

Diretivas Paralelas de Programação

Claudio Schepke, Vinícius Garcia Pinto

- Introdução - Conceitos
- Programação em OpenMP
- Programação em OpenACC
- Fechamento

Introdução - Métricas de Desempenho

Modelo de von Neumann: Entrada/Saída – Memória – Processamento.

Lei de Amdahl: Apenas parte do programa pode ser acelerado.

Paralelismo de laços x Paralelismo de tarefas (etapas de processamento)

Tempo de Processamento: segundos, ...

Ganho de Desempenho (speedup): número de vezes

Eficiência: escala entre 0 e 1, mostrando o quão paralelizável um determinado código possui.

Operações de ponto flutuante por intervalo de tempo: flop.

Introdução - Métricas de Desempenho

Consumo energético: flop / watt

Volume de processamento: +computação através do paralelismo
&& +tamanho do problema == mesmo tempo de processamento.

Contadores de hardware: coletar dados do uso do hardware.

Balanceamento de carga: distribuir o volume de processamento uniformemente entre as unidades de processamento.

Avaliar o tempo de execução paralela: todo o programa x trecho paralelizado.

Coleta de dados da execução: a frio, exclusão da 1ª, intercalado, isoladamente, ...

Estatística: média, mediana, desvio padrão, intervalo de confiança, ...

- **OpenMP** é um dos modelos de programação paralela mais usados
 - foco em processamento Sistemas Multiprocessadores
- relativamente fácil de usar
 - bom modelo para iniciar o aprendizado do paralelismo
 - útil para paralelizar programas já existentes
- suporta (oficialmente) C, C++ e Fortran

The logo for OpenMP, featuring the word "OpenMP" in a bold, teal-colored sans-serif font. A horizontal teal line is positioned below the text, starting from the left and ending under the "P". A small registered trademark symbol (®) is located to the right of the "P".

OpenMP – Open Multi-Processing

- é uma API para escrever aplicações paralelas
 - na forma de uma especificação aberta gerenciada pela OpenMP ARB (Architecture Review Boards)
- suportada por diversas ferramentas/compiladores
 - simplifica a escrita de programas multithread
 - e.g., em comparação com pthreads
- v1.0 no final dos anos 1990 pra Fortran e C
- v6.0 lançada em novembro de 2024
- o suporte é incremental (gcc e clang suportam plenamente apenas 4.5)
 - HW: arquiteturas NUMA, GPUs
 - SW: paralelismo de tarefas, novos mecanismos de sincronização
- <https://www.openmp.org/>

Visão Geral

- API
 - diretivas de compilador
 - biblioteca de rotinas
 - variáveis de ambiente
- Pilha do OpenMP (camadas)

Usuário	Aplicação do Usuário Final
Programação	Diretivas, Biblioteca, Variáveis de ambiente
Sistema	OpenMP Runtime Library (<code>libgomp</code> , <code>libomp</code>) Suporte do SO para memória compartilhada e <i>threads</i>
Hardware	Processadores/núcleos, Memória

OpenMP – Ola Mundo

- Quantas novas linhas de código são necessárias pra paralelizar este programa abaixo?

```
#include<stdio.h>

int main (){

    printf("Ola Mundo!\n");

}
```

OpenMP – Ola Mundo (agora paralelo)

- Quantas novas linhas de código são necessárias pra paralelizar este programa abaixo? **2!**

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

int main (){
    #pragma omp parallel
    printf("Ola Mundo!\n");
}
```

OpenMP – Ola Mundo (agora paralelo)

- Vejamos uma possível saída
 - executando com 4 *threads*

```
Ola Mundo!  
Ola Mundo!  
Ola Mundo!  
Ola Mundo!
```

Sintaxe / Código

- protótipos de função, tipos, etc
 - `#include<omp.h>`
- construções OpenMP fazem (extenso) uso de diretivas de compilação
 - `#pragma omp construct [clause [clause]...]`
 - e.g., `pragma omp parallel private(var1) shared(var2)`
 - associadas a blocos estruturados
 - i.e., `{ e }`

Compilação

- precisamos de um compilador com suporte à OpenMP
 - e.g., gcc, clang
- ligar a biblioteca
 - e.g., -fopenmp

Execução

- e.g, ./ola-mundo
 - roda em paralelo com n threads
 - onde n é o número de CPUs virtuais disponíveis no sistema
- usuário pode controlar o número de *threads* com a variável de ambiente OMP_NUM_THREADS
 - também há outras maneiras

Exemplo com GCC e bash

```
gcc ola-mundo.c -fopenmp -o ola-mundo-paralelo  
export OMP_NUM_THREADS=6  
./ola-mundo-paralelo
```

```
Ola Mundo!  
Ola Mundo!  
Ola Mundo!  
Ola Mundo!  
Ola Mundo!  
Ola Mundo!
```

Algumas das funções da biblioteca OpenMP

- inclusão da biblioteca

```
#include <omp.h>
```

Algumas das funções da biblioteca OpenMP

- inclusão da biblioteca

```
#include <omp.h>
```

- recuperar o identificador da *thread*

```
int omp_get_thread_num();
```

Algumas das funções da biblioteca OpenMP

- inclusão da biblioteca

```
#include <omp.h>
```

- recuperar o identificador da *thread*

```
int omp_get_thread_num();
```

- setar o número de *threads* a ser usado na região paralela

```
int omp_set_num_threads(int n_threads);
```

Algumas das funções da biblioteca OpenMP

- inclusão da biblioteca

```
#include <omp.h>
```

- recuperar o identificador da *thread*

```
int omp_get_thread_num();
```

- setar o número de *threads* a ser usado na região paralela

```
int omp_set_num_threads(int n_threads);
```

- recuperar o número de *threads* que estão executando

```
int omp_get_num_threads();
```

Algumas das diretivas de compilador do OpenMP

- criar uma região paralela

```
#pragma omp parallel  
{  
  
}
```

Algumas das diretivas de compilador do OpenMP

- criar uma região paralela

```
#pragma omp parallel  
{  
  
}
```

- criar uma região paralela definindo o escopo de algumas variáveis

```
#pragma omp parallel private(var1) shared(var2)  
{  
  
}
```

Algumas das diretivas de compilador do OpenMP

- comando(s) que apenas uma das *threads* deve executar

