УДК 681.3

К.А. Березовський

ДОСЛІДЖЕННЯ ПАРАЛЕЛЬНОГО АЛГОРИТМУ ПОБУДОВИ ДІАГРАМИ ВОРОНОГО НА ПЛОЩИНІ

Зпропонована маштабована паралельна реалізація алгоритму побудови діаграми Вороного на площині. Описано процес розробки алгоритму та програмування з використанням бібліотеки Intel Threading Building Blocks (TBB). Проведене дослідження продуктивності та наведені результати експериментів.

Вступ

Діаграми Вороного [1], отримали свою назву в честь українського математика Георгія Феодосійовича Вороного (1868-1908), що досліджував загальний *n*-вимірний випадок побудови таких об`єктів [2]. Можна нарахувати немало предметних областей, у яких зазвичай використовують такі конструкції [3], наприклад, в астрономії – ідентифікація скупчень зірок та галактик; в біології, екології, та лісництві – моделювання та аналіз конкуренції рослин; у картографії – інтеграція знімків зроблених супутником тощо.

До поняття діаграми Вороного наводять багато прикладних задач, наприклад, наступна: нехай маємо двовимірну електронну карту міста, причому кожній будівлі ставиться у відповідність адреса у традиційній формі (вулиця, номер будинку) і пара координат (x, y). Правопорядок у місті забезпечують N патрулів, які майже весь час перебувають у русі. Роботу координує диспетчерська служба, яка має інформацію щодо їхнього положення. До обов`язків диспетчерів входить:

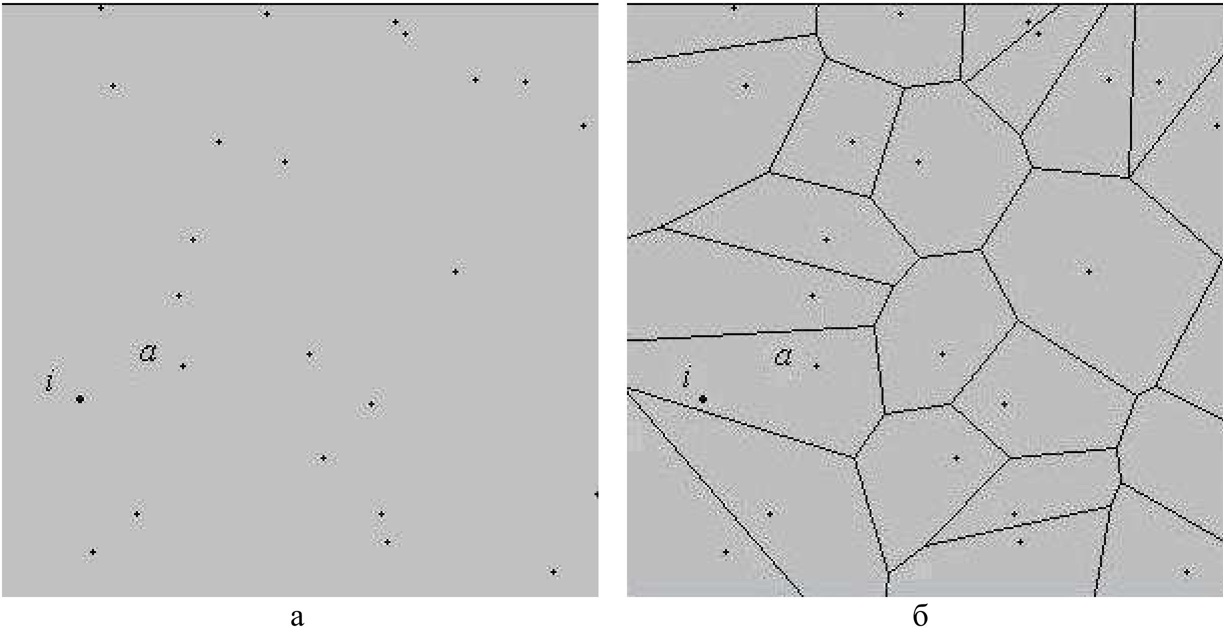
- прийняття звернень від громадян, які надають адресу инциденту в традиційній формі;
- надання відповідних вказівок міліціонерам.

