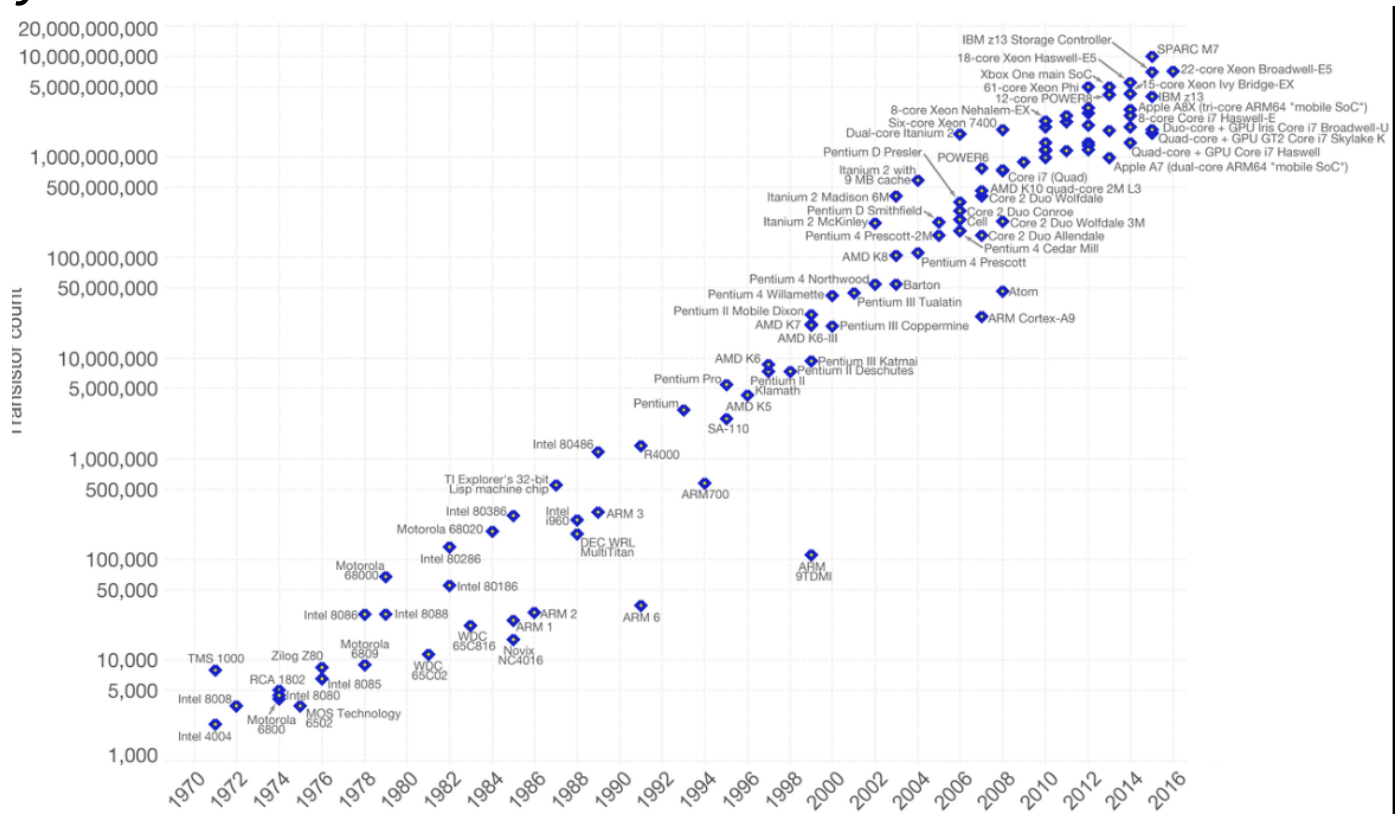
AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Intel® Threading Building Blocks

Autor: Karolina Mizera

Prawo Moore'a

Przekonanie, że „optymalna liczba tranzystorów w układzie scalonym będzie zwiększać się w kolejnych latach wykładniczo”

- 

Programowanie równoległe

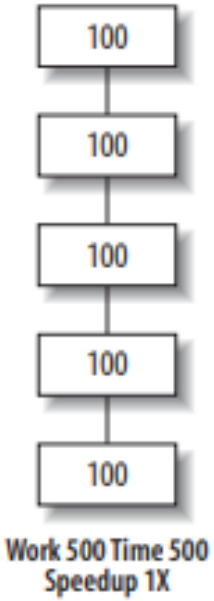
Programując równolegle skupiamy się na następujących kwestiach:

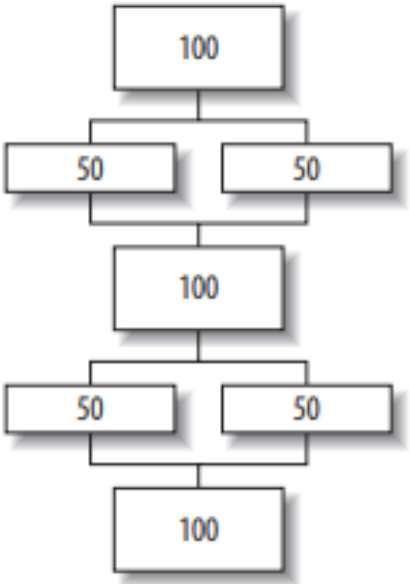
- Skalowalność
- Kompozycja programu
- Zarządzanie wątkami
- Bezbłędność/Poprawność
- Wzorce
- Abstrakcja

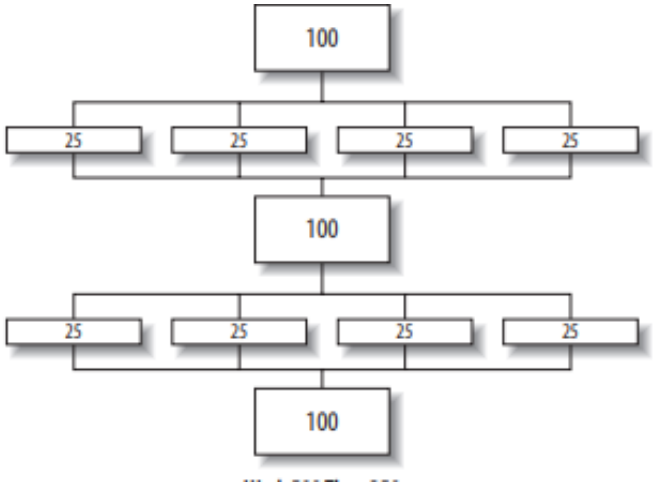
Skalowalność i przyśpieszenie
Prawo Amdahla

Skalowalność to miara jakie przyśpieszenie osiągniemy, gdy nasz program będzie przez coraz większą liczbę procesorów

Prawo Amdahla definiuje wartość maksymalnego spodziewanego zwiększenia wydajności całkowitej systemu, jeżeli przyśpieszeniu ulegnie tylko jego część.







