Intel Thread Building Blocks (TBB)

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram fortemente baseados no material elaborado por Paul Guermonprez (Intel) e Jan Verschelde (University of Illinois at Chicago)

Introdução

Já apresentamos:

- **MPI**: para programação paralela de computadores com memória distribuída
- **Pthreads**: para programação paralela de computadores com memória compartilhada em ambientes POSIX (Linux, FreeBSD, Mac OS X, ...)
- **OpenMP**: para programação paralela de computadores com memória compartilhada

Intel Thread Building Blocks (TBB) é mais uma ferramenta na mesma linha:

- Programação paralela de processadores multicore em geral (pelo menos é a propaganda...)
- Suporte comercial para: Linux, Windows, Mac OS X
- Versão open source disponível para: FreeBSD, IA Solaris, XBox 360

Intel Threading Building Blocks

UFABC

- Intel TBB é uma biblioteca que, na mesma linha de OpenMP, é voltada ao desenvolvimento de aplicações paralelas em máquinas multicore **para não especialistas**.
- O nível de programação é mais alto do que Pthreads puro
- Não exige suporte do compilador
  - Baseado em generics (templates em C++) e a partir de C++11 também utilizando funções lambda
- Talvez a principal diferença entre os concorrentes é a de que o paralelismo é especificado logicamente, sem necessariamente pensarmos em termos de threads e é, portanto, mais voltado ao paralelismo de dados
- Fonte pode ser baixado em: `http://threadingbuildingblocks.org/`

Programação baseada em tarefas

UFABC

- Em lugar de dividir a carga de trabalho entre threads (como por exemplo OpenMP), a carga é dividida em tarefas (*tasks*).
- Tarefas são muito leves
  - Iniciar e terminar uma tarefa leva em média 18 vezes menos tempo do que fazer o mesmo com um thread
  - Por outro lado as tarefas precisam ser escalonadas pelo ambiente de execução (e não pelo SO) já que não tem uma entidade única correspondente no escalonador do SO

Diferenças entre escalonadores

UFABC

- O escalonador do TBB utiliza roubo de trabalho (*work-stealing*) para efetuar o balanceamento de carga
- Quando escalonamos threads à processadores, podemos fazer a seguinte distinção:
  - **Compartilhamento de trabalho** - O escalonador tenta migrar threads entre os processadores de modo a equilibrar a carga entre eles
  - **Roubo de trabalho** - Processadores pouco utilizados roubam as tarefas diretamente dos outros processadores

Limitações

UFABC

TBB não é apropriado para

- Computações *I/O bound*
- Processamento *real time*

Isso se deve ao fato do escalonador do TBB não ser justo (*unfair*). Em alguns casos essa característica permite algumas otimizações de desempenho que não seriam possíveis de outra maneira.

Usando o TBB

Programa de exemplo

UFABC

```
1  #include "tbb/tbb.h"
2  #include <cstdio>
3
4  using namespace tbb;
5
6  class say_hello {
7      const char* id;
8      public:
9          say_hello(const char* s) : id(s) {
10         }
11         void operator( ) ( ) const {
12             printf("hello from task %s\n",id);
13         }
14 };
```

A função `main`

UFABC

```
1  int main( ) {
2      task_group tg;
3      // spawn 1st task and return
4      tg.run(say_hello("1"));
5      // spawn 2nd task and return
6      tg.run(say_hello("2"));
7      tg.wait( ); // wait for tasks to complete
8  }
```

- O método `run` cria e executa a tarefa imediatamente
- Ele é não bloqueante, ou seja, o controle da execução retorna imediatamente
- Para esperar as tarefas utilizamos o método 'wait'. Equivalente ao `join` em Pthreads.
- Note o uso de um grupo de tarefas

Compilando

UFABC

- Baixe o TBB da página oficial.
- Se estiver utilizando um Makefile, inclua as seguintes linhas:

```
TBB_ROOT=/caminho/para/o/TBB

hello_task_group:
    g++ -I$(TBB_ROOT)/include \
        -L$(TBB_ROOT)/lib \
        hello_task_group.cpp \
        -o hello_task_group -ltbb
```

Executando

UFABC

```
$ make hello_task_group
g++ -I/caminho/para/o/TBB/include \
    -L/caminho/para/o/TBB/lib \
    hello_task_group.cpp \
    -o hello_task_group -ltbb
$ ./hello_task_group
hello from task 2
hello from task 1
$
```

O `parallel_for`

Potenciação de números complexos

UFABC

Considere o seguinte problema:

