**Ministério da Educação**
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca
UNED Nova Friburgo
Bacharelado em Sistemas da Informação

# OpenMP

## Programação Paralela e Concorrente

Prof. Bruno Policarpo Toledo Freitas
bruno.freitas@cefet-rj.br


# Objetivos

- **Paralelizar programas seriais com o OpenMP**
- **Aplicar os diferentes tipos de projeto de programas paralelos utilizando OpenMP**
- **Compreender como diferentes formas de desenvolver programas paralelos com OpenMP podem afetar o desempenho de programas**

# Motivação

- **Vimos anteriormente programação concorrente com memória compartilhada utilizando *pthreads***
- **Porém, a programação com threads não é fácil pois tudo deve ser feito manualmente**
  - Mecanismos de sincronização
  - Comunicação entre threads
  - Controle de criação e término de threads

# OpenMP

- **A OpenMP é uma biblioteca de threads para sistemas de memória compartilhada**
  - Threading implícito
- **A OpenMP é mais simples do que a pthreads pois apenas requer a definição do trecho paralelizável**
  - Compilador é o responsável por fazer o “trabalho sujo”
  - Em compensação, perde-se generalidade (no sentido de problemas capazes de serem resolvidos)

# Diretivas de Pré-compilação

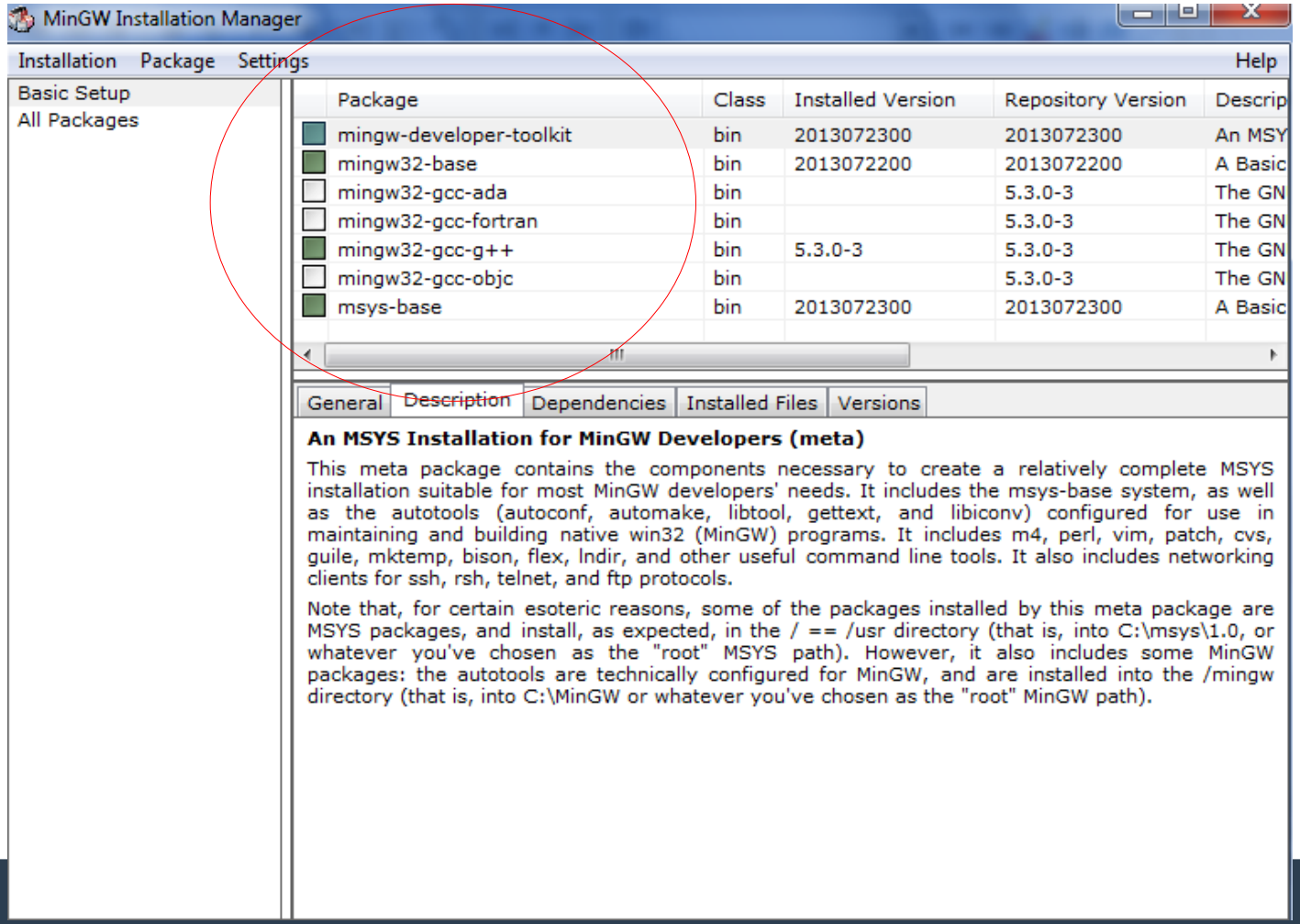
- **OpenMP funciona por meio de diretivas de compilação**
- **Em C/C++, diretivas de compilação são processadas antes do processo de compilação em si começar**
  - #include
  - #ifdef / #if / #else / #endif
- **Utiliza-se a OpenMP por meio de pragmas**
  - Diretivas de pré-compilação fora do C

