УДК 004.424

А.Ю. Дорошенко, М.В. Гнинюк

ПАРАЛЕЛЬНА РОЗПОДІЛЕНА РЕАЛІЗАЦІЯ МОДЕЛЮВАННЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Запропонована паралельна реалізація для раніше створеного інструментарію моделювання гетерогенних паралельних обчислювальних систем, побудованого на основі фреймворку GridSim. Проведена перевірка та первинне дослідження цієї реалізації на прикладі однієї прикладної задачі.

Вступ

Сьогодні паралельні обчислювальні системи все більше використовуються як в наукових, так і в промислових застосуваннях. Вони дозволяють ефективно використовувати існуючі різнорідні паралельні ресурси та вирішувати задачі великих обсягів, які не можуть бути вирішені на окремих паралельних комплексах. Проте Grid-системи є складними як для створення та конфігурування, так і для програмування. При цьому підбір найбільш оптимальних конфігурацій як для структури самого Grid-середовища, так і для задач, що виконуються на ньому, недоцільно здійснювати на реальних системах через великі витрати та необхідність координації різних учасників Grid-проектів. Тому актуальною є задача моделювання, яка дозволяє без витрат на створення, підтримку та використання реальної Grid-системи провести на моделі необхідні експерименти, результати яких не будуть суттєво відрізнятися від оригіналу.

Слід зазначити, що актуальною є проблема моделювання саме гетерогенних Grid-систем, що включають до свого складу центральні процесори (CPU), а також як графічні прискорювачі (відеоадаптери, GPU) через їх високу продуктивність останніх, їх доступність за ціною і енергоефективність. Одним з популярних засобів моделювання паралельних систем (кластерів, Grid-систем), окремі вузли яких мають гетерогенний характер і містять як CPU, так і GPU-компоненти, є фреймворк GridSim [1]. Паралельна система з CPU + GPU за допомогою сутностей GridSim може бути представлена у вигляді ресурсу з двома EOM: перша EOM визначає кількість ядер CPU та їх рейтинг, друга – кількість ядер GPU та їх рейтинг. Ядра CPU та GPU моделюються у вигляді обчислювальних елементів (PE). Моделювання CPU і GPU-компонентів з використанням базових сутностей GridSim дозволить надалі спростити включення розробленої моделі в більш складні конфігурації гетерогенних систем, зокрема кластери і Grid-системи з GPU-вузлами.

У роботі [2] запропонована архітектура гнучкого і розширюваного середовища моделювання гетерогенних Grid-систем gpusim на базі Java-фреймворку GridSim. Особливістю системи є такий підхід до опису гетерогенної паралельної системи, при якому CPU і GPU-компоненти розглядаються як окремі вузли віртуального Grid-середовища. Описано прототип інструментальної системи для моделювання, заснований на використанні запропонованої архітектури, а також механізм налаштування системи на параметри конкретної паралельної системи за рахунок автоматизованого підбору параметрів моделі. Експерименти показали достатню точність побудованої моделі для великих розмірів вхідних даних та ефективність застосування системи моделювання. В роботі [3] ці дослідження продовжені на прикладі відомої прикладної задачі гравітаційної взаємодії N тіл, де було поставлено більш складне питання про пошук оптимального значення параметрf конфігурації обчислювальної системи для отримання більш ефективної програми.

В даній роботі виконано наступний

крок у розвитку інструментарію моделювання у напрямку підвищення ефективності цього інструментарію за рахунок створення його паралельної розподіленої реалізації. Матеріал статті організований наступним чином. У розділі 1 розглянуто основні засади імітаційного моделювання, наведено огляд існуючих рішень для моделювання Grid-систем та обгрунтована необхідність розробки інструментарію gpusim. У розділі 2 описана архітектура інструментарію gpusim та його паралельна розподілена реалізація. У розділах 3 та 4 розглядається застосування паралельної версії інструментарію gpusim для моделювання конкретної прикладної задачі (блочний алгоритм множення матриць. Роботу завершують висновки та напрямки подальшої роботи.

