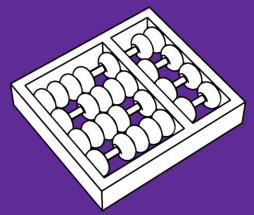
Pool de Threads, STL paralela e Sanitizer

Threads e Concorrência em C++

Hervé Yviquel
hyviquel@unicamp.br

Universidade Estadual de Campinas (Unicamp)
Instituto de Computação (IC)
Laboratório de Sistemas de Computação (LSC)

Programação em C++ • Out-Dez 2023



Plano

- Pool de Threads
- Task stealing
- STL Paralela
- Thread Sanitizer

Pool de Threads

Introdução aos Pool de Carros em Empresas

- Conceito de Pool de Carros
  - Em muitas empresas, em vez de fornecer um carro para cada funcionário, existe um 'pool de carros'
  - Isso significa que a empresa mantém um número limitado de veículos disponíveis para todos os funcionários

Introdução aos Pool de Carros em Empresas

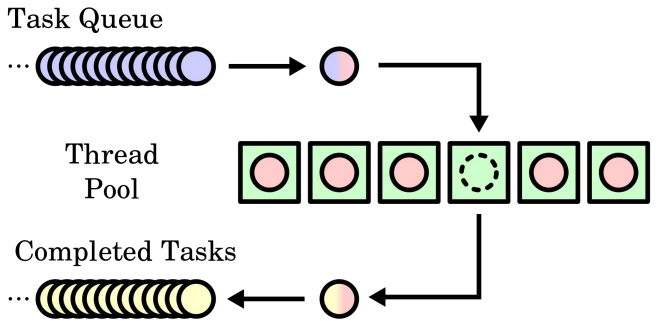
- Benefícios
  - Isso reduz custos, pois evita a necessidade de manter uma grande frota de veículos
  - Também é uma solução prática para viagens ocasionais de funcionários a clientes, fornecedores, feiras e conferências
- Necessidade Variável
  - Essa abordagem é eficiente porque a frequência das viagens varia muito entre os funcionários, podendo ocorrer mensalmente, anualmente ou até com menor frequência

Analogia com Pool de Threads

- Conceito de Pool de Threads
  - Um pool de threads em sistemas computacionais funciona de maneira similar
  - Em vez de ter uma thread separada para cada tarefa, um número limitado de threads é compartilhado entre várias tarefas
- Comparação
  - Assim como os carros no pool são usados conforme a necessidade, as threads são alocadas para tarefas conforme elas surgem
- Vantagens
  - Isso maximiza a eficiência, permitindo que o sistema lide com múltiplas tarefas simultaneamente sem o sobrecusto de criar e destruir threads constantemente

Funcionamento do Pool de Threads

- Submissão e Gerenciamento de Tarefas
  - As tarefas são enviadas ao pool de threads, onde são enfileiradas.
  - Quando uma thread fica disponível, ela seleciona uma tarefa da fila para executar
- Execução de Tarefas
  - Cada thread do pool processa uma tarefa por vez
  - Após a conclusão, a thread retorna ao pool para pegar a próxima tarefa
- Ciclo Contínuo
  - Este processo cria um ciclo contínuo de execução de tarefas, otimizando o uso dos recursos do sistema



Desafios no Design de Pool de Threads

- Questões de Design
  - Ao projetar um pool de threads, algumas questões-chave surgem
  - quantas threads utilizar, como alocar tarefas de forma eficiente e como gerenciar o tempo de espera das tarefas
- Balanceamento de Threads
  - É crucial encontrar um equilíbrio no número de threads
  - Demais threads podem sobrecarregar o sistema, enquanto poucas podem levar a uma utilização ineficiente dos recursos
- Estratégias de Alocação e Espera
  - Desenvolver um método eficaz para alocar tarefas e gerenciar tempos de espera é essencial para o desempenho otimizado do pool

Implementações de Pool de Threads

- Diversidade de Implementações
  - Existem várias maneiras de implementar um pool de threads, cada uma com suas particularidades e adequações a diferentes cenários
- Abordagens de Design
  - Algumas implementações focam na maximização do desempenho para sistemas de alta carga, enquanto outras podem ser mais simples e adequadas para tarefas leves
- Exemplos Práticos
  - Vamos explorar algumas dessas implementações

Um Pool de Threads Simples

Classes Úteis


```
template <typename T>
class threadsafe_queue {
 public:
  threadsafe_queue();
  threadsafe_queue(const threadsafe_queue&);
  threadsafe_queue& operator=(const threadsafe_queue&) = delete;
  void push(T new_value);
  bool try_pop(T& value);
  std::shared_ptr<T> try_pop();
  void wait_and_pop(T& value);
  std::shared_ptr<T> wait_and_pop();
  bool empty() const;
};

class join_threads {
  std::vector<std::thread>& threads;

 public:
  explicit join_threads(std::vector<std::thread>& threads_)
      : threads(threads_) {}
  ~join_threads() {
    for(unsigned long i = 0; i < threads.size(); ++i) {
      if(threads[i].joinable()) threads[i].join();
    }
  }
};
```

