COMPUTER SYSTEMS AND NETWORKS

Vol. 5, No. 1, 2023

DOI: https://doi.org/10.23939/csn2023.01.096

УДК 004.825

МОДИФІКАЦІЯ АЛГОРИТМУ ПУЛУ ПОТОКІВ З БАГАТЬМА ЧЕРГАМИ

Д. В. Пуйда

Національний університет "Львівська політехніка",
кафедра електронних обчислювальних машин
E-mail: dmytro.v.puida@lpnu.ua

Наведено модифікацію алгоритму пулу потоків, який запропонував Шон Парент на конференції NDC London 2017. Запропонований алгоритм не поступається оригінальному алгоритму за швидкодією та простотою реалізації, і водночас усуває потенційний недолік оригінального алгоритму, який полягає у тому, що за певних обставин кілька задач можуть виконуватися на тому самому потоці, тоді як інші потоки – перебувати в стані очікування задачі.

Основна ідея, яка запропонована в роботі, полягає у відстеженні сумарної кількості задач, які містяться в чергах пулу потоків. Коли головний потік поміщає нову задачу в якусь із черг, лічильник кількості задач збільшується на одиницю. Коли задача вилучається з черги, лічильник кількості задач зменшується на одиницю. У випадку, коли потік пулу хоче отримати задачу, він переглядає черги доти, доки йому не вдасться отримати задачу з якоїсь із черг або доки лічильник кількості задач не дорівнюватиме нулю. Якщо лічильник кількості задач дорівнює нулеві, потік переходить у стан сну до моменту, коли лічильник кількості потоків не стане знову ненульовим. Тоді один із потоків пулу прокидається і починає перевіряти черги на наявність задач. Дуже важливо зберегти рівномірний розподіл задач по чергах, оскільки це істотно впливає на швидкодію програми.

Ключові слова: багатопотоковість; пул потоків; перехоплення задач.

Вступ

Багатопотоковість має надзвичайно важливе значення у розробленні сучасного програмного забезпечення. Ефективне використання апаратних ресурсів за рахунок створення додаткових потоків дає змогу розпаралелити виконання задач, прискоривши роботу програмного продукту загалом [1]. Однак, як відомо, навіть для задач, які можна розпаралелити, створення більшої кількості потоків не завжди означає підвищення швидкодії програми. За надмірно великої кількості потоків перемикання контексту може істотно впливати на швидкодію програми і, в деяких випадках, триватиме навіть довше, ніж саме виконання корисної роботи потоками. На рис. 1 подано приклад залежності загального часу роботи програми, що складається з великої кількості незалежних простих обчислювальних задач, які можна розпаралелити, від кількості потоків, які виконують ці задачі. Як видно з рис. 1, починаючи з певного моменту збільшення кількості потоків не приводить до зменшення загального часу роботи програми. Експеримент виконано на персональному комп'ютері з процесором 13th Gen Intel(R) Core(TM) i7-1365U, 1800 MHz, 10 ядер, 12 логічних процесорів.

Збільшення кількості потоків інколи може навіть призводити до зниження швидкодії програмних продуктів.

Отже, очевидно, що загальна кількість потоків у програмному продукті повинна бути обмеженою і кількість потоків не повинна неконтрольовано збільшуватися за жодних обставин. Для обмеження кількості потоків у програмному продукті призначений інструмент, названий "пул

потоків" (thread pool). Поняття пулу потоків немає у стандарті C++ 23, який є найновішою версією стандарту C++ на момент написання цієї статті [2]. Тому розробники програмного забезпечення змушені використовувати сторонні бібліотеки або писати власні реалізації пулу потоків.