# Instalação

- **No GCC (GNU/Linux) a API OpenMP é instalado automaticamente**
- **No Windows com a IDE Codeblocks, é necessário reinstalar o compilador:**
  - https://sourceforge.net/projects/mingw/

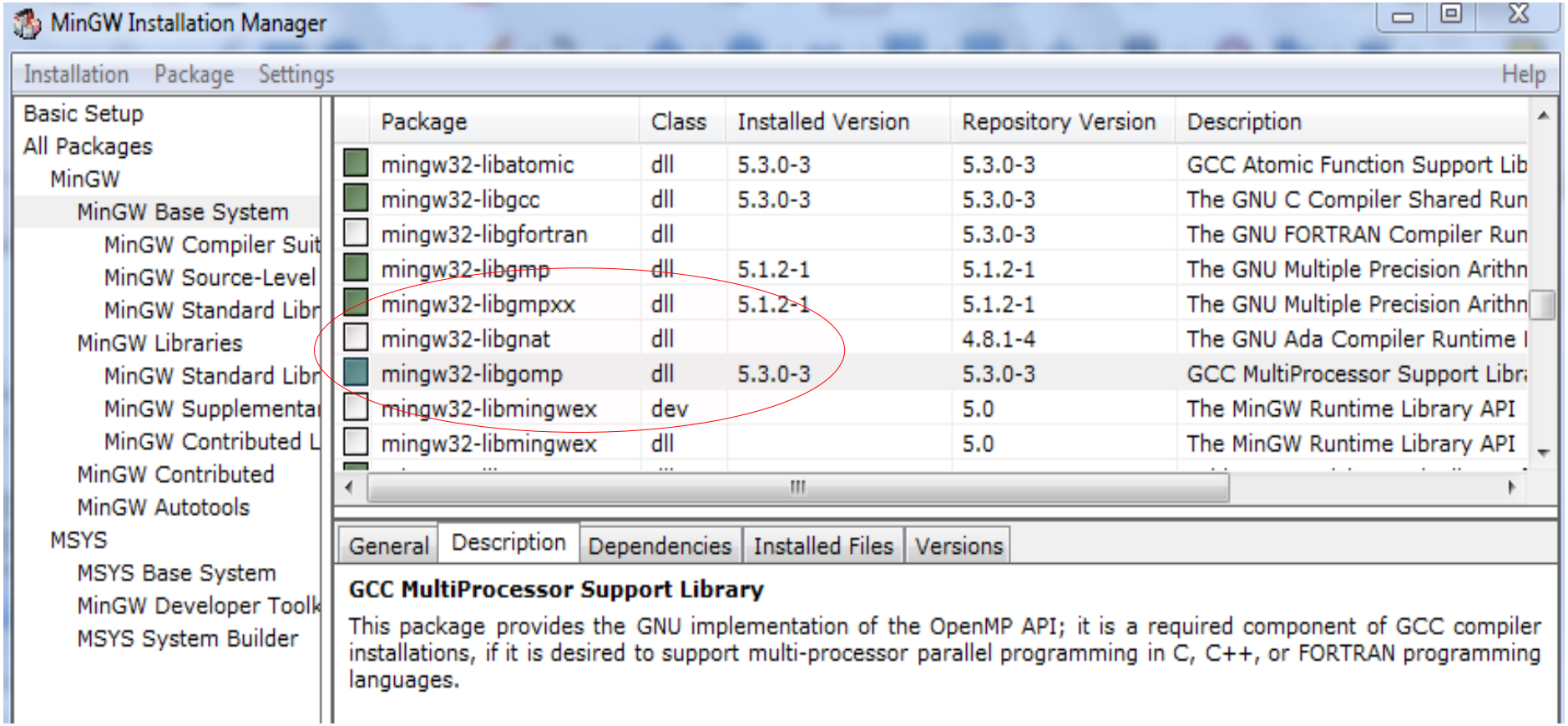
