Lista jednokierunkowa

```
p = head;
while (p) {
    process(p);
    p = p->next;
}
```

Rozważmy przebieg przez kolejne pozycje listy. W pierwotnym OpenMP (rozwijanym w języku Fortran) problem inny niż zwykła pętla nie był brany pod uwagę.

Pierwsze rozwiązanie (niezbyt eleganckie i wymagające potrójnego przebiegu przez listę):

```
while ( p != nullptr ) {
  p = p->next;
  ++count;
}
// tworzymy tablicę parr
// o rozmiarze count
p = head;
for( i=0; i<count; ++i ) {
  parr[i] = p;
  p = p->next;
}
#pragma omp parallel
{
#pragma omp for schedule(static,1)
  for( i=0; i<count; ++i )
  processwork( parr[i] );
}
```

pierwszy przebieg w celu określenia rozmiaru listy (za pomocą zmiennej count)

tworzymy pomocniczą tablicę parr w celu umieszczenia w niej kolejnych pozycji z listy

kopiujemy kolejne wskaźniki z listy to tablicy

na końcu klasyczna pętla for równolegle obliczana, ale jest to de facto zrównoleglenie na tablicy parr, a nie na pierwotnej liście

Lista jednokierunkowa (+ kontener)

```
std::vector<node*> nodelist;
for (p = head; p != nullptr; p = p->next)
   nodelist.push_back(p);
int j = (int)nodelist.size();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < j; ++i)
   processwork(nodelist[i]);
```

Trochę sprawniejsze rozwiązanie z użyciem std::vector, nadal jednak dalekie od eleganckiego.

skopiowanie wszystkich wskaźników z listy do wektora, następnie pobranie jego końcowego rozmiaru

pętla równoległa po wektorze (jak po tablicy)

Kilka wyników (względne porównanie):

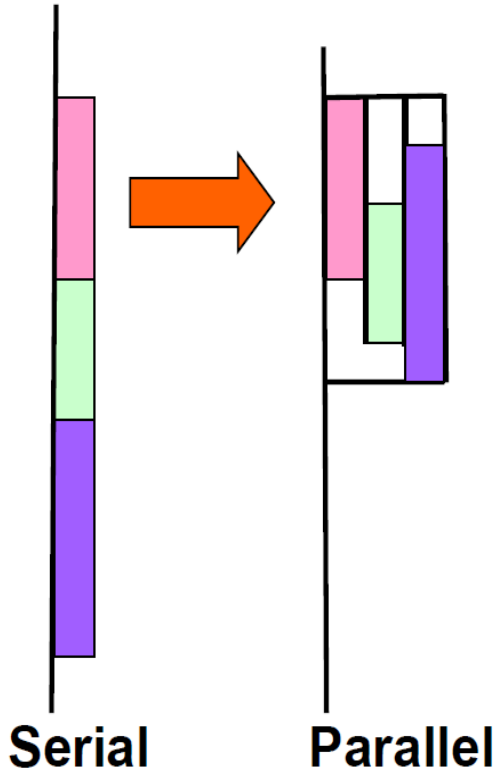
	C++ (default)	C++ (static,1)	Poprzedni program
1 wątek	37 s.	49 s.	45 s.
2 wątki	47 s.	32 s.	28 s.

Powyższe rozwiązania są trudne do zaakceptowania.
Aby możliwe były operacje równoległe poza tablicami i pętlami for, konieczne było obsłużenie bardziej ogólnych struktur danych. Zostało to zrealizowane począwszy od OpenMP ver. 3 za pomocą "zadań" (tasks).

Tasks

Zadania (**tasks**) to niezależne jednostki pracy.

- Zadania składają się z:
- **kod** do wykonania
- środowisko danych
- zmienne kontroli wewnętrznej (ICV)
- Wątki wykonują pracę każdego zadania.
- System decyduje o czasie wykonania zadań
- Zadania mogą zostać odroczone
- Zadania mogą być wykonywane natychmiast



- Konstrukcja **task** przypomina sekcje (**section**), jednak wykonanie sekcji ograniczone jest do bloku **sections**
- Wykonanie zadania może być odłożone w czasie, nie są również na stałe przypisane do jednego wątku

Definicje

Konstrukcja zadania (**task construct**) - dyrektywa zadania oraz jej blok strukturalny

