Programação Concorrente

Referencias - 2018

Vitor Albuquerque de Paula

Indice

Aula 2 (arquiteturas paralelas): 2
Aula 3 (projeto de algoritmos paralelos): 3
Aula 4 (pthreads): 6
Aula 5 (OpenMP): 10
Aula 6 (MPI): 15
Conceitos básicos: 15
Coletivas 15
Conceitos 15
Aula 7 (Comunicação coletiva em MPI): 19
Cuda 23
2. Programming Model 23
2.1. Kernels 24
2.2. Thread Hierarchy 24
2.3. Memory Hierarchy 26
2.4. Heterogeneous Programming 27
2.5. Compute Capability 28
Erlang 30
Códigos Entregues 35
Aula 05 - OpenMP - C/OpenMP - Encontrar Qtde de Primos - ENG COMP 35
Aula 06 - MPI - C/MPI PARA ENG COMP 36
Aula 7 - MPI Coletivas e Spawn - Entrega da Aplicação MPI com coletivas e spawn TA 40
Aula 7 - MPI Coletivas e Spawn - Entrega da Aplicação MPI com coletivas e spawn TB 40
Aula 07 - MPI Coletivas e Spawn - Entrega da Aplicação MPI com coletivas e spawn para ENG COMP 43
Aula 09 - TA - Multiplicação de Matrizes - OpenMP e MPI - TURMA A 43
Aula 09 - TB - Produto Escalar - OpenMP e MPI TURMA B -- FUI MAL 45
Aula 09 - TB - DESCOBRIR - OpenMP e MPI ENG COMP 46
Aula 11 - TA - CUDA 1 - TA 48
Aula 11 - TB - CUDA - TB -- Tirou 6.5 49
Aula 11 - TB - CUDA - ENG COMP 51
Aula 12 - CUDA - Soma Matrizes Quadradas BCC - Tirou 10 52
Aula 12 - CUDA - Adição de 3 matrizes ENG COMP 54
Aula 13 - Erlang - Fibonnaci TA 56
Aula 13 - Erlang - Encontrar o vlr mínimo em uma lista TB -7,5 57
Aula 13 - Erlang - QuickSort - ENG COMP 57
Aula 14 - Erlang - ENG COMP 59

Aula 2 (arquiteturas paralelas):

Computação paralela é caracterizada pelo uso de várias unidades de processamento ou processadores para executar uma computação de forma mais rápida. Ela baseia-se no fato de que o processo de resolução de um problema pode ser divido em tarefas menores, que podem ser realizadas simultaneamente através de algum tipo de coordenação.
Paralelismo de baixo nível são os recursos e técnicas aplicados diretamente em uma CPU, isto é, Paralelismo no nível do Chip que pode ser feito de diversas formas:

- **no nível de instrução**: consiste em emitir múltiplas instruções por ciclo de relógio. Há duas variedades de CPUs: processadores superescalares(podem emitir de 2 a 6 instruções) e processadores VLIW (Palavra de Instrução Muito Longa - podem emitir de acordo com o número de unidade funcional).
    - **Multithreading no Chip** – permite que a CPU gerencie múltiplos threads de controle ao mesmo tempo na tentativa de mascarar as protelações (retardações) que ocorrem quando em uma referencia à memória não encontra nada nos cachês L1 e L2 e o processador fica esperando.
        - **Multithreading de granulação fina:** esse mascara as protelações executando os threads segundo uma política de alternância circular, com thread diferente em ciclos consecutivos.
        - **Multithreading de granulação grossa:** Pipeline muda de thread a cada stall
        - **Multithreading simultâneo:** Cada stall pode se identificado isoladamente.
    - **Multiprocessadores com um único Chip**
        - **Multiprocessadores Homogêneos em um Chip** – é possível colocar duas ou mais CPUs de grande capacidade em um único chip.
            - **Chip com pipeline dual –** Há somente um chip, mas ele tem um segundo pipeline, o que pode dobrar a taxa de execução de instruções.
            - **Chip com dois núcleos:** Há dois ou mais núcleos separados no chip e cada um contem uma CPU completa.
        - **Multiprocessadores heterogêneos em um Chip –** contêm múltiplos núcleos projetados especificamente para aplicações audiovisuais (eletrônicos, TVs, Console de jogos etc.).

- **Co-processadores:** é uma CPU de computador usado para suplementar as funções do microprocessador principal. As operações podem ser de ponto flutuante aritmético, computação gráfica, processamento de sinais, processamento de cadeias de caracteres ou encriptação.
    - **Processadores de rede:** são projetados com múltiplos núcleos heterogêneos. Em função do contexto específico, cada núcleo é responsável por uma parte do processamento de rede
        - **Processamento de pacotes**: Os processadores de rede tratam os pacotes como processamento de entrada e processamento de saída, todos os pacotes passam por esses estágios.
            - **Caminho do pacote**: Verificação de erros de transmissão; Extração do campo; Classificação de pacote; Seleção de

Caminho; Determinação da rede de destino; Consulta de rota; Fragmentação e reconstrução; Computação;Gerenciamento de cabeçalho; Gerenciamento de fila; Geração de soma e verificação; Contabilidade; Coleta de dados estatísticos;

- **Maneiras de aumentar desempenho:** Aumentar a velocidade do clock do processador de rede; Introduzir mais PPEs e paralelismo; Pipeline mais profundo; Adicionar processadores especializados ou ASICs; Adicionar mais barramentos internos; Substituir SDRAM por SRAM

- **Criptoprocessadores –** usam-se co-processadores criptográficos, que possuem uma peça especial que os habilita realizar criptografia com muito mais velocidade que os processadores normais.

- **MULTIPROCESSADORES DE MEMÓRIA COMPARTILHADA:**

  Multiprocessadores versus Multicomputadores – A diferença é a presença ou ausência de memória compartilhada, que interfere no modo como são construídos, projetados e programados. Os multicomputadores são projetos mais fáceis e mais baratos. Os multiprocessadores são difíceis de construir, mas fáceis de programar.

  - **Multiprocessadores**: um computador paralelo no qual todas as CPUs compartilham uma memória comum. São populares pela capacidade de dois ou mais processos se comunicarem lendo e escrevendo na memória. Vantagens: simetria, espaço de endereçamento simples, "cacheamento", coerência, comunicação de memória
    - **UMA:** Uniform Memory Access - Todos os processadores têm acesso uniforme à memória, i.e. Os tempos de acesso são iguais. Computadores conhecidos também por multiprocessador simétrico (SMP); Multi-core.
    - **NUMA:** Non-Uniform Memory Access - O tempo de acesso aos dados depende da sua localização. O acesso local é mais rápido.
    - **COMA:** Cache Only Memory Architecture - As únicas memórias existentes são cache associadas a cada processador. O modelo pode ser visto como um caso particular das máquinas **NUMA.**
  - **Multicomputadores**: conhecido também como SMD (Sistema de Memória Distribuída), trata-se de uma arquitetura paralela, todos os CPU tem sua própria memória privada, acessível somente a eles. A Comunicação entre CPU ocorre por meio de troca de mensagens usando a rede de interconexão, a comunicação entre processos é por meio de software do tipo *send e receive*.