# Instalação

- **Durante instalação do MingW, marcar:**



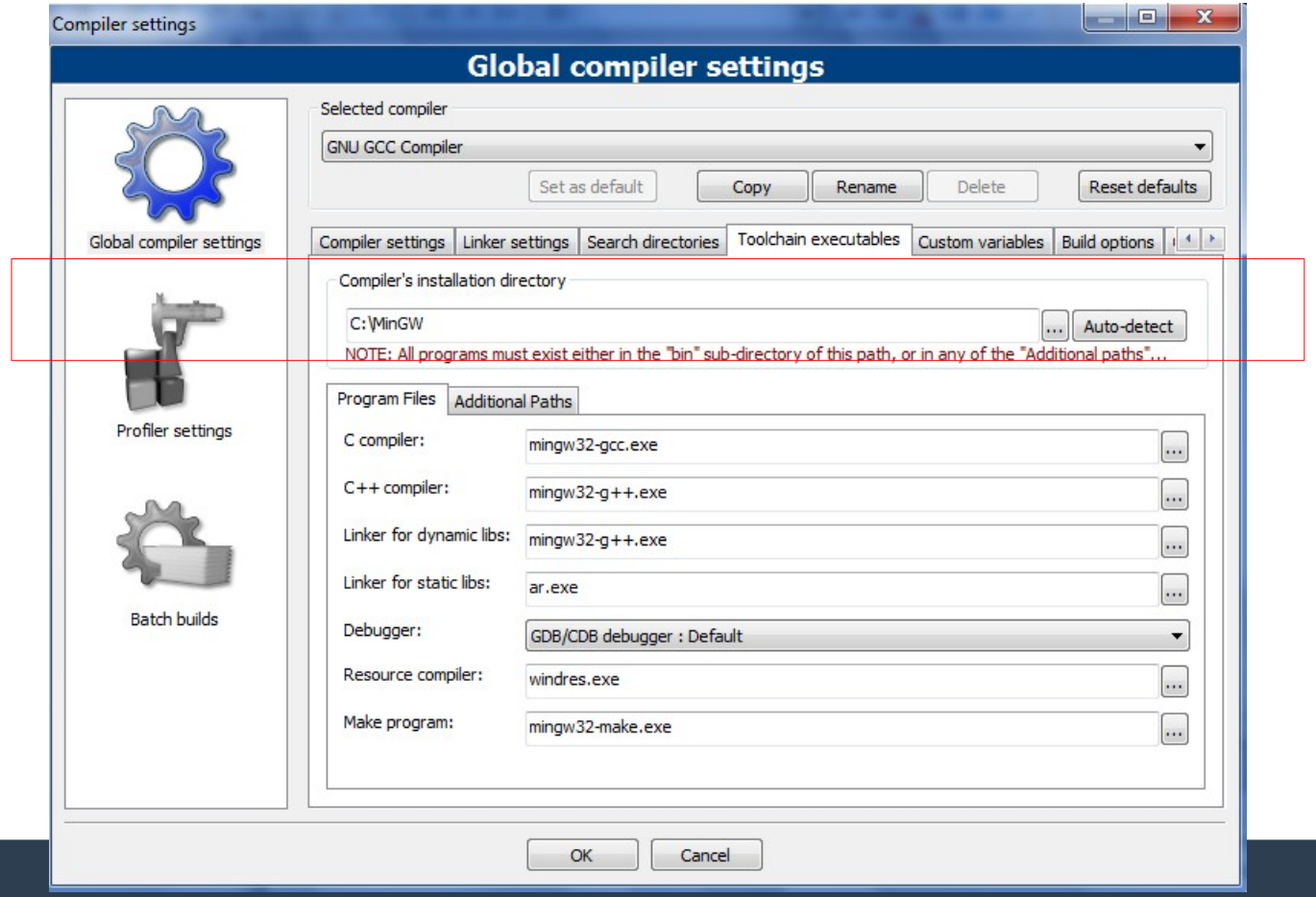
# Instalação

- **Após a instalação do MingW, aparecerá um ícone para configurar a instalação. Marcar::**