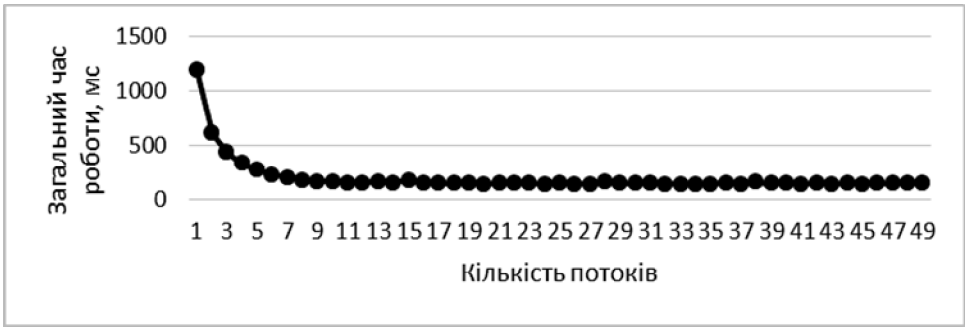

Рис. 1. Приклад залежності часу роботи програми від кількості потоків

Пул потоків зазвичай складається із двох основних компонент: потоки (workers) та черги задач (task queues). В найпростішій реалізації, під час створення, пул потоків створює певну наперед задану кількість додаткових потоків, кожен з яких спочатку переходить у стан очікування задачі. Коли головний потік хоче виконати задачу на додатковому потоці, задача поміщається в чергу задач пулу потоків. Коли в черзі з'являється задача, один з потоків пулу в певний момент часу вилучає задачу з черги і виконує її. Після виконання задачі, якщо інших задач у черзі немає, потік знову переходить у стан очікування задачі. Отже, додаткові потоки можуть створюватися лише в пулі потоків, який власне і контролює кількість додаткових потоків у системі. Жодні інші компоненти системи у цьому випадку не можуть створювати додаткових потоків. Натомість вони поміщають свої задачі в чергу пулу потоків і ці задачі виконуються на потоках, які раніше уже створив пул.

Огляд літературних джерел

Ефективна реалізація пулу потоків дуже важлива для ефективного використання потоків програмним продуктом. Уже створено багато алгоритмів та реалізацій пулу потоків (див., наприклад, [3–8]).

Пул потоків з однією чергою задач вважають неефективним у високонавантажених системах. Оскільки різні потоки можуть незалежно поміщати задачі в чергу та вилучати їх, черга задач повинна бути безпечною у багатопотоковому середовищі. Це може призводити до небажаної ситуації, коли всі потоки пулу конкурують за ексклюзивний доступ до того самого ресурсу, тобто до черги задач. Для того, щоб уникнути такої ситуації, часто будують пули потоків з кількома чергами задач. Доволі типова модель, в якій кожен із потоків пулу має свою чергу. В такому випадку, коли потоку потрібно отримати задачу для виконання, він спершу перевіряє власну чергу задач. Якщо потокові не вдається одержати задачу зі своєї черги задач, він може отримати її для виконання з черги задач, яка належить іншому потокові. Такий підхід часто називають перехопленням задач (work stealing).

Існує багато алгоритмів перехоплення задач. Один із найпростіших і водночас доволі ефективних підходів запропоновано в [9]. Припустимо, що ми маємо n потоків та n черг задач. Кожна із цих черг може бути захищена власним мютексом. Вважатимемо, що потік з номером i не може отримати задачу з черги з номером j, якщо ця черга у цей момент часу захищена мютексом або порожня. Існує ключова відмінність від найпростішої схеми з однією чергою та одним мютексом. У найпростішому випадку з однією чергою та одним мютексом, якщо в момент доступу, черга задач виявляється захищеною мютексом, потік не має іншого виходу, окрім як чекати і спробувати здобути доступ до тієї ж черги через деякий час. Якщо ж черг кілька, і в момент доступу виявляється, що черга захищена мютексом, потік може спробувати доступитися до іншої черги, яка в цей момент

може виявитися не захищеною і не порожньою. Якщо ж всі інші черги виявляються захищеними мютексом або порожніми деякий час, потік зупиняється і чекає на задачу із власної черги.