```
#pragma omp single  
{  
  
}
```

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

int main (){
    int id, total;
    #pragma omp parallel
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

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

int main (){
    int id, total;
    #pragma omp parallel
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

- Qual o problema deste código?

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

int main (){
    int id, total;
    #pragma omp parallel
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

- Qual o problema deste código?
 - acesso concorrente às variáveis id e total

Escopo das variáveis

- gerenciar o acesso (concorrente) às variáveis é um desafio
- em programas sequenciais há apenas variáveis globais (ao programa inteiro) ou locais (à função/região)
 - reflexo na acessibilidade e no tempo de vida
- com paralelismo temos mais níveis de escopo e tempo de vida
 - variáveis privadas à uma *thread*
 - variáveis compartilhadas entre as *threads* mas locais a uma região do programa
 - variáveis que em certo momento são privadas mas que ao final são reduzidas a valor único compartilhado

Escopo das variáveis – comportamento padrão em OpenMP

- são compartilhadas as variáveis declaradas fora da região paralela, estáticas e alocadas dinamicamente
 - i.e., há somente uma instância da variável
- são privadas as variáveis declaradas dentro da região paralela (e aquelas de funções chamadas a partir desta região)
 - i.e., há uma instância para cada *thread*
- comportamento padrão pode ser modificado
 - **shared**
 - **private**
 - **firstprivate**

OPENMP – OLA MUNDO 2.0 (CORRIGIDO)

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

int main (){
    int id, total;
    #pragma omp parallel shared(total) private(id)
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

- Agora o código está corrigido!

OPENMP – OLA MUNDO 2.0 (CORRIGIDO)

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

int main (){
    int id, total;
    #pragma omp parallel shared(total) private(id)
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

- Agora o código está corrigido!
- Se o código for executado 10x, a saída será sempre a mesma?

OPENMP – OLA MUNDO 2.0 (CORRIGIDO)

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

int main (){
    int id, total;
    #pragma omp parallel shared(total) private(id)
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

- Agora o código está corrigido!
- Se o código for executado 10x, a saída será sempre a mesma?
- Quantas vezes Tchau aparece?

OPENMP – OLA MUNDO 2.0 (CORRIGIDO)

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

int main (){
    int id, total;
    #pragma omp parallel shared(total) private(id)
    {
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!\n");
}
```

- Agora o código está corrigido!
- Se o código for executado 10x, a saída será sempre a mesma?
- Quantas vezes Tchau aparece?
- Tchau pode aparecer no meio dos Olas?

- programa começa executando a função `main` sequencialmente
 - como qualquer programa C
- a diretiva **parallel** cria uma região paralela
 - quanto o código atinge tal diretiva
 1. *threads* são criadas
 2. cada *thread* executa o código associado à região paralela de maneira independente
 3. há uma barreira implícita ao final da região paralela
- ao longo da execução do programa, várias regiões paralelas podem ser criadas e destruídas
 - modelo conhecido como *Fork-Join*

OPENMP – ESCOPO DAS VARIÁVEIS (EXEMPLO `private` E `shared`)

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

int main (){
    int id = -99, total = -98;
    #pragma omp parallel shared(total) private(id)
    {
        printf("id vale %d, total vale %d \n", id, total);
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!, id vale %d, total vale %d \n", id,
        total);
}
```

```
id vale 31883, total vale -
98
id vale 31883, total vale -
98
id vale 31883, total vale -
98
id vale 31883, total vale -
98
Ola Mundo! (2 de 4)
Ola Mundo! (3 de 4)
Ola Mundo! (0 de 4)
Ola Mundo! (1 de 4)
Tchau!, id vale -
99, total vale 4
```

OPENMP – ESCOPO DAS VARIÁVEIS (EXEMPLO firstprivate)

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

int main (){
    int id = -99, total = -98;
    #pragma omp parallel shared(total) firstprivate(id)
    {
        printf("id vale %d, total vale %d \n", id, total);
        id = omp_get_thread_num();

        #pragma omp single
        total = omp_get_num_threads();

        printf("Ola Mundo! (%d de %d) \n", id, total);
    }
    printf("Tchau!, id vale %d, total vale %d \n", id,
        total);
}
```

```
id vale -
99, total vale -98
id vale -
99, total vale -98
id vale -
99, total vale -98
id vale -
99, total vale -98
Ola Mundo! (3 de 4)
Ola Mundo! (2 de 4)
Ola Mundo! (0 de 4)
Ola Mundo! (1 de 4)
Tchau!, id vale -
99, total vale 4
```

- o código da região paralela é executado igualmente por todas as *threads*
 - nem sempre faz sentido ou é necessário
- diretivas **single**, **master** e **masked**
 - restringe a execução
 - à qualquer *thread* com **single** (com barreira implícita)
 - à *thread* com ID 0 com **master** (sem barreira implícita)
 - à algumas *threads* com **masked** se usada com **filter**
- diretiva **critical**
 - restringe a execução dos comandos à uma *thread* por vez
 - i.e., delimita uma região crítica

OPENMP – DIVISÃO/RESTRIÇÃO DE TRABALHO

```
#include<stdio.h>
#include<omp.h>
int main (){

#pragma omp parallel num_threads(50)
{
    printf("Thread %d diz Ola \n", omp_get_thread_num());

#pragma omp master
    printf("Thread Id=%d, apenas a master deve ter mostrado isso\n", omp_get_thread_num());

#pragma omp masked
    printf("Thread Id=%d, apenas a zero deve ter mostrado isso\n", omp_get_thread_num());

#pragma omp masked filter(3)
    printf("Thread Id=%d, apenas a thread 3 deve ter mostrado isso\n", omp_get_thread_num());

#pragma omp masked filter(omp_get_thread_num()%3)
    printf(">> Thread Id=%d, apenas algumas threads devem ter mostrado isso\n", omp_get_thread_num());

}
}
```

OPENMP – DIVISÃO/RESTRIÇÃO DE TRABALHO

- paralelismo é frequentemente empregado em programas que possuem operações custosas dentro de laços de repetição
- diretiva **omp for** divide as iterações entre as *threads*
 - não pode existir dependência entre elas

```
#pragma omp parallel private(id) num_threads(4)
{
    id = omp_get_thread_num();
    printf("Thread %d executando!\n", id);

    #pragma omp for
    for(int i = 0; i < MAX; i++)
        v[i] = id;
}
for(int i = 0; i < MAX; i++)
    printf("%d ", v[i]);
```