1. Засоби імітаційного моделювання

Імітаційне моделювання є найбільш потужним, а іноді і єдиним методом дослідження динамічної поведінки складних систем [4]. Імітаційна модель відтворює поведінку модельованої системи у часі, для якого використовуються три категорії часу: фізична, модельна і процесорна. Фізичний час стосується системи, що моделюється, модельний – відтворення фізичного часу в моделі, а процесорним часом називається час виконання імітаційної моделі на комп'ютері. Співвідношення фізичного і модельного часу визначається специфікою системи і задається величиною діапазону фізичного часу, прийнятого за одиницю модельного часу.

Змістом імітаційного моделювання є просування модельного часу і виконання подій, пов'язаних з певними його значеннями, а основним завданням імітаційного моделювання є правильне відображення порядку подій у системі, що моделюється, на порядок виконання подій у моделі.

Моделювання складних систем може вимагати значних витрат процесорного часу. Тому важливим завданням системи імітаційного моделювання є зменшення процесорного часу, наприклад, за рахунок паралельного виконання подій на мультипроцесорній техніці. Хоча у деяких прикладних галузях, навпаки, може виникнути потреба у штучному збільшення процесорного часу, наприклад, для моделей тренажерів з метою наблизити віртуальне середовище, що генерується комп'ютером, до реального.

У даній роботі розглядається задача мінімізації часу імітаційного моделювання виконання Grid-програм на відеографічних процесорах за рахунок розпаралелювання програми моделювання.

На сьогоднішній день розроблено чимало інструментальних засобів моделювання Grid-систем. Серед них імітаційні моделі використовуються для більш точних вимірів продуктивності Grid-систем при конкретних значеннях параметрів. Вони дозволяють отримати довільну точність моделювання за рахунок врахування всіх необхідних деталей. Але на практиці потрібно досягати балансу між точністю моделювання та обчислювальною складністю.

Серед існуючих рішень можна виділити імітаційну модель, розроблену для дослідження ефективності методів планування ресурсів для різних інтенсивностей потоків завдань у Grid [5] і її програмну реалізацію GRID_Scheduler_Model. Моделюється гомогенна Grid-система, і для підтримки гетерогенних Grid модель потребує доопрацювання, а розширюваність у реалізації не передбачена.

На даний момент існує кілька інструментаріїв (фреймворків), які дозволяють розробнику на їх основі створити свою модель Grid і проводити на ній експерименти. Огляд і порівняльний аналіз найбільш функціональних та стабільних фреймворків MicroGrid, OptorSim, SimGrid, GridSim, Bricks наведено в [6, 7]. На підставі порівняння можна зробити висновок, що найбільш зручним і функціональним є GridSim [1], фреймворк для мови програмування Java.

Окремим питанням є моделювання часу виконання задач на GPU, зокрема такі аспекти їх функціонування запропоновано детальну аналітичну модель як час виконання та споживання енергії [8, 9], а також

порівняння продуктивності CPU та GPU [10, 11]. Застосування таких засобів моделювання дозволяє досягти великої точності моделювання, але потребують значних ресурсів для виконання моделі. До того ж вони не передбачають вбудовування у моделі Grid-середовища.

У роботах [2, 3] запропоновані спеціалізовані засоби імітаційного моделювання gpusim, засновані на інструментарії GridSim і спрямовані на моделювання роботи GPU з метою визначення оптимальних значень параметрів прикладних задач, що забезпечують найбільше прискорення виконання цих задач на GPU. Для збільшення ефективності використання таких засобів актуальною задачею є розпаралелювання програм самого імітаційного моделювання, що і є метою даної роботи.

2. Розподілено – паралельна система моделювання gpusim

2.1. Основні концепції фреймворку Hazelcast. Hazelcast – це платформа з відкритим програмним кодом для Java, яка використовується для побудови кластерів та масштабованого розподілу даних. Серед властивостей платформи основними є такі:

- велика швидкість – тисячі операцій на секунду;
- відмовостійкість – без втрати данних після аварій;
- динамічне масштабування – по мірі додавання нових серверів;
- легке використання – досить додати у свій проект один jar файл.