- **Taxonomia de computadores paralelos**
  - **SISD** – faz uma coisa por vez, tem fluxo de dados e um fluxo de instruções;
  - **SIMD** – tem uma única Unidade de Controle que emite uma instrução por vez, mas possui múltiplas ALUs para executá-las
  - **MISD**– Múltiplas instruções operando no mesmo dado (chip com pipeline);
  - **MIMD**– Múltiplas CPUs independentes, operando como um sistema maior;

Aula 3 (projeto de algoritmos paralelos):

Metodologia PCAM: particionamento, comunicação(ênfase na concorrência e escalabilidade), agrupamento e mapeamento(ênfase para localidade e eficiência).

- **Particionamento:** Explora as oportunidades para a execução paralela e identifica como a concorrência da aplicação é caracterizada
  - **Paralelismo de dados:** execução de uma mesma atividade sobre diferentes partes de um conjunto de dados. Os dados determinam a concorrência da aplicação e a forma como o cálculo deve ser distribuído na arquitetura
  - **Paralelismo de tarefa:** execução paralela de diferentes atividades sobre conjuntos distintos de dados. Identificação das atividades concorrentes da aplicação e como essas atividades são distribuídas pelos recursos disponíveis
  - **Análise**:
    - A partição evita a redundância nos cálculos e requisitos de armazenamento dos dados?
    - As tarefas têm todas um tamanho comparável?
    - O número de tarefas é proporcional ao tamanho do problema?
    - É possível identificar mais que uma partição alternativa?
- **Comunicação:** tarefas poderão executar concorrentemente mas podem não ter execução independente. Tipicamente, os cálculos efetuados por uma tarefa precisam de dados associados a outras tarefas (comunicação de dados). À comunicação correspondem duas fases:
  - Define-se a estrutura de canais que ligam, direta ou indiretamente, as tarefas que precisam de dados (consumidoras), com as tarefas que possuem os dados (produtoras)
  - Especificam-se as mensagens que deverão ser enviadas ou recebidas nos canais
  - Requisitos de comunicação e as operações de comunicação necessárias:
    - local/global
    - estruturado/não-estruturado
    - estático/dinâmico
    - assíncrono/síncrono
  - **Análise:**
    - O número de operações realizadas por todas as tarefas é equivalente?
    - Cada tarefa comunica apenas com um número pequeno de tarefas vizinhas?
    - As comunicações podem ser realizadas concorrentemente?
    - As várias computações podem ser realizadas concorrentemente?
- **Agrupamento:** nas duas primeiras fases, criou-se um algoritmo abstrato, não específico para uma máquina. Em alguns casos criar mais tarefas do que o número de processadores pode ser ineficiente.
  - Com o agrupamento pretende-se passar do mundo abstrato para a realidade:
    - Adaptar o algoritmo genérico para uma máquina paralela.

        - Agrupar tarefas em um grupo menor de tarefas de maior tamanho, mais apropriado
- **Mapeamento:** especifica-se onde deverão ser executadas as tarefas Para atingir esse objetivo podem usar-se as seguintes estratégias:
    - As tarefas que podem ser executadas concorrentemente são atribuídas a diferentes processadores, para aumentar a concorrência
    - As tarefas que se comunicam frequentemente são alocadas no mesmo processador, para aumentar a localidade

Exemplos:

- **PCAM + Master-Slave:**
    - **Partição:**são identificadas as tarefas que são geridas pelo master
    - **Comunicação:** Neste tipo de algoritmo a comunicação é essencialmente entre o master e os slave para a envio de tarefas e para o retorno de resultados
    - **Aglomeração:** Visando a redução dos custos de comunicação; podem ser enviadas várias tarefas em cada mensagem
    - **Mapeamento:** O mapeamento é efetuado através da distribuição dos slave pelos processadores. Podem ser colocados vários slave por nodo, incluindo o processador do máster
- **PCAM + Pipeline:**
    - **Partição:** corresponde à identificação das fases de algoritmo que podem ser executadas em paralelo
    - **Comunicação:** é efetuada entre os elementos da cadeia para transmissão dos dados a processar.
    - **Aglomeração:** é efetuada através da junção de elementos consecutivos da cadeia e da junção de vários dados numa só mensagem
    - **Mapeamento:** pode ser regular se as fases do processamento forem de complexidade equivalente
- **Exemplo da aula:**

Particionamento:O particionamento foi feito pensando em separar o arquivo de entrada em um número definido de blocos, em que cada tarefa destinada a isso irá ler cada um esses blocos. Além dessas tarefas, há também a separação de outras tarefas para analisar cada um desses blocos e obter o resultado esperado. Por fim, há a separação de tarefas para pegar cada um dos resultados e escrever no arquivo de saída.

Comunicação: A comunicação é feita em relação a cada bloco do arquivo de entrada. A tarefa que leu o bloco i se comunicará com a tarefa que irá analisar e obter o resultado do bloco i, e, por fim, essa irá se comunicar com a tarefa que irá escrever o resultado da pesquisa no bloco i no arquivo de saída.

Aglomeração: Houve aglomeração de todas as tarefas de leitura do arquivo de entrada em um único processo, em que esse processo irá enviar uma cópia de um bloco do vetor de entrada para cada um dos outros processos. Cada bloco terá tamanho igual ao tamanho da entrada dividido pelo número de processos, exceto ao último processo.

Para o último processo, o tamanho será igual a TAM - (npes-1)*nlocal, em que TAM é o tamanho total do vetor de entrada, npes o número de processos e nlocal o tamanho do bloco para os demais processos. Houve também aglomeração no processo da escrita no

arquivo de saída, em que um só processo recebe e escreve todos os resultados, em ordem.

Mapeamento: A aglomeração de leitura de dados será destinada a um processador só, enquanto cada uma das tarefas de leitura dos blocos do arquivo de entrada, posterior à leitura, é feita em cada processador disponível. Por fim, a cada vez que a procura em um bloco i é concluída, ela já se torna disponível para ser escrita no arquivo de saída, processo que também será destinado a um único processador.

Aula 4 (pthreads):

pthread[_<object>]_<operation>: <operation> describes the operation to be performed and the optional <object> describes the object to which this operation is applied. For functions which are involved in the manipulation of threads, the speci□cation of <object> is omitted.

Data types are named according to the syntax form: **pthread_<object>_t**

- pthread_t: Thread ID
- pthread_mutex_t: Mutex variable
- pthread_cond_t: Condition variable
- pthread_key_t: Access key
- pthread_attr_t: Thread attributes object
- pthread_mutexattr_t: Mutex attributes object
- pthread_condattr_t: Condition variable attributes object
- pthread_once_t: One-time initialization control context

. In this model, the programmer has to partition the program into a suitable number of

user threads which can be executed concurrently with each other. The programmer cannot control the scheduler of the operating system and has only little in□uence on the library scheduler.