Entrada: $n \in \mathbb{Z}_{>0}, d \in \mathbb{Z}_{>0}, x \in \mathbb{C}^n$

Saída: $y \in \mathbb{C}^n, y_k = x_k^d, \text{para } k = 1, 2, \ldots, n$

- É preciso tomar cuidado para evitar overflows (aqui pegamos números no círculo unitário)
- Representamos os números complexos em C++ como uma *template class*
- Para criar um complexo com o tipo `double` fazemos:

```
1  #include <complex>
2  using namespace std;
3  typedef complex<double> dcmplx;
```

Código da potenciação

UFABC

```cpp
#include <cstdlib>
#include <cmath>
// generates a random complex number
// on the complex unit circle
dcmplx random_dcmplx (void);
```

Calculamos $e^{2\pi i\theta} = \cos(2\pi\theta) + i\sin(2\pi\theta)$ para valores aleatórios de $\theta \in [0, 1]$

```cpp
dcmplx random_dcmplx (void) {
    int r = rand();
    double d = ((double) r)/RAND_MAX;
    double e = 2*M_PI*d;
    dcmplx c(cos(e),sin(e));
    return c;
}
```

Imprimindo os vetores

UFABC

```
1  #include <iostream>
2  #include <iomanip>
3  // writes the array of n doubles in x
4  void write_numbers(int n, dcmplx *x) {
5    for(int i=0; i<n; i++)
6      cout << scientific << setprecision(4)
7           << "x[" << i << "] = ( " << x[i].real()
8           << " , " << x[i].imag() << ")\n";
9  }
```

- Observe a declaração local de `i` no laço.
- Também modificamos a saída para formato científico e utilizamos os métodos `real()` e `imag()`

Computando as potências

UFABC

```
1  // Percorre x e y de tamanho n. Devolve y[i]=x[i]^d
2  void compute_powers (int n, dcmplx *x,
3                       dcmplx *y, int d) {
4      for(int i=0; i < n; i++) {
5              dcmplx r(1.0,0.0);
6              // pow é eficiente demais.
7              // Fazemos na mão.
8              for(int j = 0; j < d; j++)
9                  r = r*x[i];
10             y[i] = r;
11         }
12 }
```

O `for` interno é deliberadamente lento (por exemplo não utiliza o método de elevar ao quadrado repetidamente)

Executando

```
$ /tmp/powers_serial 2 3 1
x[0] = ( -7.4316e-02 , 9.9723e-01)
x[1] = ( -9.0230e-01 , 4.3111e-01)
x[0] = ( 2.2131e-01 , -9.7520e-01)
x[1] = ( -2.3152e-01 , 9.7283e-01)
$ time /tmp/powers_serial 1000 1000000 0
real 0m20.139s
user 0m20.101s
sys 0m0.000s
```

- 1º parâmetro: n
- 2º parâmetro: d
- 3º parâmetro: verbose

Paralelizando o programa com `parallel_for`

UFABC

- Para utilizarmos o `parallel_for` é preciso criar uma classe
- Também é possível utilizar funções lambda para simplificar um pouco a implementação
- O `parallel_for` particiona o *range* original e distribui os subranges aos trabalhadores de modo que:
  - A carga seja balanceada
  - Use as caches de maneira eficiente
  - Escale bem em processadores com muitos cores

A classe ComputePowers

UFABC

```
1  class ComputePowers {
2      dcmplx *const c; // Entrada
3      int d; // Potência
4      dcmplx *result; // Saída
5  public:
6      ComputePowers(dcmplx x[], int deg, dcmplx y[])
7          : c(x), d(deg), result(y) {}
8      void operator()
9      (const blocked_range<size_t>& r) const {
10         for(size_t i = r.begin(); i != r.end(); ++i) {
11             dcmplx z(1.0,0.0);
12             for(int j = 0; j < d; j++)
13                 z = z * c[i];
14             result[i] = z;
15         }
16     }
17 };
```

tbb/blocked_range.h

UFABC

```
1 #include "tbb/blocked_range.h"
2 template<typename Value> class blocked_range
```