Завдяки розподіленості структур даних, можливості розподіленого кешування, гнучкості кешування та інтеграції з функціями Spring та Hibernate Hazelcast перетворюється на систему промислового використання у як платформа для створення кластерів.

2.2. Можливості Hazelcast. Основні можливості Hazelcast у частині розподіленої паралельної обробки включають такі:

- розподілені функції java.util. (Queue, Set, List, Map та інші);
- розподілена підтримка індексування та запитів;
- підтримка транзакцій шифрування на рівні сокетів для безпеки кластерів;
- віддалений доступ до кластера для Java-клієнтів;
- динамічне розширення кластера та динамічний перерозподіл після падіння одного з вузлів кластера.

2.3. Коли застосовувати Hazelcast? Hazelcast застосовують, коли потрібно вирішити такі задачі:

- організація обміну даними/станом серед багатьох серверів та кешування даних (розподілений кеш);
- кластеризація за стосунку та забезпечення захищенної комунікації між серверами;
- спільне використання даних у пам’яті;
- розподіл завантаження на багато серверів та паралельне виконання задач;
- забезпечення відмово-стійкого управління даними.

2.4. Архітектура розподіленої паралельної системи моделювання. У першій версії системи моделювання gpusim [2] реалізовано послідовну процедуру симуляції задачі на GPU за допомогою засобів GridSim [1]. Послідовна версія виконує всі симуляційні задачі повністю, проте це може займати досить тривалий час, особливо при збільшенні кількості симуляцій та рівня складності задачі, тому вирішено зробити gpusim розподілено-паралельною системою.

На сьогодні всі процесори, що використовуються у вузлах мультипроцесорних систем, є багатоядерними і виконують задачі паралельно, що дає змогу прискорити виконання програм. Але отримання приросту продуктивності за рахунок паралельності є обмеженим фізичними можливостями конкретного комп’ютера на якому виконується симуляція. Тому вирішено зробити gpusim розподілено-паралельним. Це дає досліднику можливість нарощувати обчислювальну по-

тужність шляхом об'єднання комп'ютерів у кластер.

Кластер містить декілька окремих обчислювальних машин, що використовуються спільно і працюють як одна система для вирішення тих чи інших задач, наприклад, для підвищення продуктивності, забезпечення надійності, спрощення, адміністрування тощо. Обчислювальний кластер потрібен для збільшення швидкості обрахунків за допомогою паралельних обчислень.

Для організації кластера використовується згадана вище система Hazelcast. На даному етапі розроблено три застосунки у вигляді jar файлів:

GpuSimV2.jar, GpuSimClusterInstance.jar та GpuSimClusterClient.jar.

Симулятор GpuSimV2 – це оболонка на мові програмування Java над фреймворком GridSim, основною задачею якої є налаштування GridSim на основі даних конфігураційного файлу, запуск симуляції, збір і вивід статистики у файл.

Cluster Instance – являє собою застосунок, що запускається на комп'ютері, який має стати частиною кластера. Програма шукає інші члени у мережі та приєднується до кластера.

Cluster Client – застосунок, який стає частиною кластера та запускає симуляцію на утвореному кластері, шляхом розділення симуляцій поміж членами кластера.

2.5. Реалізація розроблених застосунків. Для збільшення швидкодії процесу симуляції, запропонована розподілено-паралельна система.

Роглянемо побудову елементів GpuSimClusterInstance та GpuSimClusterClient.

Екземпляр кластера GpuSimClusterInstance – це компонент, який дозволяє перетворити комп'ютери, які з'єднані між собою локальною мережею, на кластер. Для цього треба запустити GpuSimClusterInstance таким чином:
java –classpath GpuSimClusterInstance.jar com.mgnyniuk.base.Main

У результаті запуску декількох екземлярів буде утворений кластер:

```
Members [2] {
    Member [192.168.100.37]:5701
    Member [192.168.100.37]:5702 this
}
```

GpuSimClusterInstance складається з таких пакетів:

- com.mgnyniuk.base
- com.mgnyniuk.parallel

До пакета com.mgnyniuk.base належить клас Main, який має метод *main*(*String*[] *args*), який запускає кластер та його конфігурує:

```
public static void main(String[] args) {
    Config cfg = new Config();
    cfg.setProperty("hazelcast.icmp.timeout",
"2000000000");
    HazelcastInstance hz =
=Hazelcast.newHazelcastInstance(cfg);

    outputMap = hz.getMap("outputMap");
  }
```

За допомогою конфігурації ми можемо задавати різні параметри для налаштування кластера. OutputMap – це структура даних для зберігання результатів симуляції у кластері.