Creating and Merging Threads

int **pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start routine)(void *), void *arg)**: the □rst argument is a pointer to an object of type pthread t which is also referredtoasthreadidenti□er(TID); The second argument is a pointer to a previously allocated and initialized attribute object of type pthread_attr_t, the argument value NULL causes the generation of a thread with default attributes; The third argument speci□es the function start routine() which will be executed by the generated thread; The fourth argument is a pointer to the argument value with which the thread function start routine() will be executed

A thread can determine its own thread identi□er by calling the function **pthread_t pthread_self().**

To compare the thread ID of two threads, the function int **pthread_equal (pthread_t t1, pthread_t t2)**, this function returns the value 0 if t1 and t2 do not refer to the same thread.

The maximum number of threads that can be started can be determined by calling **maxThreads = sysconf (_SC_THREAD_THREADS_MAX)**

A thread can terminate it self explicitly by calling the function **void pthread_exit (void *valuep)**, the argument valuep speci□es the value that will be returned to another thread which waits for the termination of this thread using pthread join().

```
#include <pthread.h>

typedef struct {
  int size, row, column;
  double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void *thread_mult (void *w) {
  matrix_type_t *work = (matrix_type_t *) w;
  int i, row = work->row, column = work->column;
  work -> MC[row][column] = 0;
  for (i=0; i < work->size; i++)
    work->MC[row][column] += work->MA[row][i] * work->MB[i][column];
  return NULL;
}

int main() {
  int row, column, size = 8, i;
  double MA[8][8], MB[8][8], MC[8][8];
  matrix_type_t *work;
  pthread_t thread[8*8];
  for (row=0; row<size; row++)
    for (column=0; column<size; column++) {
      work = (matrix_type_t *) malloc (sizeof (matrix_type_t));
      work->size = size;
      work->row = row;
      work->column = column;
      work->MA = MA; work->MB = MB; work->MC = MC;
      pthread_create (&(thread[column + row*8]), NULL,
                      thread_mult, (void *) work);
    }

  for (i=0; i<size*size; i++)
    pthread_join (thread[i], NULL);
}
```

A thread can wait for the termination of another thread by calling the function **int pthread_join (pthread t thread, void **valuep)**: thread speci□es the thread ID of the thread for which the calling thread waits to be terminated; valuep speci□es a memory address where the return value of this thread should be stored. The thread calling pthread join() is blocked until the speci□ed thread has terminated. If several threads wait for the termination of the same thread,using pthread join(), all waiting threads are blocked until the speci□ed thread has terminated. But only one of the waiting threads successfully stores the return value. For all other waiting threads, the return value of pthread_join() is the error value ESRCH.

The preservation of the internal data structure of a thread after its termination can be avoided by calling the function int **pthread_detach (pthread t thread)**.

Pthreads program for the multiplication of two matrices MA and MB. A separate thread is created for each element of the output matrix MC. A separate data structure work is provided for each of the threads created

Thread Coordination with Pthreads

To avoid race conditions, these concurrent accesses must be coordinated. To perform such coordinations, Pthreads provide mutex variables and condition variables.

Mutex Variables

Mutex variable denotes a data structure of the prede□ned opaque type **pthread_mutex_t**, used to ensure mutual exclusion when accessing common data, i.e., it can be ensured that only one thread at a time has exclusive access to a common data structure: locked or unlocked.

Before an access to the common data structure, the accessing thread locks the corresponding mutex variable using a speci□c Pthreads function. After each access to the common data structure, the accessing thread unlocks the corresponding mutex variable.

When a thread A tries to lock a mutex variable that is already owned by another thread B, thread A is blocked until thread B unlocks the mutex variable.

Allocate statically: mutex = PTHREAD MUTEX INITIALIZER

Allocate dynamically: int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutex_attr_t*attr),

If a mutex variable that has been initialized dynamically is no longer needed, it can be destroyed by calling the function **int pthread_mutex_destroy (pthread mutex_t *mutex)**.

A thread can lock a mutex variable mutex by calling the function **int pthread mutex lock (pthread mutex t *mutex).** Which one of the waiting threads is unblocked may depend on their priorities and the scheduling strategies used.

A thread which is owner of a mutex variable mutex can unlock mutex by calling the function **int pthread_mutex_unlock (pthread_mutex_t *mutex)**.

A thread can check without blocking whether a mutex variable is owned by another thread with **int pthread_mutex_trylock (pthread_mutex_t *mutex).** This procedure ensures that it is not possible for different threads to insert new elements at the same time.

```
typedef struct node {
  int index;
  void *data;
  struct node *next;
} node_t;

typedef struct list {
  node_t *first;
  pthread_mutex_t mutex;
} list_t;

void list_init (list_t *listp)
{
  listp->first = NULL;
  pthread_mutex_init (&(listp->mutex), NULL);
}

void list_insert (int newindex, void *newdata, list_t *listp)
{
  node_t *current, *previous, *new;
  int found = FALSE;

  pthread_mutex_lock (&(listp->mutex));
  for (current = previous = listp->first; current != NULL;
       previous = current, current = current->next)
  {
    if (current->index == newindex) {
      found = TRUE; break;
    }
    else
      if (current->index > newindex) break;
  }
  if (!found) {
    new = (node_t *) malloc (sizeof (node_t));
    new->index = newindex;
    new->data = newdata;
    new->next = current;
    if (current == listp->first) lstp->first = new;
    else previous->next = new;
  }
  pthread_mutex_unlock (&(lstp->mutex));
}
```

Pthreads implementation of a linked list. The function list insert() can be called by different threads concurrently which insert new elements into the list. In the form presented, list insert() cannot be used as the start function of a thread, since the function has more than one argument. To be used as start function, the arguments of list insert() have to be put into a new data structure which is then passed as argument. The original arguments could then be extracted from this data structure at the beginning of list insert().

Mutex Variables and Deadlocks

Adeadlock may occur if the threads use a different order for locking the mutex variables. The occurrence of deadlocks can be avoided by using a □xed locking order for all threads or by employing a backoff strategy.

- **□xed locking order**, each thread locks the critical mutex variables always in the same prede□ned order
- **backoff strategy**, each participating thread can lock the mutex variables in its individual order, and it is not necessary to use the same prede□ned order for each thread. But a thread must back off when its attempt to lock a mutex variable fails. In this case, the thread must release all mutex variables that it has previously locked successfully. The use of a backoff strategy typically leads to larger execution times

Condition Variables

A condition variable is an opaque data structure which enables a thread to wait for the occurrence of an arbitrary condition without active waiting. The mutex variable is used to protect the evaluation of the speci□c condition which is waiting to be ful□lled.

Allocate statically:pthread_cond_t cond = PTHREAD_COND_INITIALIZER

Allocate dynamically: int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr): cond is the address of the condition variable to be initialized and attr is the address of an attribute data structure for condition variables.