Um `blocked_range` representa um intervalo na forma $[i, j)$ que pode ser dividido de maneira recursiva.

```
1 void operator() (const blocked_range<size_t>& r) const {
2     for(size_t i = r.begin(); i != r.end(); ++i) {
3     ...
```

tbb/blocked_range.h

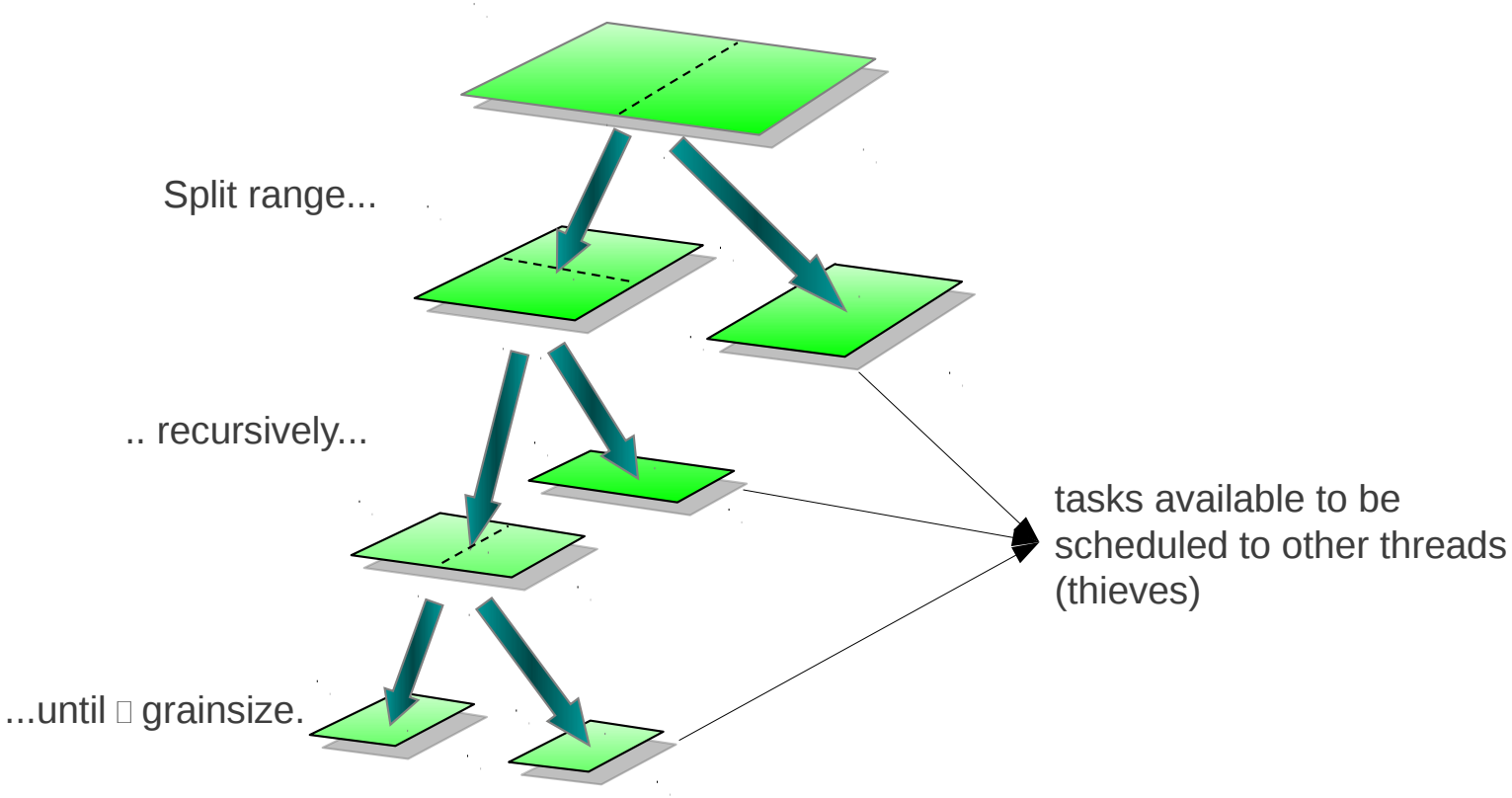
UFABC

- A biblioteca já fornece alguns ranges: `blocked_range`, `blocked_range2d`, e `blocked_range3d`
- Você pode definir o seu próprio implementando os métodos:

```
1 //Copy constructor
2 MyRange::MyRange (const MyRange&)
3 //Destructor
4 MyRange::~MyRange()
5 bool MyRange::is_empty() const
6 //True if range can be partitioned
7 bool MyRange::is_divisible() const
8 //Splitting constructor; splits r into two subranges
9 MyRange::MyRange(MyRange& r, split)
```

UFABC

blocked_range2d



Chamando o `parallel_for`

UFABC

```
#include "tbb/tbb.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"
using namespace tbb;
```

É preciso modificar a chamada à função `compute_powers` na função `main` de:

```
compute_powers(dim, r, s, deg);
```

Para:

```
task_scheduler_init
  ↪  init(task_scheduler_init::automatic);
parallel_for(blocked_range<size_t>(0, dim),
             ComputePowers(r, deg, s));
```

