Task Parallel Library

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt

prezentacja Michał Albrycht

Agenda

- O potrzebie zrównoleglania
- Przykłady użycia TPL
- Tasks and Replicable Tasks
- Rozdzielanie zadań
  - Workgroup
  - WorkThread
- Duplicating Queue
- Wydajność

Problem ze zrównoleglaniem

- Przyrost mocy obliczeniowej procesorów, nie poprzez przyspieszanie zegarów, a poprzez dodawanie dodatkowych rdzeni.
- Aby w pełni wykorzystac moc procesora, należy jednocześnie obciążyć wszystkie dostępne rdzenie.
- Własnoręczne kodowanie obsługi puli wątków, w celu zrównoleglenia jednej pętli for jest demotywujące.

Mnożenie macierzy - sekwencyjnie

```
void MatrixMult(int size, double[,] m1,
                double[,] m2, double[,] result){
  for(int i = 0; i < size; i++){
    for(int j = 0; j < size; j++){
      result[i, j] = 0;
      for(int k = 0; k < size; k++){
        result[i, j] += m1[i, k] * m2[k, j];
      }
    }
  }
}
```

Mnożenie macierzy - TPL

```
void ParMatrixMult(int size, double[,] m1,
                   double[,] m2, double[,] result){
  Parallel.For(0, size, delegate(int i){
    for(int j = 0; j < size; j++){
      result[i, j] = 0;
      for(int k = 0; k < size; k++){
        result[i, j] += m1[i, k] * m2[k, j];
      }
    }); // Źle postawiony nawias
  });
}
```

Mnożenie macierzy – TPL cz.2

- Pobiera trzy argumenty: od (włącznie), do (wyłącznie) oraz delegację.
- Delegacja to odpowiednik funkcji nienazwanych w C# (lub jeśli ktoś woli – lambda wyrażeń).
- Delegacja pobiera indeks, a jako treść zawiera niezmieniony algorytm mnożenia macierzy.
- Automatycznie zostały przechwycone zmienne m1, m2 i result.

TPL – krótki opis

- Podstawowe jednostki: Task i ReplicableTask
- Brak gwarancji zrównoleglenia
- Kolejka zadań przydzielona do grupy wątków
- Work-stealing strategy

QuickSort - sekwencyjnie

```
static void SeqQuickSort<T>(T[] dom, int lo, int hi)
   where T : IComparable<T>
{
   if(hi - lo <= Threshold){
      InsertionSort(dom, lo, hi);
      return;
   }
   int pivot = Partition(dom, lo, hi);
   SeqQuickSort(dom, lo, pivot - 1);
   SeqQuickSort(dom, pivot + 1, hi);
}
```

QuickSort – TPL

```
static void ParQuickSort<T>(T[] dom, int lo, int hi)
      where T : IComparable<T>
{
  if(hi - lo <= Threshold){
    InsertionSort(dom, lo, hi);
    return;
  }
  int pivot = Partition(dom, lo, hi);
  Parallel.Do(
    delegate{ ParQuickSort(dom, lo, pivot - 1); },
    delegate{ ParQuickSort(dom, pivot + 1, hi); }
 );
}
```

Agregacja

Chcemy policzyć sumę liczb pierwszych mniejszych od 10000.

```
int sum = 0;
for (int i = 0; i < 10000; i++){
  if(isPrime(i))
      sum += i;
}
```

```
int sum = 0;
Parallel.For(0, 10000, delegate(int i){
  if(isPrime(i)){
    lock (this){ sum += i; }
  }
});
```

Agregacja - TPL

```
int sum = Parallel.Aggregate(
   0, 10000,   // domain
   0,          // initial value
   delegate(int i){ return (isPrime(i) ? i : 0) },
   delegate(int x, int y){ return x+y; }
);
```

- Pierwsza delegacja jest stosowana do każdego elementu.
- Druga stosowana jest do lokalnej zmiennej agregującej i wyników pierwszej delegacji.