Przyśpieszenie i wydajność

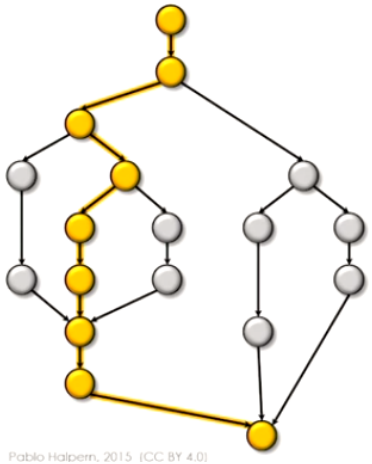
Praca - T_1 – czas, jaki zajmie wykonanie danego zadania jednemu proc.

Najdłuższa gałąź, krytyczna ścieżka – T_∞ - czas potrzebny do wykonania najdłuższej sekwencyjnej ścieżki

T_P – czas jaki zajmie wykonanie tego zadania P procesorom

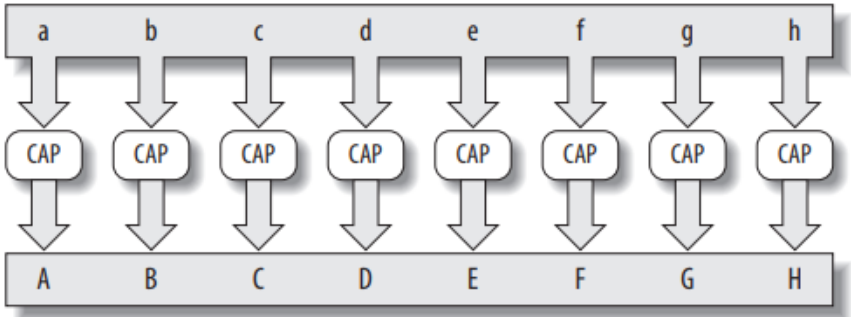
Przyśpieszenie na P procesorach – T_1/T_P

Równoległość – T_1/T_∞ -maksymalne teoretyczne przyśpieszenie na nieskończonej liczbie procesorów

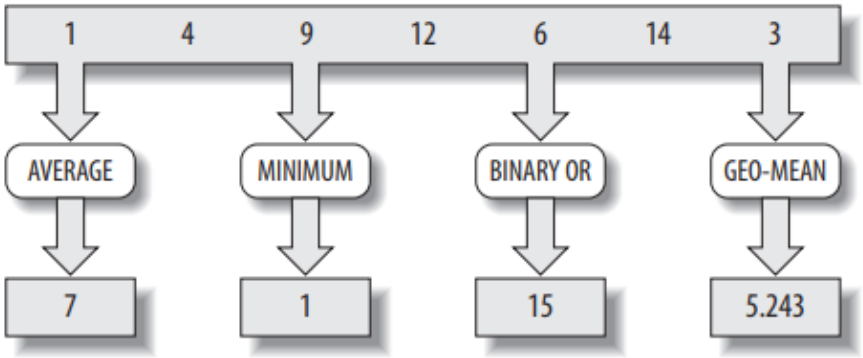


Kompozycja – z jaką równoległością mamy do czynienia?

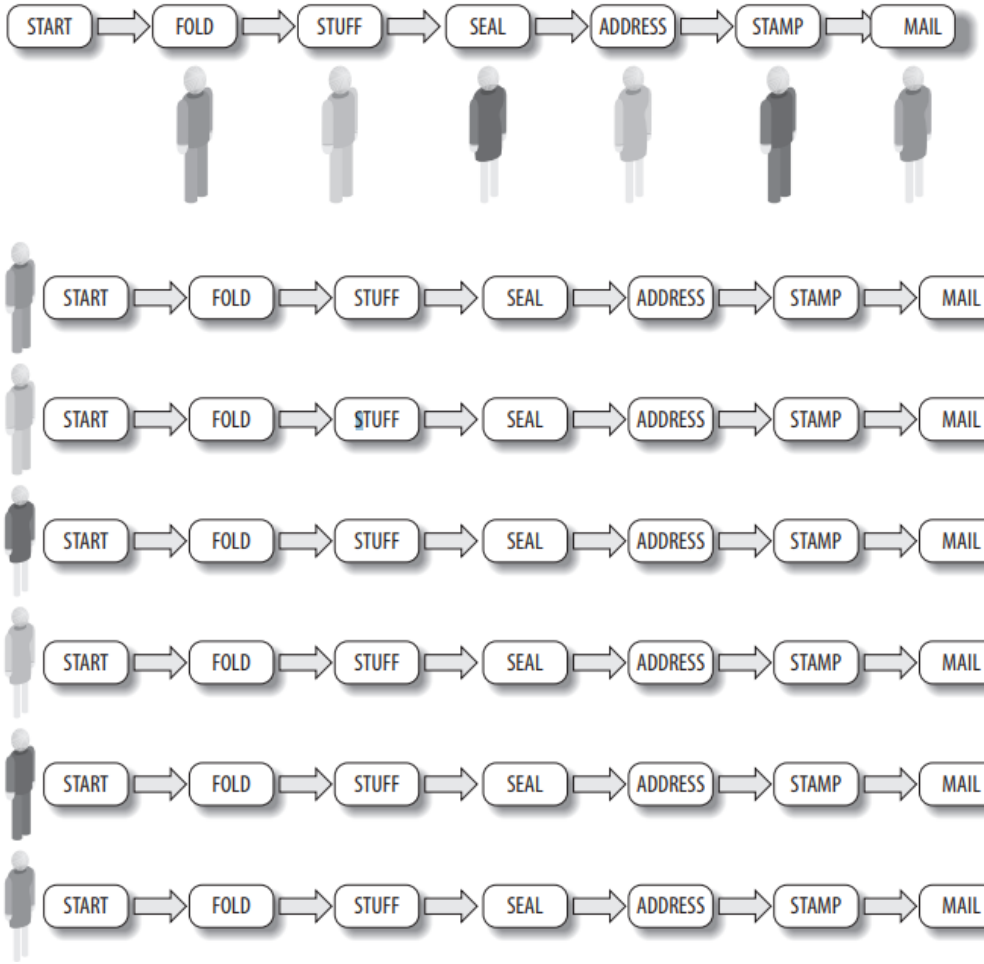
Równoległość danych – na każdym elemencie ze zbioru danych przeprowadzamy operację

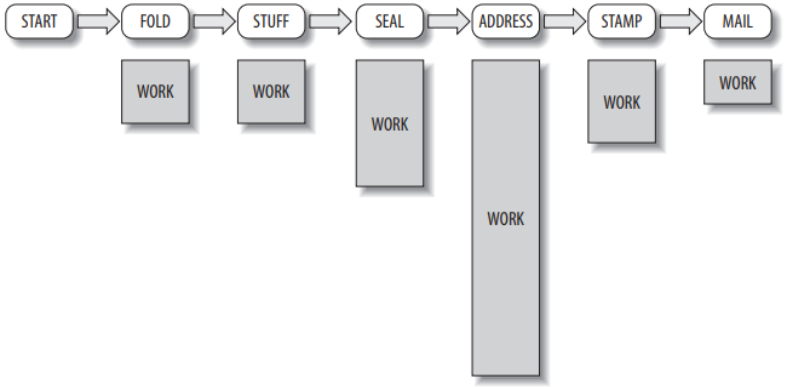


Równoległość zadań – różne, niezależne zadania operujące na tym samym zbiorze danych