Необхідно розробити механізм, за яким диспетчерська служба могла б швидко давати доручення лише тим міліціонерам, які знаходяться найближче до місця інциденту. Математично суть задачі можна представити таким чином. Маємо площину. Нехай S – множина її точок, в яких знаходяться патрулі у деякий момент часу (рис. 1, а), i – точка, в якій стався інцидент. Наша мета полягає у тому, щоб знайти таку точку $a \in S$, що $dist(i,a) = \min_{b \in S} dist(i,b)$, тобто визначити міліціонерів, які зможуть прибути на місце раніше за інших. Періодично диспетчерська служба визначає координати патрулів (координати точок множини S). За точками цієї множини будується діаграма Вороного $Vor(S)$ (рис. 1, б). Коли надходить інформація про інцидент, то вона трансформується з традиційної форми (вулиця, номер будинку) у координатну (точка i). При цьому визначається, якому многокутнику Вороного із побудованої діаграми належить i, відповідний екіпаж (точка a) отримує доручення.

Можливий випадок коли i лежить на ребрі діаграми, таким чином „найближчими” до i є одразу декілька патрулів (не більше ніж три [1]). У такому разі система обирає одну з них. Діаграма Вороного для розв`язання зазначеної задачі може мати досить великі розміри. Крім того вона не є статичною. У загальному випадку, патрулі постійно знаходяться у русі, тому через невеликі проміжки часу слід перебудовувати діаграму, враховуючи зміни. Саме тому для досягнення необхідної ефективності слід розробляти паралельні алгоритми побудови діаграми Вороного, що дозволяє розв`язувати задачу в реальному часі.

З геометричних міркувань на множину точок S зазвичай [1] накладається обмеження, що жодні чотири її точки не лежать на одному колі. Це забезпечує, що у будь-якій вершині діаграми Вороного перетинається рівно три ребра, і гарантує існування єдиного центра описаного кола,

© К.А. Березовський, 2009



а б

Рис. 1, а) схематичне зображення положення патрулів; б) діаграма Вороного $Vor(S)$

що проходить через відповідні точки S. Зазначена умова є основною для перевірки допустимості вхідних даних. При її порушенні побудова діаграми не є можливою [1].

1. Алгоритм побудови діаграми Вороного

У роботі [1] наведено послідовний алгоритм, що є гарною основою для побудови його паралельного аналога. Нехай N – множина натуральних чисел, $\{\}$ – порожня множина, S – множина точок площини, елементами якої є упорядковані пари $p(x, y)$, де $x, y \in N$. Відношення лінійного порядку R на S введемо так:
нехай $p'(x', y') \in S$, $p''(x'', y'') \in S$, тоді $p' R p''$ якщо $\begin{bmatrix} x' < x'' \\ x' = x'', y' < y'' \end{bmatrix}$. Нагадаємо, що згідно означення [4] відношення R має такі властивості:

- рефлексивність – $\forall p(pRp)$;
- антисиметричність – $\forall p_1 \forall p_2 (p_1 R p_2 \wedge p_2 R p_1 \Rightarrow p_1 = p_2)$;
- транзитивність – $\forall p_1 \forall p_2 \forall p_3 (p_1 R p_2 \wedge p_2 R p_3 \Rightarrow p_1 R p_3)$;
- порівнюваність – $\forall p_1 \forall p_2 (p_1 R p_2 \vee p_2 R p_1)$.

Тепер маємо змогу виконати розріз множини S на дві підмножини S_1 та S_2. Причому $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \{\}$, $\forall p_1 \forall p_2 (p_1 \in S_1, p_2 \in S_2 \Rightarrow p_1 R p_2)$. Використавши підхід „розподіляй і володарюй” [5], отримаємо наступний послідовний алгоритм.