Fibonacci

```
static int ParFib(int n){
  if(n <= 8) return Fib(n);
  var f2 = new Future<int>(() => ParFib(n-2));
  int f1 = ParFib(n-1);
  return (f1 + f2.Value);
}
```

- Wywołanie f2.Value czeka na zakończenie obliczeń i przekazuje wartość.
- Wszystkie „prawe gałęzie” liczone równolegle.

Task

- Reprezentuje skończone zadanie obliczeniowe.
- Konstruktor pobiera akcje do wykonania.
- Wyjątki przekazuje przy wywołaniu Wait().
- Wykonywany na puli wątków.

```
delegate void Action();
class Task{
  Task(Action action);
  void Wait();
  bool IsCompleted{ get; }
  ...
}
```

Paralell.Do

```
static void Do(Action action1, Action action2){
   Task t = new Task(action1); // do potentially in parallel
   action2();                  // call action2 directly
   t.Wait();                   // wait for action1
}
```

Przypadek szczególny, dla dwóch operacji

Replicable Tasks

- Konstruktor pobiera akcję do wykonania.
- Różni się od Task tym, że będzie wykonywane wielokrotnie.
- Nie jest dostępny dla programisty.

```
static void For(int from, int to, Action<int> body){
  int index = from;
  var rtask = new ReplicableTask(delegate{
    int i;
    while ((i = InterLocked.Increment(ref index)) <= to){
      body(i-1);
    }
  });
  rtask.Wait();
}
```

Future i ReplicableFuture

Future i ReplicableFuture to odpowiedniki Task i ReplicableTask przekazujące wyniki.

```
static T Aggregate<T>(int from, int to, T init,
                      Func<T> body,
                      Func<T,T,T> combine){
  int index = from;
  var rfuture = new ReplicableFuture<T>(
    delegate{
      int i;
      T   acc = init;
      while ((i = Interlocked.Increment(ref index)) <= to){
        acc = combine(acc, body(i-1));
      }
      return acc;
    },
    combine
 );
  return rfuture.Value;
}
```

Podział pracy

- Worker group dla każdego procesora.
- Worker group zawiera od jednego do kilku wątków (Worker thread).
- W danej chwili pracuje tylko jeden wątek z danej Worker group.
- Jeśli dany Worker thread zostaje zablokowany, dodatkowy Worker jest dodawany do grupy.

Podział pracy

- Na każdą Worker group przypada jedna, dwustronna kolejka zadań.
- Dostępne operacje to Push, Pop, Take.
- Push i Pop to standardowe operacje dodawanie i zdjęcia zadania z kolejki.
- Jeśli Worker Thread widzi, że nie ma już zadań w kolejce, to staje się złodziejem.
- Operacja Take kradnie zadanie z końca innej kolejki zadań.

Podział pracy

- Task może być w jednym z 3 stanów: Init, Running, Done
- Kiedy wątek chce wykonać pobrane zadanie, to zachowuje się różnie w zależności od jego stanu:
  - Init - zaczyna wykonywać zadanie
  - Running – ktoś mu ukradł zadanie, czeka na jego zakończenie
  - Done – ktoś mu ukradł, ale już skończył liczyć
- Stan Init jest najczęstszym przypadkiem, dlatego jego opłaca się optymalizować.

Podział pracy

- Implementacja kolejki zadań ma bezpośredni wpływ na skuteczność kradzieży, a więc również na wydajność.
- Dotychczas podobne rozwiązania były oparte o THE protocol wymyślony przez Dijkstrę w 1965r.
- Jednak to wymaga silnego modelu pamięci.
- Atomowe CompareAndSwap() do operowaniu na stanie zadania.
- Wymaga słabego modelu pamięci.

Total Sort Order

- Aby zachodził TSO, należy spełnić 4 aksjomaty
  - $\forall SS'.\ S <_m S' \vee S' <_m S$
  - $S <_p S' \Rightarrow S <_m S'$
  - $L <_p op \Rightarrow L <_m op$
    - L zawsze daje ostanią wartość trzymaną w pamięci
- Architektura x86 nie spełnia TSO!
- Spełnia, po przyjęciu dodatkowych założeń.