Реалізація підходу, запропонованого в [9], надзвичайно проста і водночас забезпечує добрі результати бенчмаркінгу. Однак такий підхід має і певні недоліки. Один з недоліків полягає в тому, що за певних обставин кілька задач можуть виконуватися на тому самому потоці, тоді як інші потоки – очікувати задачі. Розглянемо найпростіший приклад [9], показаний на рис. 2.

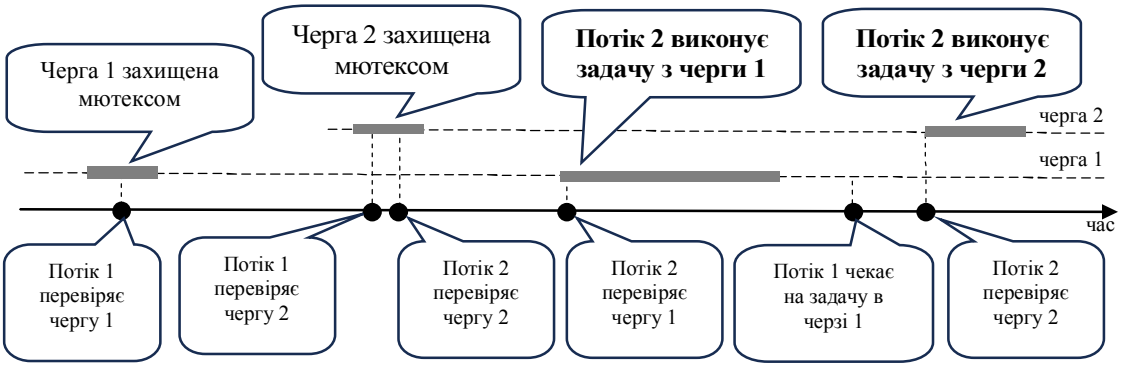

Рис. 2. Приклад проблеми в підході

У наведеному прикладі два потоки і дві черги. Припустимо, що в деякий момент часу потік 1 намагається отримати задачу зі своєї черги 1, яка в цей момент виявляється захищеною мютексом, оскільки головний потік поміщає в неї задачу. Після цього потік 1 намагається одержати задачу з черги 2, яка, припустимо, також виявляється захищеною мютексом, оскільки головний потік, попередньо помістивши задачу в чергу 1, зараз поміщає задачу в чергу 2. Припустимо, що відразу після цього потік 2 також намагається отримати задачу з черги 2, яка в цей момент все ще залишається захищеною мютексом. Після цього потік 2 може перевірити чергу 1, в якій вже є задача. Отже, потік 2 вилучає задачу з черги 1 і розпочинає її виконання. Після цього потік 1, вже переглянувши інші черги, зосереджується на черзі 1 і очікує в ній задачу, не помічаючи, що в черзі потоку 2 вже є задача, котра чекає на виконання, тоді як потік 2 виконує задачу з черги потоку 1. Отже, у цьому прикладі потік 2 виконує дві задачі, а потік 1 не виконує жодної.

Постановка задачі

Мета цієї статті – запропонувати простий варіант вирішення описаної вище проблеми в підході [9]. Важливо, щоб реалізація алгоритму залишалась такою ж простою, як і в оригінальному варіанті алгоритму [9], і щоб новий алгоритм не втратив ефективності порівняно з оригінальним варіантом.

Результати досліджень

На перший погляд видається, що очевидне рішення полягає в тому, щоб, якщо не вдається отримати задачу з іншої черги, потік пулу не переходив у стан очікування задачі у своїй черзі, а продовжував неперервно перевіряти всі черги на предмет появи задач. Однак таке рішення було би доволі невдалим як мінімум з погляду навантаження на систему. Такий підхід був би аналогічним до класичного “busy wait”, що загалом не рекомендований для розроблення програмного забезпечення. Переведення потоків, які не виконують активної роботи, в стан сну дозволяє операційній системі ефективніше використовувати апаратні ресурси, тому переведення “пасивних” потоків у стан сну є важливим нюансом у розробленні багатопотокового програмного забезпечення.