Um Pool de Threads Simples


```
class thread_pool {
  std::atomic_bool done;
  threadsafe_queue<std::function<void()>> work_queue;
  std::vector<std::thread> threads;
  join_threads joiner;
  void worker_thread() {
    while(!done) {
      std::function<void()> task;
      if(work_queue.try_pop(task)) {
        task();
      } else {
        std::this_thread::yield();
      }
    }
  }
```

```
public:
  thread_pool() : done(false), joiner(threads) {
    unsigned const thread_count = std::thread::hardware_concurrency();
    try {
      for(unsigned i = 0; i < thread_count; ++i) {
        threads.push_back(
            std::thread(&thread_pool::worker_thread, this));
      }
    } catch(...) {
      done = true;
      throw;
    }
  }
  ~thread_pool() { done = true; }
  template <typename FunctionType>
  void submit(FunctionType f) {
    work_queue.push(std::function<void()>(f));
  }
};
```

Um Pool de Threads Simples

- Definição e funcionamento
  - Pool de threads com número de threads igual ao suporte de hardware (`std::thread::hardware_concurrency()`)
  - Tarefas adicionadas à fila e executadas por threads disponíveis; sem espera ativa pela conclusão da tarefa
- Implementação do Pool de Threads
  - Usa vetor de threads e fila segura para gerenciar tarefas
  - Função `submit()` para adicionar tarefas encapsuladas em `std::function<void()>` à fila
- Gerenciamento de Threads e Exceções
  - Threads iniciadas no construtor; tratamento de exceções para limpeza segura
  - Importância da ordem de declaração dos membros para destruição segura

Um Pool de Threads Simples

- A Função `worker_thread`
  - Loop de execução de tarefas da fila; pausa se a fila estiver vazia.
  - Aguarda a flag 'done' para terminar o processamento.
- Limitações
  - Adequado para tarefas independentes
  - Pode haver problemas em casos complexos como deadlock
- Alternativas
  - Uso de `std::async` para casos mais simples
  - Exploração de implementações mais avançadas em situações complexas

Mecanismo de Espera de Tarefa

Evolução na Gestão de Tarefas em Pool de Threads

- Em exemplos anteriores com threads explícitas, a thread mestre aguardava a conclusão das threads recém-criadas para finalizar a tarefa geral
- Com pools de threads, é necessário esperar pela conclusão das tarefas submetidas ao pool, não das próprias threads
- Processo semelhante aos exemplos baseados em std::async aguardando pelos futures

Evolução na Gestão de Tarefas em Pool de Threads

- Complexidade Adicional
  - No pool de threads simples, a espera pelas tarefas requer o uso manual de variáveis de condição e futures, adicionando complexidade ao código
- Solução Proposta
  - Mover essa complexidade para dentro do pool de threads, permitindo esperar diretamente pelas tarefas com a função submit(), retornando um manipulador de tarefa
- Integração com Futures
  - Uso de std::packaged_task<> para facilitar a transferência de resultados e espera pelas tarefas, requerendo uma nova abordagem para armazenar tarefas na fila devido à não copiabilidade dos std::packaged_task<>

Implementação da Espera


```
class function_wrapper {
  struct impl_base {
    virtual void call() = 0;
    virtual ~impl_base() {}
  };
  std::unique_ptr<impl_base> impl;
  template <typename F>
  struct impl_type : impl_base {
    F f;
    impl_type(F&& f_) : f(std::move(f_)) {}
    void call() { f(); }
  };

 public:
  template <typename F>
  function_wrapper(F&& f) : impl(new impl_type<F>(std::move(f))) {}
  void operator()() { impl->call(); }
  function_wrapper() = default;
  function_wrapper(function_wrapper&& other)
      : impl(std::move(other.impl)) {}
  function_wrapper& operator=(function_wrapper&& other) {
    impl = std::move(other.impl);
    return *this;
  }
  function_wrapper(const function_wrapper&) = delete;
  function_wrapper(function_wrapper&) = delete;
  function_wrapper& operator=(const function_wrapper&) = delete;
};
```

Implementação da Espera


```
class thread_pool {
  std::atomic_bool done;
  threadsafe_queue<function_wrapper> work_queue;

  void worker_thread() {
    while(!done) {
      function_wrapper task;
      if(work_queue.try_pop(task)) {
        task();
      } else {
        std::this_thread::yield();
      }
    }
  }

 public:
  template <typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type> submit(
      FunctionType f) {
    typedef typename std::result_of<FunctionType()>::type result_type;
    std::packaged_task<result_type()> task(std::move(f));
    std::future<result_type> res(task.get_future());
    work_queue.push(std::move(task));
    return res;
  }
  // rest as before
};
```

std::accumulate

std::accumulate

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );                    (1)  (until C++20)
template< class InputIt, class T >
constexpr T accumulate( InputIt first, InputIt last, T init );          (1)  (since C++20)

template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );                                     (2)  (until C++20)
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );                           (2)  (since C++20)
```

Computes the sum of the given value `init` and the elements in the range [`first` , `last`).

1) Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with `acc = acc + *i` (until C++20) `acc = std::move(acc) + *i` (since C++20) for every iterator `i` in the range [`first` , `last`) in order.

2) Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with `acc = op(acc, *i)` (until C++20) `acc = op(std::move(acc), *i)` (since C++20) for every iterator `i` in the range [`first` , `last`) in order.

If `op` invalidates any iterators (including the end iterators) or modifies any elements of the range involved, the behavior is undefined.

Implementação de Accumulate Paralelo