Duplicating Queue

- Implementuje dwustronną kolejkę zadań.
- Operacje: Push, Pop, Take.
- Take zwiększa head o 1 (kradnie z początku).
- Pop zmniejsza tail o 1.
- Push zwieksza tail o 1.
- TailMin zapobiega gubieniu elementów.

```
class DupQueue{
   Task[]  tasks;
   int     size;
   int     tailMin   = ∞;
   volatile int tail = 0;
   volatile int head = 0;
   ...
}
```

Duplicating Queue – wykorzystanie TSO

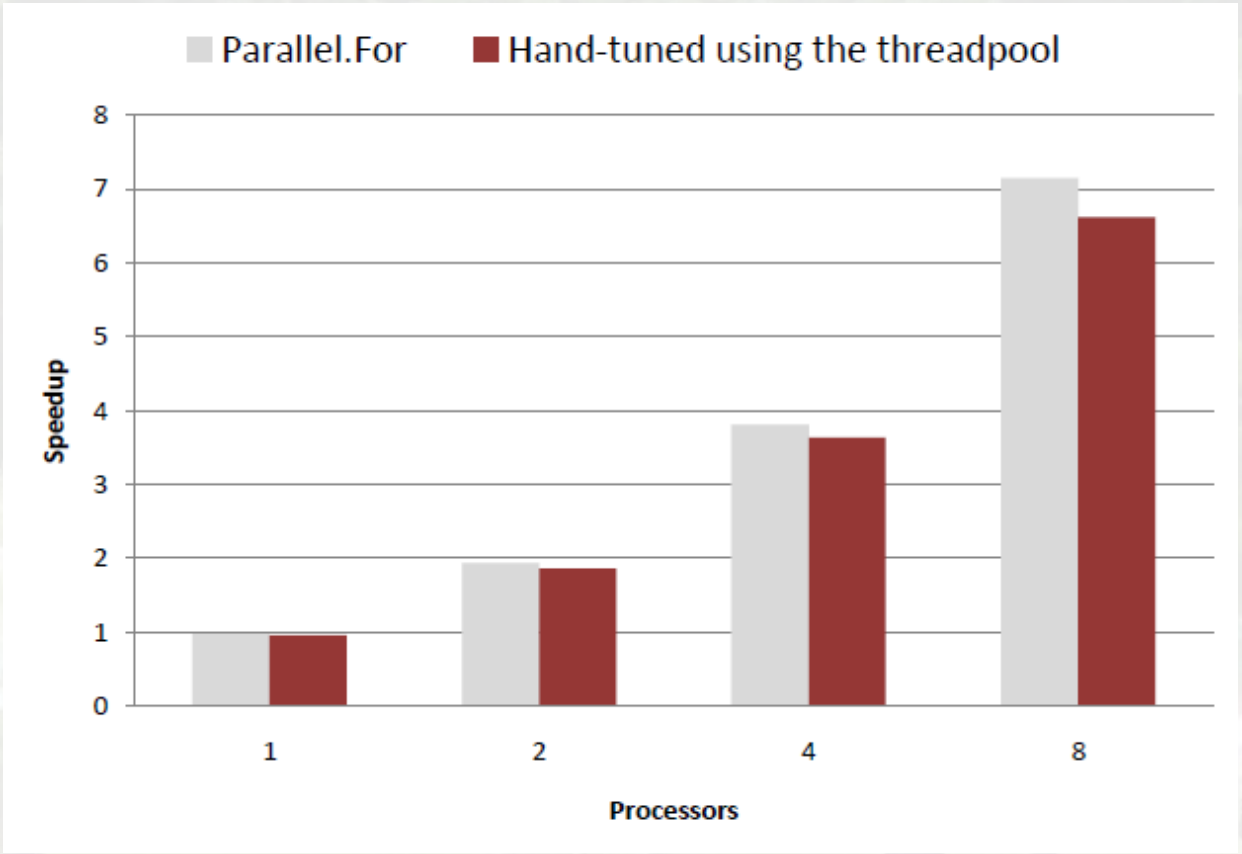
- Operacja Take() bierze locka do kolejki.
- Operacje Push() i Pop() tylko czasami biorą locka.
- Ponieważ pracuje tylko jeden Worker Thred to jednoczesny dostęp może być tylko pomiędzy jednym złodziejem (tym który założył locka) i aktualnie działającym wątkiem.
- $$L^{w}_{\texttt{tail}} <_m L^{t}_{\boxed{\texttt{tail}}} <_m S^{w}_{\texttt{tail}} \text{ and } S^{w}_{\texttt{tail}} <_p L^{w}_{\texttt{tail}}$$
- Worker Thread może widzieć nieaktualny *head.*
- Złodziej może widzieć nieaktualne *tail.*