Kompozycja – przykład teoretyczny





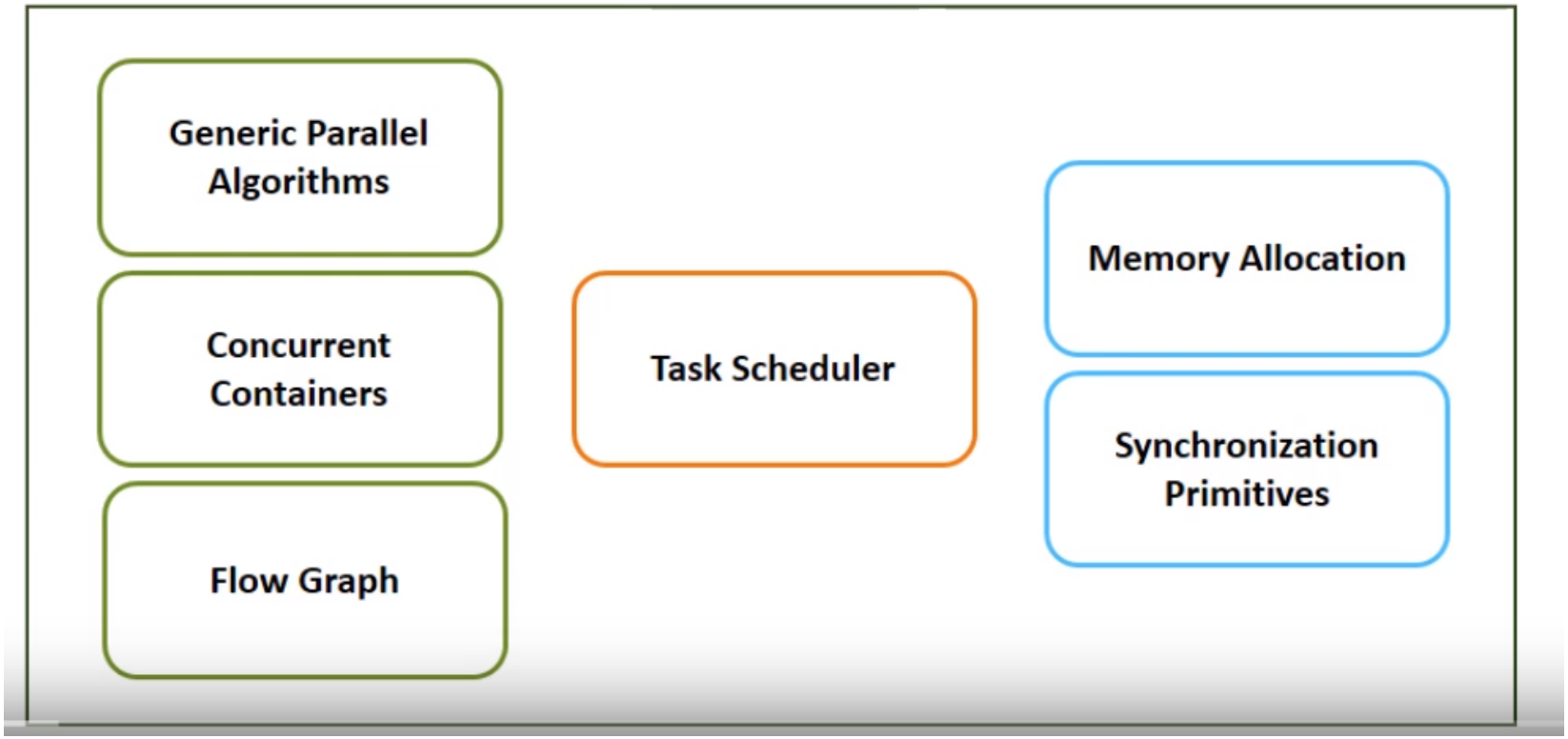
Intel® Threading Building Blocks

- Biblioteka szablonów napisana w języku C ++ umożliwiająca programowanie równoległe zgodnie ze standardem ISO C++.
- Open-source
- Łączy w sobie zarówno proste w użyciu algorytmy czy kontenery jak i niskopoziomowe założenia
- Abstrakcja na zarządzanie wątkami w postaci harmonogramu zadań

Zalety Intel TBB

1. Łatwa do uzyskania dla programisty skalowalność programu.
2. Wysokopoziomowy interface do zarządzania wątkami.
3. Współpraca z innymi wielowątkowymi bibliotekami.
4. Biblioteka oparta na generycznym programowaniu.
5. Bardzo duże podobieństwo do STL.
6. Building Blocks – wiele łatwo łączących się elementów.
7. Zapewnia mechanizmy programowania zagnieżdżonego.

Budowa Intel TBB



HARMONOGRAM ZADAŃ
TASK SCHEDULER

Task-based programing

- zadanie do wykonania - pojedyncza jednostka obliczeniowa

  Intel TBB mapuje zadania na fizyczne wątki.

  Zalety:
- Dopasowuje równoległość do dostępnych zasobów
- Pozawala na lepsze zbalansowanie zadań
- Implementuje wysokopoziomowe programowanie wielowątkowe
- Odciąża programistę z zarządzania fizycznymi wątkami

task class

-abstrakcyjna klasa bazowa, instancja tej klasy to obiekt C++

- stworzenie instancji tej klasy wymaga implementacji metody **task::execute()**

- obiekty klasy *task* są automatycznie niszczone przez *task_scheduler* po wykonaniu metody *execute()*

Atrybuty:

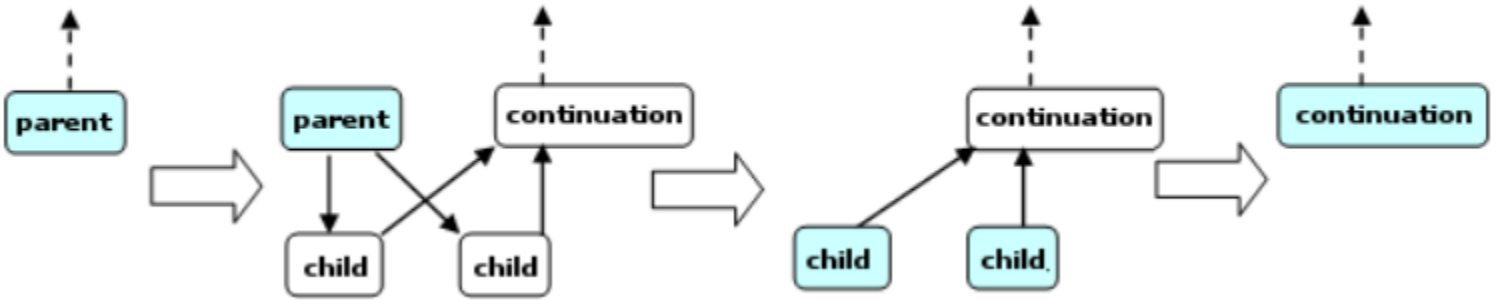
successor - następnik

refcount – liczba tasków, które mają ten task jako rodzic

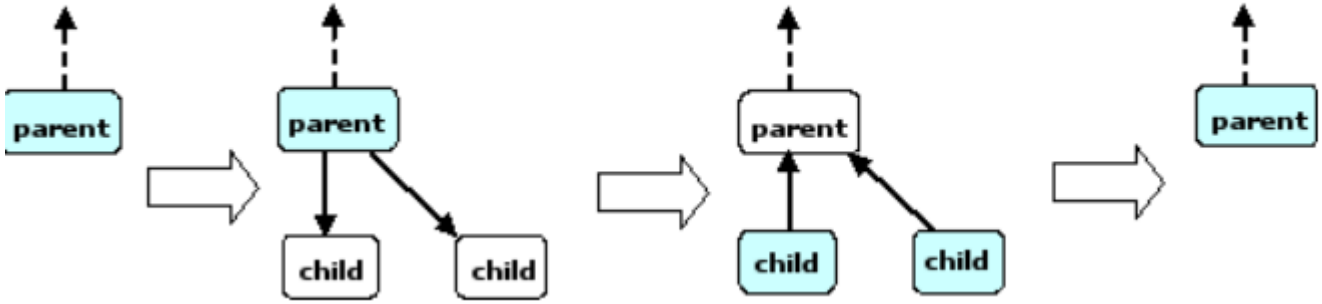
Harmonogram zadań

- Silnik, który zarządza wykonywaniem równoległym wykonywaniem pętli
- Mapuje zadania w fizyczne wątki
- Zarządza wątkami w sposób wydajny i ukrywa złożoność tych operacji
- Generyczne algorytmy Intel TBB bazują na tym mechaniźmie
- Pozwala na na implementację 2 typowych wzorców programowania równoległego:
- task_scheduler_init()

Continuation-passing Style



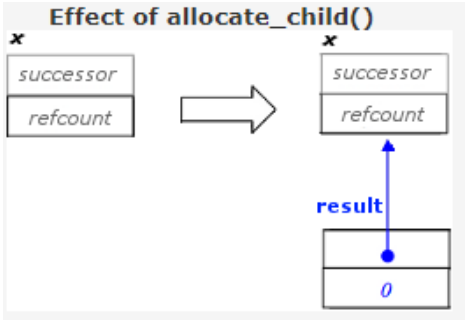
Blocking Style

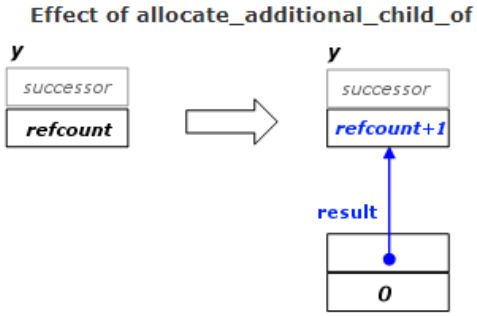


Alokacja nowych task'ów

- Alokacja pamięci na task zachodzi za pomocą jednego z przeładowanych operatorów new.
- Sama alokacja nie konstruuje task'a → zwraca proxy, które będzie argumentem przeładowanego operatora new dostarczonego przez bibliotekę
- Alokacja nowego zadania powinna zostać wykonana zanim wcześniejszy task (poprzednik) nie zostanie wykonywany → wyjątek allocate_additional_child_of(old_task)

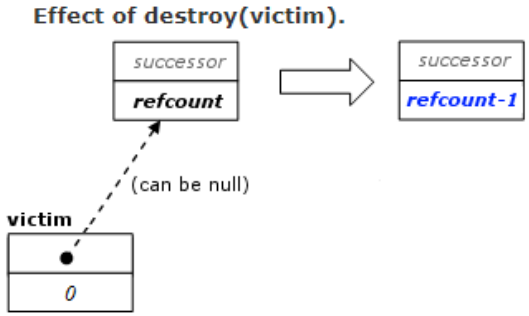
Typy alokacji nowych zadań





- new (x.allocate_child()) T
- new (x.allocate_continution()) T
- new(task::allocate_additional_child_of(t)) T
- new(task::allocate_root(task_group_context& group)) T

Typy są niszczone w momencie zakończenia metody execute(), lub za pomocą metody **static void destroy(task& victim)**;



Kradzież pracy – work stealing

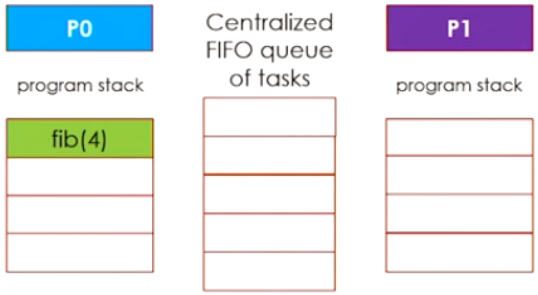
- Zadania są identyfikowane dynamicznie w trakcie execute()
- Zadania są wykonywane przez zadaną pulę wątków
- Gdy dochodzimy do momentu rozdzielenia jednego zadania na dwa(split), jedno z zadań jest odkładane na kolejkę zadań wątku
- Każdy wątek, którego kolejka zadań jest pusta może „ukraść task” z kolejki innego wątku.

  -współdzielona kolejka dwukierunkowa (TBB)

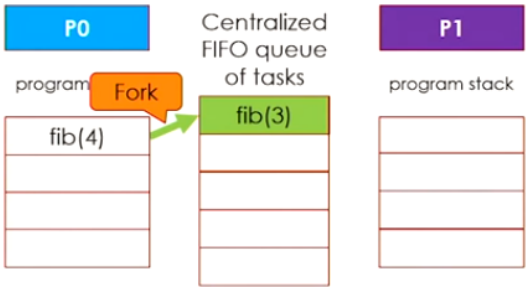
  -centralizowana kolejka

Rodzaje zarządzania zadaniami-kolejka zcentralizowana

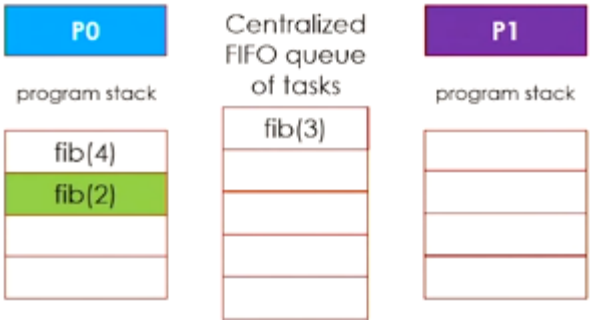
1.



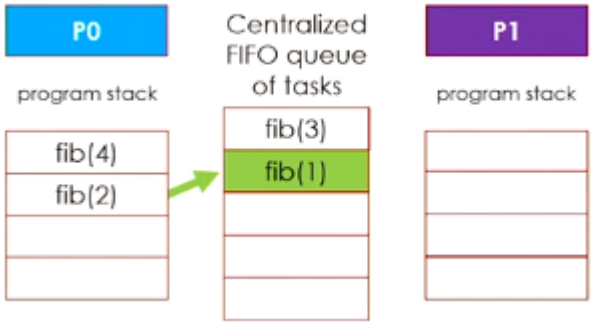
2.



