

OpenMP

**Slides baseados em tutorial de Tim Mattson da
Intel**

O que é OpenMP ?

- **Uma especificação para um conjunto de diretivas de compilação, rotinas de biblioteca e variáveis de sistema que podem ser utilizadas para especificar paralelismo baseado em memória compartilhada**
- **Portável, incluindo plataformas Unix e Windows NT**
- **Disponível em implementações Fortran e C/C++**
- **Definida e endossada por um grupo grande de fabricantes de software e hardware**
- **Suporta paralelismo de granulosidade fina e grossa**

Origens

- **No início dos anos 90, fabricantes de máquinas com memória compartilhada forneciam extensões para programação paralela em Fortran**
- **As implementações permitiam ao usuário inserir diretivas para indicar onde loops deveriam ser paralelizados e os compiladores eram responsáveis pela paralelização**
- **Implementações funcionalmente parecidas, mas não portáveis e começaram a divergir**
- **ANSI X3H5 em 1994 foi a primeira tentativa de padronização**

Origens

- **A especificação padrão OpenMP começou em 1997 partindo do padrão X3H5 graças ao aparecimento de novas arquiteturas de máquinas de memória compartilhada**
- **Alguns parceiros na especificação:**
 - Compaq, HP, Intel, IBM, Silicon, Sun
 - Absoft, GENIAS, Myrias, The Portland Group
 - ANSYS, Dash, ILOG CPLEX, Livermore, NAG
- **A API para Fortran foi liberada em Outubro de 1997 e para C/C++ no final de 1997**
- **Última versão OpenMP 5.0 (Novembro 2018)**

Objetivos

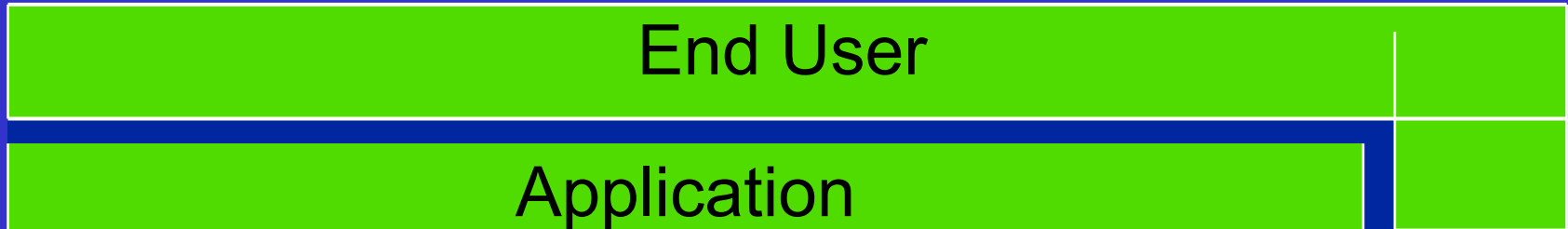
- **Prover um padrão para uma variedade de plataformas e arquiteturas baseadas em memória compartilhada**
- **Estabelecer um conjunto limitado e simples de diretivas para programação utilizando memória compartilhada**
- **Prover capacidade para paralelizar um programa de forma incremental**
- **Implementar paralelismo com granulosidade fina e grossa**
- **Suportar Fortran, C e C++**

Modelo de programação

- **Paralelismo baseado em threads:**
 - Se baseia na existência de processos consistindo de várias threads
- **Paralelismo explícito:**
 - Modelo de programação explícito e não automático, permitindo total controle da paralelização ao programador
- **Modelo fork-join**
 - Os programas OpenMP começam como um único processo denominado *master thread*, que executa seqüencialmente até encontrar a primeira construção para uma região paralela
 - FORK: a master thread cria um time de threads paralelos
 - As instruções que estão dentro da construção da região paralela são executadas em paralelo pelas diversas threads do time
 - JOIN: quando as threads finalizam a execução das instruções dentro da região paralela, elas sincronizam e terminam, ficando somente ativa a master thread

OpenMP Basic Defs:

User layer



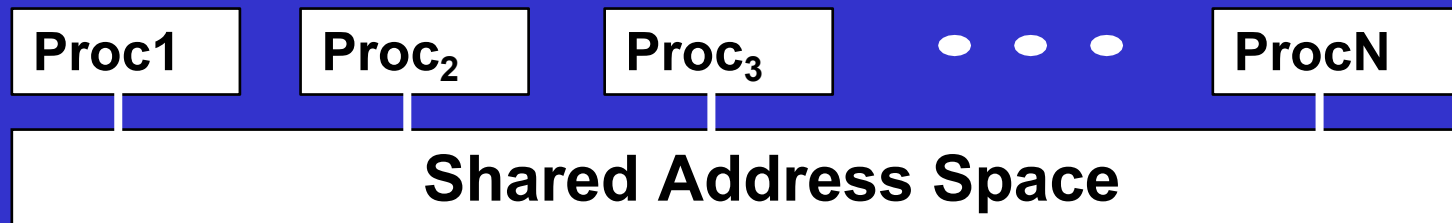
Prog-Layer



System layer



HW



Modelo de programação

- **Baseado em diretivas de compilação:**
 - o paralelismo é especificado através do uso de diretivas para o compilador que são inseridas em um código Fortran ou C/C++
- **Suporte a paralelismo aninhado:**
 - construções paralelas podem ser colocadas dentro de construções paralelas e as implementações podem ou não suportar essa característica
- **Threads dinâmicas:**
 - o número de threads a serem utilizadas para executar um região paralela pode ser dinamicamente alterado

Exercício 1: Hello World (uma thread)

```
void main ( )  
{  
  int ID = 0;  
  printf("hello(%d) ", ID);  
  printf("world(%d) \n", ID);  
}
```

Diretivas C/C++

- **Formato**

#pragma omp	nome da diretiva	[cláusula, ...]	newline
Necessária para todas as diretivas	Uma diretiva válida que deve aparecer depois da pragma e antes das cláusulas	Opcional. Podem aparecer em qualquer ordem e repetidas quando necessário	Necessária. Deve ser inserido após o bloco estruturado que está dentro da diretiva.

- **Exemplo:**

- #pragma omp parallel default(shared) private(beta,pi)

- Seguem o padrão de diretivas de compilação para C/C++
- Cada diretiva se aplica no máximo a próxima instrução, que deve ser um bloco estruturado

Construtor de região PARALLEL

- **Fork-join**
 - Quando uma thread chega na região paralela, ela cria um time de threads e se torna a mestre do time. Ela faz parte do time e tem o número 0.
 - O código que começa no início da região paralela é duplicado e todas as threads o executam
 - Existe uma barreira implícita no final da seção paralela, e somente o mestre continua a executar após esse ponto
- **O número de threads na região paralela é definido por:**
 - Uso da rotina **omp_set_num_threads ()**
 - Na diretiva **#pragma omp parallel num_threads ()**
 - Uso da variável de ambiente **OMP_NUM_THREADS**
 - Default da implementação

Exercício 1: Hello World (multithreaded)

```
#include <omp.h>
void main ( )
{
    #pragma omp parallel
    {
        int ID = 0;
        printf("hello(%d)", ID);
        printf("world(%d)\n", ID);
    }
}
```

Arquivo header OpenMP

Início da região paralela
com número default de threads

Final da região paralela

Exercício 1: Hello World (multithreaded)

```
#include <omp.h>
void main ( )
{
#pragma omp parallel
{
int ID = 0;
printf("hello(%d)", ID);
printf("world(%d)\n", ID);
}
```

Opções para compilação e
ligação
gcc -fopenmp

Exercício 1: Hello World (multithreaded)

Cada thread imprime seu número:

```
#include <omp.h>
void main ( )
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d)", ID);
        printf("world(%d)\n", ID);
    }
}
```

Rotina da biblioteca para
obter número da thread

Criação de threads

- **Para criar 4 threads na região paralela:**

```
double A[1000]
omp_set_num_threads(4)
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    fa(A, ID);
}
```