До пакета com.mgnyniuk.parallel належать класи, які відповідають за паралельну функціональність системи та ConfigurationUtil, який включає такі засоби для роботи з конфігураціями, як серіалізація конфігурацій та десеріалізація результатів та збереження у OutputMap.

Основним класом у паралельній частині є Runner, який також використовує функціональність, яка з'явилась у Java 1.5 і підримується в Hazelcast – Executor framework. Це дозволяє асинхронно виконувати поставлені задачі та виконувати іншу роботу водночас. Є також можливість керувати часом виконання задачі, а саме:

– задача виконується вчасно, отримується результат та продовжується робота;

– якщо задача виконується більше часу ніж очікувалось, то є можливість скасувати її та продовжити роботу.

Клас Runner реалізує інтерфейси Callable<Boolean> та Serializable.

```
public Boolean call() throws IOException {

ConfigurationUtil.deserializeConfigs(gridSim
ConfigList, startIndex);
    ThreadListener threadListener = new
ThreadListener();
    for (int j = 0; j <
< overallProcessesQuantity/partProcessesQua
ntity; j++) {
    for (int i = startIndex; i < startIndex +
+ partProcessesQuantity; i++) {

   NotifyingThread notifyingThread = new
WorkerThread("GpuSimV2.jar",
   String.format(CONFIG, (i + j *
partProcessesQuantity)),
   String.format(OUTPUT, (i + j *
partProcessesQuantity)));

notifyingThread.addListener(threadListener);
      notifyingThread.start();
       }
while
(threadListener.quantityOfEndedThreads ! =
= partProcessesQuantity) {
System.out.print(threadListener.quantityOfEn
dedThreads);
          continue;
       }

threadListener.quantityOfEndedThreads = 0;
    }
ConfigurationUtil.loadOutputs(startIndex,part
ProcessesQuantity);
    return true;
}
```

Метод *call*() викликається при виконанні симуляції на кожному члені кластера. Він виконує такі функції:

– серіалізація отриманих конфігурацій;

– виконання розпаралелювання симуляцій шляхом запуску кожного GpuSimV2.jar у новому потоці, який створює для кожного працюючого GpuSimV2.jar свій процес операційної системи. Система може створювати "порції" паралельних потоків, розмір яких визначається змінною partProcessesQuantity. Це дає змогу оптимально використовувати обчислювальну потужність комп'ютера який виконує задачу симуляції. Після закінчення "порції", якщо не опрацьовані всі конфігурації, то запускається нова і т. д., тоді коли симуляція закінчується, то файли отримані з GpuSimV2 десеріалізуються та зберігаються в outputMap. В outputMap свої результати складаються всі учасники кластера.

Клієнт та екземпляр кластера GpuSimClusterClient – це компонент, який запускає симуляцію на кластері та стає складовою кластера.

GpuSimClusterClient запускається таким чином:

java –classpath GpuSimClusterClient.jar com.mgnyniuk.base.Main

GpuSimClusterClient складається аналогічно до GpuSimClusterClient з пакетів com.mgnyniuk.base та com.mgnyniuk.parallel.

Склад пакетів є таким самим, як і у GpuSimClusterInstance. Відрізнаються тільки реалізацією метода *main*(*String*[] *args*), більш широкою функціональністю ConfigurationUtil та доповненним методом *call*() класу Runner для обробки додаткових симуляцій, які залишаються від ділення задачі на встановлені "порції" паралельних потоків. ConfigurationUtil також відповідає за підготовку конфігурацій для симуляцій запропонованої задачі на GPU.

Метод *main*(*String*[] *args*), є відповідальним за запуск симуляції на Gridi.