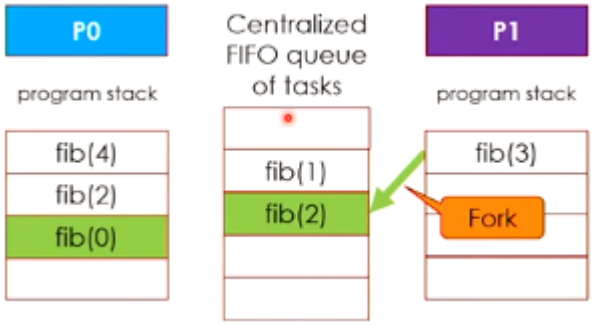
3.



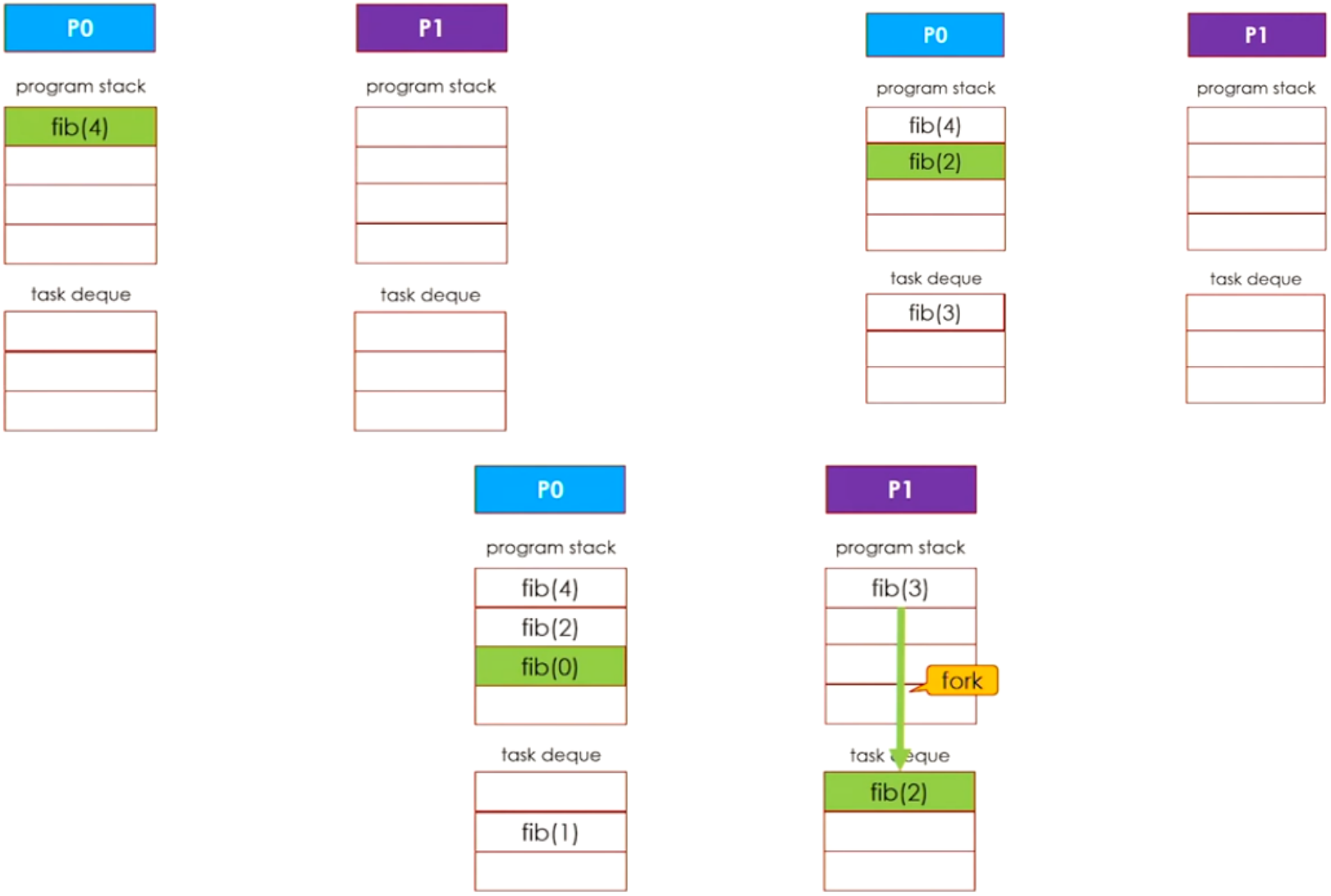
4.



5.



Rodzaje zarządzania zadaniami -rozproszona kolejka dwukierunkowa



Algorytm działania – work-stealing

Po wykonaniu task t wątek wybiera następny task do wykonania w następującej kolejności:

1. Task zwrócony przez t.execute()
2. Następca t (successor), jeżeli t było jego ostatnim poprzednikiem
3. Zostanie wykonany task, który zostanie pobrany z końca własnej kolejki dwukierunkowej wątku.
4. Task, z ‘powinowactwem’ wobec watku
5. Task ściągnięty z szacowanego początku kolejki współdzielonej
6. Task ściągnięty z początku kolejki tasków innego wątku.

Grupy zadań (Task Groups)

Wysokopoziomowy interface zarządzania zadaniami

```
template<typename Func> task_handle;
template<typename Func> task_handle<Func> make_task( const Func& f );
enum task_group_status;
class task_group;
class structured_task_group;
bool is_current_task_group_canceling();
```

class task_group

Klasa reprezentująca współbieżną egzekucję grupy zadań.

Zadania do grupy można dodawać dynamicznie .

- Metoda run(...)

Class task_group

```
class task_group
{
  public:
    task_group();
    ~task_group();
    //run() -> task wykonywuje funkcje f
    template <typename Func>
    void run(const Func &f);

    // Wspierane od C++11->lambda
    template <typename Func>
    void run(Func &&f);

    template <typename Func>
    void run(task_handle<Func> &handle);
    // {run(f); return wait()}, gwarantuje, ze f zostanie wykonane w tym samym wątku
    template <typename Func>
    task_group_status run_and_wait(const Func &f);

    template <typename Func>
    task_group_status run_and_wait(task_handle<Func> &handle);
    //czeka na wszystkie inne zadania w grupie aż się wykonają lub zostaną anulowane
    task_group_status wait();
    bool is_canceling();
    //anuluje wykonywanie wszystich zadań w grupie
    void cancel();
};
```

```
enum task_group_status {
    not_complete,
    complete,
    canceled
};
```

Split

Splittable Concept	
Pseudo-Signature	**Semantics**
`X::X(X& x, split)`	Split `x` into `x` and newly constructed object.

Konstruktor dekompozycyjny pozwala na rozdzielenie instancji obiektu na dwie części.

