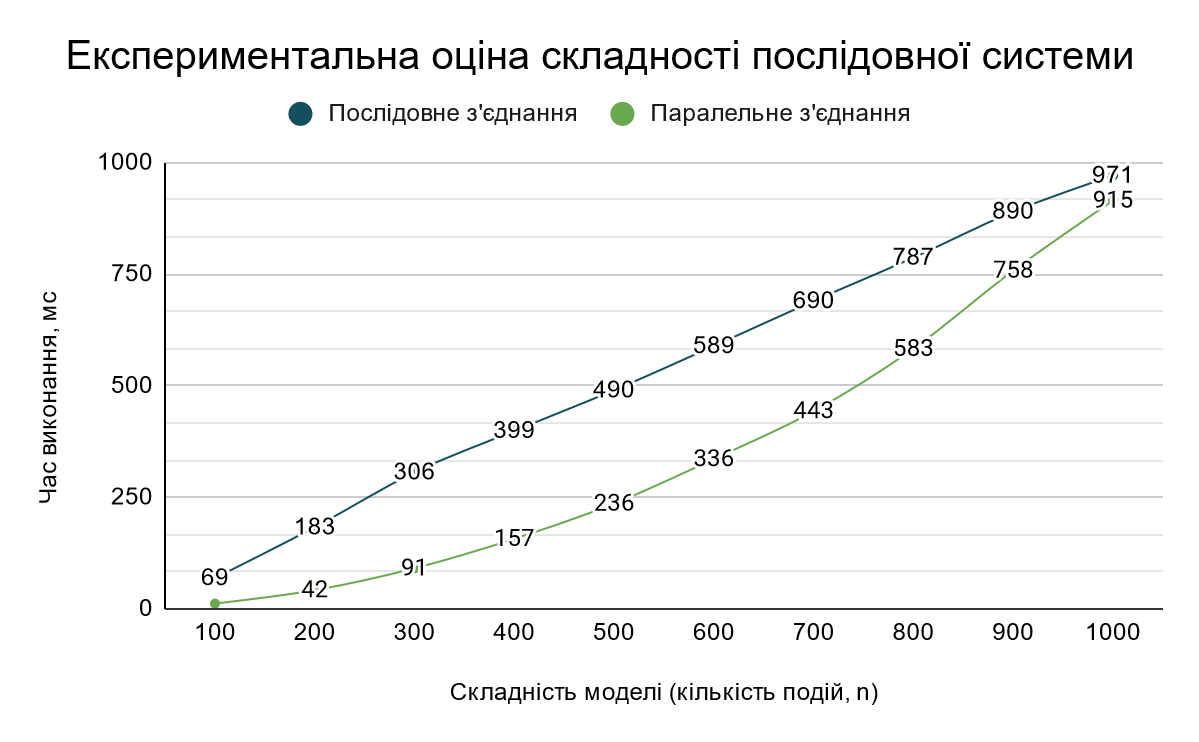


Рисунок 1.9 – Порівняння результатів паралельної та послідовної моделі

# ВИСНОВКИ

У ході виконання лабораторної роботи було створено імітаційні моделі систем масового обслуговування, що дозволило дослідити їх продуктивність та ефективність у послідовному й паралельному виконанні. Було проведено серію експериментів, що охоплюють 10 прогонів бенчмарку з різною кількістю подій від 100 до 1000 із кроком 100, використовуючи експоненційний генератор з математичним очікуванням 1. Результати кожного прогону зафіксовані у вигляді вихідних даних, які демонструють поведінку системи при обробці різного навантаження.

Метрики, отримані в процесі експериментів, дозволили оцінити продуктивність системи для послідовних моделей. Зокрема, було розраховано кількість елементарних операцій на одну подію (EOPs) та складність системи у вигляді формули O(v⋅timeMod⋅k). Для кожної ітерації виконано візуалізацію метрик, що дозволило визначити залежності між параметрами системи та її продуктивністю. Теоретичні значення складності підтвердили відповідність експериментальних даних розрахунковим прогнозам.

Окрім цього, було проаналізовано роботу паралельних моделей системи, результати яких порівняно з послідовними моделями. Порівняння показало, що паралельні моделі демонструють вищу продуктивність при великій інтенсивності подій завдяки можливості одночасної обробки кількох різних задач паралельно.