Вхід. Множина S, що складається з точок площини.

Вихід. Множина многокутників Вороного.

Крок 1. Розділити множину S на дві приблизно рівні підмножини S_1 та S_2, $S_1 \cap S_2 = \{\}$.

Крок 2. Рекурсивно побудувати $Vor(S_1)$ та $Vor(S_2)$.

Крок 3. Побудувати ламану d („розділяючий ланцюг”), що розділяє S_1 та S_2.

Крок 4. Вилучити всі ребра діаграми $Vor(S_2)$, що розташовані зліва від d, та всі ребра $Vor(S_1)$, що розташовані справа від d. Отримаємо $Vor(S)$ – діаграму Вороного для множини S.

Зазначимо, що початкове розбиття S, може бути виконане за час $O(N)$ (за допомогою алгоритму пошуку медіани). Крок 4 може бути виконаний за час $O(|S_1| + |S_2|) = O(N)$. Часова склад-

ність послідовного алгоритму становить $O(n \log n)$.

1.1. Паралельний алгоритм побудови діаграми Вороного

Розділимо множину точок S лінійно на частини. Кількість груп точок дорівнює кількості вузлів обчислювальної системи, що виділені для виконання задачі. Паралельний алгоритм має вигляд:

Вхід. Множина S, що складається з точок площини.

Вихід. Множина многокутників Вороного.

Крок 1. Визначити кількість наявний вузлів P, $(P \geq 2)$. Розділити множину S на P приблизно рівних підмножин $S_1, S_2, \ldots S_P$, причому $S_i \cap S_j = \{\}$, $i = \overline{1, P}$, $j = \overline{1, P}$, $i \neq j$.

Крок 2. Паралельно і рекурсивно побудувати $Vor(S_1), Vor(S_2), \ldots Vor(S_P)$.

Крок 3. $I := P$. Виконати K разів, де

$$K = \begin{bmatrix} \log_2 P, \{\log_2 P\} = 0 \\ \log_2 P + 1, \{\log_2 P\} \neq 0 \end{bmatrix}.$$

Крок 3, а) паралельно побудувати $\left\lceil \frac{I}{2} \right\rceil$ ламаних (розділяючих ланцюгів), що лежать між S_{i-1} та S_i для $i = \overline{2, \left(2 * \left\lceil \frac{I}{2} \right\rceil\right)}$;

Крок 3, б) паралельно вилучити всі ребра діаграми $Vor(S_{i-1})$, розташовані зправа відповідної ламаної, та всі ребра діаграми $Vor(S_i)$, розташовані зліва відповідної ламаної, для $i = \overline{2, \left\lceil \frac{I}{2} \right\rceil}$, отримати діаграму $Vor(S_j)$, для $j = \overline{1, \left(\left\lceil \frac{I}{2} \right\rceil + \mathrm{mod}(I, 2)\right)}$, $I := \left\lceil \frac{I}{2} \right\rceil + \mathrm{mod}(I, 2)$.

У роботі [6] показано, що на моделі абстрактної паралельної машини CREW PRAM [7], часова складність даного алгоритму становить $O(\log^2 n)$.

2. Аналіз вимог до засобів розробки

Першим кроком у реалізації паралельного алгоритму є вибір мови програмування та бібліотеки високорівневих інструментів (якщо вона існує). Програмні засоби мають задовольняти таким головним вимогам:

- швидкодія результуючого (виконуваного) коду;
- якомога вищій рівень абстракції, що сприятиме збільшенню швидкості програмування, а також дозволить розробнику мислити категоріями задач, а не потоків;
- можливість роботи на різних архітектурах та платформах;
- маштабованість, тобто автоматичне визначення кількості ядер процесора, та їхнє ефективне використання.