- cláusula opcional **schedule** permite controlar a distribuição das iterações entre as *threads*

```
#pragma omp for schedule(static, 3)
for(int i = 0; i < MAX; i++)
    v[i] = id;
```

- cláusula **reduction** permite combinar globalmente as soluções parciais calculadas localmente por cada *thread*
 - cada *thread* possui uma instância privada da variável
 - atualizações parciais sempre ocorrem nesta instância
 - ao final, cada *thread* incorpora o valor local à variável original
 - de maneira segura e sem necessidade de sincronização explícita

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

- diretiva task oferece outra estratégia de divisão de trabalho
 - tarefas são trechos de código que podem executar em paralelo
 - tarefas são executadas por *threads*
 - desvincula o algoritmo da plataforma
 - usualmente temos muito mais tarefas do que *threads*
- cada tarefa possui um ambiente de dados
- tarefas podem executar em qualquer ordem em qualquer *thread*
 - podemos adicionar pontos de sincronização
- algumas cláusulas opcionais
 - `if`
 - `priority`

OPENMP – DIVISÃO/RESTRIÇÃO DE TRABALHO

```
1  #pragma omp task
2  {
3      printf("tarefa A!! \n");
4      x = 123;
5  }
6  #pragma omp task
7  {
8      printf("tarefa B!! \n");
9      y = -10;
10 }
11 printf("outra mensagem qualquer\n");
12 #pragma omp taskwait
13 printf("as duas tarefas acabaram e x*y=%d\n", x*y);
```

OPENMP – DIVISÃO/RESTRIÇÃO DE TRABALHO

- cláusula **depend** permite especificar o modo de acesso aos dados compartilhados
 - permite sincronizações finas

```
1  #pragma omp task depend(out:x)
2  {
3      printf("tarefa A!! \n");
4      x = 123;
5  }
6  #pragma omp task depend(out:y)
7  {
8      printf("tarefa B!! \n");
9      y = -10;
10 }
11 printf("outra mensagem qualquer\n");
12 #pragma omp task depend(in:x) depend(in:y)
13 printf("as duas tarefas acabaram e x*y=%d\n", x*y);
```

OPENMP – EXEMPLO COM parallel for (SELECTIONSORT)

```
1 // código completo em: https://gitlab.com/viniciusvgp/exemplos-openmp.git
2 void selection_sort(int *v, int n){
3     int i, j, min, min_local, tmp;
4     for(i = 0; i < n - 1; i++){
5         #pragma omp parallel default(shared) private(j, min_local)
6         {
7             min_local = i;
8             #pragma omp single
9             min = i;
10
11             #pragma omp for
12             for(j = i + 1; j < n; j++){
13                 if(v[j] < v[min_local])
14                     min_local = j;
15
16                 #pragma omp critical
17                 {
18                     if(v[min_local] < v[min])
19                         min = min_local;
20                 }
21             }
22             tmp = v[i];
23             v[i] = v[min];
24             v[min] = tmp;
25         }
26     }
```

OPENMP – EXEMPLO COM task (MERGESORT)

```
1 // código completo em: https://gitlab.com/viniciusvgp/exemplos-openmp.git
2 void merge_sort(int vetor[], int tam) {
3     int metade = tam / 2;
4     if (tam > 1) {
5         #pragma omp task if(metade > MIN_PAR)
6         merge_sort(vetor, metade);
7
8         #pragma omp task if(metade > MIN_PAR)
9         merge_sort(vetor + metade, tam - metade);
10
11         #pragma omp taskwait
12         merge(vetor, tam);
13     }
14     return;
15 }
```

OPENMP – EXEMPLO COM task depend (FATORAÇÃO DE CHOLESKY)

```
1 // código completo em: https://gitlab.com/viniciusvgp/exemplos-openmp.git
2 for(k = 0; k < NumBlocos; k++) {
3     p = MAX_PRIO - 2*NumBlocos - 2*k;
4     #pragma omp task depend(inout: Akk[0:TamBloco]) firstprivate(Akk)
        priority(p)
5     potrf(Akk, OrdemBloco);
6     for(i = k+1; i < NumBlocos; i++)
7         p = MAX_PRIO - 2*NumBlocos - 2*k - i;
8         #pragma omp task depend(in: Akk[0:TamBloco]) depend(inout: Aik[0:
            TamBloco]) firstprivate(Akk, Aik) priority(p)
9         trsm(Akk, Aik, OrdemBloco);
10    for(i = k+1; i < NumBlocos; i++) {
11        p = MAX_PRIO - 2*NumBlocos - 2*k - i;
12        #pragma omp task depend(in: Aik[0:TamBloco]) depend(inout: Aii[0:
            TamBloco]) firstprivate(Aii, Aik) priority(p)
13        syrk(Aik, Aii, OrdemBloco);
14        for(j = k+1; j < i; j++) // continua...
```

OPENACC DIRECTIVES

- The parallel directive
- The kernels directive
- The loop directive
- Fundamental differences between the kernels and parallel directive
- Expressing parallelism in OpenACC