```
template <typename Iterator, typename T>
struct accumulate_block {
  void operator()(Iterator first, Iterator last, T& result) {
    result = std::accumulate(first, last, result);
  }
};

template <typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init) {
  unsigned long const length = std::distance(first, last);
  if(!length) return init;
  unsigned long const block_size = 25;
  unsigned long const num_blocks =
      (length + block_size - 1) / block_size;
  std::vector<std::future<T> > futures(num_blocks - 1);
  thread_pool pool;
  Iterator block_start = first;
  for(unsigned long i = 0; i < (num_blocks - 1); ++i) {
    Iterator block_end = block_start;
    std::advance(block_end, block_size);
    futures[i] = pool.submit([=] {
      accumulate_block<Iterator, T>()(block_start, block_end);
    });
    block_start = block_end;
  }
  T last_result = accumulate_block<Iterator, T>()(block_start, last);
  T result = init;
  for(unsigned long i = 0; i < (num_blocks - 1); ++i) {
    result += futures[i].get();
  }
  result += last_result;
  return result;
}
```

Accumulate Paralelo

- Abordagem de Blocos vs. Threads
  - Mudança de foco do número de threads para o número de blocos de tarefas (num_blocks), maximizando a escalabilidade do pool de threads
- Dimensionamento de Blocos
  - Importância de dividir o trabalho em blocos pequenos o suficiente para valer a pena serem processados concorrentemente, ajustando-se ao número de threads disponíveis.
  - Evitar blocos muito pequenos devido ao overhead de submeter tarefas ao pool, executar e gerenciar retornos com std::future<>, o que pode tornar o pool mais lento do que a execução em uma única thread.

Accumulate Paralelo

- Facilidade de Uso:
  - Não é necessário gerenciar embalagem de tarefas, obtenção de futures ou armazenamento de objetos std::thread – o pool de threads cuida disso através da função submit().
- Segurança de Exceções
  - O pool de threads lida com a segurança de exceções, propagando exceções através dos futures e abandonando tarefas não concluídas em caso de falhas, garantindo a conclusão das threads do pool.
- Limitações:
  - Embora eficaz para tarefas independentes, esta abordagem pode não ser ideal para tarefas que dependem de outras tarefas no mesmo pool de threads

Dependencia entre Tarefas

Quicksort Paralelo

- Exemplo do Quicksort
  - Divisão dos dados em torno de um pivô, com conjuntos de dados sendo ordenados recursivamente e combinados
- Desafío de Deadlock
  - Risco de deadlock ao esperar pela ordenação de blocos, resolvido por threads processando blocos da pilha enquanto esperam
- Limitações do Pool de Threads Simples
  - Problemas potenciais com todos os threads aguardando por tarefas não agendadas devido à limitação de threads
- Solução Proposta
  - Modificação do pool de threads para processar automaticamente tarefas pendentes, adicionando uma função para executar tarefas da fila

Quicksort Paralelo


```
template <typename T>
struct sorter {
  thread_pool pool;
  std::list<T> do_sort(std::list<T>& chunk_data) {
    if(chunk_data.empty()) {
      return chunk_data;
    }
    std::list<T> result;
    result.splice(result.begin(), chunk_data, chunk_data.begin());
    T const& partition_val = *result.begin();
    typename std::list<T>::iterator divide_point = std::partition(
        chunk_data.begin(), chunk_data.end(),
        [&](T const& val) { return val < partition_val; });
    std::list<T> new_lower_chunk;
    new_lower_chunk.splice(new_lower_chunk.end(), chunk_data,
                           chunk_data.begin(), divide_point);
    std::future<std::list<T> > new_lower = pool.submit(std::bind(
        &sorter::do_sort, this, std::move(new_lower_chunk)));
    std::list<T> new_higher(do_sort(chunk_data));
    result.splice(result.end(), new_higher);
    while(new_lower.wait_for(std::chrono::seconds(0)) ==
          std::future_status::timeout) {
      pool.run_pending_task();
    }
    result.splice(result.begin(), new_lower.get());
    return result;
  }
};
template <typename T>
std::list<T> parallel_quick_sort(std::list<T> input) {
  if(input.empty()) {
    return input;
  }
  sorter<T> s;
  return s.do_sort(input);
}
```

```
void thread_pool::run_pending_task() {
  function_wrapper task;
  if(work_queue.try_pop(task)) {
    task();
  } else {
    std::this_thread::yield();
  }
}
```

Quicksort Paralelo

- Implementação e Simplificação
  - A lógica de gerenciamento de threads é movida para o pool de threads, simplificando a implementação do Quicksort
- Gerenciamento de Tarefas
  - Submissão de tarefas ao pool e execução de tarefas pendentes enquanto espera
- Considerações de Desempenho
  - Necessidade de abordar o problema de acesso múltiplo à mesma fila para evitar impacto negativo no desempenho

Reduzindo a Contenção na Fila de Tarefa

Contenção

- Problema de Contenção
  - Com o aumento do número de processadores, cresce a disputa pela única fila de trabalho compartilhada, afetando o desempenho.
- Cache Ping-Pong
  - Mesmo com filas sem bloqueio, a contínua troca de cache entre threads é um problema significativo

Filas Privadas

- Filas de Trabalho Separadas por Thread
  - Uso de uma variável thread_local para que cada thread tenha sua própria fila de trabalho, além da fila global
- Funcionamento
  - Threads postam novos itens em suas filas locais e acessam a fila global apenas se não houver trabalho em suas filas individuais
- Dinâmica de Submissão
  - submit() coloca tarefas na fila local se o thread pertencer ao pool;
  - caso contrário, usa a fila do pool
- Execução de Tarefa
  - run_pending_task() verifica primeiro a fila local e, se vazia, recorre à fila do pool