Метод виконує такі дії:

– визначає кількість членів кластера;

– розділяє задачі для членів кластера і вказує кількість паралельних потоків на які ділити задачі та ініціює їх обробку;

– після закінчення обробки десе-

ріалізує outputMap та зберігає результати;

– вимірює час виконання симуляцій на кластері.

3. Моделювання задачі множення матриць

3.1. Задача множення матриць блочним алгоритмом. Для перевірки точності та актуальності розробленого середовища моделювання гетерогенних Grid-систем обрана задача множення матриць за допомогою блочного алгоритму [12].

Результатом множення матриці A розмірністю $m \times n$ й матриці B розмірності $n \times l$ є матриця C розмірності $m \times l$, кожен елемент якої обчислюється наступним чином:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj},\ i = 0,..,m,\ j = 0,..,l.$$

Цей алгоритм вимагає $m \cdot n \cdot l$ операцій множення й додавання елементів матриць. При множенні квадратних матриць розмірності $n \times n$ кількість виконаних операцій має порядок $O(n^3)$. Оскільки для обчислення одного елемента результуючої матриці потрібні тільки відповідні рядок та стовпець матриць, що множаться, цей алгоритм природно розділяється на незалежні потоки обчислень.

Якщо ми множимо дві квадратні матриці розмірністю $n \times n$, то результуюча матриця C буде тієї самої розмірності, що й вхідні матриці A і B. Спочатку матриця C розбивається на прямокутні блоки розмірністю $k \times p$. Кожен з цих блоків обчислюється незалежно окремою задачею, для чого потрібно передати блок матриці A розмірністю $n \times k$ й відповідний блок матриці B розмірністю $p \times n$.

Перевагою такого алгоритму є те, що задачу можна розбивати на довільну кількість підзадач, а не тільки на квадратні блоки. Також відсутність будь-яких залежностей між підзадачами дозволяє ефективно виконувати обчислення у випадку коли кількість наявних процесорів значно менша за кількість підблоків – у такому випадку підзадачі отримують нові блоки «за готовністю». Недоліком є додаткові витрати пам'яті, які виникають при частковому дублюванні вхідних даних для різних підзадач. Загальна кількість операцій не відрізняється від послідовного алгоритму й має порядок $O(n^3)$.

В роботі [14] також побудовано емпіричну модель часу виконання блочного алгоритму на GPU. Ця модель має наступний характер:

$$T_{gpu} = N^2 T_{st.global} + N^3 T_{ld.global},$$

де $T_{ld.global}$ та $T_{st.global}$ – параметри, що визначають час завантаження та збереження даних при роботі з глобальною пам'яттю GPU. Ці параметри є частиною опису моделі і мають підбиратись для конкретних обчислювальних вузлів експериментальним шляхом.

3.2. Експериментальний модуль множення матриць блочним алгоритмом. На основі емпіричної моделі часу виконання задачі множення матриць на GPU [14] розроблені генератор експериментів симулятора і обробник статистики, що входять до складу експериментального модуля MatrixMultiply. Вхідні параметри генератора – розмір блоку, мінімальний і максимальний розмір матриці, а також інкремент розміру матриці. Оброблювач статистики отримує вихідні дані симулятора, перетворює їх для наочного відображення кінцевому користувачу, а також порівнює час кожної з симуляцій з результатами експерименту на реальній паралельній системі.

Генератор має ряд попередньо встановлених параметрів, що не відносяться безпосередньо до експерименту, але впливають на час симуляції:

- кількість обчислювальних елементів CPU і GPU, а також їх рейтинги в одиницях MIPS (million instructions per second):

cpuMachinePECount,

cpuMachinePERating,

gpuMachinePECount,

gpuMachinePERating;

- пропускна здатність підключення між ресурсами і розподільниками завдань: resourceBaudRate, linkBaudRate;

- вартість використання ресурсів Grid – resourceCostPerSec;

- вартість операцій роботи із пам'яттю – це завантаження та збереження даних:

loadOperationCost,

saveOperationCost.

Дані параметри залежать від внутрішньої структури Grid-системи, відповідної моделі, і для проведення подальших досліджень з моделлю, необхідно знайти їх точні значення за допомогою модуля оптимізації констант.