It can be destroyed by calling the function **int pthread_cond_destroy (pthread_cond_t *cond)**

It is not allowed that different threads associate different mutex variables with a condition variable at the same time. A condition variable should only be used for a single condition to avoid deadlocks or race conditions.

A thread must □rst lock the associated mutex variable mutex with pthread mutex lock() before it can wait for a speci□c condition to be ful□lled using the function **int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex).**

Two cases can occur for this usage pattern for condition variables:

- If the speci□ed condition is ful□lled when executing the code segment from above, the function pthread cond wait() is not called.
- If the speci□ed condition is not ful□lled, pthread cond wait() is called

It should be ensured that a thread which is waiting for a condition variable is woken up only if the speci□ed condition is ful□lled

Pthreads provide two functions to wake up (signal) a thread waiting on a condition variable:

- **int pthread_cond_signal (pthread_cond_t *cond),** wakes up a single thread waiting on the condition
- **int pthread_cond_broadcast (pthread_cond_t *cond),** wakes up all threads waiting on the condition

The advantage of not protecting the call of pthread cond signal() or pthread cond broadcast() by the mutex variable is the chance that the mutex variable may not have an owner when the waiting thread is woken up.

To wait for a condition, Pthreads also provide the function **int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *time);** maximum waiting time is speci□ed using type struct timespec { time_t tv_sec; long tv_nsec; }: tv_sec speci□es the number of seconds and tv_nsec speci□es the number of additional nanoseconds. The time parameter speci□es an absolute clock time rather than a time interval.

Aula 5 (OpenMP):

OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

#pragma omp directive [clause list]: this directive is responsible for creating a group of threads

The clause list is used to specify conditional parallelization, number of threads, and data handling:

- **Conditional Parallelization:** The clause if (scalar expression) determines whether the parallel construct results in creation of threads
- **Degree of Concurrency:** The clause num_threads (integer expression) specifies the number of threads that are created by the parallel directive
- **Data Handling:** The clause private (variable list) indicates that the set of variables specified is local to each thread – i.e., each thread has its own copy of each variable in the list. The clause firstprivate (variable list) is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive. The clause shared (variable list) indicates that all variables in the list are shared across all the threads

Exemplo: #pragma omp parallel if (is_parallel == 1) num_threads(8) \ private (a) shared (b) firstprivate(c)

The usage of the reduction clause is **reduction (operator: variable list)**. This clause performs a reduction on the scalar variables specified in the list using the operator (+, *, -, &, |, ^, &&, and ||).

The **omp_get_num_threads()** function returns the number of threads in the parallel region and the **omp_get_thread_num()**function returns the integer i.d. of each thread.

The for Directive

Is used to split parallel iteration spaces across threads: **#pragma omp for [clause list].** The last iteration of the for loop update the value of a variable.

Exemplo:

```
1    #pragma omp parallel default(private) shared (npoints) \
2                    reduction(+: sum) num_threads(8)
3  {
4      sum=0;
5      #pragma omp for
6      for (i = 0; i < npoints; i++) {
```

```
7        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
8rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
9if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
10(rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
11          sum ++;
12     }
13   }
```

Assigning Iterations to Threads

The schedule clause of the for directive deals with the assignment of iterations to threads: **schedule(scheduling_class[, parameter])**. Supports 4 scheduling classes: static, dynamic, guided, and runtime

- **Static:** schedule(static[, chunk-size]). This technique splits the iteration space into equal chunks of size chunk-size and assigns them to threads in a round-robin fashion.
- **Dynamic:** schedule(dynamic[, chunk-size]). The iteration space is partitioned into chunks given by chunk-size. If no chunk-size is specified, it defaults to a single iteration per chunk.
- **Guided:** schedule(guided[, chunk-size]). The solution to this problem (also referred to as an edge effect) is to reduce the chunk size as we proceed through the computation. This is the principle of the guided scheduling class. The chunk-size refers to the smallest chunk that should be dispatched.
- **Runtime:** In this case the environment variable OMP_SCHEDULE determines the scheduling class and the chunk size.

Clause – **nowait**, which can be used with a for directive to indicate that the threads can proceed to the next statement without waiting for all other threads to complete the for loop execution. We can use the nowait clause to save idling and synchronization.

Exemplo:

```
1      #pragma omp parallel
2      {
3          #pragma omp for nowait
4              for (i = 0; i < nmax; i++)
5                  if (isEqual(name, current_list[i])
6                      processCurrentName(name);
7#pragma omp for
8              for (i = 0; i < mmax; i++)
9                  if (isEqual(name, past_list[i])
10                     processPastName(name);
11     }
```

The sections Directive

OpenMP supports such non-iterative parallel task assignment using the sections directive: **#pragma omp sections [clause list]{[#pragma omp section /*structured block */**

The lastprivate clause, in this case, specifies that the last section (lexically) of the sections directive updates the value of the variable. The nowait clause specifies that there is no implicit synchronization among all threads at the end of the sections directive. Note that it is illegal to branch in and out of section blocks.

Merging Directives

OpenMP allows the programmer to merge the parallel directives to parallel for and parallel sections, re-spectively:**#pragma omp parallel for default (private) shared (n)**

Nesting parallel Directives

Instead of nesting three for directives inside a single parallel directive, we have used three parallel for directives. This is because OpenMP does not allow for, sections, and single directives that bind to the same parallel directive to be nested. To generate a new set of threads, nested parallelism must be enabled using the OMP_NESTED environment variable.

Synchronization Constructs in OpenMP

A barrier is one of the most frequently used synchronization primitives: **#pragma omp barrier.** On encountering this directive, all threads in a team wait until others have caught up, and then release

The single and master Directives:

- A Single directive specifies a structured block that is executed by a single (arbitrary) thread: **#pragma omp single [clause list]**. It can take clauses private, firstprivate, and nowait; the first thread enters the block; If the nowait clause has been specified at the end of the block, then the other threads proceed; This directive is useful for computing global data as well as performing I/O.
- A Master directive is a specialization of the single directive in which only the master thread executes the structured block: **#pragma omp master**

Critical Sections:

OpenMP provides a **critical directive** for implementing critical regions: **#pragma omp critical [(name)]** the optional identifier name can be used to identify a critical region. The critical directive ensures that at any point in the execution of the program, only one thread is within a critical section, all others must wait until it is done before entering the named critical section.

The critical directive is a direct application of the corresponding mutex function in Pthreads. We must reduce the size of the critical sections as much as possible (in terms of execution time) to get good performance. No jumps are permitted into or out of the block.

The **atomic directive**, for incrementing or adding to an integer atomic updates to memory locations.

It is important to note that the atomic directive only atomizes the load and store of the scalar variable. All atomic directives can be replaced by critical directives provided they have the same name. However, the availability of atomic hardware instructions may optimize the performance of the program.

The ordered Directive:

In many circumstances, it is necessary to execute a segment of a parallel loop in the order in which the serial version would execute it: **#pragma omp ordered.**

It must be within the scope of a for or parallel for directive. The for or parallel for directive must have the ordered clause specified. Only a single thread can enter an ordered block when all prior threads have exited.