Ідея рішення, запропонованого у цій статті, полягає у відстеженні сумарної кількості задач, які перебувають у чергах пулу потоків. Коли головний потік поміщає нову задачу в якусь із черг, лічильник кількості задач збільшується на одиницю, а коли вилучає задачу з черги, – зменшується на одиницю. Коли потік пулу хоче отримати задачу, він переглядає черги доти, доки йому не

вдасться отримати задачу з якоїсь із черг або доки лічильник кількості задач не дорівнюватиме нулю. Коли лічильник кількості задач дорівнює нулю, потік переходить у стан сну до моменту, коли лічильник кількості потоків не стане знову ненульовим. Коли лічильник кількості потоків стає ненульовим, один із потоків пулу прокидається і починає перевіряти черги на наявність задач. Дуже важливо у такому разі зберегти рівномірний розподіл задач по чергах, оскільки це суттєво впливає на швидкодію програми.

Для реалізації такого пулу потоків використаємо стандарт C++ 20, оскільки це перший стандарт C++, який підтримує std::atomic<T>::wait. У реалізації алгоритму, яка пропонується, застосуємо std::atomic<T>::wait для очікування моменту появи задач у чергах. Якщо необхідно підтримувати старіші стандарти C++, std::atomic<T>::wait, очевидно, легко замінити за допомогою std::condition_variable. Швидкодію рішення з використанням std::condition_variable не досліджено.

Як і у випадку оригінальної реалізації [9], нам потрібна безпечна в багатопотоковому середовищі черга для зберігання задач. Зауважимо, що, на відміну від черги [9], в цьому випадку нам не потрібен std::condition_variable. Опишемо необхідний інтерфейс черги задач:

```
using Task = std::function<void()>;

class TaskQueue {
 public:
  [[nodiscard]] bool TryPush(Task&& task);
  [[nodiscard]] bool TryPop(Task& task);
  void Push(Task&& task);

 private:
  std::queue<Task> queue_;
  std::mutex mutex_;
};
```

Метод TryPush намагається помістити задачу в чергу. Якщо операція успішна, TryPush повертає true. Якщо ж черга в цей момент захищена мютексом, TryPush повертає false. Нижче наведено приклад реалізації методу TryPush:

```
bool TryPush(Task&& task) {
  const std::unique_lock lock(mutex_, std::try_to_lock);
  if (lock) {
    queue_.push(std::forward<Task>(task));
    return true;
  }
  return false;
}
```

Аналогічно, метод TryPop намагається вилучити задачу із черги. Якщо операція успішна, TryPop повертає true. Якщо ж черга в цей момент захищена мютексом або порожня, TryPop повертає false. Нижче наведено приклад реалізації методу TryPop:

```
bool TryPop(Task& task) {
  const std::unique_lock lock(mutex_, std::try_to_lock);
  if (lock && !queue_.empty()) {
    task = std::move(queue_.front());
    queue_.pop();
    return true;
  }
  return false;
}
```

Нарешті, метод Push безумовно поміщає задачу в чергу:

```
void Push(Task&& task) {
  std::lock_guard lock(mutex_);
  queue_.push(std::forward<Task>(task));
}
```

Розглянемо найпростіший інтерфейс пулу потоків. Метод Push поміщає задачу в одну із черг пулу. Приватний лічильник count_ позначає загальну кількість задач, які містяться у чергах. Приватне поле index_ призначене для рівномірного розподілу задач по чергах і використовується аналогічно до [9]. Зауважимо, що рівномірний розподіл задач по чергах є ключовим нюансом у реалізації методу Push. Під час спроби отримати задачу черги можуть переглядатися, починаючи з нульової, оскільки послідовність перегляду черг під час одержання задачі істотно не впливає на швидкодію рішення.

```
class ThreadPool {
 public:
  explicit ThreadPool(unsigned threads_count);
  void Push(Task&& task);

 private:
  const unsigned threads_count_;
  std::vector<std::thread> threads_;
  std::vector<TaskQueue> queues_;
  std::atomic<int> count_;
  std::atomic<unsigned> index_;
};
```