Executando a versão paralela

UFABC

```
$ time /tmp/powers_serial 1000 1000000 0
real 0m20.139s
user 0m20.101s
sys 0m0.000s
$ time /tmp/powers_tbb 1000 1000000 0
real 0m1.191s
user 0m35.170s
sys 0m0.043s
```

- Speedup de $\frac{20.139}{1.191} = 16.909$ em uma máquina com dois processadores de 8 núcleos

Exemplo 2 - Código sequencial

UFABC

```
1  const int N = 100000;
2  void change_array(float array, int M) {
3      for (int i = 0; i < M; i++){
4          array[i] *= 2;
5      }
6  }
7  int main (){
8      float A[N];
9      initialize_array(A);
10     change_array(A, N);
11     return 0;
12 }
```

Exemplo 2 - Código paralelo

UFABC

```
1  class ChangeArrayBody {
2      float *array;
3  public:
4      ChangeArrayBody(float *a): array(a) {}
5      void operator()( const blocked_range<size_t>& r )
       ↪  const{
6          for (size_t i = r.begin(); i != r.end(); i++ ){
7              array[i] *= 2;
8          }
9      }
10 };
11 void parallel_change_array(float *array, size_t M) {
12     parallel_for(blocked_range<int>(0, M,
       ↪  IdealGrainSize),
13                  ChangeArrayBody(array));
14 }
```

Exemplo 2 - Código usando lambdas

UFABC

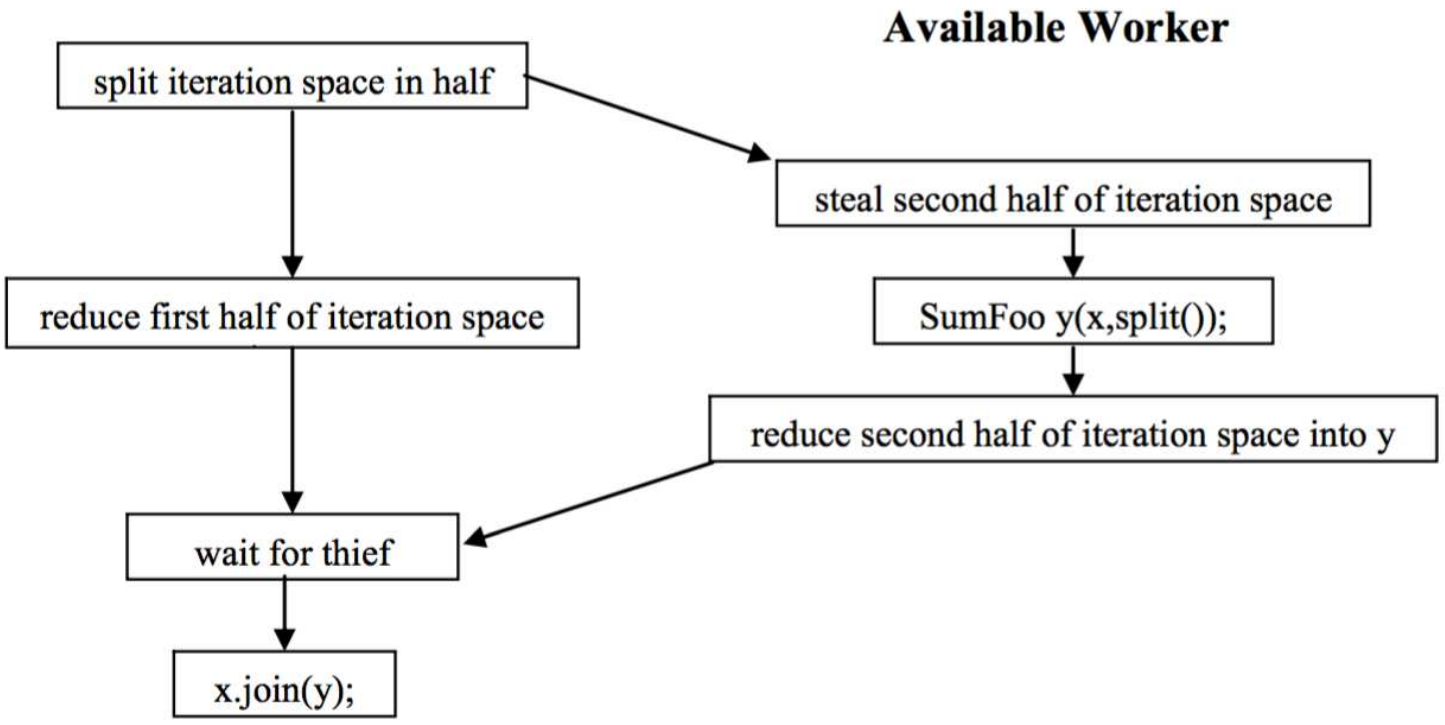
```
1  void parallel_change_array(float *array, size_t M) {
2      parallel_for(blocked_range<size_t>(0, M,
       ↪  IdealGrainSize),
3          [=](const blocked_range<size_t>& r) -> void {
4              for(size_t i = r.begin(); i != r.end(); i++)
5                  array[i] *= 2;
6          });
7  }
```