- typ split pozwala nam rozróżnić konstruktor dekompozycyjny od konstruktora kopiującego

  proportional_split
- parallel_scan, paralle_reduce
- blocked_range, blocked_range2d → analiza klasy

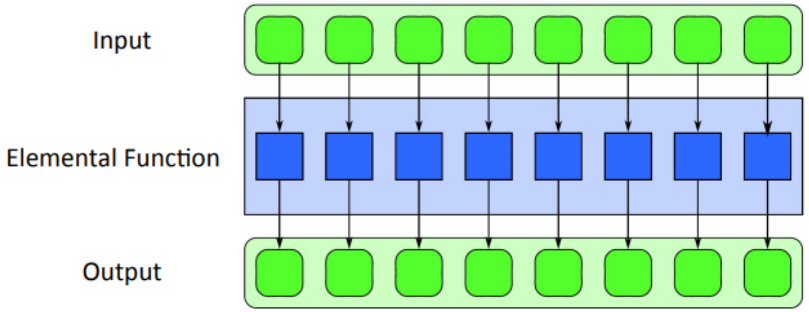
GENERYCZNE ALGORYTMY RÓWNOLEGŁE

Generyczne algorytmy równoległe

Wzorce są strukturalnym sposobem myślenia o aplikacjach i modelach programowania

- parallell_for, pararell_for_each → map
- parallell_do → workpile(map + incr.task.addition)
- parallell_reduce → reduce
- parellell_scan →scan
- parallell_pipeline → pipeline
- parallell_sort
- parallell_invoke

parallel_for → map



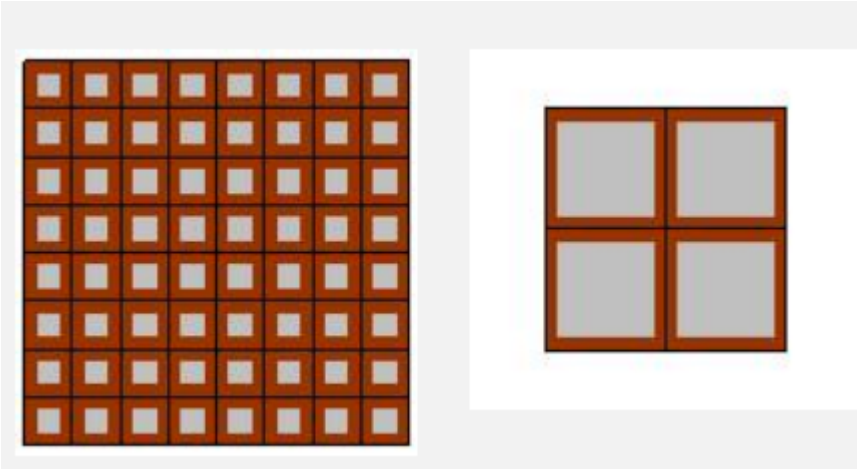

```
template<typename Index, typename Func>
Func parallel_for( Index first, Index_type last, const Func& f
                   [, partitioner[, task_group_context& group]] );

template<typename Index, typename Func>
Func parallel_for( Index first, Index_type last,
                   Index step, const Func& f
                   [, partitioner[, task_group_context& group]] );

template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body,
                   [, partitioner[, task_group_context& group]] );
```

Grain size – wielkość podziału

Wartość ziarna definiuje minimalna ilości iteracji na podzielną część zadania.



Szary obszar definiuje pracę , jaką trzeba wykonać, Brązowy – koszty podziału zadania

Wielkość ziarna a szybkość wykonania

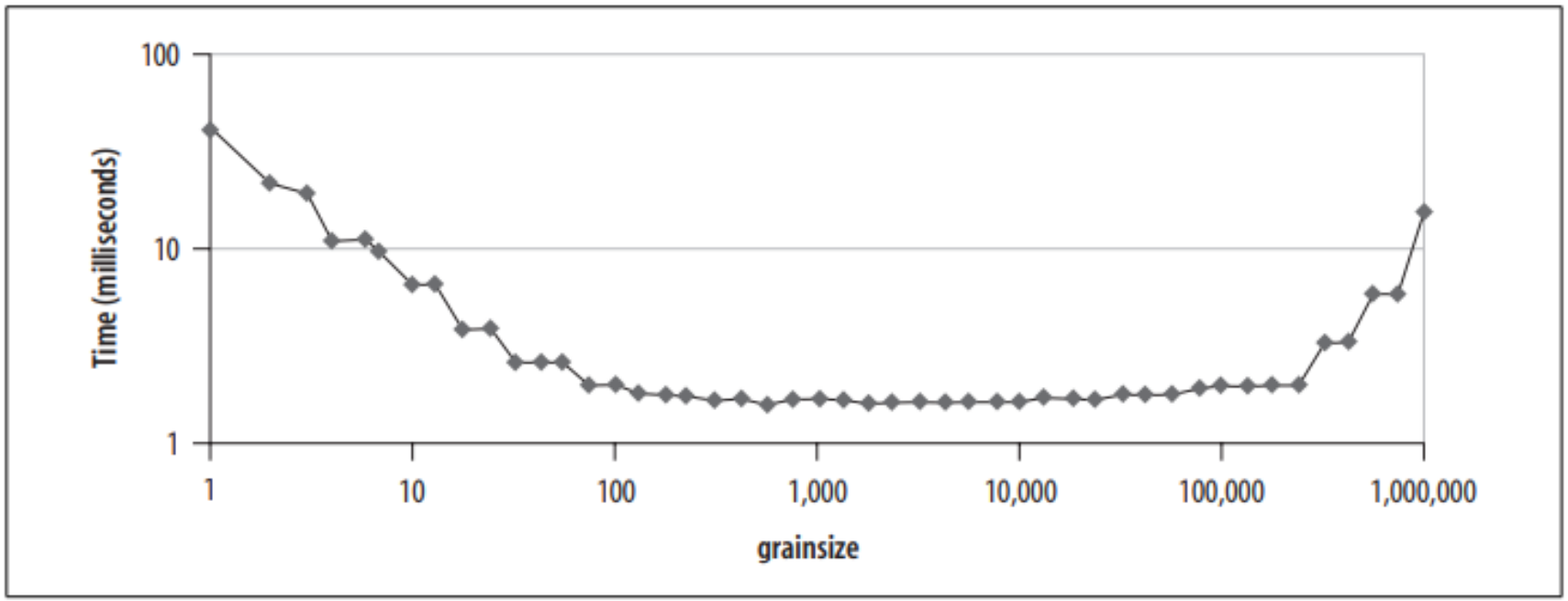


Figure 3-2. Wall clock time versus grainsize

Podział na części - partitioner

Jest dodatkowym argumentem **parallel_for i parallel_reduce.**

Silnie powiązane z **blocked_range**(i,j,**grainsizer**).