OPENACC SYNTAX

OPENACC SYNTAX

- Syntax for using OpenACC directives in code

C/C++

```
#pragma acc directive clauses  
<code>
```

Fortran

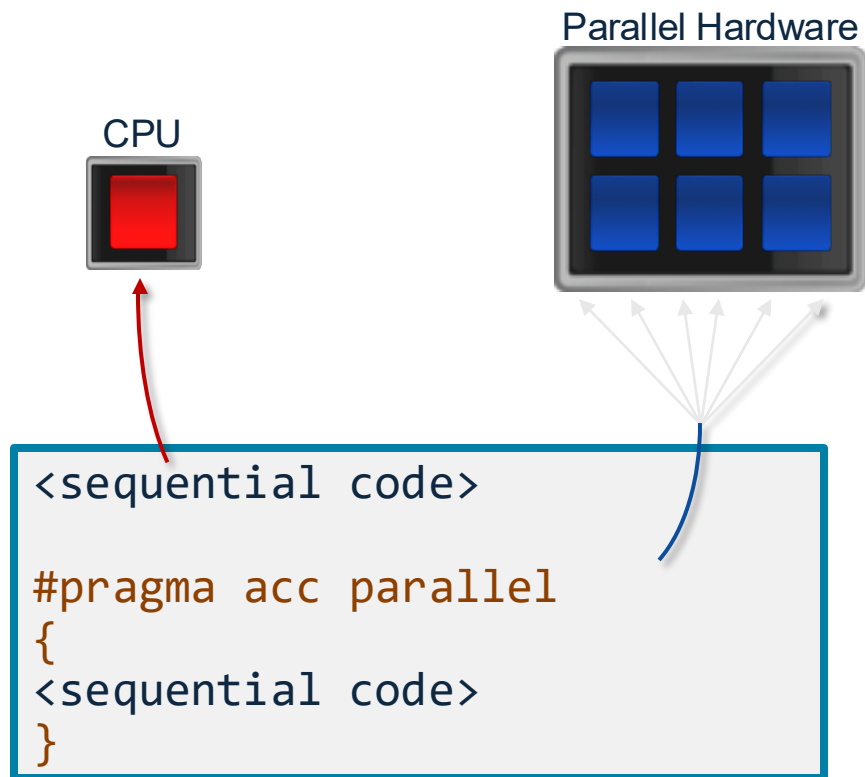
```
!$acc directive clauses  
<code>
```

- A ***pragma*** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.
- A ***directive*** in Fortran is a specially formatted comment that likewise instructions the compiler in its compilation of the code and can be freely ignored.
- “***acc***” informs the compiler that what will come is an OpenACC directive
- ***Directives*** are commands in OpenACC for altering our code.
- ***Clauses*** are specifiers or additions to directives.

OPENACC PARALLEL DIRECTIVE

OPENACC PARALLEL DIRECTIVE

- Explicit programming



- The parallel directive instructs the compiler to create parallel *gangs* on the accelerator
- Gangs are independent groups of worker threads on the accelerator
- The code contained within a parallel directive is executed redundantly by all parallel gangs

OPENACC PARALLEL DIRECTIVE

- Expressing parallelism

```
#pragma acc parallel  
{
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

```
}
```



OPENACC PARALLEL DIRECTIVE

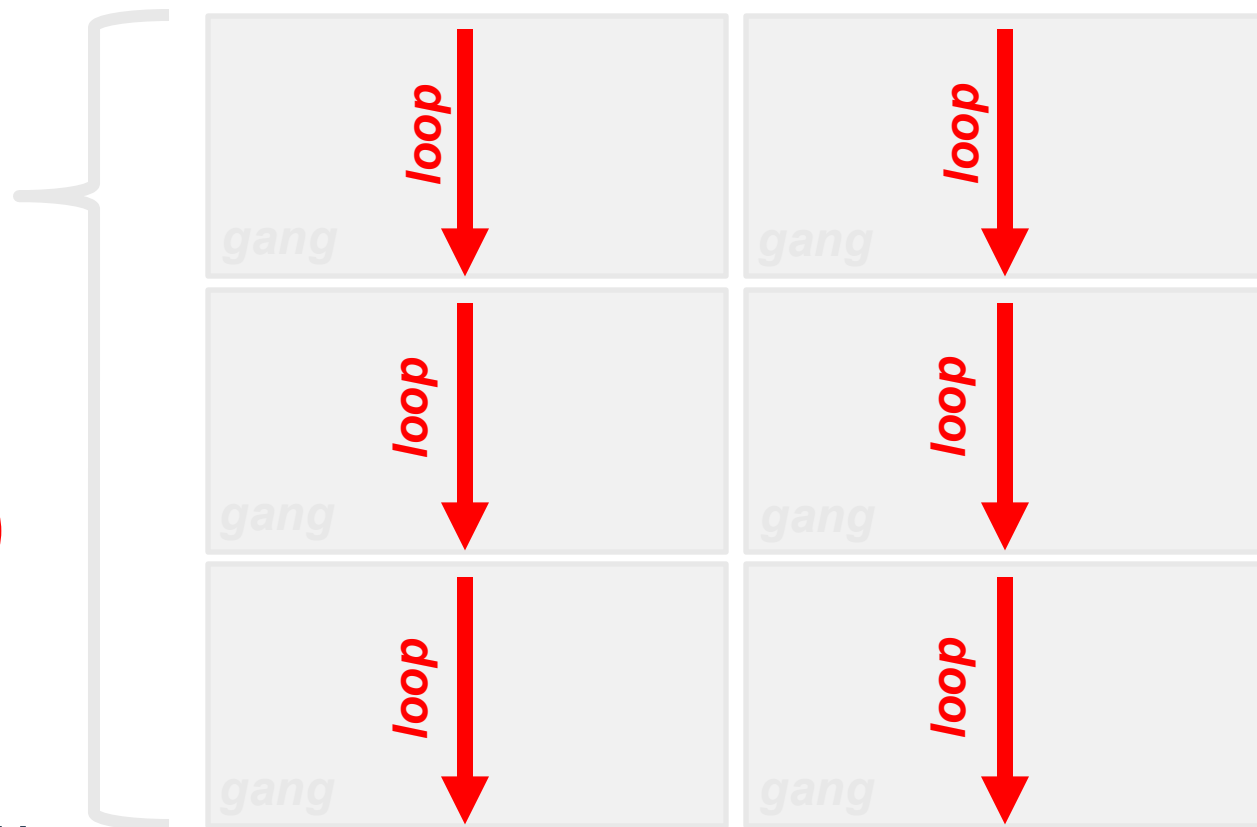
- Expressing parallelism

```
#pragma acc parallel  
{
```

```
    loop  
    for(int i = 0; i < N; i++)  
    {  
        // Do Something  
    }
```