Exemplo: Computing the cumulative sum of a list using the ordered directive

```
1       cumul_sum[0] = list[0];
2       #pragma omp parallel for private (i) \
3                   shared (cumul_sum, list,
n) ordered 4 for (i = 1; i < n; i++)
5       {
6           /* other processing on list[i] if needed */
8#pragma omp ordered
9           {
10              cumul_sum[i] = cumul_sum[i-1] + list[i];
```

```
11        }
12    }
```

The flush Directive:

Provides a mechanism for making memory consistent across threads. The flush directive provides a memory fence by forcing a variable to be written to or read from the memory system. The flush directive applies only to shared variables. **#pragma omp flush[(list)]**

A flush is implied at a barrier, at the entry and exit of critical, ordered, parallel, parallel for, and parallel sections blocks and at the exit of for, sections, and single blocks. A flush is not implied if a nowait clause is present. It is also not implied at the entry of for, sections, and single blocks and at entry or exit of a master block.

Data Handling in OpenMP:

- **Private:** If a thread initializes and uses a variable (such as loop indices) and no other thread accesses the data, then a local copy of the variable should be made for the thread
- **Firstprivate:** If a thread repeatedly reads a variable that has been initialized earlier in the program, it is beneficial to make a copy of the variable and inherit the value at the time of thread creation
- **Reduction clause:** If multiple threads manipulate a single piece of data, one must explore ways of breaking these manipulations into local operations followed by a single global operation
- **Shared:**remaining data items may be shared

If multiple threads manipulate different parts of a large data structure, the programmer should explore ways of breaking it into smaller data structures and making them private to the thread manipulating them.

Objects persist through parallel and serial blocks provided the number of threads remains the same. In contrast to private variables, these variables are useful for maintaining persistent objects across parallel regions, which would otherwise have to be copied into the master thread's data space and reinitialized at the next parallel block.

#pragma omp threadprivate(variable_list): all variables in variable_list are local to each thread and are initialized once before they are accessed in a parallel region.

Controlling Number of Threads and Processors:

- **void omp_set_num_threads (int num_threads):** sets the default number of threads that will be created
- **omp_get_num_threads** function returns the number of threads participating in a team.
- **omp_get_max_threads** function returns the maximum number of threads that could possibly be created
- **omp_get_thread_num**returns a unique thread i.d. for each thread in a team.
- **omp_get_num_procs** function returns the number of processors that are available to execute the threaded program at that point
- **omp_in_parallel** returns a non-zero value if called from within the scope of a parallel region, and zero otherwise

There are situations where it is more convenient to use an explicit lock. It must be initialized.

- This is done using the omp_init_lock function.
- When a lock is no longer needed, it must be discarded using the function omp_destroy_lock.
- It can be locked and unlocked using the functions omp_set_lock and omp_unset_lock.
- The function omp_test_lock can be used to attempt to set a lock. If the function returns a non-zero value, the lock has been successfully set.
- OpenMP also supports nestable locks that can be locked multiple times by the same thread. The lock object in this case is omp_nest_lock_t and the corresponding functions.

Environment Variables in OpenMP:

OpenMP provides additional environment variables that help control execution of parallel programs.

- **OMP_NUM_THREADS** This environment variable specifies the default number of threads created upon entering a parallel region.
- **OMP_DYNAMIC** This variable, when set to TRUE, allows the number of threads to be controlled at runtime using the omp_set_num threads function or the num_threads clause.
- **OMP_NESTED**This variable, when set to TRUE, enables nested parallelism
- **OMP_SCHEDULE** This environment variable controls the assignment of iteration spaces associated with for directives that use the runtime scheduling class. The variable can take values static, dynamic, and guided along with optional chunk size.

Explicit Threads versus OpenMP Based Programming:

- **Vantagens:** a programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc. This convenience is especially useful when the underlying problem has a static and/or regular task graph.
- **Desvantagens:** artifact of explicit threading is that data exchange is more apparent. Provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization. Tools and support for Pthreads programs is easier to find.

Aula 6 (MPI):

Conceitos básicos:

bloqueante- Espera a resposta da chamada
local - Depende apenas do processo local
coletiva - Comunicação de grupo todos devem participar, mas nao envolve necessariamente a sincronização
portável Tipo de dado que é representado igualmente em qualquer memória
equivalente Tipos construıdos da mesma forma.

Coletivas

• Barrier — Barreira
• Broadcast
• Gather — Recolher
• Scatter — Espalhar
• AllGather — Todos recebem o resultado do Gather
• AllToAll — Todos recebem a parte correspondente da mensagem, que está espalhada nos processadores
• Redução
• Redução com Scatter
• Scan — Varredura (redução progressiva por prefixo)

Conceitos

• Grupo é uma coleção ordenada de processos (ids)
• Contexto é uma propriedade que restringe o escopo de comunicação
• Comunicador é um grupo com contexto definido
• Topologia é uma estrutura virtual de conexão entre os processos (grafo)

Principles of Message-Passing Programming:

There are two key attributes that characterize the message-passing programming paradigm: it assumes a partitioned address space and it supports only explicit parallelization.

Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming.

All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data. This requirement for cooperation adds a great deal of complexity for a number of reasons.

Advantages: programmer is fully aware of all the costs; it can be efficiently implemented on a wide variety of architectures; can often achieve very high performance and scale to a very large number of processes.

Disadvantages: requires that the parallelism is coded explicitly by the programmer; programming using the message-passing paradigm tends to be hard and intellectually demanding

Structure of Message-Passing Programs:

- In the **asynchronous paradigm,** all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about.
- **Loosely synchronous** programs are a good compromise between these two extremes. Tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously.

Most message-passing programs are written using the single program multiple data (SPMD) approach.

Blocking Message Passing Operations

Send operation to return only when it is semantically safe to do so. There are two mechanisms:

- **Blocking Non-Buffered Send/Receive:** the send operation does not return until the matching receive has been encountered at the receiving process. There are no buffers used at either sending or receiving ends. A blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time. idling overhead is one of the major drawbacks of this protocol.
    - Deadlocks in Blocking Non-Buffered Operations: As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined.
- **Blocking Buffered Send/Receive:** simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well.
    - It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads.
    - Deadlocks in Buffered Send and Receive Operations: This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

Returns from the send or receive operation before it is semantically safe to do so. Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation.

Generally accompanied by a check-status operation, which indicates whether the semantics of a previously initiated transfer may be violated or not.

- In the **non-buffered case**, a process wishing to send data to another simply posts a pending message and returns to the user program
  - When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data.
  - the data being received is unsafe for the duration of the receive operation.
- In the **buffered case**, the sender initiates a DMA operation and returns immediately. The data becomes safe the moment the DMA operation has been completed. At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location.
  - Using buffers with nonblocking operation has the effect of reducing the time during which the data is unsafe.