Для експериментальної побудови паралельної реалізації алгоритму в даній роботі прийнята широковідома платформа Intel Threading Building Blocks (TBB) [8]. Було вирішено порівняти деякі аспекти цього прикладного програмного інтерфейсу (API) з іншими засобами організації багатопотокової роботи, а саме: OpenMP [9] та потоками операційної системи (ОС).

Якщо розглянути ці альтернативи із позиції складності розробки, можемо констатувати, що використання вбудованих потоків ОС є складнішим, у порівнянні з TBB та OpenMP. Одна із причин полягає у тому, що зазначені бібліотеки створюють так званий „пул потоків", і за його допомогою відбувається автоматична синхронізація та планування роботи, що приведена в таблиці.

Взявши до уваги мову програмування, що буде використовуватися, маємо такі попередні висновки. Для розробки на C або Fortran краще використати OpenMP, адже цей API зручніший для структурного програмування, що може суттєво зменшити витрати часу на розробку для не громіздких проектів. З тих же причин варто зупинити свій вибір на OpenMP, якщо пла-

Таблиця 1. Порівняння можливостей альтернатив за деякими критеріями

Критерій	TBB	OpenMP	Потоки ОС
Організація паралелізму на рівні задач	Так	Так	Ні
Підтримка декомпозиції даних	Так	Так	Ні
Складені шаблони паралельної обробки	Так	Ні	Ні
Узагальнені шаблони паралельної обробки	Так	Ні	Ні
Підтримка маштабованого вкладеного паралелізму	Так	Ні	Ні
Інтегроване розподілення навантаження	Так	Так	Ні
Статичне розподілення	Ні	Так	Ні
Паралельні структури даних	Так	Ні	Ні
Маштабоване розподілення пам`яті	Так	Ні	Ні
Задачі орієнтовані на ввід/вивід	Ні	Ні	Так
Елементи синхронізації на рівні користувача	Так	Так	Ні
Не вимагається підтримка певного компілятора	Так	Ні	Так
Підтримка у різноманітних ОС	Так	Так	Ні

нується програмування на С++ і передбачаються численні операції обробки масивів. Якщо планується широке використання об`єктно-орієнтованого підходу, шаблонів С++, то варто обрати TBB. Складність моделей програмування із вбудованими потоками операційної системи, для мов С та С++, приблизно однакова. Але той факт, що розбиття на потоки має бути описане у функціональному стилі, зумовлює використання мови С для програмування потоків ОС, більш природнім. У С++-програмах, орієнтованих на роботу з об`єктами, використання вбудованих потоків операційної системи може порушити стиль програмування та іти у розріз із проектним рішенням.

Для TBB та потоків ОС підтримка певного компілятора не є необхідною, на відміну від OpenMP. Для роботи із цим API слід використати компілятор, що розпізнає pragma-директиви, процедури та змінні оточення OpenMP [9]. Але варто зазначити, що розробники багатьох сторонніх компіляторів вже забезпечують деяку підтримку цього API.

Код, що містить компоненти OpenMP або TBB може працювати під різними ОС, водночас як реалізація на основі вбудованих потоків, потребуватиме переробки під конкретну систему. Особливі незручності виникають якщо слід переносити код між системами Windows (використовуються Windows-потоки) і Unix (Posix-потоки) [10].

API TBB грунтується на парадигмі так званого „узагальненого програмування” [11], тому слід використовувати його шаблони паралелізму тоді, коли необхідно працювати з ітераційним простором, що визначив користувач, або переслідується мета отримання результатів складних операцій редукції [8].

Вкладенний паралелізм може бути реалізований за допомогою вбудованих потоків та OpenMP. Варто пам`ятати, що розповсюдженою проблемою є занадто інтенсивне використання ресурсів. Розробники TBB декларують, що підтримці рекурсивного та вкладеного паралелізму була приділена особлива увага. Зокрема зазначається алгоритм динамічного балансування навантаження та метод захоплення задач, що реалізовані у планувальнику задач TBB [8].