# Instalação

- **Trocar o compilador do Codeblocks:**
  - Settings → Compiler → Toolchain Executables



# Programa Hello_OMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);

int main(int argc,char *argv[]){
    // Get number of threads from command line
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num threads(thread_count)
    Hello();

    return 0;
};

void Hello(void){
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n",
           my_rank,
           thread_count);
}
```

header

#pragma omp parallel
Paraleliza um trecho

**Barreira implícita**

Funções que retornam
Informações *per thread*

# Compilação

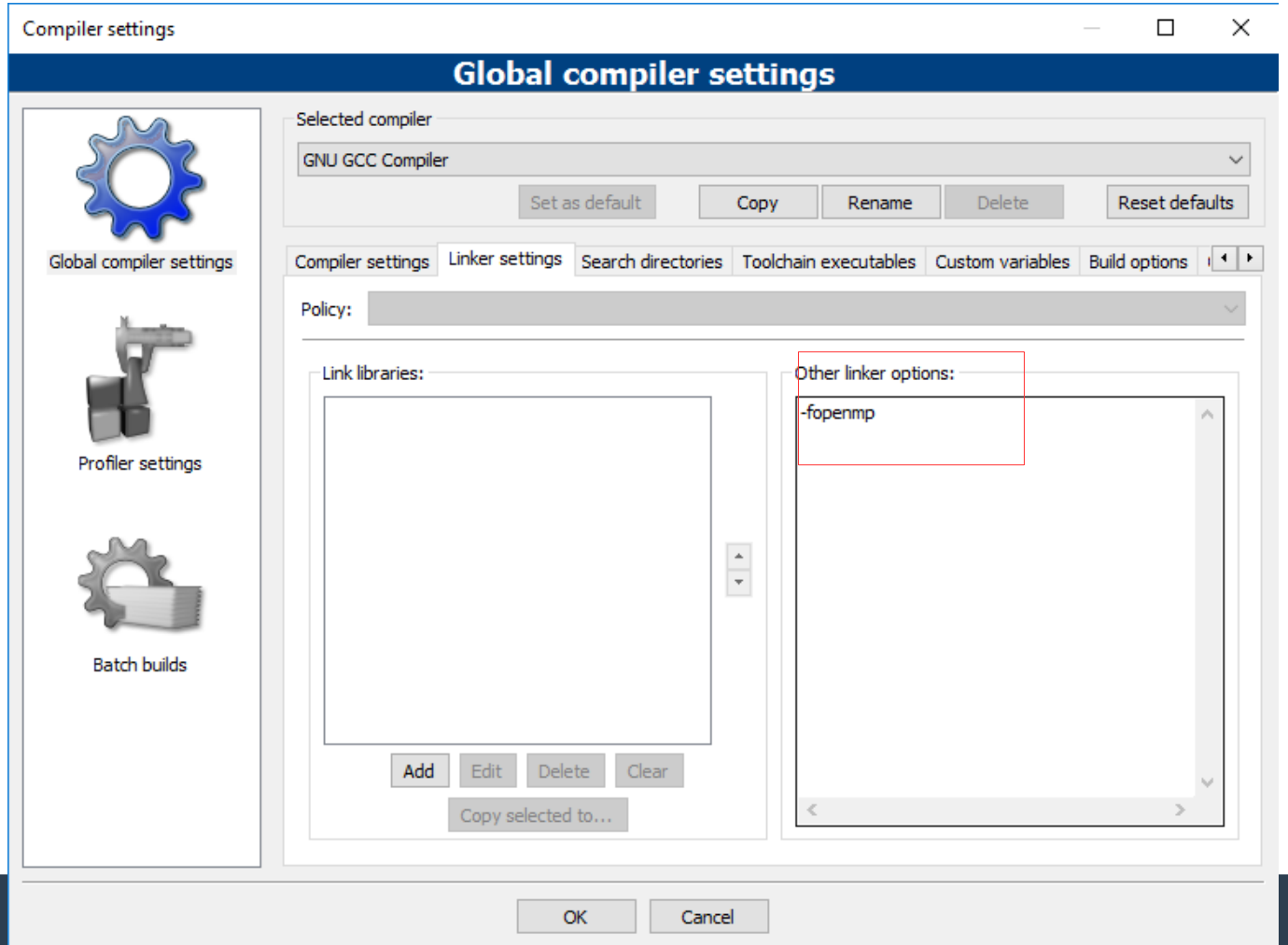
- **Para compilar um programa OpenMP na linha de comando:**

  gcc −g −Wall **−fopenmp** −o omp_hello omp_hello.c