```
}
```

This loop will be
executed redundantly
on each gang



OPENACC PARALLEL DIRECTIVE

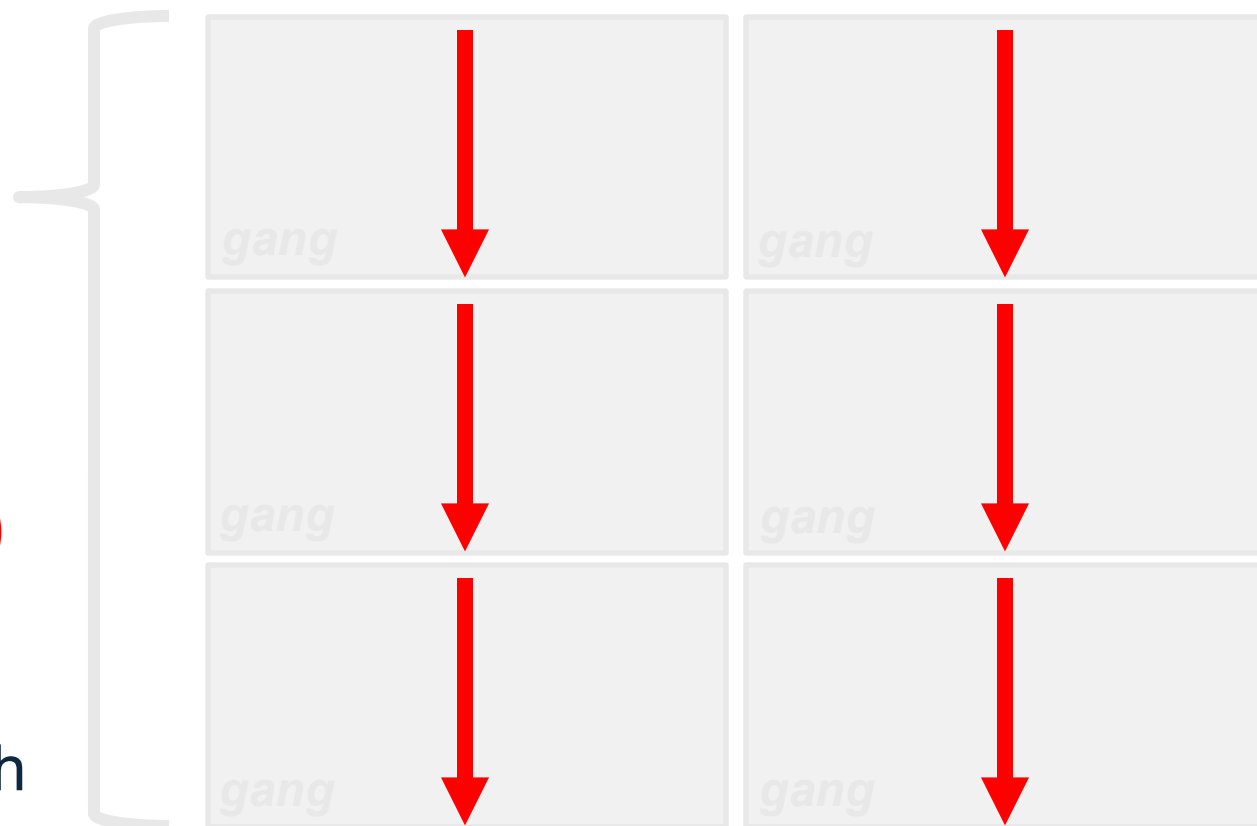
- Expressing parallelism

```
#pragma acc parallel  
{
```

```
  for(int i = 0; i < N; i++)  
  {  
    // Do Something  
  }
```

```
}
```

This means that each
gang will execute the
entire loop



OPENACC PARALLEL DIRECTIVE

- Parallelizing a single loop

C/C++

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < N; i++)
        a[i] = 0;
}
```

Fortran

```
!$acc parallel
!$acc loop
do i = 1, N
    a(i) = 0
end do
!$acc end parallel
```

- Use a **parallel** directive to mark a region of code where you want parallel execution to occur
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran
- The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

OPENACC PARALLEL DIRECTIVE

- Parallelizing a single loop

C/C++

```
#pragma acc parallel loop  
for(int i = 0; i < N; i++)  
    a[i] = 0;
```

Fortran

```
!$acc parallel loop  
do i = 1, N  
    a(i) = 0  
end do
```

- This pattern is so common that you can do all of this in a single line of code
- In this example, the parallel loop directive applies to the next loop
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency, parallel loop may produce incorrect results

OPENACC PARALLEL DIRECTIVE

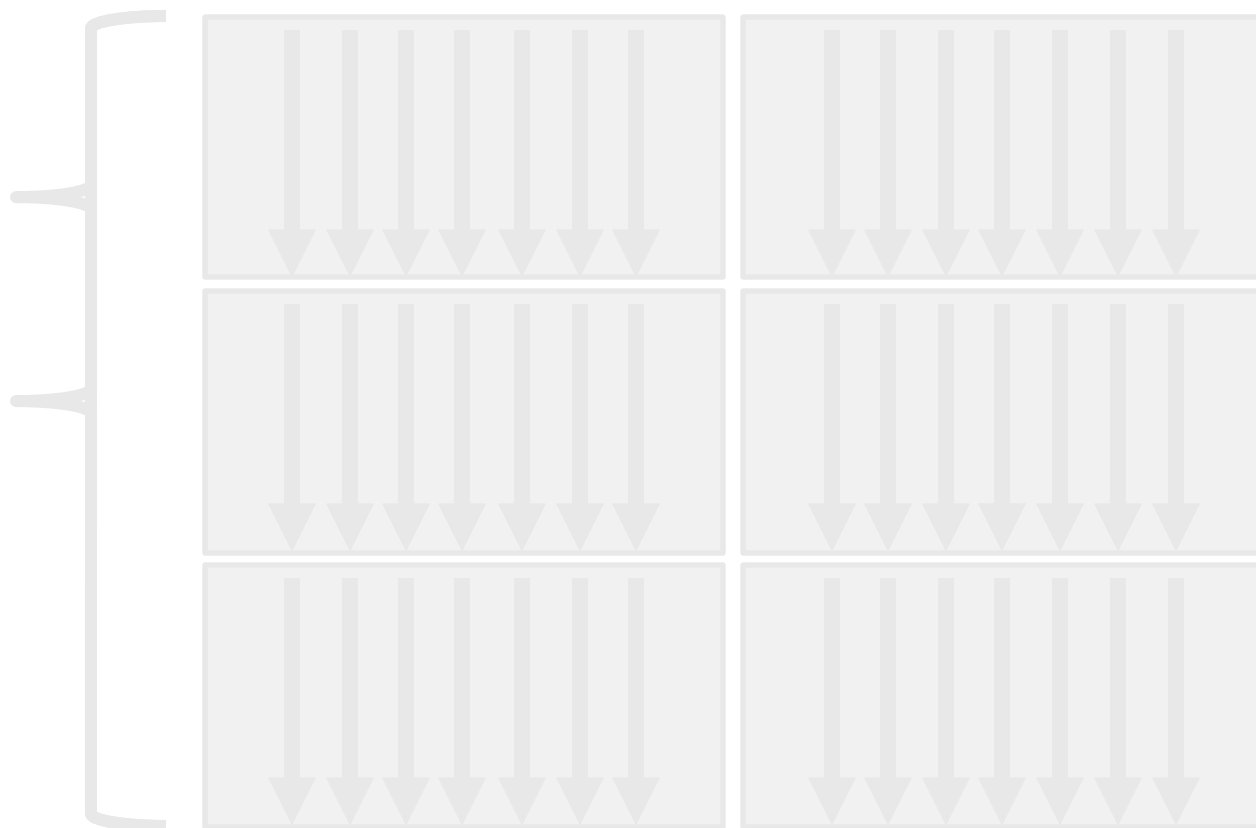
- Expressing parallelism

```
#pragma acc parallel  
{
```

```
    #pragma acc loop  
    for(int i = 0; i < N; i++)  
    {  
        // Do Something  
    }
```

```
}
```