OpenMP розроблений для роботи у багатопроцесорних системах із розподіленою пам`яттю [9]. TBB – для реалізації багатопотоковості [12]. В обох випадках за мету ставилося досягнення продуктивності. Конструктивні елементи цих API були розроблені, безпосередньо, для паралельного розбиття маштабованих даних. За умови інтенсивних обчислювальних навантажень, актуальність їхнього використання збільшується. При роботі із вбудовани-

ми потоками розробнику доведеться самостійно реалізовувати деякі алгоритми, що користувачі OpenMP або TBB мають у готовому вигляді. У такому випадку слід бі льше уваги приділити тестуваню, аби уникнути появи помилок взаємного блокування, або змагання потоків за час процесора. Але у деяких випадках вбудовані по токи ОС є беззаперечно кращим варіантом. Наприклад, коли йдеться про створення паралельного виконання на основі подій або вводу-виводу.

Як OpenMP, так і TBB задовольняють принциповим вимогам, що були поставлені на початку розділу. Зважаючи на більшу привабливість поєднання TBB з підходом об`єктно-орієнтованого програмування, було прийнято рішення зупинити свій вибір на цьому API.

3. Прикладний програмний інтерфейс для TBB та його використання

TBB, як засіб забезпечення високорівневого паралелізму містить набір компонентів для досягнення швидкодії циклів з наперед заданою кількістю ітерацій, циклів з редукцією, передумовою. Також до складу прикладного програмного інтерфейсу входять контейнери для паралельного програмування, специфічні засоби розподілу пам`яті, примітиви синхронізації. Повертаючись до контейнерів, зазначимо, що вони аналогічні тим, що використовуються в Standard Template Library (STL) [13], але розробники TBB стверджують, що „накладні витрати” у них менші [8].

Для компіляції програм, які використовують TBB необхідно інсталювати пакет, що містить відповідні класи та функції. Для виконання програм під операційною системою користувача достатньо мати динамічну бібліотеку для даної ОС. Як правило вона входить до складу пакета.

До основних алгоритмів API TBB належать:
parallel_for,
parallel_reduce,
parallel_scan,
parallel_while,
parallel_do,
pipeline,
parallel_sort.
Контейнери:
concurrent_queue,
concurrent_vector,
concurrent_hash_map.
Засоби для маштабованого виділення пам`яті:
scalable_malloc,
scalable_free,
scalable_realloc,
scalable_calloc,
scalable_allocator,
cache_aligned_allocator.
Семафори для синхронізації потоків, що виконуються одночасно:
mutex,
spin_mutex,
queuing_mutex,
spin_rw_mutex,
queuing_rw_mutex,
recursive mutex.
Атомарні (нероздільні на дрібніші) операції:
fetch_and_add,
fetch_and_increment,
fetch_and_decrement,
compare_and_swap,
fetch_and_store.
Синхронізація потоків із використанням глобальних міток часу (time stamp) та планувальник задач, що має безпосередній доступ до їхнього створення, активації та контролю.

Спеціалісти компанії Intel наголошують [14] на можливості роботи бібліотеки сумісно з потоками ОС Microsoft Windows, Posix-потоками, та API OpenMP. На сьогодні доступні реалізації TBB для операційних систем Linux, Solaris, Mac OS, Windows.

4. Розробка паралельної реалізації

Діаграма статичної структури розроблюваної програми мовою UML [15] показана на рис. 2. Вона містить класи, зв`язки між ними, їхні атрибути, методи.

Перелічені типи *Type, Direction* слугують для характеристики орієнтації прямої на площині з прямокутною системою координат.