4. Тестування розподілено – паралельної версії gpusim та аналіз отриманих результатів

4.1. Проведення тестування системи. Для проведення тестування системи вирішено використати наявні ресурси, а саме: 4 комп'ютери, які мають по 4 ядра кожний однакової обчислювальної потужності. Тестування проводилось на 256 симуляціях, що є стандартними налаштуваннями для GpuSim першої версії. Виконано 5 вимірів часу виконання симуляції:

- послідовно,
- паралельно на 4 ядрах.
- паралельно на 8 ядрах,
- паралельно на 12 ядрах,
- паралельно на 16 ядрах.

Симуляції, які виконувались на 1, 4, 8 та 16 ядрах паралельно запускались партіями по 16 потоків для оптимального використання ресурсів, на 12 ядрах, так як це 3 комп'ютери, то розмір партій дорівнював 17, остання симуляція (256 – та) додатково виконана на останньому доданому члені кластеру – клієнті.

4.2. Аналіз отриманих результатів. Отримані результати приводяться в табл. 1.

Таблиця 1

P	T, с
1	161.812
4	69.749
8	44.978
12	33.627
16	24.36

де P – кількість ядер, а T – час виконання процесу симуляції для 256 конфігурацій. На базі отриманих результатів побудовано графік залежності часу виконання від кількості задіяних ядер, який показано на рис. 1.

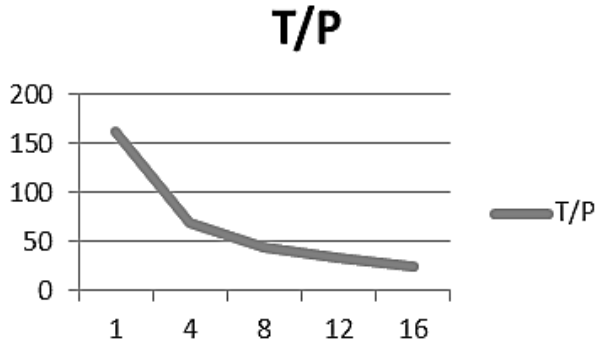

Рис. 1. Графік залежності P/T

З графіка видно істотне прискорення виконання симуляцій.

Також досліджено залежність прискорення зміни часу виконання від кількості ядер. Прискорення обчислюється за формулою:

$$S_p = T_1 / T_p .$$

За допомогою даної формули отримано табл. 2.

Таблиця 2

P	S_p
1	1
4	2.319918565
8	3.59758104
12	4.811966574
16	6.642528736

За табл. 2 побудовано графік, на якому показано, що прискорення наближається до ідеального випадку $S_p = P$ (рис. 2)

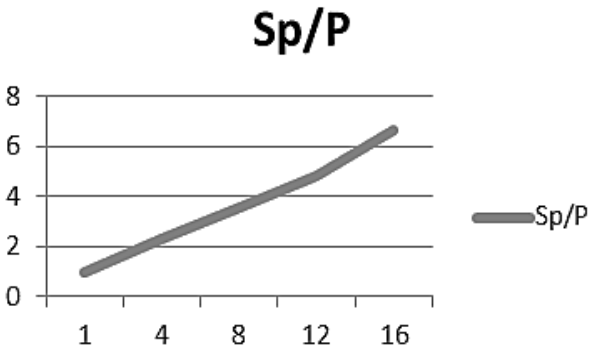

Рис. 2. Графік залежності S_p / P

Висновки

У роботі запропонована архітектура розподілено-паралельної версії GpuSim. Одною з головних задач побудови наступної версії є зменшення часу виконання симуляцій. Для досягнення цієї цілі побудовано нову архітектуру для системи моделювання gpusim.

На цей момент розроблено “ядро” розподілено-паралельної системи, яке перевірено на прикладі задачі множення матриць блочним алгоритмом. Отримані результати, які наведені вище, показують, що приріст швидкодії є істотним та прискорення наближається до графік лінійної залежності $S_p = P$, що підвереджує ефективність обраної архітектури та реалізації. Одним з головних надбань даного рішення є те, що дослідник може нарощувати ресурси для обчислень за своїм бажанням, різної конфігурації та потужності, що дає змогу досягати бажаної швидкодії системи.