Duplicating Queue

- *head == tail => kolejka jest pusta.*
- Gdyby nie *tailMin* można by zgubić zadanie.
- Działanie Duplicating Queue zostało formalnie zweryfikowane przy użyciu narzędzia *CheckFence.*

Wydajność

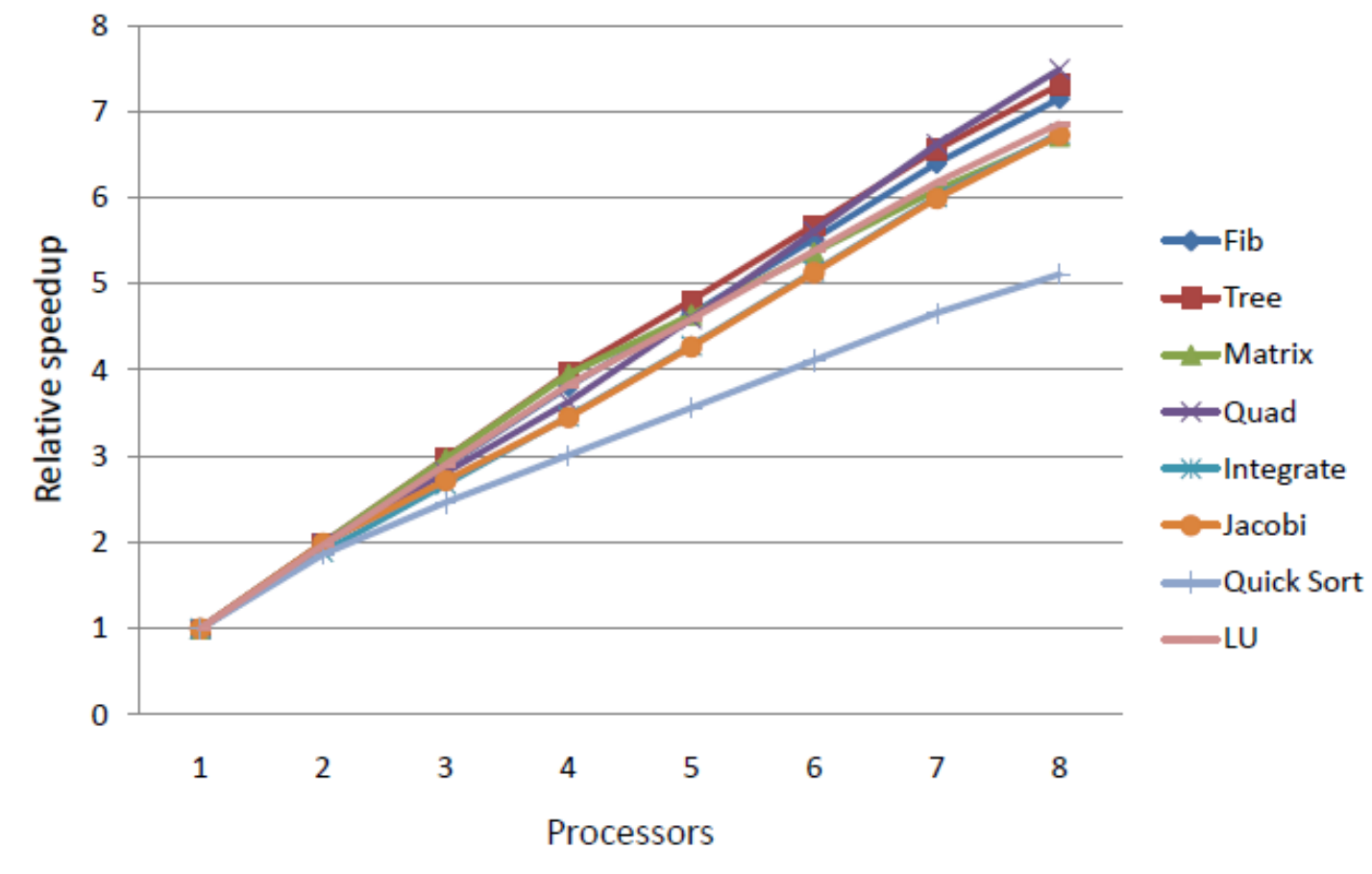
- Mnożenie macierzy: Parallel.For vs ręcznie dostrojony ThreadPool