LineFunc – клас, що описує пряму (промінь, відрізок) на площині. Він містить

коефіцієнти рівнянь як загального вигляду ($ax+by+c=0$), так і часткових випадків (наприклад, $y=kx+b$). Має методи визначення відстані від точки площини до прямої, променя або відрізку (*get_dist*()), знаходження перпендикуляра (*get_perpend_lf*()), обчислення значення відповідної функції у точці (*value*()), знаходження точки перетину прямих, променів, відрізків (*get_intesect*()). Цей клас є основою для побудови реберного списку із подвійними зв`язками (*Rsds*), оскільки за допомогою його методів визначаються вершини багатокутників діаграми Вороного.

Клас *Main* реалізує інтерфейс для порівняння інших програм з реалізацією *VoronoiDiagram.* Далі наведено фрагмент коду метода *main*():

```
...
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
...
using namespace tbb;
using namespace std;
...
static const size_t chunk_st=1000;
```

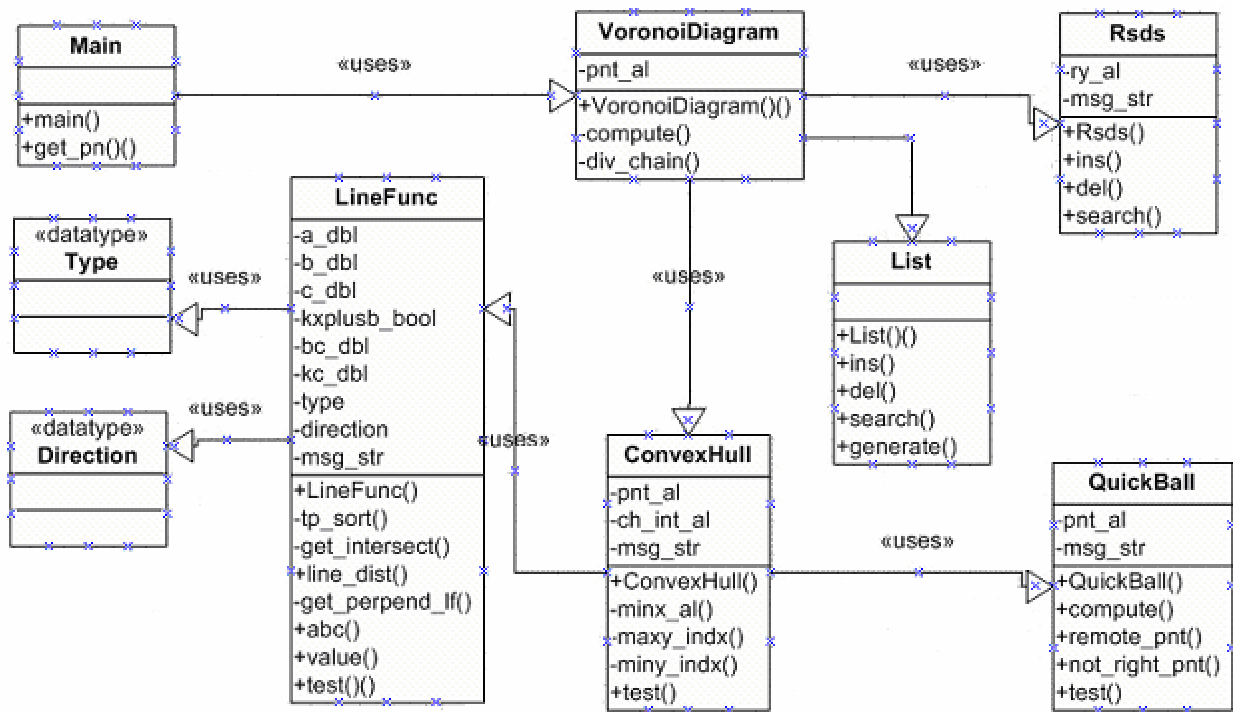


Рис. 2. Діаграма класів

Клас *List* реалізує основні операції над елементами списків та здійснює генерацію точок площини (*generate*()).

QuickBall – клас, що реалізує алгоритм побудови опуклої оболонки „*Quick-Ball”* [1] (*compute*()).