# ДОДАТОК ПРОГРАМНИЙ КОД

Programs.cs

using System;

using System.Collections.Generic;

using LabWork4.Framework.Core.Controllers;

using LabWork4.Framework.Components.Queues.Concrete;

using LabWork4.Framework.Components.Modules.Common;

using LabWork4.Framework.Components.Modules.Concrete;

using LabWork4.Framework.Components.Workers.Concrete;

using LabWork4.Framework.Components.Schemes.Concrete;

using LabWork4.Framework.Components.Tasks.Utilities.Factories.Concrete;

namespace LabWork4.Application;

file sealed class Program

{

private static void Main()

{

const float SIMULATION\_TIME = 500.0f;

const int BENCHMARK\_MILLISECONDS = 1000;

const int ITERATIONS\_COUNT = 10;

const int LINES\_COUNT = 10;

const int MODULES\_COUNT\_START = 100;

const int MODULES\_COUNT\_DELTA = 100;

const int MODULES\_COUNT\_FINISH = 1000;

// Program.CreateBenchmarkModel().RunSimulation(SIMULATION\_TIME, BENCHMARK\_MILLISECONDS);

IDictionary<int, IList<int>> samples = new Dictionary<int, IList<int>>();

for (int modulesCount = MODULES\_COUNT\_START; modulesCount <= MODULES\_COUNT\_FINISH; modulesCount += MODULES\_COUNT\_DELTA)

samples[modulesCount] = new int[ITERATIONS\_COUNT];

for (int iteration = 0; iteration < ITERATIONS\_COUNT; ++iteration)

{

Console.WriteLine($"\n|LOG| [BENCHMARK] Iteration #{iteration + 1}\n");

for (int modulesCount = MODULES\_COUNT\_START; modulesCount <= MODULES\_COUNT\_FINISH; modulesCount += MODULES\_COUNT\_DELTA)

{

SimulationModelController model = Program.CreateSequentialModel(modulesCount);

// SimulationModelController model = Program.CreateParallelModel(modulesCount, LINES\_COUNT);

model.RunSimulation(SIMULATION\_TIME);

samples[modulesCount][iteration] = model.SimulationDurationMilliseconds;

Console.WriteLine($"|LOG| [BENCHMARK] n: {modulesCount}; Duration: {model.SimulationDurationMilliseconds}ms");

}

}

for (int modulesCount = MODULES\_COUNT\_START; modulesCount <= MODULES\_COUNT\_FINISH; modulesCount += MODULES\_COUNT\_DELTA)

Console.Write($"\n|REPORT| [BENCHMARK] n: {modulesCount}; Duration (mean): {Program.CalculateDurationMean(samples[modulesCount])}ms");

}

private static BenchmarkModelController CreateBenchmarkModel()

{

DisposeModule dispose = new DisposeModule("dispose");

ProcessorModule processor = new ProcessorModule("processor", new SingleTransitionScheme(dispose), new MockExponentialWorker(1.0f), new DefaultQueue(Int32.MaxValue));

CreateModule create = new CreateModule("create", new SingleTransitionScheme(processor), new MockExponentialWorker(1.0f), new MockTaskFactory());

return new BenchmarkModelController(new Module[] { create, processor, dispose });

}

private static SimulationModelController CreateSequentialModel(int modelsCount)

{

ProcessorModule nextProcessor;

ProcessorModule previousProcessor;

IList<Module> modules = new List<Module>();

DisposeModule dispose = new DisposeModule("dispose");

modules.Add((Module)dispose);

previousProcessor = new ProcessorModule($"processor\_{modelsCount}", new SingleTransitionScheme(dispose), new MockExponentialWorker(1.0f), new DefaultQueue(Int32.MaxValue));

modules.Add((Module)previousProcessor);

for (int i = modelsCount - 1; i > 0; --i)

{

nextProcessor = new ProcessorModule($"processor\_{i}", new SingleTransitionScheme(previousProcessor), new MockExponentialWorker(1.0f), new DefaultQueue(Int32.MaxValue));

modules.Add((Module)nextProcessor);

previousProcessor = nextProcessor;

}

modules.Add(new CreateModule("create", new SingleTransitionScheme(previousProcessor), new MockExponentialWorker(1.0f), new MockTaskFactory()));

return new SimulationModelController(modules);

}

private static SimulationModelController CreateParallelModel(int modelsCount, int linesCount)

{

const float MAX\_PROBABILITY = 1.0f;

const int DIRECT\_CREATE\_CONNECTIONS\_COUNT = 1;

float flowProbability = MAX\_PROBABILITY / linesCount;

int modelsPerLineCount = (modelsCount / linesCount) - DIRECT\_CREATE\_CONNECTIONS\_COUNT;

ProcessorModule nextProcessor;

ProcessorModule previousProcessor;

IList<Module> modules = new List<Module>();

DisposeModule dispose = new DisposeModule("dispose");

modules.Add((Module)dispose);

ProbabilityScheme createScheme = new ProbabilityScheme(dispose);

for (int i = 0; i < linesCount; ++i)