Zadanie (**task**) - pakiet kodu i instrukcji do przydzielania danych utworzonych gdy wątek napotyka konstrukcję zadania

Obszar zadań (**task region**) - dynamiczna sekwencja instrukcje tworzona podczas wykonania zadania przez wątek

Tasks - składnia

```
#pragma omp task [clause [,clause]*]
```

- Możliwe klauzule
  - if (expression)
  - final(expression)
  - untied
  - mergeable
  - default(shared|firstprivate|none)
  - private(list)
  - firstprivate(list)
  - shared(list)
  - depend(list)
  - priority(value)

Gwarancja wykonania zadań

- gwarancja zakończenia wykonania zadań (**tasks**) na barierze: #pragma omp barrier lub barierze zadań: #pragma omp taskwait

```
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
```

wiele zadań foo() utworzonych tutaj, po jeden na wątek

gwarancja, że wszystkie zadania foo() tu się zakończą

jedno zadanie bar() bo jesteśmy w bloku #pragma omp single

gwarancja wykonania bar() na niejawnej barierze na końcu bloku single

Zakres ważności zmiennych w zadaniach

Przykład. Ciąg Fibonacciego
(ciąg liczb naturalnych określony rekurencyjnie)

$$F_n := \begin{cases} 0 & \text{dla } n = 0, \\ 1 & \text{dla } n = 1, \\ F_{n-1} + F_{n-2} & \text{dla } n > 1. \end{cases}$$

```
int fib(int n)
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x)
    x = fib(n-1);
  #pragma omp task shared(y)
    y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
```

n, x, y są zmiennymi prywatnymi w zadaniach

dlatego shared(x) i shared(y) umożliwia obliczenie wyniku i przekazanie go na końcu do sumy

przed obliczeniem końcowej sumy x+y postawiona jest blokada

Sekcje i zadania mogą być stosowane do wygodnego zrównoleglania algorytmów rekurencyjnych. Sekcje i zadania nie nadają się do zadań, które wykonują czasochłonne, blokujące operacje, np. związane z I/O.

Zakres ważności (przykład z listą)

Lista: wprawdzie w bloku równoległym segment #pragma omp single jest wykonany przez jeden (dowolny) wątek, ale wewnątrz pętli utworzone są zadania (tyle, ile przebiegów pętli), ich czas wykonania jest arbitralny.

```
List ml; // moja lista
Element *e;
#pragma omp parallel
#pragma omp single
{
  for(e=ml->first; e; e=e->next)
  #pragma omp task firstprivate(e)
    process(e);
}
```

gdyby nie było klauzuli firstprivate(e), to zadań proces(e) przekazano by współdzieloną zmienną: możliwe data race

sprawdźmy też numery wątków: **omp_get_thread_num()**

Przykłady (ćwiczenie) – co będą pokazywać na ekranie kolejne wersje prostego kodu (fragment w main):

```
#include <iostream>
#include "omp.h"
using namespace std;
int main() {
#pragma omp parallel num_threads(4)
{
   // ZAWARTOŚĆ
}
cout << " Koniec " << endl;
}
```

```
cout << " Raz ";
cout << " Dwa ";
cout << " Trzy ";
```

```
#pragma omp single
{
   cout << " Raz ";
   cout << " Dwa ";
   cout << " Trzy ";
}
```

```
#pragma omp single
{
   cout << " Raz ";
   #pragma omp task
   { cout << " Dwa "; }
   #pragma omp task
   { cout << " Trzy "; }
   cout << " Cztery ";
}
```

```
#pragma omp single
{
   cout << " Raz ";
   #pragma omp task
   { cout << " Dwa "; }
   #pragma omp task
   { cout << " Trzy "; }
   #pragma omp taskwait
   cout << " Cztery ";
}
```

Konstrukcja task

Kolejność wykonywania wątków zaangażowanych w zadanie:

```
#pragma omp parallel
{
  #pragma omp single
  {
    node* p = head;
    while (p) {
      #pragma omp task firstprivate(p)
        process(p);
      p = p->next;
    }
  }
}
```

1) tworzy grupę wątków w bloku równoległym

2) jeden z wątków wykonuje konstrukcję single – wtedy pozostałe wątki czekają na końcu bloku single, gdzie jest niejawna bariera synchronizująca