Клас *ConvexHull* відповідальний за організацію побудов опуклих оболонок та збереження їхніх результатів (*ConvexHull*()).

Клас *VoronoiDiagram* керує підготовкою вхідних даних, їхньою перевіркою, побудовою діаграми, визначенням розділяючого ланцюга, виправленням можливих помилок.

```
static const size_t min_x=-20000;
static const size_t min_y=-20000;
static const size_t max_x=20000;
static const size_t max_y=20000;
...
int main(int argc, char *argv[]) {
...
task_scheduler_init init;
tick_count begin_tc, end_tc, serial_tc, parallel_tc;
size_t pnt_nbr;
List s_list;
//ініціалізація множини S;
pnt_nbr = get_p_n();
s_list = new List();
```

```
s_list.generate(min_x, min_y, max_x,
max_y)
//виконання паралельної програми;
begin_tc = tick_count::now();
paral-
lel_for(blocked_range<Node*>(0,
s_list.size(),chunk_st 100), Voron-
oiDiagram(s_list));
end_tc = tick_count::now();
parallel_tc = end_tc-begin_tc;
//виконання послідовної програми;
begin_tc = tick_count::now();
SerialVoronoiDiagram(s_list)
end_tc = tick_count::now();
parallel_tc = end_tc-begin_tc;
...}
...
```

5. Експерименти

Для експериментальної перевірки ефективності розпаралелювання було виконано наступні кроки:

1) розробити послідовну програму;
2) провести її тестування;
3) частину коду, що має виконуватися, паралельно переписати із використанням компонентів TBB.

Під час виконання (п. 3), автор переконався в недоліку проектування. Оскільки раніше розроблений код не відповідав STL-стилю компонентів TBB, що призвело до виконання деякого обсягу надлишкової роботи. Експерименти з дослідження швидкодії роботи програм здійснювалися на машинах з різними процесорами: Intel Pentium III 400 МГц, та двоядерному Intel Core 2 Duo 2,53 ГГц.

Таким чином, завдяки використанню паралельної програми, на двоядерному процесорі вдалося досягти збільшення швидкості в 1,792 рази (у середньому для 10 експериментів (рис. 3)) щодо швидкості роботи послідовної програми. Крім того отримано несподіваний результат на одноядерному „повільному" процесорі. Виявилося, що паралельна версія програми працює в 1,18 (у середньому для 10 експериментів) рази повільніше ніж послідовна. Звісно, зменшення швидкості очікувалося, оскільки воно зумовлене використанням шаблонів TBB, але не передбачалося, що втрати швидкодії будуть настільки суттєвими.

Висновки

У роботі запропонована маштабована паралельна реалізація алгоритму побудови діаграми Вороного на площині. Сформульована низка вимог до програмних засобів забезпечення високорівневого паралелізму на багатоядерних однопроцесорних системах. Внаслідок аналізу функціональності трьох прикладів таких засобів