The *loop* directive informs the compiler which loops to parallelize.



OPENACC PARALLEL DIRECTIVE

- Parallelizing many loops

```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
    a[i] = 0;

#pragma acc parallel loop
for(int j = 0; j < M; j++)
    b[j] = 0;
```

- To parallelize multiple loops, each loop should be accompanied by a parallel directive
- Each parallel loop can have different loop boundaries and loop optimizations
- Each parallel loop can be parallelized in a different way
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results

OPENACC LOOP DIRECTIVE

OPENACC LOOP DIRECTIVE

- Expressing parallelism
- Mark a single for loop for parallelization
- Allows the programmer to give additional information and/or optimizations about the loop
- Provides many different ways to describe the type of parallelism to apply to the loop
- Must be contained within an OpenACC compute region (either a kernels or a parallel region) to parallelize loops

C/C++

```
#pragma acc loop  
for(int i = 0; i < N; i++)  
    // Do something
```

Fortran

```
!$acc loop  
do i = 1, N  
    ! Do something
```

OPENACC LOOP DIRECTIVE

- Inside of a parallel compute region

```
#pragma acc parallel
{
    for(int i = 0; i < N; i++)
        a[i] = 0;

    #pragma acc loop
    for(int j = 0; j < N; j++)
        a[i]++;
}
```

- In this example, the first loop is not marked with the loop directive
- This means that the loop will be “redundantly parallelized”
- Redundant parallelization, in this case, means that the loop will be run in its entirety, multiple times, by the parallel hardware
- The second loop is marked with the loop directive, meaning that the loop iterations will be properly split across the parallel hardware

OPENACC LOOP DIRECTIVE

- Parallelizing loop nests

C/C++

```
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    #pragma acc loop
    for(int j = 0; j < M; j++){
        a[i][j] = 0;
    }
}
```

Fortran

```
!$acc parallel loop
do i = 1, N
    !$acc loop
    do i = 1, M
        a(i,j) = 0
    end do
end do
```

- You are able to include multiple loop directives to parallelize multi-dimensional loop nests
- On some parallel hardware, this will allow you to express more levels of parallelism, and increase performance further
- Other parallel hardware has difficulties expressing enough parallelism for multi-dimensional loops
- In this case, inner loop directives may be ignored

REDUCTION CLAUSE

- The inner-most loop is not parallelizable
- If we attempted to parallelize it without any changes, multiple threads could attempt to write to `c[i][j]`
- When multiple threads try to write to the same place in memory simultaneously, we should expect to receive erroneous results
- To fix this, we should use the **reduction clause**

```
for( i = 0; i < size; i++ )  
    for( j = 0; j < size; j++ )  
        for( k = 0; k < size; k++ )  
            c[i][j] += a[i][k] * b[k][j];
```

WITHOUT A REDUCTION

```
#pragma acc parallel loop
for( k = 0; k < size; k++ )
    c[i][j] += a[i][k] * b[k][j];
```



When running this loop **sequentially**, the loop iterations will “**take turns**” writing to `c[i][j]`

When running this loop in **parallel**, we cannot guarantee that the threads will “**take turns**”

REDUCTION CLAUSE

- The **reduction** clause is used when taking many values and “reducing” it to a single value such as in a summation
- Each thread will have their own private copy of the reduction variable and perform a partial reduction on the loop iterations that they compute
- After the loop, the reduction clause will perform a final reduction to produce a **single global result**

```
for( i = 0; i < size; i++ )  
    for( j = 0; j < size; j++ )  
        for( k = 0; k < size; k++ )  
            c[i][j] += a[i][k] * b[k][j];
```

```
for( i = 0; i < size; i++ )  
    for( j = 0; j < size; j++ )  
        double tmp = 0.0f;  
        #pragma parallel acc loop \  
            reduction(+:tmp)  
        for( k = 0; k < size; k++ )  
            tmp += a[i][k] * b[k][j];  
        c[i][j] = tmp;
```

REDUCTION CLAUSE

- The compiler is often very good at detecting when a reduction is needed so the clause may be optional
- May be more applicable to the parallel directive (depending on the compiler)

```
for( i = 0; i < size; i++ )  
    for( j = 0; j < size; j++ )  
        double tmp = 0.0f;  
        #pragma parallel acc loop \  
            reduction(+:tmp)  
            for( k = 0; k < size; k++ )  
                tmp += a[i][k] * b[k][j];  
        c[i][j] = tmp;
```

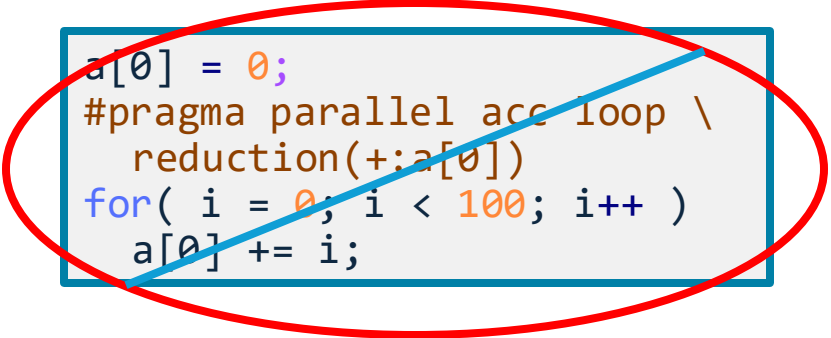
REDUCTION CLAUSE OPERATORS

Operator	Description	Example
<code>+</code>	Addition/Summation	<code>reduction(+:sum)</code>
<code>*</code>	Multiplication/Product	<code>reduction(*:product)</code>
<code>max</code>	Maximum value	<code>reduction(max:maximum)</code>
<code>min</code>	Minimum value	<code>reduction(min:minimum)</code>
<code>&</code>	Bitwise and	<code>reduction(&:val)</code>
<code> </code>	Bitwise or	<code>reduction(:val)</code>
<code>&&</code>	Logical and	<code>reduction(&&:val)</code>
<code> </code>	Logical or	<code>reduction(:val)</code>