```
class thread_pool {
  std::atomic_bool done;
  threadsafe_queue<function_wrapper> pool_work_queue;
  typedef std::queue<function_wrapper> local_queue_type;
  static thread_local std::unique_ptr<local_queue_type>
    local_work_queue;

  void worker_thread() {
    local_work_queue.reset(new local_queue_type);
    while(!done) {
      run_pending_task();
    }
  }

 public:
  template <typename FunctionType>
  std::future<typename std::result_of<FunctionType()>::type> submit(
    FunctionType f) {
    typedef typename std::result_of<FunctionType()>::type result_type;
    std::packaged_task<result_type()> task(f);
    std::future<result_type> res(task.get_future());
    if(local_work_queue) {
      local_work_queue->push(std::move(task));
    } else {
      pool_work_queue.push(std::move(task));
    }
    return res;
  }

  void run_pending_task() {
    function_wrapper task;
    if(local_work_queue && !local_work_queue->empty()) {
      task = std::move(local_work_queue->front());
      local_work_queue->pop();
      task();
    } else if(pool_work_queue.try_pop(task)) {
      task();
    } else {
      std::this_thread::yield();
    }
  }
};
```

- Desafio da Distribuição Desigual de Trabalho
  - Pode resultar em um thread com muitas tarefas enquanto outros ficam inativos, especialmente em tarefas desbalanceadas como no Quicksort
- Solução para Desbalanceamento
  - Permitir que threads 'roubem' tarefas das filas uns dos outros se suas próprias filas e a fila global estiverem vazias
  - Esta estratégia de balanceamento é chamada de *task stealing* ou *work stealing*

Aprimorando o Pool de Threads com Roubo de Tarefas

- Contexto do Roubo de Tarefas
  - Para maximizar a eficiência, threads sem tarefas devem poder 'roubar' tarefas de outras threads com filas cheias
- Requisitos de Implementação
  - Registro de Filas: Cada thread deve registrar sua fila no pool de threads ou receber uma do pool
  - Sincronização de Dados: Proteção dos dados na fila de trabalho para manter a integridade e sincronização

Aprimorando o Pool de Threads com Roubo de Tarefas

- Uso de mutex para proteger a fila de trabalho, assumindo que o roubo de tarefas é um evento raro
- Estrutura da Fila de Tarefas
  - Wrapper em torno de std::deque: Protege acessos com um bloqueio de mutex.
  - Operações de Fila: push() e try_pop() no início da fila; try_steal() no final para minimizar a contenção.
- Uso no Pool de Threads:
  - Filas de Roubo de Trabalho: Cada thread tem uma work_stealing_queue ao invés de uma fila padrão.
  - Acesso e Roubo de Tarefas: Threads tentam pegar tarefas da própria fila, da fila global ou de outras threads.
  - Distribuição de Tarefas: Iteração entre as filas de todas as threads para roubar tarefas, evitando sobrecarga em uma única thread.

Fila


```
class work_stealing_queue {
 private:
  typedef function_wrapper data_type;
  std::deque<data_type> the_queue;
  mutable std::mutex the_mutex;

 public:
  work_stealing_queue() {}
  work_stealing_queue(const work_stealing_queue& other) = delete;
  work_stealing_queue& operator=(const work_stealing_queue& other) =
      delete;
  void push(data_type data) {
    std::lock_guard<std::mutex> lock(the_mutex);
    the_queue.push_front(std::move(data));
  }
  bool empty() const {
    std::lock_guard<std::mutex> lock(the_mutex);
    return the_queue.empty();
  }
  bool try_pop(data_type& res) {
    std::lock_guard<std::mutex> lock(the_mutex);
    if(the_queue.empty()) {
      return false;
    }
    res = std::move(the_queue.front());
    the_queue.pop_front();
    return true;
  }
  bool try_steal(data_type& res) {
    std::lock_guard<std::mutex> lock(the_mutex);
    if(the_queue.empty()) {
      return false;
    }
    res = std::move(the_queue.back());
    the_queue.pop_back();
    return true;
  }
};
```

Pool de Threads