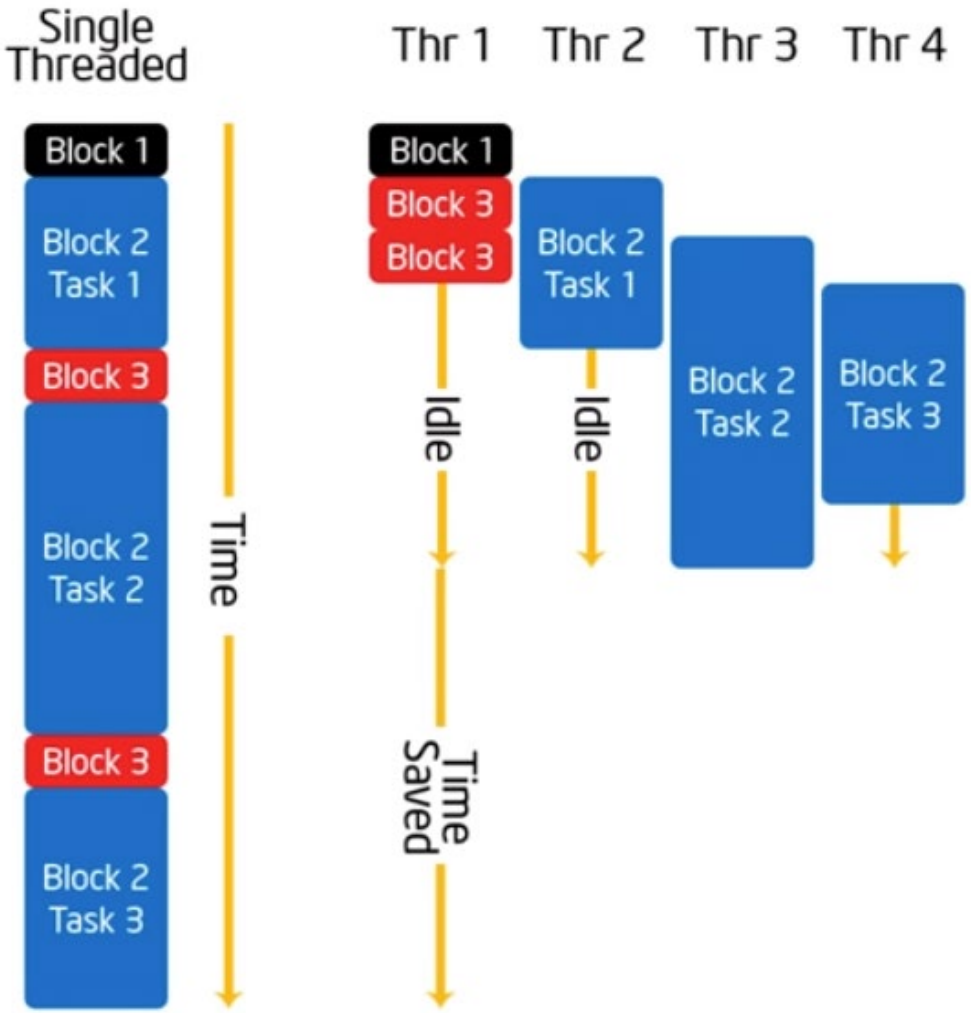
3) pojedynczy wątek w bloku single tworzy zadanie (task), które ma własną kopię lokalnej zmiennej p

4) w takiej sytuacji wątki czekające na barierze wykonują zadanie (task)

5) gdy wszystkie wątki zakończą pracę (zadanie), dalsze wykonanie programu przechodzi poza barierę bloku single

Konstrukcja task

```
#pragma omp parallel
{
  #pragma omp single
  {
    node* p = head;
    while (p) {
      #pragma omp task firstprivate(p)
        process(p);
      p = p->next;
    }
  }
}
```