UFABC

- Em termos de desempenho são equivalentes
- Nem sempre é possível utilizar funções lambda
- Funções lambda fazem parte do padrão C++11
  - Alguns compiladores podem exigir um parâmetro adicional para especificar a versão do C++ (`-std=c++0x`)

parallel_reduce

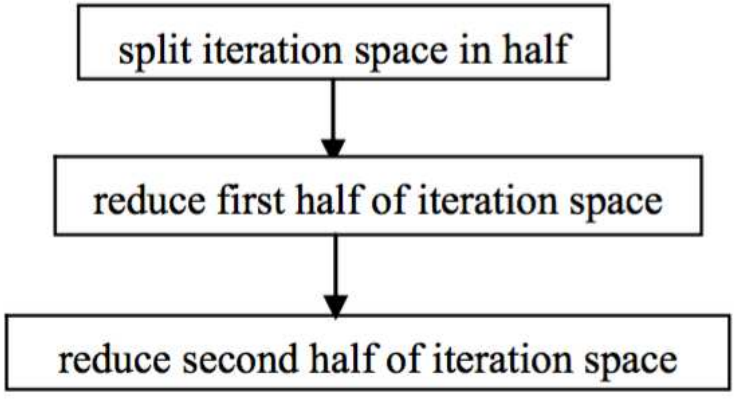
Uma ótima aplicação de roubo de trabalho



Fonte: Intel TBB Tutorial

E no caso de não haver nenhum trabalhador disponível?

UFABC



No Available Worker

Fonte: Intel TBB Tutorial

A classe SumIntegers

UFABC

```
1  class SumIntegers {
2    int *data;
3  public:
4    int sum;
5    SumIntegers (int *d) : data(d), sum(0) {}
6    void operator() (const blocked_range<size_t>& r) {
7        int s = sum; // É preciso acumular
8        int *d = data;
9        size_t end = r.end();
10       for(size_t i = r.begin(); i != end; ++i)
11           s += d[i];
12       sum = s;
13   }
14   // O construtor com split
15   SumIntegers (SumIntegers& x, split) : data(x.data), sum(0) {}
16   void join (const SumIntegers& x) {//combina resultados
17       sum += x.sum;
18   }
19 };
```

Chamando a função

UFABC

```
1  int ParallelSum (int *x, size_t n) {
2      SumIntegers S(x);
3      parallel_reduce(blocked_range<size_t>(0, n), S);
4      return S.sum;
5  }
6
7  int main (int argc, char *argv[] ) {
8      ...
9      task_scheduler_init init
10         (task_scheduler_init::automatic);
11     int s = ParallelSum(d,n);
12     cout << "A soma é " << s
13          << " e deveria ser: " << n*(n+1)/2
14          << endl;
15     return 0;
16 }
```

Outras funcionalidades

Algoritmos disponíveis no TBB 1/2

UFABC

- `parallel_for` e `parallel_for_each`: execução paralela de iterações de laços com balanceamento de carga automático. As iterações precisam ser independentes.
- `parallel_reduce`: execução paralela de iterações de laços sem dependência onde é necessário efetuar uma redução (ex. soma de elementos distintos) com balanceamento de carga automático
- `parallel_scan`: execução paralela e balanceada da computação de prefixos paralelos
- `parallel_pipeline`: execução paralela através de um pipeline com a opção de uso de filtros sequenciais ou paralelos.

Algoritmos disponíveis no TBB 2/2

UFABC

- `parallel_do`: execução paralela balanceada de iterações de laços sem dependência com a possibilidade de adicionar trabalho à fila durante a execução
- `parallel_sort`: auto-explicativo
- `parallel_invoke`: execução paralela de *function objects* ou ponteiros para funções

Para saber mais

Para saber mais

UFABC

- Página oficial: https://www.threadingbuildingblocks.org/
- Repositório TBB: https://github.com/01org/tbb
- Intel Threading Building Blocks Tutorial - https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf
- Parallel Programming with Intel Threading Building Blocks - https://www.csd.uoc.gr/~hy529/TBB.pdf