```
class thread_pool {
  typedef function_wrapper task_type;
  std::atomic_bool done;
  threadsafe_queue<task_type> pool_work_queue;
  std::vector<std::unique_ptr<work_stealing_queue> > queues;
  std::vector<std::thread> threads;
  join_threads joiner;
  static thread_local work_stealing_queue* local_work_queue;
  static thread_local unsigned my_index;

  void worker_thread(unsigned my_index_) {
    my_index = my_index_;
    local_work_queue = queues[my_index].get();
    while(!done) {
      run_pending_task();
    }
  }

  bool pop_task_from_local_queue(task_type& task) {
    return local_work_queue && local_work_queue->try_pop(task);
  }

  bool pop_task_from_pool_queue(task_type& task) {
    return pool_work_queue.try_pop(task);
  }

  bool pop_task_from_other_thread_queue(task_type& task) {
    for(unsigned i = 0; i < queues.size(); ++i) {
      unsigned const index = (my_index + i + 1) % queues.size();
      if(queues[index]->try_steal(task)) {
        return true;
      }
    }
    return false;
  }
```

```
public:
 thread_pool() : done(false), joiner(threads) {
   unsigned const thread_count = std::thread::hardware_concurrency();
   try {
     for(unsigned i = 0; i < thread_count; ++i) {
       queues.push_back(std::unique_ptr<work_stealing_queue>(
           new work_stealing_queue));
     }
     for(unsigned i = 0; i < thread_count; ++i) {
       threads.push_back(
           std::thread(&thread_pool::worker_thread, this, i));
     }
   } catch(...) {
     done = true;
     throw;
   }
 }

 ~thread_pool() { done = true; }
 template <typename FunctionType>
 std::future<typename std::result_of<FunctionType()>::type> submit(
     FunctionType f) {
   typedef typename std::result_of<FunctionType()>::type result_type;
   std::packaged_task<result_type()> task(f);
   std::future<result_type> res(task.get_future());
   if(local_work_queue) {
     local_work_queue->push(std::move(task));
   } else {
     pool_work_queue.push(std::move(task));
   }
   return res;
 }

 void run_pending_task() {
   task_type task;
   if(pop_task_from_local_queue(task) ||
      pop_task_from_pool_queue(task) ||
      pop_task_from_other_thread_queue(task)) {
     task();
   } else {
     std::this_thread::yield();
   }
 }
};
```

Melhorias Possiveis

- Pool Versátil
  - O pool aprimorado está adequado para muitos usos, mas ainda há espaço para otimizações específicas
- Redimensionamento Dinâmico e Interrupção de Threads
  - Aspectos não explorados que podem otimizar ainda mais o uso de CPU e a gestão de threads.

STL Paralela

Sortear com STL


```
std::vector<int> data = { 8, 9, 1, 4 };

std::sort(std::begin(data), std::end(data));

if (std::is_sorted(data)) {
    Std::cout << " Data is sorted!" << std::endl;
}
```

Normal sequential sort algorithm

```
std::vector<int> data = { 8, 9, 1, 4 };

std::sort(std::execution_policy::par,
          std::begin(data), std::end(data));

if (std::is_sorted(data)) {
    Std::cout << " Data is sorted!" << std::endl;
}
```

Extra parameter to STL algorithms enable parallelism

Política de Execução: Classes de política padrão

Type	Vectorization	Parallelization
Sequenced	X	X
Unsequenced	V	X
Parallel	X	V
Parallel & unsequenced	V	V

Definidas no namespace `execution`

- Política sequencial
  - Nunca executa em paralelo, execução sequencial ordenada
  - `constexpr sequenced_policy sequenced;`
- Política paralela
  - Pode usar a thread do chamador, mas pode abranger outras (std::thread)
  - As invocações não se entrelaçam em uma única thread
  - `constexpr sequenced_policy par;`
- Paralelo não sequencial
  - Pode usar a thread do chamador ou outras (por exemplo, std::thread)
  - Múltiplas invocações podem ser entrelaçadas em uma única thread
  - `constexpr sequenced_policy par_unseq;`

Usar execution policies


```
using std::execution_policy;


// May execute in parallel
std::sort(par, std::begin(data), std::end(data))
// May be parallelized and vectorized
std::sort(std::par_unseq, std::begin(data), std::end(data));
// Will not be parallelized/vectorized
std::sort(std::sequenced, std::begin(data), std::end(data));
// Vendor-specific policy, read their documentation!
std::sort(custom_vendor_policy, std::begin(data), std::end(data));
```

Propagando a política para o usuário final


```
using std::execution_policy;

template<typename Policy, typename Iterator>
void library_function(Policy p, Iterator begin,
                      Iterator end) {
   std::sort(p, begin, end);
   std::for_each(p, begin, end,
                 [&](Iterator::value_type e&) { e ++;}) ;
   std::for_each(std::sequenced, begin, end,
                 non_parallel_operation) ;
}
```

Parallel Overloads Disponiveis

Table 1 — Table of parallel algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

For Each