REDUCTION CLAUSE

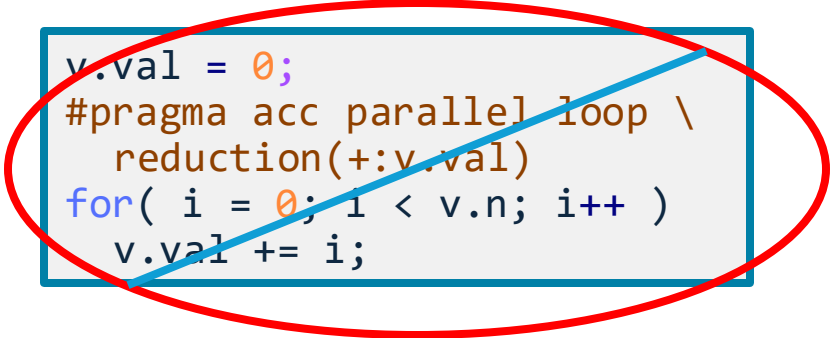
- Restrictions

- The reduction variable may not be an array element



```
a[0] = 0;  
#pragma parallel acc loop \  
reduction(+:a[0])  
for( i = 0; i < 100; i++ )  
a[0] += i;
```

- The reduction variable may not be a C struct member, a C++ class or struct member, or a Fortran derived type member

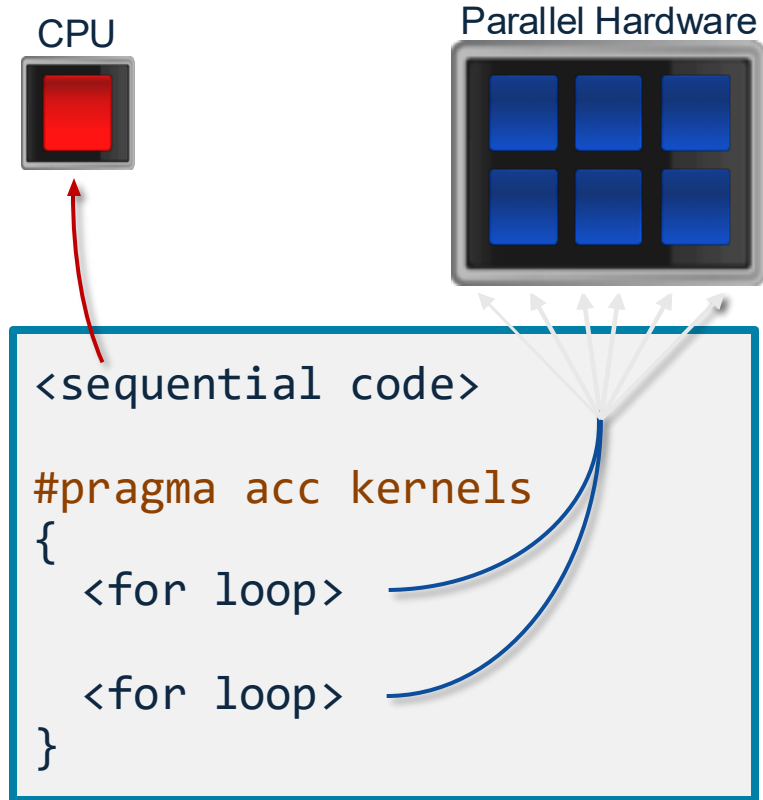


```
v.val = 0;  
#pragma acc parallel loop \  
reduction(+:v.val)  
for( i = 0; i < v.n; i++ )  
v.val += i;
```

OPENACC KERNELS DIRECTIVE

OPENACC KERNELS DIRECTIVE

- Compiler directed parallelization



- The kernels directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe and profitable to do so
- The kernels directive can be applied to regions containing multiple loop nests

OPENACC KERNELS DIRECTIVE

- Parallelizing a single loop

C/C++

```
#pragma acc kernels  
for(int i = 0; j < N; i++)  
    a[i] = 0;
```

Fortran

```
!$acc kernels  
do i = 1, N  
    a(i) = 0  
end do  
!$acc end kernels
```

- In this example, the kernels directive applies to the next for loop
- The compiler will take the loop, and attempt to parallelize it on the parallel hardware
- The compiler will also attempt to optimize the loop
- If the compiler decides that the loop is not parallelizable, it will not parallelize the loop

OPENACC KERNELS DIRECTIVE

- Parallelizing many loops

C/C++

```
#pragma acc kernels
{
    for(int i = 0; i < N; i++)
        a[i] = 0;

    for(int j = 0; j < M; j++)
        b[j] = 0;
}
```

Fortran

```
!$acc kernels
do i = 1, N
    a(i) = 0
end do

do j = 1, M
    b(j) = 0
end do
!$acc end kernels
```

- In this example, we mark a region of code with the kernels directive
- The kernels region is defined by the **curly braces** in C/C++, and the **!\$acc kernels** and **!\$acc end kernels** in Fortran
- The compiler will attempt to parallelize all loops within the kernels region
- Each loop can be parallelized/optimized in a different way