MPI: the Message Passing Interface

int **MPI_Init(int *argc, char ***argv)** and int **MPI_Finalize().** Both MPI_Init and MPI_Finalize must be called by all the processes, otherwise MPI's behavior will be undefined.

Communicators:

A communication domain is a set of processes that are allowed to communicate with each other. Information about communication domains is stored in variables of type MPI_Comm

MPI defines a default communicator called MPI_COMM_WORLD which includes all the processes involved in the parallel execution

Getting Information:

int MPI_Comm_size(MPI_Comm comm, int *size) determine the number of processes

int MPI_Comm_rank(MPI_Comm comm, int *rank): determine the label of the calling process. The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Sending and Receiving Messages:

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Each message has an integer-valued tag associated with it. This is used to distinguish different types of messages. The message-tag can take values ranging from zero up to the MPI defined constant MPI_TAG_UB

After a message has been received, the status variable can be used to get information about the MPI_Recv operation. typedef struct MPI_Status { int MPI_SOURCE; int MPI_TAG; int MPI_ERROR; }; MPI_SOURCE and MPI_TAG store the source and the tag of the received message.

Sending and Receiving Messages Simultaneously:

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

The send and receive buffers must be disjoint, and the source and destination of

the messages can be the same or different

Overlapping Communication with Computation:

- **Non-Blocking Communication Operations:** These functions are MPI_Isend and MPI_Irecv.
  - MPI_Isend starts a send operation but does not complete, that is, it returns before the data is copied out of the buffer. int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  - MPI_Irecv starts a receive operation but returns before the data has been received and copied into the buffer. int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
  - Functions allocate a request object and return a pointer to it in the request variable. This request object is used as an argument in the MPI_Test and MPI_Wait functions to identify the operation whose status we want to query or to wait for its completion.
    - int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status): tests whether or not the non-blocking send or receive operation identified by its request has finished. It returns flag = {true} (non-zero value in C) if it completed
    - int MPI_Wait(MPI_Request *request, MPI_Status *status): blocks until the non-blocking operation identified by request completes.

Aula 7 (Comunicação coletiva em MPI):

Broadcast Operation:

One speci□c process of a group of processes sends the same data block to all other processes of the group: **int MPI_Bcast (void *message, int count, MPI_Datatype type, int root, MPI_Comm comm),** root denotes the process which sends the data block. This process provides the data block to be sent in parameter message. The other processes specify in message their receive buffer. The parameter count denotes the number of elements in the data block, type is the data type of the elements of the data block. MPI Bcast() is a collective communication operation, i.e., each process of the communicator comm must call the MPI Bcast() operation.

Data blocks sent by MPI Bcast() cannot be received by an MPI Recv() operation. The MPI runtime system guarantees that broadcast messages are received in the same order in which they have been sent by the root process.

Collective MPI communication operations are always blocking. For the root process, this means that control can be returned as soon as the message has been copied into a system buffer and the send buffer speci□ed as parameter can be reused. The execution of a collective communication operation does not involve a synchronization of the participating processes.

Reduction Operation:

Each participating process provides a block of data that is combined with the other blocks using a binary reduction operation: **int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)**, This must be an associative operation. MPI provides a number of prede□ned reduction operations which are also commutative:

- MPI_MAX Maximum
- MPI_MIN Minimum
- MPI_SUM Sum
- MPI_PROD Product
- MPI_LAND Logical and
- MPI_BAND Bit-wise and
- MPI_LOR Logical or
- MPI_BOR Bit-wise
- int MPI Op create (MPI User function *function, int commute, MPI Op *op): cria nova função

For an MPI Reduce() operation, all participating processes must specify the same values for the parameters count, type, op, and root.

Exemplo:MPI program for the parallel computation of a scalar product

```
int j, m, p, local_m;
float local_dot, dot;
float local_x[100], local_y[100];
MPI_Status status;

MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
MPI_Comm_size( MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf("%d",&m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j < local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce(&local_dot, &dot,1, MPI_FLOAT, MPI_SUM,0, MPI_COMM_WORLD);
```

Gather Operation:

Each process provides a block of data collected at a root process. For p processes,thedatablockcollectedattherootprocessis p times larger than the individual blocks provided by each process: **int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm).**

For p processes the effect of the MPI Gather() call can also be achieved if each process, including the root process, calls a send operation MPI_Send (sendbuf, sendcount, sendtype, root, my rank, comm) and the root process executes p receive operations MPI_Recv (recvbuf+i*recvcount*extent, recvcount, recvtype, i, i, comm, &status).

Each process must specify the same root process root, same element data type and the same number of elements to be sent.

MPI provides a variant of MPI Gather() for which each process can provide a different number of elements to be collected. The variant is MPI Gatherv(), rrecvcountisreplacedbyanintegerarrayrecvcounts of length p where recvcounts[i] denotes the number of elements provided by process i; there is an additional parameter displs after recvcounts: displs[i] speci□es at which position of the receive buffer of the root process the data block of process i is stored. For a correct execution of MPI Gatherv(), the parameter sendcount speci□ed by process i must be equal to the value of recvcounts[i].

Exemplo: Example for the use of MPI Gatherv()

```
MPI_Comm comm;
int sbuf[100];
int my_rank, root = 0, gsize, *rbuf, *displs, *rcounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if  (my_rank == root) {
    MPI_Comm_size (comm, &gsize);
    rbuf = (int *) malloc(gsize*stride*sizeof(int));
    displs = (int *) malloc(gsize*sizeof(int));
    rcounts = (int *) malloc(gsize*sizeof(int));
    for (i = 0; i < gsize; i++) {
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
}
MPI_Gatherv(sbuf,100,MPI_INT,rbuf,rcounts,displs,MPI_INT,root,comm);
```

Scatter Operation:

For a scatter operation, a root process provides a different data block for each participating process: **int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype**

sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm): where sendbuf is the send buffer provided by the root process root which contains a
data block for each process of the communicator comm. Each data block contains sendcount elements of type sendtype. In the send buffer, the blocks are ordered in rank order of the receiving process. The data blocks are received in the receive buffer recvbuf provided by the corresponding process.

For p processes, the effects of MPI Scatter() can also be achieved by letting the root process execute p send operations

There is a generalized version MPI Scatterv() of MPI Scatter() for which the root process can provide data blocks of different sizes: **MPI Scatterv()**: integer parameter sendcount is replaced by the integer array sendcounts where sendcounts[i] denotes the number of elements and there is an additional parameter displs after sendcounts: displs[i] speci□es from which position in the send buffer of the root process the data block for process i should be taken. No entry of the send buffer is sent to more than one process.