```
template<class ExecutionPolicy, class InputIterator, class Function>
void for_each(ExecutionPolicy && exec,InputIterator first,
InputIterator last, Function f);

template<class ExecutionPolicy, class InputIterator, class Size, class
Function>
InputIterator for_each_n(ExecutionPolicy && exec,
                         InputIterator first, Size n,
                         Function f) ;

template<class InputIterator, class Size, class Function>
InputIterator for_each_n(InputIterator first, Size n, Function f);
```

- for_each
  - Aplica a função f aos elementos no intervalo [primeiro, último)
- for_each_n
  - Aplica a função f aos elementos em [primeiro, primeiro + n)

Reduce

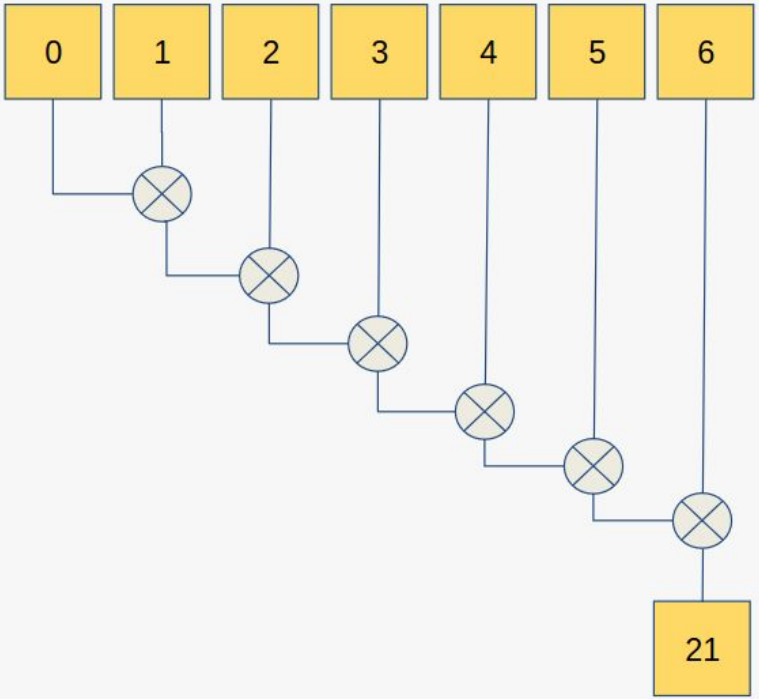

```
template < class InputIterator >
typename iterator_traits < InputIterator >:: value_type
reduce ( InputIterator first , InputIterator last ) ;

template < class InputIterator , class T >
T reduce ( InputIterator first , InputIterator last , T init ) ;

template < class InputIterator , class T , class BinaryOperation >
T reduce ( InputIterator first , InputIterator last , T init ,
BinaryOperation binary_op ) ;
```

- Implementa uma operação de redução
  - a ordem do binary_op não é relevante
- O equivalente sequencial é `accumulate`

Accumulate

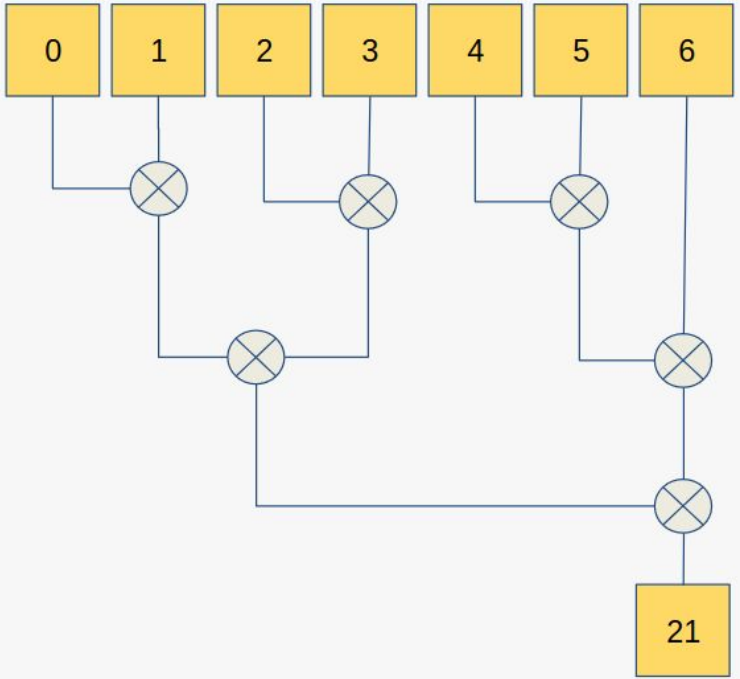



```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::accumulate(std::begin(v1), nElems, 1);
```

Only one core is used for the different additions.

Reduce



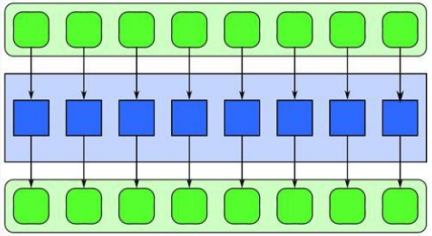

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::reduce(std::execution_policy::par,
                    std::begin(v1), nElems, 1);
```

If operation is commutative and associative, can be run in parallel.
Reduction uses all cores!

Transform

Transform



```
std::transform(std::execution::par,
               v1.begin(), v1.end(),
               v2.begin(), output.begin(),
               [=](int val1, int val2)
                  { return val1 + val2 + 1; });
```

- transform (aka *map*)
  - Aplica uma função a um intervalo de entrada e armazena o resultado em um intervalo de saída
  - A operação é realizada fora de ordem.