На базі розробленого “ядра” буде розроблений інтерфейс для виконання симуляцій, а також для конфігурування роботи розподілено – паралельної системи.

1. *GridSim.* A Grid Simulation Toolkit for Resource Modelling and Application Scheduling for Parallel and Distributed Computing [Электроний ресурс]. – Режим доступу: http://www.buyya.com/gridsim/. – 01.11.2013 р.
2. *Оконський І.В., Дорошенко А.Ю., Жереб К.А.* Інструментальні засоби моделювання гетерогенних середовищ заснованих на відеографічних прискорювачах // Проблеми програмування. – 2013. – № 1. – С. 107–115.
3. *Дорошенко А.Ю., Оконський І.В., Жереб К.А. та ін.* Використання засобів моделювання для визначення оптимальних параметрів виконання програм на відеографічних прискорювачах // Проблеми програмування. – 2013. – № 2. – С. 23–31.
4. *Лоу А.М., Кельтон А.Д.* Имитационное моделирование. – СПб: Издательская группа BHV, 2004. – 847 с.
5. *Минухин С.В., Знахур С.В.* Исследование эффективности методов планирования ресурсов для различных интенсивностей потоков заданий в Грид // Радіоелектронні і комп’ютерні системи. – 2012. – № 1 (53). – С. 165–171.
6. *Петренко А.І.* Комп’ютерне моделювання грід-систем // Электроника и связь. – 2010. – № 5. – С. 40–48.
7. *Кореньков В.В., Нечаевский А.В.* Пакеты моделирования DATAGRID // Системный анализ в науке и образовании. – 2009. – № 1. – С. 1–15.
8. *Baghsorkhi S.S., Delahaye M., Patel S.J., et al.* An adaptive performance modeling tool for GPU architectures // SIGPLAN Not. – 2010. – Vol. 45, N 5. – P. 105–114.
9. *Hong S., Kim H.* An integrated GPU power and performance model // SIGARCH Comput. Archit. News. – 2010. – Vol. 38, N 3. – P. 280–289.
10. *Kerr A., Diamos G., Yalamanchili S.* Modeling GPU-CPU workloads and systems // Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. – 2010.– P. 31–42.
11. *Kerr A., Diamos G., Yalamanchili S.* A characterization and analysis of PTX kernels // IEEE International Symposium on Workload Characterization (IISWC 2009). – 2009. – P. 3–12.
12. *Жереб К.А., Ігнатенко О.П.* Моделювання задачі множення матриць для відеографічних прискорювачів // Матеріали конференції "Високопродуктивні обчислення" (HPC-UA 2012). Київ, 08–10 жовтня 2012. – С. 174–181.
13. *Qt* Developer Network [Электроний ресурс]. – Режим доступу: http://qt-project.org/. – 01.11.2012 г.

14. *Intel* Core i7-3770k Processor [Электроний ресурс]. – Режим доступу: http://ark.intel.com/products/65523. – 01.11.2012 г.
15. *GeForce* GTX 650 Specifications [Электроний ресурс]. – Режим доступу: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-650/specifications . – 01.11.2012 г.
16. *Hazelcast* Documentation [Электроний ресурс]. – Режим доступу: http://www.hazelcast.com/docs/3.1/manual/single_html/#

Одержано 16.07.2013

Про авторів:

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор, завідувач відділу теорії
комп'ютерних обчислень Інституту
програмних систем НАН України,
професор кафедри автоматики і управління в технічних системах НТУУ "КПІ",

Гнинюк Максим Володимирович,
студент факультету інформатики та
обчислювальної техніки,
кафедри автоматики і управління
в технічних системах НТУУ "КПІ".

Місце роботи авторів:

Національний технічний
університет України "КПІ"
03056, Київ-56,
проспект Перемоги, 37
Тел.: (044) 236 7989

Інститут програмних систем
НАН України,
03680, Київ-187,
проспект Академіка Глушкова, 40.
Тел.: (044) 526 1538.
E-mail: dor@isofts.kiev.ua
mgnyniuk@gmail.com