- **Para compilar um programa OpenMP no Codeblocks:**
  - Settings → Compiler → Compiler Settings
    - Other Compiler Options
  - Settings → Compiler → Linker Settings
    - Other Linker Options
  - Colocar -fopenmp em ambas opções

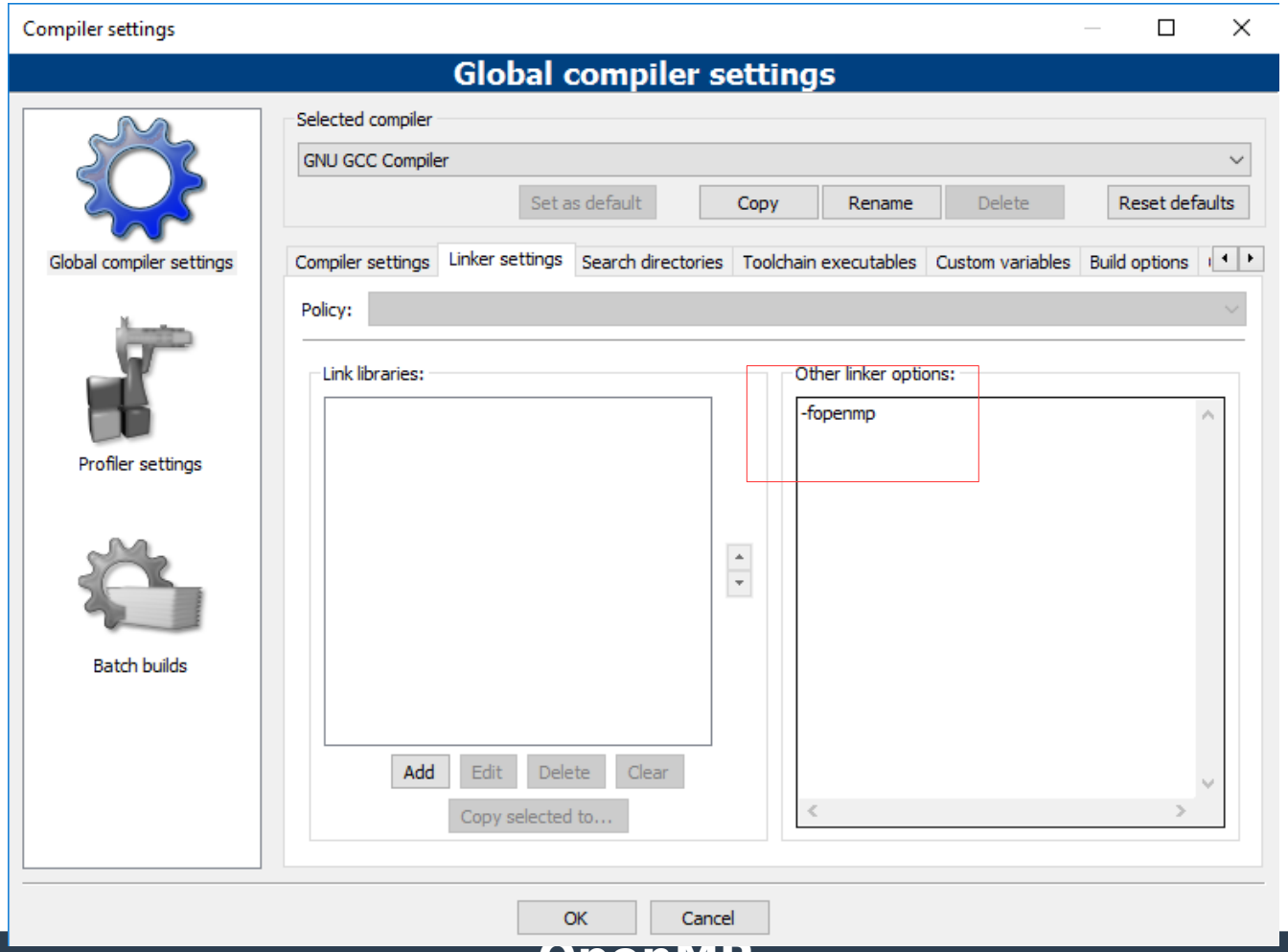
# Codeblocks
## Compilação (configuração global)

- **Settings → Compiler → Compiler Settings**



# Codeblocks
## Ligação (configuração global)

- **Settings → Compiler → Linker Settings**



# Verificação de suporte

- **Boa prática de programação em C: escrever código alternativo quando não há suporte do compilador ao OpenMP**
- **A verificação do suporte ao OpenMP é feita pela macro _OPENMP**
  - Se definida: utiliza o código com OMP
  - Se não: troca por implementações monothread

# Verificação de suporte