Ustala on strategię wykonywania równoległych pętli, a dokładniej ich podziału. Domyślnie: auto_partitioner();

affinity_partitioner

-automatycznie wybiera wielkość ziarna

-optymalizuje dostęp do pamięci cache

-równomiernie dystrybuuje dane pomiędzy wątki

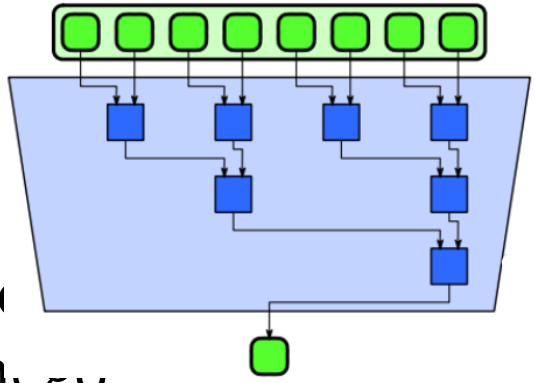
Kiedy używać:

- przeprowadzamy kilka operacji na jednej zestawie danych
- pętla przeprowadza często operacje na zmiennej
- dana już znajdująca się w cache pasują do tych w pętli

partitioner

`simple_partitioner`	Chunksize bounded by grain size.	$g/2 \leq chunksize \leq g$
`auto_partitioner` (default)[4]	Automatic chunk size.	$g/2 \leq chunksize$
`affinity_partitioner`	Automatic chunk size, cache affinity and uniform distribution of iterations.	
`static_partitioner`	Deterministic chunk size, cache affinity and uniform distribution of iterations without load balancing.	$\max(g/3, problem\_size/num\_of\_resources) \leq chunksize$

parallel_reduce → reduction



- Redukcja wykonuje operacje na każdym el zbioru i zwraca wynik tych operacji do jedn elementu
- Często wymaga zamiany kolejności w wejściowym zbiorze, zeby pozwolić na równoległość
- Zależności pomiędzy kolejnym stopniami wykonywania algorytmy

Reduction with `parallel_reduce`

- Good when
  - Operation is non-commutative
  - Tiling is important for performance

```
sum = parallel_reduce(
   blocked_range<int>(0,n),
   0.f,
   [&](blocked_range<int> r, float s) -> float
   {
      for( int i=r.begin(); i!=r.end(); ++i )
         s += a[i];
      return s;
   },
   std::plus<float>()
);
```

Recursive range

Identity value.

Reduce subrange

Initial value for reducing subrange r. Must be included!

Functor for combining subrange results.

parallel_invoke

```
template<typename Func0, typename Func1>
void parallel_invoke(const Func0& f0, const Func1& f1);


template<typename Func0, typename Func1, typename Func2>
void parallel_invoke(const Func0& f0, const Func1& f1, const Func2& f2);
template<typename Func0, typename Func1, ..., typename Func9>
void parallel_invoke(const Func0& f0, const Func1& f1, ..., const Func9& f9);
```

- Pozwala na wykonywanie od 2 do 10 funkcji równolegle
- Każdy argument parallel_invoke musi mieć zdefiniowany operator()
- Jako argumenty może przyjmować funktory, lambdy lub wskaźniki do funkcji

Mierzenie czasu

Kiedy staramy się mierzyć wydajność programów równoległych ‘**czas rzeczywisty**’, a nie czas wskazywany przez CPU **tbb::tick_count** zapewnia prosty interface do mierzenia czasu rzeczywistego.

- tick_count::**now()** zwraca czas rzeczywisty

  Różnica/suma obiektów tick_count jest typu **interval_t** i może być konwerowana na **sekundy.**

SKALOWALNE ALOKATOR PAMIĘCI

Alokacja pamięci – potencjalne problemy

- Skalowalność – klasyczne alokując pamięć zakładają globalną blokadę na kopiec. W ten sposób zwiększając liczbę procesorów wykonujących alokację możemy spowolnić program! Każdy kontener STL ma domyślnie zaimplementowaną alokacje ,która nie jest bezpieczna w kontekście wątków.

- Fałszywe współdzielenie – wiele wątków używa pamięci, która została zaalokowana blisko siebie. Procesor przechowuje pamięć w tzw. cache lines – dostęp do pojedynczej linii powinien być tylko przez jeden wątek.

Rozwiązanie: alokatory Intel TBB

- tbb::scalable_allocator<T>

  - zapewnia skalowalność, nie rozwiązuje problemu współdzielenia

- tbb::cache_aligned_allocator<T>

  -zapewnia skalowalność i poprawne współdzielenie

  -dwa obiekty są na pewno w różnych cache line, jeżeli oba zostały zaalokowane za pomocą tego alokatora

  -duży koszty pamięci, jeżeli mamy styczność z małymi obiektami

WSPÓŁBIEŻNE KONTENERY

Krótki opis ogólny

Umożliwiają wielu wątkom dostęp do danych i możliwość zmiany ich wartości.

Współbieżność kontenerów TBB została uzyskana poprzez:

- Fine-grained locking – blokowana jest tylko ta część programu, która faktycznie ulega modyfikacji
- Mechanizm Lock-free – inne wątki widząc zmiany w kontenerze automatycznie je przetwarzają

Kontenery STL z nie implementują współbieżnego modyfikowalnego dostępu i są oparte na mechanizmie kopiowania. Jednakże nie mają kosztów wynikających z wielowątkowości.

Dostępne kontenery:

- **concurrent_queue**
- concurrent_bounded_queue
- concurrent_priority_queue
- **concurrent_vector**
- concurrent_unordered_map and concurrent_unordered_multimap
- **concurrent_hash_map**
- concurrent_unordered_set and concurrent_unordered_multiset

tbb::concurrent_vector

Zawiera interface znany z vectora STL:

- reserve, clear, swap, empty,
- Rozmiar: size, max_size, resize, capacity,
- Metody dostępowe: operator [], at(), front, back.

Iteratory:

- cbegin, rbegin, begin , crbegin, cend , rend, end, crend

Metody, którymi można rozszerzać współbieżnie kontener:

- push_back
- grow_by(std::initializer_list<T>), grow_by()
- emplace_back

  Rekursywnie podzielny zakres:

  (const_)range_type range(size_t grainsize=1) (const)
- range_type – iteruje po wektorze i pozwala na dostęp R/W
- const_range_type – dostęp R/O

  Metoda shrink_to_fit()

Tbb::concurrent_queue (fifo)

- Współbieżne operacje pop and push
- Podobny interface do STL std::queue. Różnice:

  Brak metod front() i back()
- Metoda try_pop

  metoda unsafe_size() może zwrócić niepoprawną wartość, jeżeli try_pop jest współbieżnie wykonywane

```
void push( const T& source )
```

```
void push(T&& elem)
```

```
template<typename... Arguments> void
emplace(Arguments&&... args);
```

```
bool try_pop ( T& destination )
```

```
void clear()
```

```
size_type unsafe_size() const
```

```
bool empty() const
```

```
allocator_type get_allocator() const
```

Interface:

```
concurrent_queue( const allocator_type& a =
allocator_type() )
```

```
concurrent_queue( const concurrent_queue& src, const
allocator_type& a = allocator_type() )
```

```
template<typename InputIterator> concurrent_queue(
InputIterator first, InputIterator last, const
allocator_type& a = allocator_type() )
```

```
concurrent_queue( concurrent_queue&& src )
```

```
concurrent_queue( concurrent_queue&& src, const
allocator_type& a )
```

```
~concurrent_queue()
```

```
void push( const T& source )
```

```
void push(T&& elem)
```

```
template<typename... Arguments> void
emplace(Arguments&&... args);
```

```
bool try_pop ( T& destination )
```

```
void clear()
```

```
size_type unsafe_size() const
```

```
bool empty() const
```

```
allocator_type get_allocator() const
```

- Ograniczone wsparcie dla iteratorów

```
iterator unsafe_begin()
```

```
iterator unsafe_end()
```

```
const_iterator unsafe_begin() const
```

```
const_iterator unsafe_end() const
```

ITERATORY

Rodzaje:

```
#include "tbb/iterators.h"
```

- Couting Iterator – random access iterator dla algorytmów STL

  - pozwala na jednoczesne używanie algorytmów STL w ciele algorytmów pochodzących z Intel TBB, jak i odwrotnie/
- Zip Iterator - random access interator

  - umożliwia jednoczesną iterację po wielu kontenerach w algorytmach STL.