Pomiar czasu wykonania programu

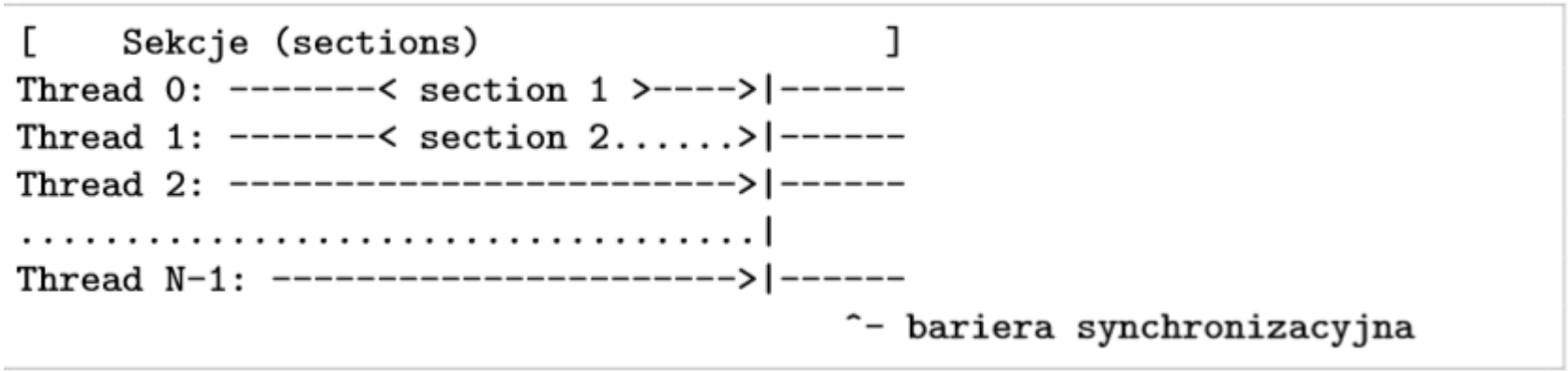
- OpenMP udostępnia funkcję `double omp_get_wtime()`, która zwraca liczbę sekund od pewnego ustalonego momentu w przeszłości
- Rozdzielczość pomiaru można pobrać za pomocą funkcji `double omp_get_wtick()`, jest to najkrótszy mierzalny okres czasu

```
#include <iostream>
#include <thread>
#include <iomanip>
#include "omp.h"
using namespace std;
```

```
int main() {
    double start = omp_get_wtime();
    this_thread::sleep_for(1234ms);
    double stop = omp_get_wtime();
    cout << "Start: " << setprecision(4) << start
         << " stop: " << stop << " stop-start: "
         << stop-start << endl;
    cout << "Precision: " << omp_get_wtick()
         << endl;
}
```

Sekcje a zadania

- Zadania (ang. tasks) i sekcje (ang. sections) są do siebie podobne, jednak istnieje między nimi kilka różnic
- Sekcje definiowane są wewnątrz bloku sections i żaden z wątków roboczych nie opuści bloku dopóki wszystkie sekcje nie zostaną zakończone



- N wątków napotyka blok sections zawierający jedynie dwie sekcje
- Dwa wątki wykonują sekcje
- Pozostałe N-2 wątków musi czekać

Sekcje a zadania

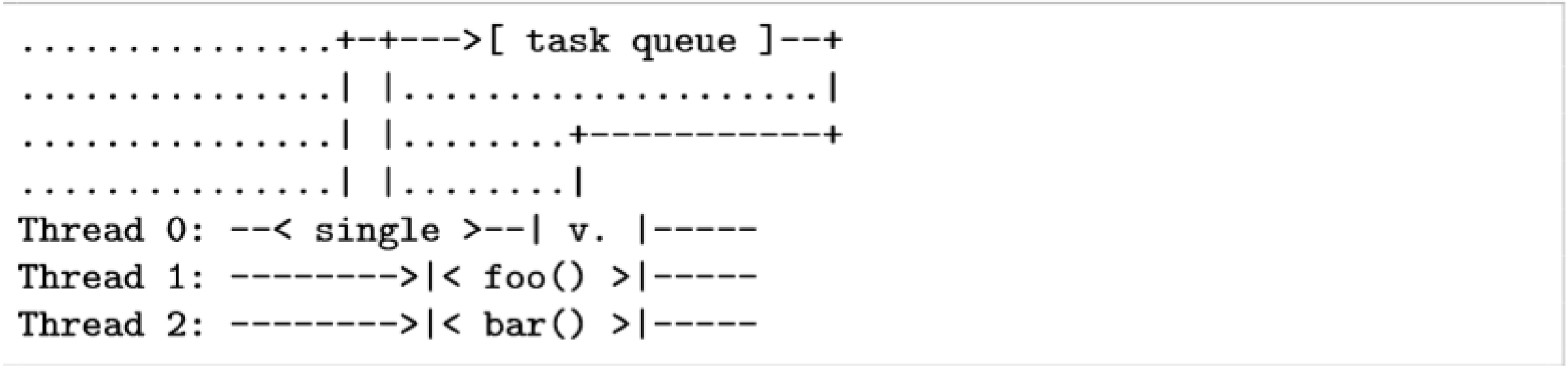
- Zadania są kolejkowane i wykonywane we wskazanych punktach synchronizacji
- Dopuszczalna jest migracja zadań między wątkami roboczymi, tzn. może się zdarzyć, że zadanie w trakcie wykonywania zostanie przeniesione z wątku 1 na 2

```
1 // sections
2 ...
3 #pragma omp sections
4 {
5    #pragma omp section
6    foo();
7    #pragma omp section
8    bar();
9 }
```

```
1 // tasks
2 ...
3 #pragma omp single nowait
4 {
5    #pragma omp task
6    foo();
7    #pragma omp task
8    bar();
9 }
10 #pragma omp taskwait
```