Transform reduce

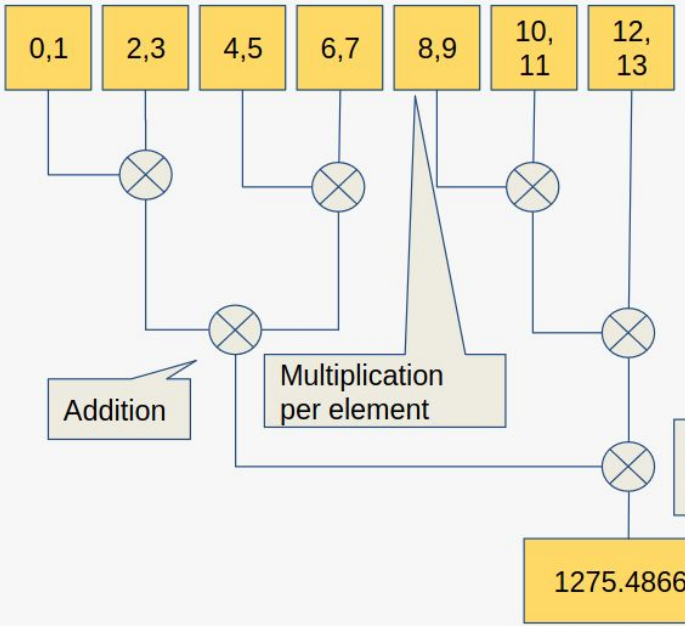

```
template<class ExecutionPolicy,
         class ForwardIt1, class ForwardIt2, class T>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, T init);
```
(4) (since C++17)

```
template<class ExecutionPolicy,
         class ForwardIt1, class ForwardIt2, class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2,
                   T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2);
```
(5) (since C++17)

```
template<class ExecutionPolicy,
         class ForwardIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt first, ForwardIt last,
                   T init, BinaryOp binary_op, UnaryOp unary_op);
```
(6) (since C++17)

- transform_reduce
  - aplica uma função a um intervalo de entrada e depois aplica a operação binária para reduzir os valores

Exemplo de Transform Reduce




```
struct elem {
float a;
float b;
} elem;
```

```
auto transformReduce = [&]() {
   float pi = PI;
   float res = std::experimental::parallel::transform_reduce(
       snp, std::begin(v), std::end(v), [=](elem x) { return x.a * x.b * pi; },
       0,                                                  // map multiplication
       [=](float a, float b) { return a + b; });  // reduce addition
```

Apply to each element of v

Neutral element of reduction

Reduction function: addition

Exemplo de for_each paralelo


```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::begin(v1), nElems, 1);

std::for_each(std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

Traditional for each uses only one core, rest of the die is unutilized!

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::execution_policy::par,
            std::begin(v1), nElems, 1);

std::for_each(std::execution_policy::par,
              std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

Workload is distributed across cores!

Benchmarks

The following Table shows algorithms which show the highest level of parallel speedup when run on a 14-core i7-12700 laptop processor, processing an array of 100 Million 32-bit integers:

Algorithm	seq	unseq	par	par_unseq	Parallel Speedup
sort(std::	11	n/s	73	74	**6.7**
sort(dpl::	11	11	109	107	**9.9**
stable_sort(std::	12	n/s	44	44	**3.7**
stable_sort(dpl::	10	12	115	117	**9.8**
merge(std::	300	n/s	296	312	**1.0**
merge(dpl::	311	311	2963	2941	**9.5**
all_of(std::	2381	n/s	9551	9930	**4.2**
all_of(dpl::	2894	3663	11249	13055	**4.5**

Windows 11
Visual Studio 2022
Intel C++ compiler and
Intel OneAPI 2023.1

Benchmarks de mais algoritmos

Algorithm	seq	unseq	par	par_unseq	Parallel Speedup
adjacent_difference(std::	2415	n/s	3905	4095	**1.7**
adjacent_find(std::	3126	n/s	11419	11642	**3.7**
adjacent_find(dpl::	3122	3211	11739	12052	**3.9**
all_of(std::	2381	n/s	9551	9930	**4.2**
all_of(dpl::	2894	3663	11249	13055	**4.5**
any_of(std::	2961	n/s	10753	10186	**3.6**
any_of(dpl::	2950	4090	11152	11700	**4.0**
copy(std::	3395	n/s	3317	3080	**1.0**
copy(dpl::	2865	2518	3250	4181	**1.5**
count(std::	3758	n/s	7541	8403	**2.2**
count(dpl::	3284	3058	6329	8850	**2.7**
equal(std::	3003	n/s	6382	6993	**2.3**
equal(dpl::	2262	2930	7002	6892	**3.1**
fill(std::	3030	n/s	3125	3056	**1.0**
fill(dpl::	2795	2889	4268	6459	**2.2**
max_element(std::	3922	n/s	3614	3759	**1.0**
max_element(dpl::	2545	2529	10471	10830	**4.3**
merge(std::	300	n/s	296	312	**1.0**
merge(dpl::	311	311	2963	2941	**9.5**
inplace_merge(std::	274	n/s	272	272	**1.0**
inplace_merge(dpl::	277	277	1420	1351	**5.1**
sort(std::	11	n/s	73	74	**6.7**
sort(dpl::	11	11	109	107	**9.9**
stable_sort(std::	12	n/s	44	44	**3.7**
stable_sort(dpl::	10	12	115	117	**9.8**

Mais Nucleos, Mais Desempenho

Some of the standard C++ algorithms scale better than others as the number of processor cores increases. The following Table shows performance of single-core serial and multi-core parallel algorithms on a 48-core Intel Xeon 8275CL AWS node when processing an array of 100 Million 32-bit integers:

Algorithm	seq	unseq	par	par_unseq	Parallel Speedup
adjacent_difference(std::	1531	n/s	8865	8836	**5.8**
copy(std::	1279	n/s	1280	1277	**1.0**
copy(dpl::	1229	1229	7143	7143	**5.8**
count(std::	2355	n/s	29923	30054	**12.8**
count(dpl::	2320	2315	20368	19876	**8.8**
fill(std::	2670	n/s	2639	2625	**1.0**
fill(dpl::	2789	2804	2918	9920	**3.5**
max_element(dpl::	1706	1713	20325	20964	**12.2**
merge(std::	130	n/s	131	130	**1.0**
merge(dpl::	128	128	2687	2819	**22.0**
sort(std::	10	n/s	73	81	**8.1**
sort(dpl::	9	9	165	162	**18.3**
stable_sort(std::	10	n/s	63	63	**6.3**
stable_sort(dpl::	10	10	161	163	**16.3**

STL paralelo com GPU

Algorithm	seq	un-seq	par	par_unseq	GPU Speedup
max_element(std::	1600	1613	1620	1581	1.0
adjacent_differ-ence(std::	2052		2062	996	0.5
adjacent_find(std::	2963		2947	37	
all_of(std::	3652		3752	34	
any_of(std::	3652		3584	37	
count(std::	2999		2987	1627	
equal(std::	3839		3716	37	
copy(std::	4421		4525	1529	
merge(std::	201		197	387	
inplace_merge(std::	183		181		
sort(std::	15	15	15	Segmentation Fault	
stable_sort(std::	17	17	17	Segmentation Fault	

After following NVidia's instructions on the above site, performance on Windows 11 WSL (Ubuntu) executing GPU accelerated C++ Standard algorithms is slower than single-core CPU algorithms on an Alienware Dell laptop with a GeFore RTX 3060 lap-top GPU.

Depuração

ThreadSanitizer