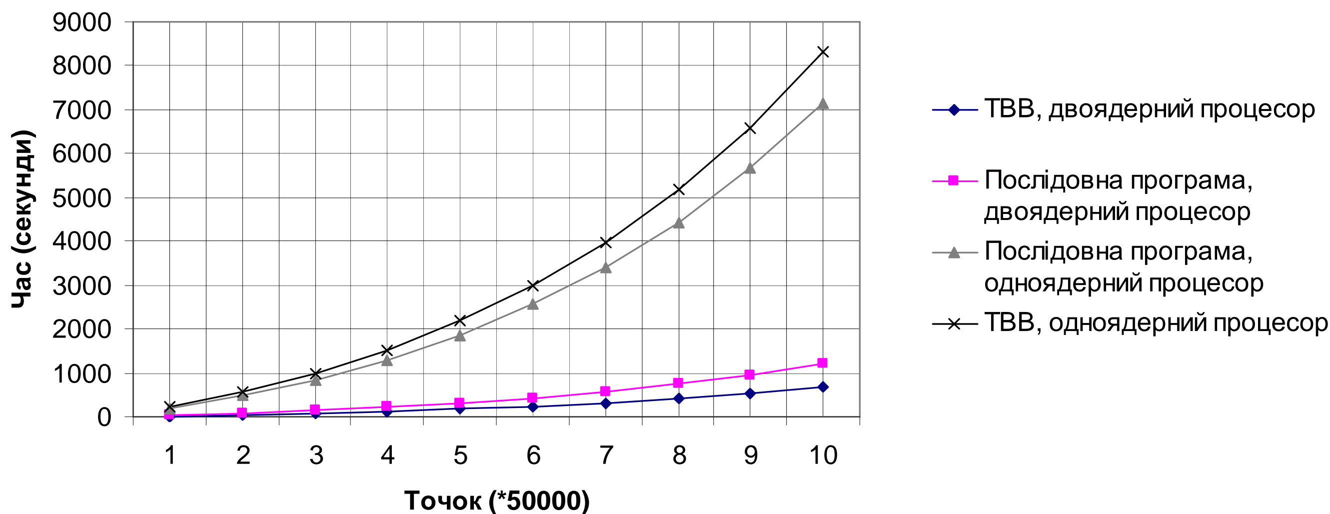


Рис. 3. Залежність часу виконання від кількості точок площини

було зроблено висновок, що Intel Threading Building Blocks (TBB) найбільш повно відповідає поставленим вимогам. Розглянуто компоненти цього прикладного програмного інтерфейсу, використавши які, було виконано паралельну реалізацію алгоритму побудови діаграми Вороного. Представлено результати експерименту для визначення ефективності роботи реалізацій послідовного та паралельного алгоритмів на одноядерному та двоядерному процесорах.

1. *Препарата Ф., Шеймос М.* Вычислительная геометрия: Введение: Пер. с англ. – М.: Мир, 1989. – 478 с.
2. *Вороний* Георгій Феодосійович. – http://uk.wikipedia.org/wiki/Вороний_Георгій_Феодосійович
3. *Drysdale S.* Voronoi Diagrams: Applications from Archeology to Zoology. – http://www.ics.uci.edu/~eppstein/gina/scot.drysdale.html
4. *Верещагин Н., Шень А.* Лекции по математической логике и теории алгоритмов. – ftp://ftp.mccme.ru/users/shen/logic/sets/part1pdf.zip
5. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов: Пер. с англ. – М.: Мир, 1979. – 536 с.
6. *Aggarwal A., Chazelle B., Guibas L., O`Dunlaing C., Yap C.* Parallel Geometry. – Algorithmica, 1998. – 328 p.
7. *Parallel* Random Access Machine. – http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine
8. *Reinders J.* Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. – O'Reilly Media, 2007. – 303 p.
9. *OpenMP.* – https://computing.llnl.gov/tutorials/openMP/
10. *Posix.* – http://en.wikipedia.org/wiki/Posix
11. *Standard* Template Library. http://en.wikipedia.org/wiki/Standard_Template_Library
12. *Intel* Threading Building Blocks. – http://en.wikipedia.org/wiki/TBB
13. *Дейтел П.Дж., Дейтел Х.М.* Как программировать на C++: Пер. с англ. – 5-е изд., – М.: Бином пресс, 2008. –1454 с.
14. *Threading* Building Blocks 2.1. – http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm
15. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя: Пер. с англ. – 2-е изд., стер. – М.: Питер ДМК, 2004. – 429 с.

Отримано 04.12.2008

Про автора:

Березовський Костянтин Анатолійович,
студент 2-го курсу, факультету кібернетики Київського національного університету імені Тараса Шевченка,
Проспект Академіка Глушкова 2, корп. 6 ,
Тел.: 521 3554
8 – (097) 173 6420
e-mail: kos:berezovskyi@gmail.com