Exemplo: Example for the use of an MPI Scatterv() operation

```
MPI_Comm comm;
int rbuf[100];
int my_rank, root = 0, gsize, *sbuf, *displs, *scounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   sbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   scounts = (int *) malloc(gsize*sizeof(int));
   for (i=0; i<gsize; i++) {
       displs[i] = i*stride; scounts[i]=100;
   }
}
MPI_Scatterv(sbuf,scounts,displs,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

Multi-broadcast Operation:

Each participating process contributes a block of data which could, for example, be a partial result from a local computation. All blocks will be provided to all processes. There is no distinguished root process, since each process obtains all blocks provided: **int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)**

Foran **MPI Allgather()** operation, each process must contribute a data block of the same size: int MPI_Allgatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm). The parameters have the same meaning as for MPI Gatherv().

Multi-accumulation Operation:

For a multi-accumulation operation, each participating process performs a separate single-accumulation operation for which each process provides a different block of data. a multi-accumulation operation in MPI has the same effect as a single-accumulation operation followed by a singlebroadcast operation which distributes the accumulated data block to all processes

int MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI Op op, MPI_Comm comm), where sendbuf is the send buffer in which each process provides its local data block. The parameter recvbuf speci□es the receive buffer in which each process of the communicator comm collects the accumulated result. Both

buffers contain count elements of type type. The reduction operation op is used. Each process must specify the same size and type for the data block.

