AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Intel® Threading Building Blocks

Patryk Chodur

Podstawowe informacje

- Jest to biblioteka szablonowa oparta o funkcjonalność wbudowaną w język C++
- Wersja komercyjna (Windows, Linux, macOS) oraz Open Source (kom. + FreeBSD, PowerPC, Solaris, Xbox 360 oraz QNX Neutrino) na licencji Apache v2.0
- Zadania zamiast wątków
- Wysokopoziomowość przy minimalizacji strat wydajności
- Działa z innymi bibliotekami do obsługi wielowątkowości (np. wątki POSIX czy Windows)

Task scheduler

- Silnik biblioteki odpowiedzialny za zarządzanie zadaniami
- Zaimplementowany w formie klasy

```
namespace tbb{
  task_scheduler_init (int max_threads=automatic,
  stack_size_type thread_stack_size=0 );
}
```

- Rozkłada zadania pod kątem największej efektywności
- Zadania nie muszą być wykonywane równolegle
- Task stealing
- Od wersji 2.2 jawne tworzenie obiektu task_scheduler_init nie jest wymagane
- Zgodnie z zaleceniami Intela najlepiej zostawić pusty konstruktor

przyklad1.cpp

tbb::blocked_range

- Obiekt ułatwiający iterację oraz dzielenie
- Konstruktory

```
namespace tbb {
  template <typename Value>
  class blocked_range
    blocked_range(Value begin, Value end, size_t
    grainsize=1);
    blocked_range(const blocked_range& r);
    blocked_range(blocked_range& r, split);
    blocked_range(blocked_range& r, proportional_split&
    proportion);
  };}
```

- Grainsize
- Split concept

przyklad2.cpp

Kontrola rozmiaru chunków

- Zależy od grainsize i użytego partitionera
- blocked_range dzieli się na mniejsze obszary rozmiaru minimalnie 1/2 grainsize
- auto_partitioner
- affinity_partitioner (optymalizacja cache)
- static_partitioner (dzieli na tyle ile jest rdzeni)
- simple_partitioner (gwarantuje, że 1/2G < grainsize < G)

Podstawowe algorytmy

tbb::paralell_for

- **Wykonuje zależną od argumentów ilość iteracji funkcji, funktora, bądź lambdy**

```
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]] );
```

- **Jeśli wykorzystujemy obiekty funkcyjne paralell_for tworzy wiele kopii obiektu**

przyklad4.cpp

tbb::paralell_reduce

- **Wykonuje zależną od argumentów ilość iteracji funkcji, funktora, bądź lambdy**

```
template<typename Range, typename Value, typename Func, typename
Reduction>
  Value parallel_reduce( const Range& range,
    const Value& identity, const Func& func,
    const Reduction& reduction,
    [, partitioner[, task_group_context& group]] );
template<typename Range, typename Body>
  void parallel_reduce( const Range& range, Body& body
  [, partitioner[, task_group_context& group]] );
```

- **Obiekt funkcyjny w drugim przypadku powinien posiadać splitting constructor**
- **Obiekty dzielone zależnie od potrzeby (operator() nie powinien kasować wyników dla wcześniejszych zakresów)**
- **paralell_deterministic_reduce**

przyklad5.cpp
przyklad6.cpp

tbb::parallel_pipeline

- **Wykonuje serię funkcji, każdą w określonym przez programistę trybie (parallel, serial_out_of_order, serial_in_order)**

```
template<typename Index, typename Func>
  void parallel_pipeline( size_t max_number_of_live_tokens,
  const filter_t<void,void>& filter_chain
  [, task_group_context& group] );
```

przyklad7.cpp

Filtry

- **Filtry można tworzyć za pomocą konstruktora, funkcji tbb::make_filter lub przez łączenie za pomocą operatora &**

```
template<typename T, typename U, typename Func>
class filter_t ( filter::mode mode, const Func& f );
template<typename T, typename U, typename Func>
  filter_t<T,U> make_filter( filter::mode mode, const Func& f );
template<typename T, typename V, typename U>
  filter_t<T,U> operator&( const filter_t<T,V>& left,
    const filter_t<V,U>& right );
```