```
$ cat simple_race.cc
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

```
$ clang++ simple_race.cc -fsanitize=thread -fPIE -pie -g
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (exe+0x000000006e66)

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (exe+0x000000006ed6)

  Thread T1 (tid=26328, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:19 (exe+0x000000006f39)

  Thread T2 (tid=26329, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:20 (exe+0x000000006f63)
==================
ThreadSanitizer: reported 1 warnings
```

Comparação

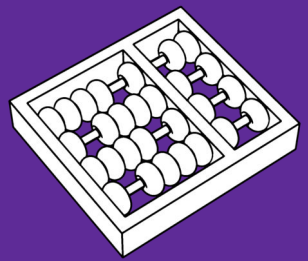
Comparação de Bibliotecas de Threads

Feature	Java	POSIX C	Boost threads	C++11
Starting threads	`java.lang.thread` class	`pthread_t` type and associated API functions: `pthread_create()`, `pthread_detach()`, and `pthread_join()`	`boost::thread` class and member functions	`std::thread` class and member functions
Mutual exclusion	`synchronized` blocks	`pthread_mutex_t` type and associated API functions: `pthread_mutex_lock()`, `pthread_mutex_unlock()`, etc.	`boost::mutex` class and member functions, `boost::lock_guard<>` and `boost::unique_lock<>` templates	`std::mutex` class and member functions, `std::lock_guard<>` and `std::unique_lock<>` templates
Monitors/ waits for a predicate	`wait()` and `notify()` methods of the `java.lang.Object` class, used inside `synchronized` blocks	`pthread_cond_t` type and associated API functions: `pthread_cond_wait()`, `pthread_cond_timed_wait()`, etc.	`boost::condition_variable` and `boost::condition_variable_any` classes and member functions	`std::condition_variable` and `std::condition_variable_any` classes and member functions
Atomic operations and concurrency-aware memory model	`volatile` variables, the types in the `java.util.concurrent.atomic` package	N/A	N/A	`std::atomic_xxx` types, `std::atomic<>` class template, `std::atomic_thread_fence()` function
Thread-safe containers	The containers in the `java.util.concurrent` package	N/A	N/A	N/A
Futures	`java.util.concurrent.future` interface and associated classes	N/A	`boost::unique_future<>` and `boost::shared_future<>` class templates	`std::future<>`, `std::shared_future<>` and `std::atomic_future<>` class templates
Thread pools	`java.util.concurrent.ThreadPoolExecutor` class	N/A	N/A	N/A
Thread interruption	`interrupt()` method of `java.lang.Thread`	`pthread_cancel()`	`interrupt()` member function of `boost::thread` class	N/A

Resumo

- Pool de Threads
  - Esperar uma tarefa
  - Dependencia entre tarefas
  - Task stealing
- STL Paralela
  - For each
  - Reduce
- Thread Sanitizer

Obrigado!
Merci!



UNICAMP

Pallete

Bubbles

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

DRACULA

Tables

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Emojis