- taskwait działa podobnie do barrier – powoduje, że wątki robocze zanim przejdą dalej muszą najpierw wykonać wszytkie zakolejkowane zadania
- dyrektywa single powoduje, że zadania zostaną wygenerowane tylko przez jeden wątek roboczy

Sekcje a zadania



- W przykładzie zadania tworzone są przez pierwszy wątek, pozostałe dwa dzięki klauzuli nowait kontynuują obliczenia do napotkania klauzuli taskwait
- Wtedy pobierają zadania z kolejki i je wykonują
- Jeżeli nie ma jawnie określnonych punktów synchronizacji, to OpenMP może wykonywać zadania w dowolnym momencie wewnątrz bloku parallel

Ciąg Fibonacciego

```cpp
#include <iostream>
#include <omp.h>
using namespace std;

const unsigned N = 5;
const unsigned FS = 38;

struct node {
    int data;
    int fibdata;
    node* next;
};

int fib(int n) {
    if (n < 2) {
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

void processwork(node* p)
{
    int n = p->data;
    p->fibdata = fib(n);
}
```

```cpp
node* init_list(node* p) {
    int i;
    node* head { nullptr };
    node* temp { nullptr };

    head = new node;
    p = head;
    p->data = FS;
    p->fibdata = 0;
    for (i=0; i<N-1; ++i) {
        temp = new node;
        p->next = temp;
        p = temp;
        p->data = FS + i + 1;
        p->fibdata = i + 1;
    }
    p->next = nullptr;
    return head;
}
```

```cpp
int main() {
    double start, end;
    node *p { nullptr };
    node *temp { nullptr };
    node *head { nullptr };
    cout << N << " liczb ciagu, od " << FS << endl;
    p = init_list(p);
    head = p;

    start = omp_get_wtime();
    {
        while (p != nullptr) {
            processwork(p);
            p = p->next;
        }
    }
    end = omp_get_wtime();

    p = head;
    while (p != nullptr) {
        cout << p->data << " : " << p->fibdata << endl;
        temp = p->next;
        delete p;
        p = temp;
    }
    delete p;
    cout << "Czas: " << end-start << " sekund\n";
}
```

Ciąg Fibonacciego

```
int main() {
    double start, end;
    node *p { nullptr };
    node *temp { nullptr };
    node *head { nullptr };
    cout << N << " liczb ciagu, od " << FS << endl;
    p = init_list(p);
    head = p;

    start = omp_get_wtime();
    {
        while (p != nullptr) {
            processwork(p);
            p = p->next;
        }
    }
    end = omp_get_wtime();

    p = head;
    while (p != nullptr) {
        cout << p->data << " : " << p->fibdata << endl;
        temp = p->next;
        delete p;
        p = temp;
    }
    delete p;
    cout << "Czas: " << end-start << " sekund\n";
}
```

```
start = omp_get_wtime();
#pragma omp parallel
{
  #pragma omp master
  cout << "Liczba watkow: " << omp_get_num_threads() << endl;
  #pragma omp single
  {
      p = head;
      while (p != nullptr) {
          #pragma omp task firstprivate(p)
          {
            processwork(p);
          }
          p = p->next;
      }
  }
}
end = omp_get_wtime();
```