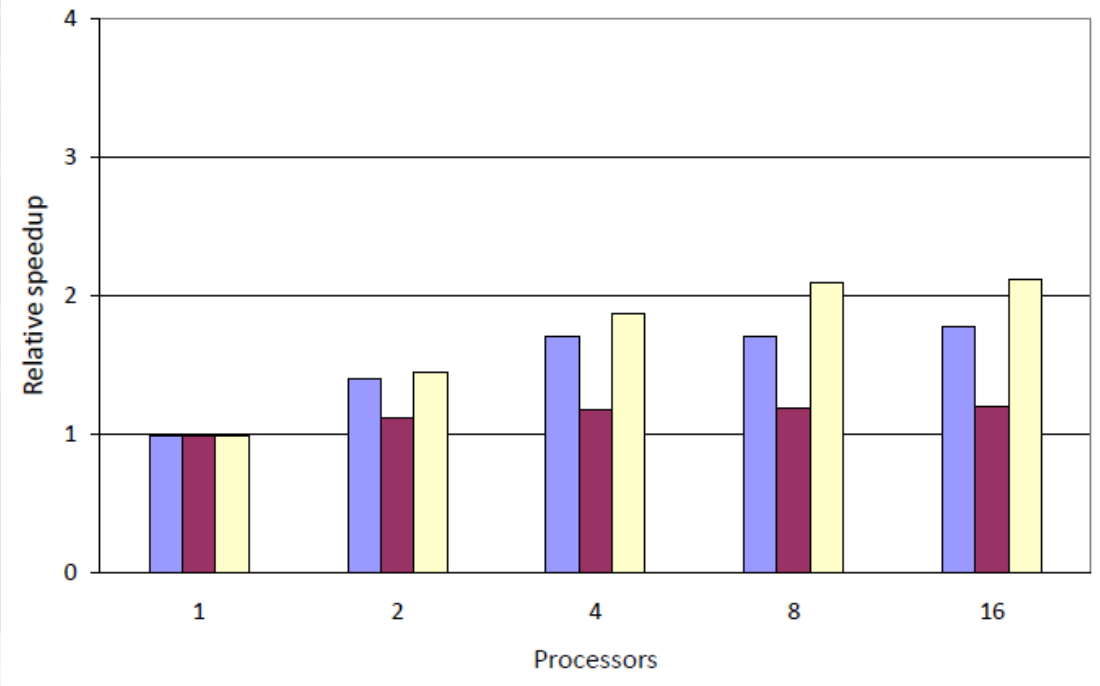
- 99% wydajności na maszynie z 1 procesorem!

Wydajność

- Standardowy zestaw testów zaczerpnięty z CILK i Java fork/join library



Wydajność



- Drobna przeróbka w MSAGLu.
- Dostajemy 50% szybszy program.
- Jeśli zrównoleglimy 50% programu, to dostaniemy program najwyżej 2 razy szybszy.

Wydajność Duplicating Queue

- Porównanie wydajności Duplicating Queue i rozwiązania opartego o THE protocol

	speedup	steal	switch	migrate	workers
DUP	3.89	29	20	6	20
THE	2.78	37	2452	9596	53

Bibliografia

- Prezentacja artukułu „The Design of a Task Parallel Library”, OOPSLA 2009

Dziękuję