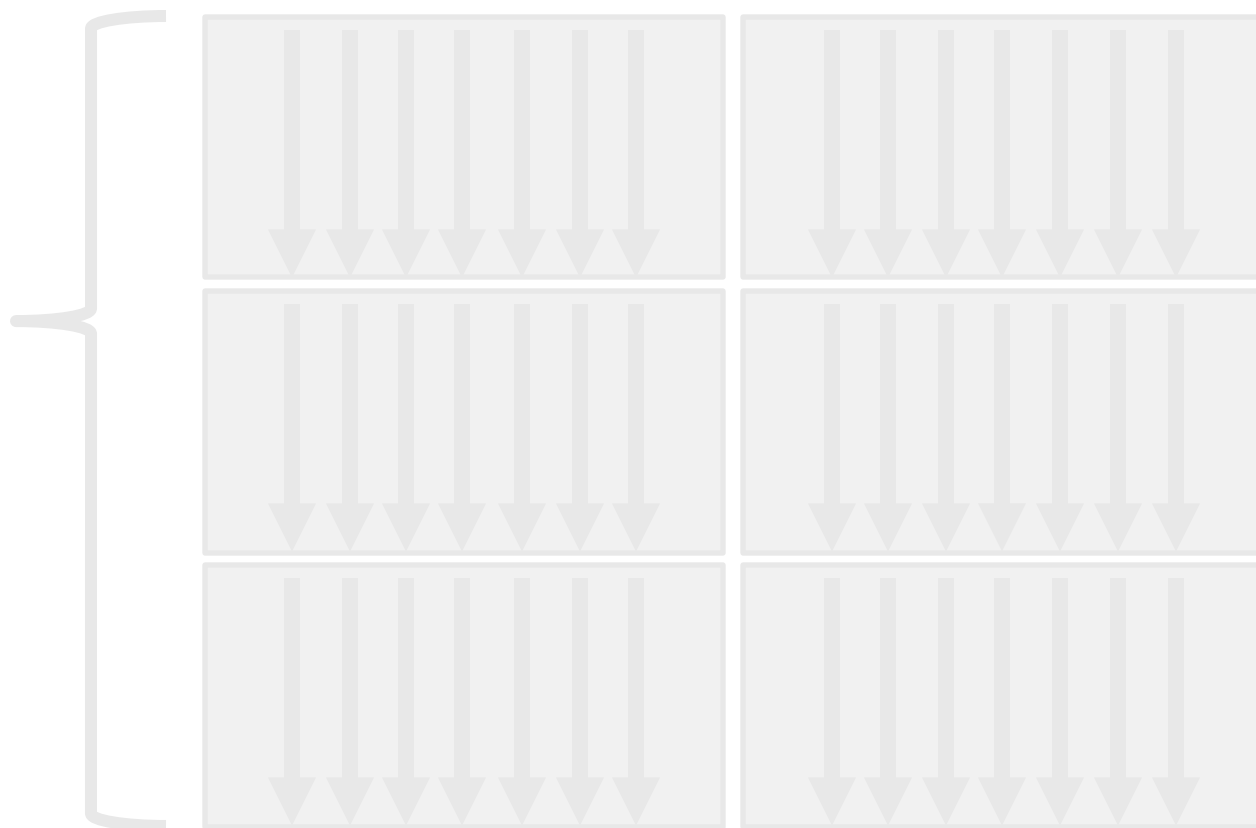
EXPRESSING PARALLELISM

- Compiler generated parallelism

```
#pragma acc kernels
```

```
{  
  
    for(int i = 0; i < N; i++)  
    {  
        // Do Something  
    }  
  
    for(int i = 0; i < M; i++)  
    {  
        // Do Something Else  
    }  
  
}
```

With the *kernels*
directive, the *loop*
directive is implied.



EXPRESSING PARALLELISM

- Compiler generated parallelism

```
#pragma acc kernels
```

```
{
```

```
  for(int i = 0; i < N; i++)
```

```
  {
```

```
    // Do Something
```

```
  }
```

```
  for(int i = 0; i < M; i++)
```

```
  {
```

```
    // Do Something Else
```

```
  }
```

```
}
```

This process can happen
multiple times within the
kernels region.

Each loop can have a different number of gangs, and those gangs can be organized/optimized completely differently.



OPENACC KERNELS DIRECTIVE

- Fortran array syntax

```
!$acc kernels  
a(:) = 1  
b(:) = 2  
c(:) = a(:) + b(:)  
!$acc end kernels
```

```
!$acc parallel loop  
c(:) = a(:) + b(:)
```

- One advantage that the kernels directive has over the parallel directive is Fortran array syntax
- The parallel directive must be paired with the loop directive, and the loop directive does not recognize the array syntax as a loop
- The kernels directive can correctly parallelize the array syntax

KERNELS VS PARALLEL

Kernels

- Compiler decides what to parallelize with direction from user
- Compiler guarantees correctness
- Can cover multiple loop nests

Parallel

- Programmer decides what to parallelize and communicates that to the compiler
- Programmer guarantees correctness
- Must decorate each loop nest

When fully optimized, both will give similar performance.

COMPILING PARALLEL CODE

COMPILING PARALLEL CODE (PGI)

CODE

```
7: #pragma acc parallel loop
8: for(int i = 0; i < N; i++)
9:     a[i] = 0;
```

COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

FEEDBACK

```
main:
    7, Generating Multicore code
    8, #pragma acc loop gang
```

COMPILING PARALLEL CODE (PGI)

CODE

```
7:  #pragma acc kernels
8:  for(int i = 0; i < N; i++)
9:    a[i] = 0;
```

COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

FEEDBACK

```
main:
    8, Loop is parallelizable
      Generating Multicore code
      8, #pragma acc loop gang
```

COMPILING PARALLEL CODE (PGI)

CODE

```
7: #pragma acc kernels
8: for(int i = 1; i < N; i++)
9:   a[i] = a[i-1] + a[i];
```

Non-parallel loop

COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

FEEDBACK

main:

- 8, Loop carried dependence of a-> prevents parallelization
- Loop carried backward dependence of a-> prevents vectorization

COMPILING PARALLEL CODE (PGI)

CODE

```
7: #pragma acc parallel loop
8: for(int i = 1; i < N; i++)
9:   a[i] = a[i-1] + a[i];
```

Non-parallel loop

COMPILING

```
$ pgcc -fast -acc -ta=multicore -Minfo=accel main.c
```

FEEDBACK

```
main:
    7, Generating Multicore code
    8, #pragma acc loop gang
```

KEY CONCEPTS

- By end of this module, you should now understand
 - The parallel, kernels, and loop directives
 - The key differences in functionality and use between the kernels and parallel directives
 - When and where to include loop directives
 - How the parallel and kernel directives conceptually generate parallelism