- **Cada thread executa uma cópia do código que está no bloco estruturado**

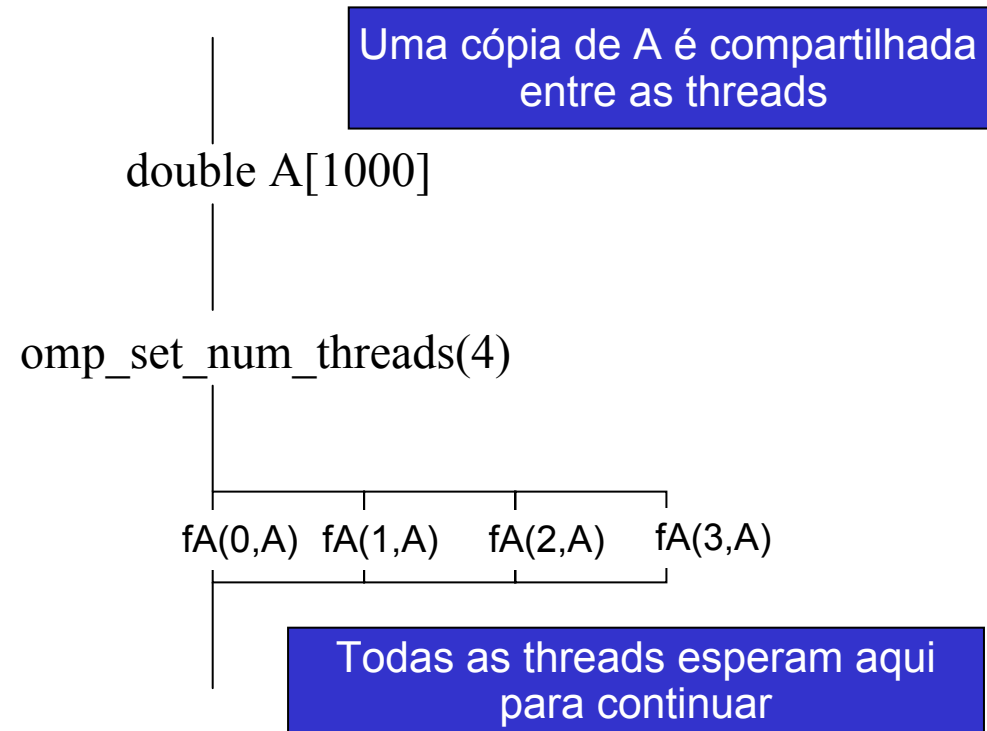
Criação de threads

- **Para criar 4 threads na região paralela:**

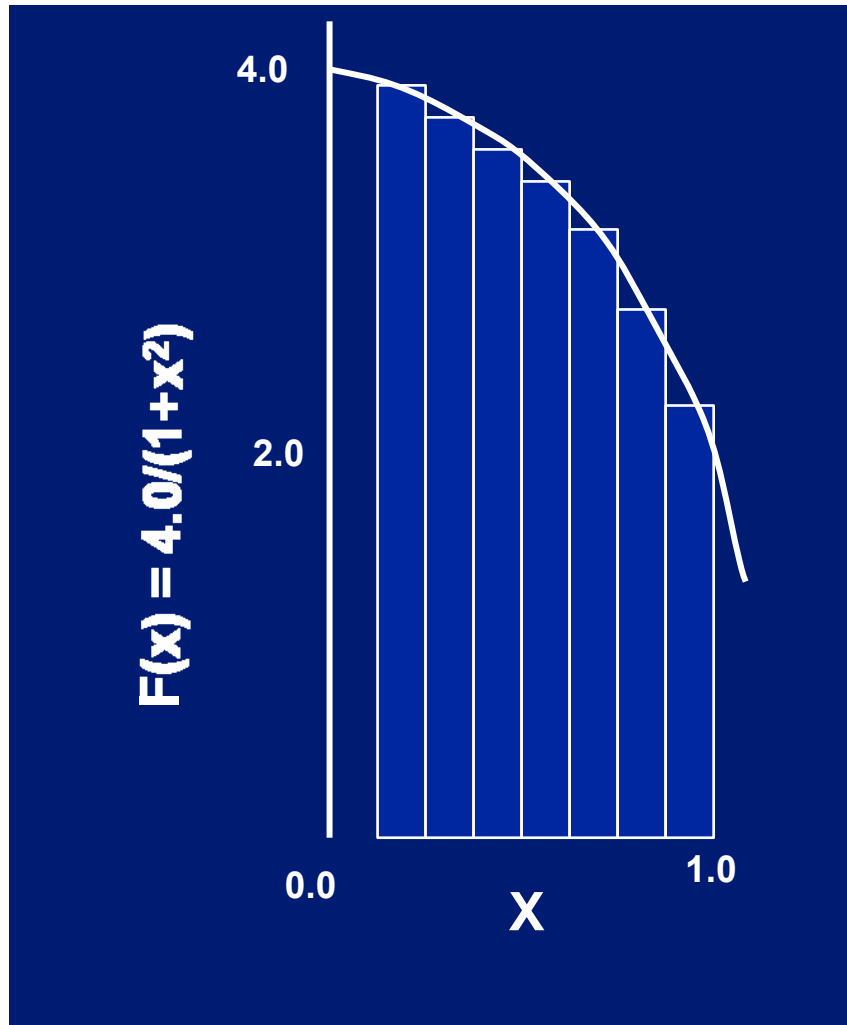
```
double A[1000]
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    fa(A, ID);
}
```

- **Cada thread executa uma cópia do código que está no bloco estruturado**

Criação de threads



Integração numérica



Matematicamente, sabemos que:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

Podemos aproximar a integral por uma soma de retângulos:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Onde cada retângulo tem largura Δx e altura $F(x_i)$ no meio do intervalo i

Programa sequencial Pi

```
static long num_steps = 100000;
double step;
void main ()
{  int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
   for (i=0;i< num_steps; i++){
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Programa paralelo Pi

- **Crie uma versão paralela do programa Pi usando uma construção paralela**
- **Divida o número de steps entre as threads**
- **Crie um vetor compartilhado para armazenar as somas parciais**
- **Utilize as rotinas:**
 - `omp_set_num_threads`
 - `omp_get_thread_num`
 - `omp_get_num_threads`

Programa paralelo Pi

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
}
```