{

previousProcessor = new ProcessorModule($"processor\_{i}\_{0}", new SingleTransitionScheme(dispose), new MockExponentialWorker(1.0f), new DefaultQueue(Int32.MaxValue));

modules.Add((Module)previousProcessor);

for (int j = modelsPerLineCount - 1; j >= 0; --j)

{

nextProcessor = new ProcessorModule($"processor\_{i}\_{j}", new SingleTransitionScheme(previousProcessor), new MockExponentialWorker(1.0f), new DefaultQueue(Int32.MaxValue));

modules.Add((Module)nextProcessor);

previousProcessor = nextProcessor;

}

createScheme.Attach(previousProcessor, flowProbability);

}

modules.Add(new CreateModule("create", createScheme, new MockExponentialWorker(1.0f), new MockTaskFactory()));

return new SimulationModelController(modules);

}

private static float CalculateDurationMean(IList<int> samples)

{

float durationTotal = 0.0f;

for (int i = 0; i < samples.Count; ++i)

durationTotal += samples[i];

return durationTotal / samples.Count;

}

}

BenchmarkModelController.cs

using System;

using System.Linq;

using System.Diagnostics;

using System.Collections.Generic;

using LabWork4.Framework.Common;

using LabWork4.Framework.Components.Modules.Common;

namespace LabWork4.Framework.Core.Controllers;

internal sealed class BenchmarkModelController : IStatisticsPrinter

{

private readonly Stopwatch stopwatch;

private readonly IList<Module> modules;

private float timeNext;

private float timeCurrent;

internal BenchmarkModelController(IList<Module> modules)

{

if (modules == null)

throw new ArgumentNullException($"{nameof(modules)} cannot be null.");

this.stopwatch = new Stopwatch();

this.timeNext = 0.0f;

this.modules = modules;

this.timeCurrent = 0.0f;

}

internal int SimulationDurationMilliseconds => this.stopwatch.Elapsed.Milliseconds;

internal void RunSimulation(float simulationTime, int milliseconds)

{

IList<Module> nextModules;

this.timeNext = this.modules.Min(module => module.TimeNext);

this.stopwatch.Restart();

while (timeNext < simulationTime)

{

if (this.stopwatch.ElapsedMilliseconds >= milliseconds)

break;

this.timeCurrent = this.timeNext;

foreach (Module module in this.modules)

module.TimeCurrent = this.timeCurrent;

nextModules = this.modules.Where(module => module.TimeNext == this.timeCurrent).ToList();

foreach (Module module in nextModules)

module.CompleteTask();

this.timeNext = this.modules.Min(module => module.TimeNext);

this.PrintIntermediateStatistics();

}

this.stopwatch.Stop();

this.PrintFinalStatistics();

}

public void PrintIntermediateStatistics()

{

foreach (Module module in this.modules)

module.PrintIntermediateStatistics();

}

public void PrintFinalStatistics()

{

foreach (Module module in this.modules)

module.PrintFinalStatistics();

}

}

SimulationModelController.cs

using System;

using System.Linq;

using System.Diagnostics;

using System.Collections.Generic;

using LabWork4.Framework.Common;

using LabWork4.Framework.Components.Modules.Common;

namespace LabWork4.Framework.Core.Controllers;

internal sealed class SimulationModelController : IStatisticsPrinter

{

private readonly Stopwatch stopwatch;

private readonly IList<Module> modules;

private float timeNext;

private float timeCurrent;

internal SimulationModelController(IList<Module> modules)

{

if (modules == null)

throw new ArgumentNullException($"{nameof(modules)} cannot be null.");

this.stopwatch = new Stopwatch();

this.timeNext = 0.0f;

this.modules = modules;

this.timeCurrent = 0.0f;

}

internal int SimulationDurationMilliseconds => this.stopwatch.Elapsed.Milliseconds;

internal void RunSimulation(float simulationTime)

{

IList<Module> nextModules;

this.timeNext = this.modules.Min(module => module.TimeNext);

this.stopwatch.Restart();

while (timeNext < simulationTime)

{

this.timeCurrent = this.timeNext;

foreach (Module module in this.modules)

module.TimeCurrent = this.timeCurrent;

nextModules = this.modules.Where(module => module.TimeNext == this.timeCurrent).ToList();

foreach (Module module in nextModules)

module.CompleteTask();

this.timeNext = this.modules.Min(module => module.TimeNext);

// this.PrintIntermediateStatistics();

}

this.stopwatch.Stop();

// this.PrintFinalStatistics();

}

public void PrintIntermediateStatistics()

{

foreach (Module module in this.modules)

module.PrintIntermediateStatistics();

}

public void PrintFinalStatistics()

{

foreach (Module module in this.modules)

module.PrintFinalStatistics();

// Console.WriteLine($"|LOG| [SYSTEM] Duration: {this.stopwatch.Elapsed}");

}

}