Конструктор класу ThreadPool приймає аргумент threads_count, що визначає кількість потоків, які потрібно створити. Нижче наведено приклад реалізації конструктора класу ThreadPool:

```
ThreadPool(const unsigned threads_count)
    : threads_count_{threads_count},
      queues_{threads_count},
      count_{0},
      index_{0} {
  for (unsigned i = 0; i != threads_count; ++i) {
    threads_.emplace_back([this] { Run(); });
  }
}
```

Нижче наведено приклад реалізації приватного методу Run. Зауважимо, що лічильник count_ також використовується як індикатор завершення роботи пулу потоків. Коли пул потоків завершує роботу, в змінну count_ поміщається від'ємне число. Це дає змогу уникнути появи додаткового об'єкта класу std::atomic<T> для організації завершення роботи потоків пулу.

```
void Run() {
  Task task;
  while (count_.load() >= 0) {
    while (GetTask(task)) {
      task();
    }
    count_.wait(0, std::memory_order::relaxed);
  }
}
```

Реалізація приватного методу GetTask:

```
bool GetTask(Task& task) {
  do {
    for (auto& queue : queues_) {
      if (queue.TryPop(task)) {
        count_.fetch_sub(1);
        return true;
      }
    }
  } while (count_.load() > 0);
  return false;
}
```

Нарешті, реалізація методу Push аналогічна оригінальній реалізації [9], однак містить також інкрементування лічильника count_ у разі додавання задачі. Зауважимо, що здійснювати кілька проходів, щоб спробувати помістити задачу в чергу, не є необхідним, а оптимальне значення кількості проходів може відрізнятися у різних ситуаціях і потребує окремого дослідження. Тому, щоб спростити реалізацію, виконаємо лише один прохід для розміщення задачі в одну із черг. На практиці, здебільшого, задача буде розміщена під час цього одного проходу за допомогою методу TryPush.

```
void Push(Task&& task) {
  count_.fetch_add(1);
  const auto i = index_.fetch_add(1, std::memory_order::relaxed);
  for (unsigned j = 0; j != threads_count_; ++j) {
    if (queues_[(i + j) % threads_count_].TryPush(std::forward<Task>(task))) {
      count_.notify_one();
      return;
    }
  }
  queues_[i % threads_count_].Push(std::forward<Task>(task));
  count_.notify_one();
}
```

Аналіз швидкодії

Для аналізу швидкодії пропонованого алгоритму використано дослідження, описані в роботі [10]. Бенчмаркінг, запропонований у роботі [10], показує, що швидкодія алгоритму, запропонованого в цій статті, співмірна зі швидкодією оригінального алгоритму [9]. Нижче наведено приклад результатів бенчмаркінгу в роботі [10]. У цих результатах оригінальний алгоритм [9] позначено назвою Sean Parent Work Stealing. Модифікований алгоритм, запропонований в цій статті, позначено назвою Sean Parent Work Modified. Значення, одержані за допомогою Sean Parent Work Modified, близькі до значень, отриманих за допомогою оригінального Sean Parent Work Stealing. У деяких випадках модифікований алгоритм навіть може демонструвати дещо кращий результат, ніж оригінальний алгоритм, однак різниця між значеннями незначна, у межах похибки вимірювання. Дослідження виконано на персональному комп'ютері із процесором 13th Gen Intel(R) Core(TM) i7-1365U, 1800 MHz, 10 ядер, 12 логічних процесорів.

```
< gladiatior >              < forking >   < joining >   < total >
Sean Parent Naive            0.871500ms  1040.808400ms  1041.679900ms
Sean Parent Multiqueue       1.442100ms  1017.190800ms  1018.632900ms
Sean Parent Work Stealing    0.816300ms   925.348600ms   926.164900ms
Sean Parent Work Modified    1.108700ms   922.597300ms   923.706000ms

< gladiatior >              < forking >   < joining >   < total >
Sean Parent Naive           89.231000ms  1533.984500ms  1623.215500ms
Sean Parent Multiqueue      49.679200ms  1653.557200ms  1703.236400ms
Sean Parent Work Stealing   88.714700ms  1564.351200ms  1653.065900ms
Sean Parent Work Modified   72.752800ms  1548.310400ms  1621.063200ms
```