Couting interator

```cpp
#include <vector>
#include <algorithm>
#include <tbb/parallel_for.h>
#include <tbb/iterators.h>

int main() {
    std::vector<int> vec(1000000);

    tbb::parallel_for( tbb::blocked_range<int>(0, vec.size(), 100),
    [&vec](tbb::blocked_range<int>& r) {
        using c_it = tbb::counting_iterator<int>;
        std::copy(c_it(r.begin()), c_it(r.end()), vec.begin());
    });
    return 0;
}
```

```cpp
#include <vector>
#include <algorithm>
#include <tbb/parallel_for.h>
#include <tbb/iterators.h>

int main() {
    const int N = 100000;
    float a[N];

    std::for_each(tbb::counting_iterator<int>(0),
    tbb::counting_iterator<int>(N),
    [&a](int i){
        if(i%2 == 0)
            a[i] = i*i;
        else
            a[i] = i;
    });
    return 0;
}
```

Zip Iterator

```cpp
#include <algorithm>
#include <tuple>
#include <vector>
#include <tbb/iterators.h>

int main() {
    const int N = 100000;
    std::vector<float> a(N), b(N), c(N);

    tbb::counting_iterator<int> cnt0(0), cntN(N);
    std::for_each(cnt0, cntN, [&a](int i){ a[i] = i*i; });
    std::for_each(cnt0, cntN, [&b](int i){ b[i] = i*i*i; });

    auto start = tbb::make_zip_iterator(a.begin(), b.begin(), c.begin());
    auto end   = tbb::make_zip_iterator(a.end(), b.end(), c.end());

    std::for_each(start, end, [](const std::tuple<float&, float&, float&>& v) {
        std::get<2>(v) = std::get<0>(v) + std::get<1>(v);
    });

    return 0;
}
```

FLOW GRAPH

Grafy

- Alternatywa dla równoległych pętli.

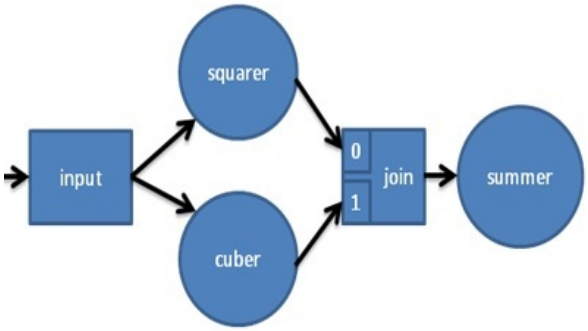
  3 główne komponenty do implementacji grafów:

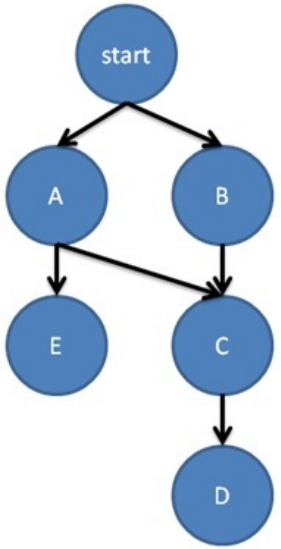
  - **graph** object – kolekcja **węzłów** i **krawędzi,** które są połączone poprzez porty

  Klasa **graph**, metoda **wait_for_all()**

  Grafy kierunkowe

- Graf przepływu zależności
- Graf przepływu wiadomości





Typy węzłów

Węzły funkcyjne – przetwarzają wiadomości i rozsyłają do potomków

- Cecha charakterystyczna – argument funkcyjny

```
template < typename Input, typename Output = continue_msg,
           typename Policy = queueing,
           typename A = cache_aligned_allocator<Input> >
class function_node;
// template< typename Body > function_node( graph &g, size_t concurrency, Body body )

template < typename Input, typename Output,
           typename Policy = queueing,
           typename Allocator=cache_aligned_allocator<Input> >
class multifunction_node;

template< typename Output, typename Policy = void >
class continue_node;

template< typename Output>
class source_node;
```

Węzły udostępniające właściwości

broadcast_node

- Rozsyła przychodzące wiadomości do wszystkich następników

limiter_node

- Użytkownik definiuje liczbę wiadomości rozsyłanych

join_node

- Posiada krotkę input_port, wspiera odbiór wiadomości różnego typu od wielu węzłów, zwraca krotkę, składającą się z danych z input_port

composite_node

- Pozwala na implementację wielu węzłów naraz, input_port jest krotką
- Definiujemy węzły, które interpretują InputTuple i OutputTuple

split_node

- Dzieli zadania pomiędzy węzły przypisane output_port

MUTEX | OPERACJE ATOMOWE | WYJĄTKI

Mutex’y w Intell TBB

- Mutex – kontroluje ile wątków wykonuje danych blok kodu jednocześnie
- Ochrona przed takimi problemami jak np. wyścig o dane.
- Cechy:

  - skalowalność

  - „sprawiedliwość”

  - blokowania rekursywne

  - yield and block

Rodzaje mutex’ów w Intel TBB

Mutex	Scalable	Fair	Recursive	Long Wait	Size
mutex	OS dependent	OS dependent	no	blocks	≥ 3 words
recursive_mutex	OS dependent	OS dependent	✓	blocks	≥ 3 words
spin_mutex	no	no	no	yields	1 byte
speculative_spin_mutex	HW dependent	no	no	yields	2 cache lines
queuing_mutex	✓	✓	no	yields	1 word
spin_rw_mutex	no	no	no	yields	1 word
speculative_spin_rw_mutex	HW dependent	no	no	yields	3 cache lines
queuing_rw_mutex	✓	✓	no	yields	1 word
null_mutex[6]	moot	✓	✓	never	empty
null_rw_mutex	moot	✓	✓	never	empty

- R/W mutex
- Potencjalne problemy – deadlock, convoyiny
- Rozwiązanie – atomic_operatrion

Wyjątki i przerywanie task'ów

Typowe obsłużenie wyjątku wewnątrz algorytmu TBB:

1. **Wyjątek został złapany**, wszystkie następujące po nim możliwe wyjątki zostają zignorowane.

2.**Algorytm zostaje anulowany** → wszystkie następne iteracje nie zostaną wykonane.

3.**Wszystkie wykonywane części algorytmu zostają zatrzymane.** Wyjątek zostaje obsłużony przez wątek, który wywołał algorytm.

TBB a procesory Intel

- CPU jest częścią hardware'u, która jest programowalna i kontrolowana przez software.
- Mikrokod – implementuje bardziej skomplikowane instrukcje procesora. Rozbija instrukcje CISC na instrukcje RISC
- Mikrokody Intel są szyfrowane i tajne

Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Źródła

- „Intel Threading Building Blocks Outfitting C++ for Multi-Core Processor Parallelism” James Reinders
- https://software.intel.com/en-us/tbb-documentation
- https://www.threadingbuildingblocks.org/
- Wykład z konferencji „CppCon 2015” autorstwa Pablo Halpern: “Work Stealing"