```
#ifdef _OPENMP
#include <omp.h>
#endif


#ifdef _OPENMP
   int my_rank = omp_get_thread_num() ;
   int thread_count = omp_get_num_threads() ;
#else
   int my_rank = 0;
   int thread_count = 1;
#endif
```

# Diretivas

- **As diretivas do OpenMP seguem o seguinte formato:**

  #pragma omp *diretiva [opções]*

# #pragma omp parallel

- **Paraleliza um trecho do programa**
- **É formado um time de threads após sua chamada**
  - Thread principal é a mestre
  - Demais threads são escravas
- **Escopo das variáveis:**
  - Variáveis declaradas antes da diretiva são compartilhadas
  - Variáveis declaradas no bloco são locais
    - Parâmetros são inicializados com os valores antes da diretiva

# #pragma omp parallel

- **Exemplo 1: Soma dos elementos de um vetor**

  Padrão *reduction*

# Sincronizações
## Regiões críticas

- **Regiões críticas são definidas com:**

  #pragma omp critical

- **Regiões críticas puramente aritméticas podem ser definidas com**

  #pragma omp atomic

**Regra geral: minimizar ao máximo o uso de sincronizações**

# Sincronizações
## Barreiras

- **Barreiras são definidas por:**

  #pragma omp barrier

- **As barreiras automáticas podem ser desabilitadas com:**

  #pragma omp nowait

  – Avaliar com cuidado quando desabilitar barreiras!

# Sincronizações

- **Exemplo 2: Soma dos elementos de 2 vetores ponto-a-ponto**
  - Padrão fork-join
  - Barreira necessária ...

# #pragma omp parallel for

- ***Loops* são candidatos perfeitos para serem paralelizados**
- **Paraleliza um *loop* for**

# #pragma omp parallel for
## reduction

- **Reduções são bastante comuns em programação paralela**
- **A construção *reduction* implementa uma redução**

  reduction(*operador*: *variável*)

  - Operador pode ser (+,-, *, ^, ||, &&)
  - Variável só pode ser “pura”
    - Não vale indexação de vetores, por exemplo

# #pragma omp parallel for
## reduction

- **Exemplo: soma dos elementos de um vetor**

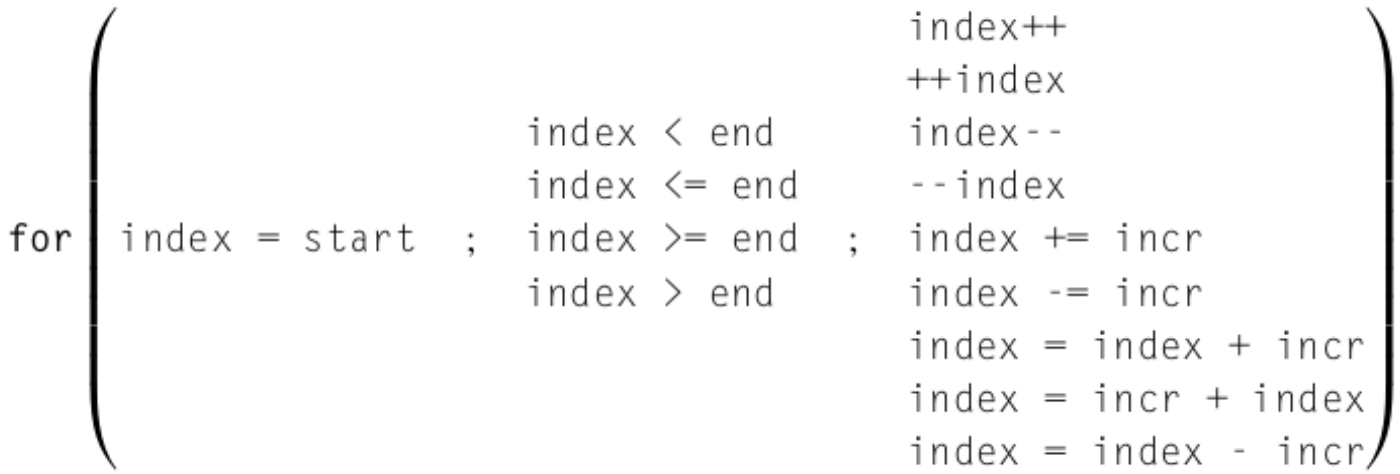
```
int soma = 0;