Висновки

Модифікація алгоритму, запропонована у статті, не поступається оригінальному алгоритму за швидкодією та простотою реалізації, і водночас усуває потенційний недолік оригінального алгоритму, який полягає у тому, що за певних обставин кілька задач можуть виконуватися на тому самому потоці, тоді як інші потоки – перебувати в стані очікування задачі. Ідею, на якій ґрунтується викладена модифікація алгоритму, а також запропоновану реалізацію засобами C++ 20, можна використати для створення пулу потоків для проєктів різної складності.

Список літератури

1. Amdahl G. The validity of the single processor approach to achieving large-scale computing capabilities, Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N. J., AFIPS Press, 483–485.

2. ISO International Standard ISO/IEC 14882:2020(E). Programming Language C++.

3. David Chase and Yossi Lev, Dynamic circular work-stealing deque, SPAA, 21–28. ACM, 2005. DOI: 10.1145/1073970.1073974.

4. K. Agrawal et al. Executing task graphs using work-stealing, IEEE IPDPS, April 2010, 1–12. DOI: 10.1109/IPDPS.2010.5470403.

5. Nhat Minh Lé et al., Correct and efficient work-stealing for weak memory models, PPoPP, 69–80. ACM, 2013. DOI: 10.1145/2442516.2442524.

6. T. Huang et al., Cpp-Taskflow: Fast task-based parallel programming using modern C++. In IEEE IPDPS, May 2019, 974–983. DOI: 10.1109/IPDPS.2019.00105.

7. C.-X. Lin, T.-W. Huang and M. D. F. Wong, An Efficient Work-Stealing Scheduler for Task Dependency Graph, 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), Hong Kong, 2020, 64–71. DOI: 10.1109/ICPADS51040.2020.00018.

8. Intel oneAPI Threading Building Blocks. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html

9. Sean Parent, Better Code: Concurrency, NDC { London } 2017. URL: https://sean-parent.stlab.cc/presentations/2016-08-08-concurrency/2016-08-08-concurrency.pdf

10. Thread Pool Benchmark. URL: https://github.com/Red-Portal/thread-pool-benchmark/

MODIFICATION OF A THREAD POOL ALGORITHM WITH MULTIPLE TASK QUEUES

D. Puida

Lviv Polytechnic National University
Computer Engineering Department

In this paper, the author suggests a modification of the thread pool algorithm that was presented by Sean Parent at NDC London 2017. The suggested algorithm is as simple as the original implementation and demonstrates similar performance, while eliminating a potential drawback of the original implementation consisting in the fact that under certain circumstances, multiple tasks can be executed on the same thread, while other threads may be waiting for a task.

The suggested idea consists in tracking the total number of tasks that are in the queues of the thread pool. When the main thread pushes a new task to one of the queues, the tasks counter is incremented. When a task is removed from the queue, the task counter is decremented. When a thread wants to get a task, it keeps checking the queues until it succeeds in getting a task from one of the queues, or until the tasks counter becomes equal to zero. When the tasks counter becomes equal to zero, the thread becomes idle until the counter becomes non-zero again. Then, one of the threads wakes up and starts checking the queues. An important point is to maintain even distribution of tasks in the queues since it has a significant impact on the performance of the algorithm.

Key words: multithreading; thread pool; task stealing.