- **filtry są łączone za pomocą operatora &**
- **Pierwszy parametr to typ argumentu funkcji, a drugi to typ zwracany przez funkcję w filtrze**
- **Typy zwracane przez funkcje w filtrach muszą się zgadzać z typami przyjmowanymi przez funkcje w kolejnych filtrach**
- **Pierwszy parametr pierwszego filtra oraz drugi ostatniego muszą być typu void**
- **Pierwsza funkcja przyjmuje jako argument referencję do typu tbb::flow_control, a gdy przetworzy ostatni element musi wywołać metodę stop()**

tbb::parallel_do

- **Uruchamia w sposób sekwencyjny równoległe zadania**
- **Nie potrzebuje znać ilości iteracji**

```
template<typename InputIterator, typename Body>
  void parallel_do( InputIterator first, InputIterator last,
                    Body body[, task_group_context& group] );
template<typename Container, typename Body>
  void parallel_do( Container c, Body body[, task_group_context&
  group] );
```

- **Wykonuje się od elementu wskazywanego przez first do elementu przed last [first,last)**
- **Dodatkowe zadania mogą zostać dodane za pomocą metody add opcjonalnego argumentu parallel_do_feeder**

przyklad8.cpp

Obsługa wyjątków

- Propagacja zależy od wsparcia dla std::exception_ptr
- W przypadku braku wsparcia przekazywane jest podsumowanie typu tbb::captured_exception
- Wyjątek jest przekazywany do nadrzędnego zadania, a wszystkie podrzędne zadania są przerywane
- Jeśli nie chcemy przerywać innych zadań muszą być zadeklarowane jako osobna grupa zadań

przyklad9.cpp

Kontenery

- Biblioteka udostępnia alternatywne kontenery które umożliwiają bezproblemową obsługę wielozadaniowości. Są to między innymi:
  - concurrent_vector
  - concurrent_queue
  - concurrent_priority_queue
  - concurrent_hash_map
  - concurrent_unordered_set
  - concurrent_unordered_map

tbb::concurrent_vector

- Przypomina w dużej mierze std::vector
- Nie posiada metody insert() ani erase()
- Tworzy wiele mniejszych tablic, w przeciwieństwie do std::vector, choć dzięki metodzie shrink_to_fit() można złączyć mniejsze wektory w jeden ciągły
- Nie wszystkie metody są bezpieczne przy pracy na wielu wątkach, w szczególności metoda clear()

przyklad10.cpp

tbb::concurrent_queue

- Zdejmowanie elementu jest realizowane za pomocą pojedynczej funkcji try_pop()
- Metoda push() gwarantuje poprawną kolejność
- concurrent_bounded_queue

przyklad11.cpp

Mutex

- Służy do blokowania dostępu do zasobów, których jednoczesna obsługa na wielu wątkach jest ryzykowna
- Dostępne w Intel TBB rodzaje to:

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

Atomic expressions

- Są szybsze i bezpieczniejsze od mutexów
- Mają limitowany zestaw instrukcji

`= x`	read the value of `x`
`x =`	write the value of `x`, and return it
`x.fetch_and_store(y)`	do `x=y` and return the old value of `x`
`x.fetch_and_add(y)`	do `x+=y` and return the old value of `x`
`x.compare_and_swap(y,z)`	if `x` equals `z`, then do `x=y`. In either case, return old value of `x`.

przyklad12.cpp

tbb::task_group

- Reprezentują grupy zadań, które mogą zostać wykonane równolegle
- Zadania można dodawać w trakcie wykonywania wcześniejszych
- task_group_status
- task_handle
- is_current_task_group_canceling

przyklad13.cpp

tbb::task

- Umożliwia tworzenie zadań samemu
- Zadania zawsze muszą być alokowane za pomocą przeładowanego operatora new
- Zadanie musi dziedziczyć po klasie tbb::task oraz posiadać metodę execute() zwracającą wskaźnik na tbb::task
- Główne zadanie wywołujące inne musi mieć argument tbb::task::allocate_root, a funkcje pochodne tbb::task::allocate_child
- trzeba ustawić ref_counter na ilość zadań tworzonych w trakcie jednego wywołania zadania na ilość zadań potomnych + 1 dla zadania bieżącego
- Po zaalokowaniu zadanie uruchamia się metodą spawn

przyklad14.cpp

Źródła

https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf

https://software.intel.com/en-us/tbb-user-guide

https://software.intel.com/en-us/tbb-reference-manual