#pragma omp for reduction( + : soma)
for (int i = 0; i < MAX; i++){
    soma += vetor [ i ];
}
```

# #pragma omp parallel for
## Limitações

- **Loop deve ter número conhecido de iterações**
- **Não utilizar *break* dentro do loop**
- **Não funciona com demais loops**
- **Loop for deve estar na forma canônica**

# #pragma omp parallel for
## Forma canônica

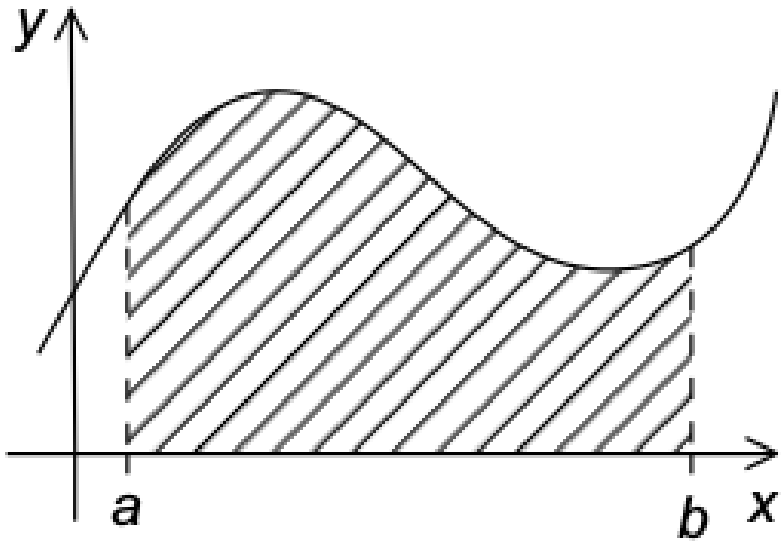



- **index somente podem ser inteiros ou ponteiros**
- **start/end/incr devem ter tipos compatíveis**
- **start/end/incr não podem ser alteradas na execução do código**
- **index só pode ser alterado pela expressão de incremento**

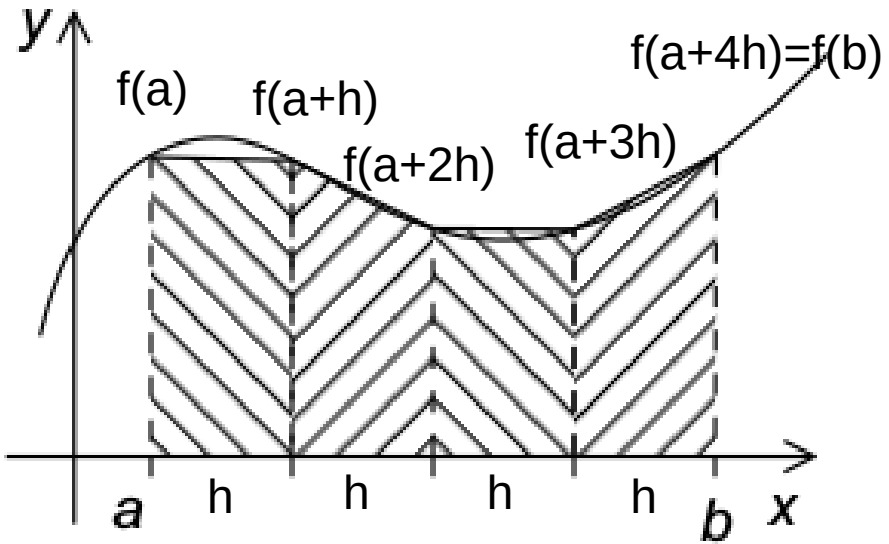
# #pragma omp parallel for

- **Exemplo: Aproximação de área de superfícies por trapézios**

$$Área\,do\,trapézio = Base\,média * altura \Rightarrow A = \frac{(b+B)}{2} * h$$




(a)




(b)

# #pragma omp parallel for
## Dependência de dados

- **Se alguma das condições for violada, o compilador simplesmente irá gerar um erro.**
- **Porém, certas situações são mais sutis**
  - Passarão pelo compilador
  - Porém, irão gerar problemas em tempo de execução:
    - Exemplo: Sequência de Fibonacci

# Escopo de variáveis

- **Dois modificadores podem definir explicitamente o escopo de variáveis:**
  - private(variavel): é criada uma cópia da variável para cada thread
  - shared(variavel): variável se torna compartilhada entre as threads

# Exercício: escopo de variáveis Cálculo de π

- **Uma maneira de calcular é por meio do seguinte somatório:**

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right]$$

1) Implemente o programa serial equivalente?
2) Utilize a OpenMP para paralelizá-lo

**Problema: O valor do $(-1)^k$ depende do fator inicial!**

# Escalonamento de Threads

- **O escalonamento padrão de threads da OpenMP é estático**
  - Cada bloco possui (iterações/threads) de tamanho
- **Porém, certas aplicações podem ser prejudicadas por tal política**
  - Exemplo: tempo de execução do corpo do for é proporcional à iteração:

```
for (int i = 0; i < SIZE; i++){
      for ( int j = 0; j < i; j ++){
            x [  i  ]  +=  j;
      }
}
```

# Escalonamento de Threads
## Modificador *schedule(tipo [,chunksize])*

- **O modificador *schedule(tipo,chunksize)* modifica como o OpenMP escalona as threads**
  - *chunksize*: quantidade de iterações em sequência que são distribuídas para cada thread

| | |
|---|---|
| Thread 0: | 0,3,6,9 |
| Thread 1: | 1,4,7,10 |
| Thread 2: | 2,5,8,11 |

tipo=static
chunksize=1

| | |
|---|---|
| Thread 0: | 0,1,6,7 |
| Thread 1: | 2,3,8,9 |
| Thread 2: | 4,5,10,11 |

tipo=static
chunksize=2

# Escalonamento de Threads
## Modificador *schedule(tipo [,chunksize])*

- **tipo=static**
  - Iterações são distribuídas na compilação
- **tipo=dynamic**
  - Iterações são distribuídas em tempo de execução quando a thread termina o seu bloco
- **tipo=guided**
  - Cada bloco de iterações terminadas diminui o tamanho dos próximos blocos, proporcional à quantidade de threads
- **tipo=runtime**
  - Obtém o tipo de escalonamento pela variável de ambiente OMP_SCHEDULE

# Qual escalonamento?

- **Blocos menores implicam overhead maior**
- **Quase sempre o padrão é mais recomendado**
- **Porém:**
  - Se o custo de execução de uma iteração de um loop cresce/decresce, blocos menores podem favorer a paralelização
  - Se o custo de execução de uma iteração de um loop não pode ser determinado, recomenda-se testar cada caso com *runtime*

# Temporização

- **double *omp_get_wtime()***
  - Retorna a quantidade de segundos transcorridos desde o início do programa

# Programação concorrente

- **Sections: explicita as partes paralelas do programa**
  - Podem executar códigos diferentes

```
#pragma omp sections nowait
{
  #pragma omp section
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
  #pragma omp section
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
} /*-- End of sections --*/
```

# Programação concorrente

- **Task: threads geram “tarefas” como no padrão despachante-operário**

```
#pragma omp parallel
{
   #pragma omp single
     {
        printf("A ");
        #pragma omp task
          {printf("race ");}
        #pragma omp task
          {printf("car ");}
   }
} // End of parallel region
```

# Exercício 1

- **Paralelizar o algoritmo Monte-Carlo para o cálculo do PI:**

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between −1 and 1;
   y = random double between −1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

# Exercício 2

- **Considere o exercício do cálculo aproximado do PI. Suponhamos que queiramos comparar as diferentes aproximações do somatório, de 1 até 10000**

# Referências


- **PACHECO, P. An Introduction to Parallel Programming**. McGraw-Hill, 2011.
  - Cap. 5: Seções 5.1.2, 5.1.3, 5.3, 5.4, 5.5, 5.5.1, 5.5.2

# Referências

- Minicurso de Introdução ao OpenMP 2023:

  https://www.youtube.com/watch?v=z8sEIvreP80

  https://www.inf.ufrgs.br/~msserpa/MC-SD02-I.zip
- Introdução a Programação Paralela e Vetorial

  https://www.youtube.com/watch?v=z8sEIvreP80