```
int m, local_m, n, p;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M];
float c[MAX_N], sum[MAX_N];
local_m = m/p;
for (i=0; i<n; i++) {
   sum[i] = 0;
   for (j=0; j<local_m; j++)
      sum[i] = sum[i] + a[i][j]*local_b[j];
}
MPI_Allreduce (sum, c, n, MPI_FLOAT, MPI_SUM, comm);
```

We consider the use of a multi-accumulation operation for the parallel computation of a matrix–vector multiplication.

Total Exchange:

For a total exchange operation, each process provides a different block of data for each other process. The operation has the same effect as if each process performs a separate scatter operation or as if each process performs a separate gather operation.

int MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

If data blocks of different sizes should be exchanged, the vector version must be used: int **MPI Alltoallv** (void *sendbuf, int *scounts, int *sdispls, MPI_Datatype sendtype, void *recvbuf, int *rcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm).

Process Creation and Management:

A number of MPI processes can be started by calling the function **int MPI_Comm_spawn (char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int errcodes[]):** The parameter command speci□es the name of the program to be executed by each oftheprocesses,argv[]containstheargumentsforthisprogram.Incontrasttothe standardCconvention,argv[0]isnottheprogramnamebutthe□rstargumentfor the program. An empty argument list is speci□ed by MPI ARGV NULL. The parameter maxprocs speci□es the number of processes to be started. If the MPI runtime system is not able to start maxprocs processes, an error message is generated. The parameter info speci□es an MPI Info data structure with (key, value) pairs providing additional instructions for the MPI runtime system on how to start the processes. The parameter root speci□es the number of the root process from which the new processes are spawned. Only this root process provides values for the preceding parameters. The parameter errcodes is an array with maxprocs entries in which the status of each process to be spawned is reported.

Multiple MPI programs or MPI programs with different argument values can be spawned by calling the function **int MPI_Comm_spawn_multiple** (int count, char *commands[], char **argv[], int maxprocs[], MPI Info infos[], int root, MPI Comm comm, MPI Comm *intercomm, int errcodes[]).

Cuda

Benefícios

- Leitura paralela - o código pode ler de endereços arbitrários na memória;
- Memória compartilhada - CUDA expõe uma região de memória compartilhada rápida (16KB em tamanho) que podem ser compartilhados entre threads. Isso pode ser usado como um cache de usuário, permitindo maior largura de banda do que é possível utilizando textura lookups;
- Downloads mais rápidos e readbacks para a GPU;
- Suporte completo para operações de números inteiros e operações de bitwise;

Limitações

- A renderização de texturas não é suportado;
- As cópias realizadas entre uma memória e outra podem gerar algum problema na performance das aplicações;
- Ao contrário do OpenCL, o CUDA está disponível apenas para placas de vídeo fabricadas pela própria NVIDIA. Caso seja usado em outro tipo de placa, o CUDA funcionará corretamente, entretanto a performance será bem limitada;

A GPU se torna mais apta para o trabalho de processamento paralelo por ter sido desenvolvida para atender à demanda de processos de computação 3D em alta resolução e em tempo real. Assim, com o passar do tempo, as GPUs modernas se tornaram muito eficientes ao manipular grandes quantidades de informações. O fluxo de processamento em CUDA não é tão complexo. Para começar, os dados são copiados da memória principal para a unidade de processamento gráfico. Depois disso, o processador aloca o processo para a GPU, que então executa as tarefas simultaneamente em seus núcleos. Depois disso, o resultado faz o caminho inverso, ou seja, ele é copiado da memória da GPU para a memória principal. Na unidade de processamento gráfico, todo esse processamento é feito dentro dos núcleos CUDA (conhecidos como CUDA Cores), os quais podem ser comparados com os núcleos de um processador comum. Por isso, quanto mais núcleos CUDA tiver a placa de vídeo, melhor.

2. Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C. An extensive description of CUDA C is given in Programming Interface.
Full code for the vector addition example used in this chapter and the next can be found in the vectorAdd CUDA sample.

2.1. Kernels

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.
A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new`<<<...>>>`execution configuration syntax (see C Language Extensions). Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable.
As an illustration, the following sample code adds two vectors A and B of size N and stores the result into vector C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the N threads that execute `VecAdd()` performs one pair-wise addition.

2.2. Thread Hierarchy

For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.
The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + y D_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + y D_x + z D_x D_y)$.
As an example, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*:

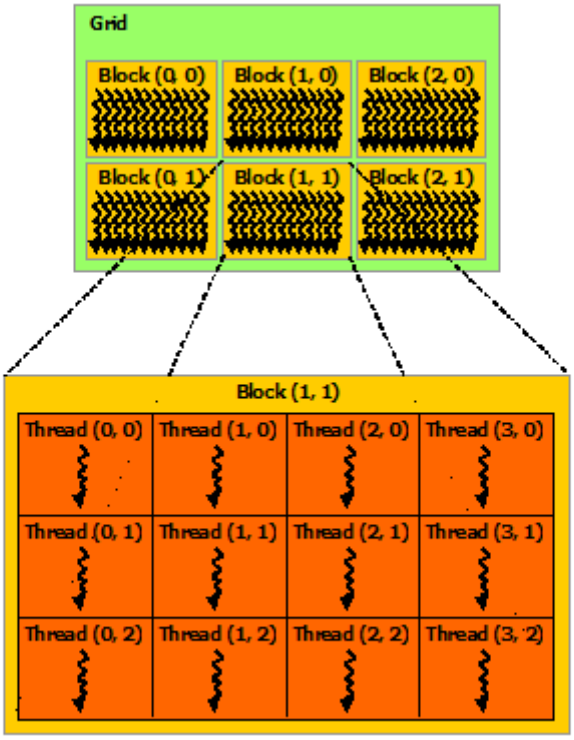
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
```

```
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.
However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.
Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 6. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

Figure 6. Grid of Thread Blocks



The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified as in the example above.
Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdxvariable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

Extending the previous MatAdd() example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 5, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the __syncthreads() intrinsic function; __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Shared Memory gives an example of using shared memory. In addition to __syncthreads(), the Cooperative Groups API provides a rich set of thread-synchronization primitives.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and __syncthreads() is expected to be lightweight.

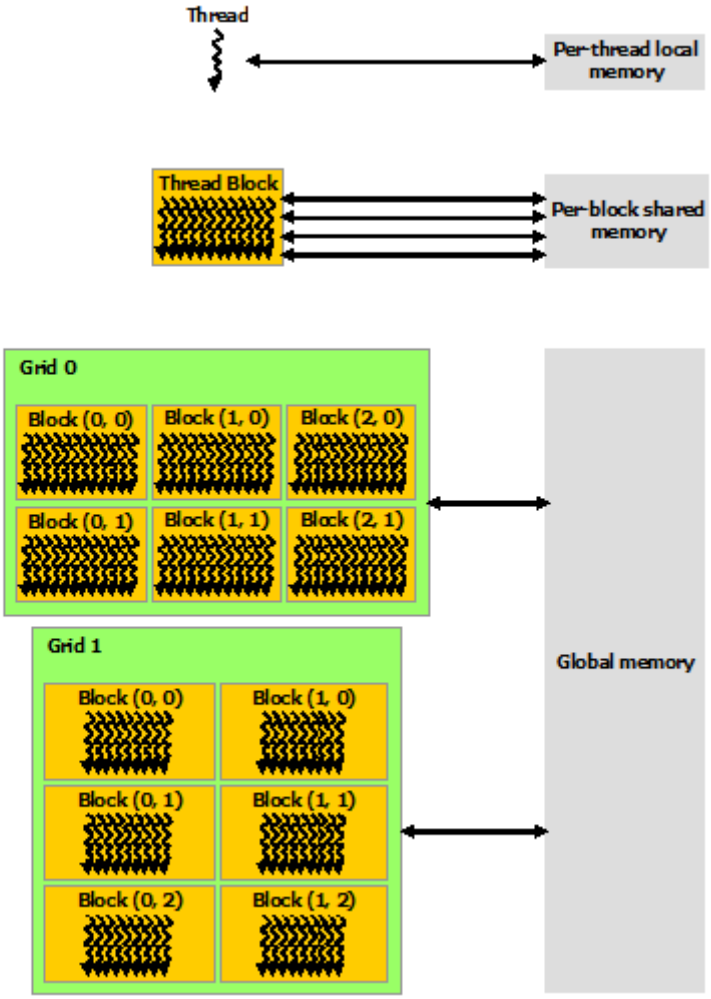
2.3. Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 7. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see Device Memory Accesses). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Texture and Surface Memory).

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Figure 7. Memory Hierarchy



2.4. Heterogeneous Programming

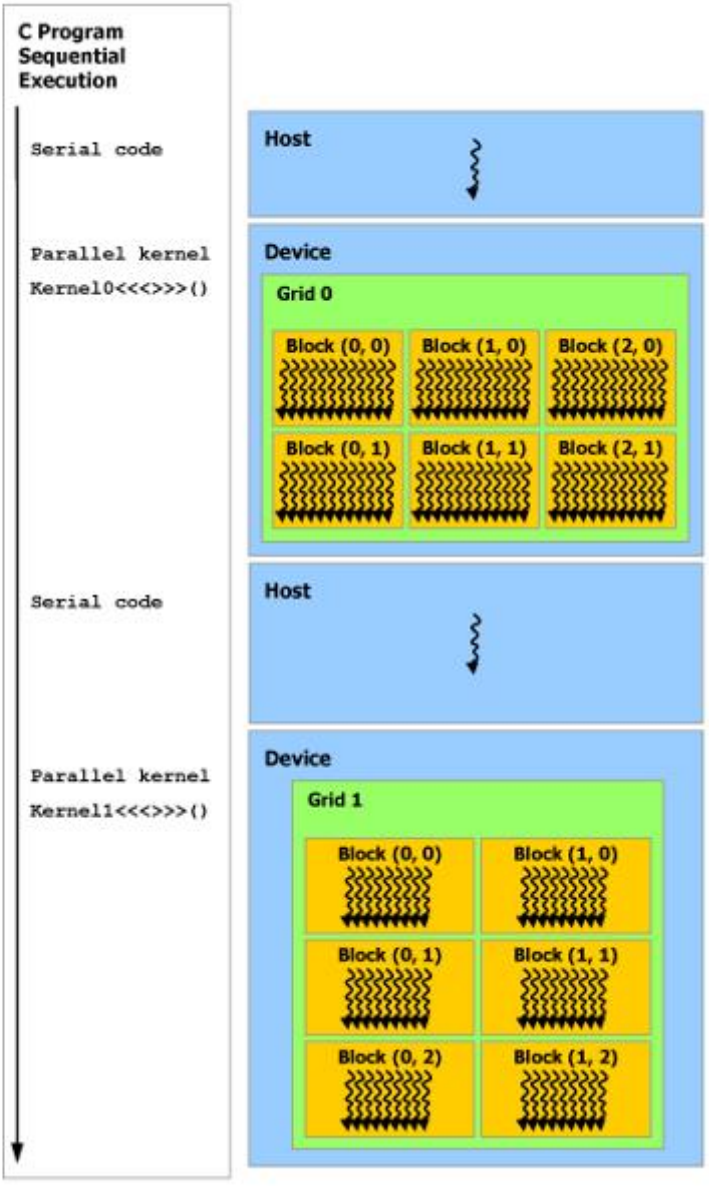
As illustrated by Figure 8, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the hostrunning the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (described in Programming Interface). This includes device memory allocation and deallocation as well as data transfer between host and device memory.

Unified Memory provides managed memory to bridge the host and device memory spaces. Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space. This capability enables oversubscription of device memory and can greatly simplify the task of porting applications by eliminating the

need to explicitly mirror data on host and device. See Unified Memory Programming for an introduction to Unified Memory."

Figure 8. Heterogeneous Programming



Note: Serial code executes on the host while parallel code executes on the device.

2.5. Compute Capability

The compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU

hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.
The compute capability comprises a major revision number X and a minor revision number Y and is denoted by X.Y.
Devices with the same major revision number are of the same core architecture. The major revision number is 7 for devices based on the Volta architecture, 6 for devices based on the Pascal architecture, 5 for devices based on the Maxwell architecture, 3 for devices based on the Kepler architecture, 2 for devices based on the Fermi architecture, and 1 for devices based on the Tesla architecture.
The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.
Turing is the architecture for devices of compute capability 7.5, and is an incremental update based on the Volta architecture.
CUDA-Enabled GPUs lists of all CUDA-enabled devices along with their compute capability.
Compute Capabilities gives the technical specifications of each compute capability.

Note: The compute capability version of a particular GPU should not be confused with the CUDA version (e.g., CUDA 7.5, CUDA 8, CUDA 9), which is the version of the CUDA *software platform*. The CUDA platform is used by application developers to create applications that run on many generations of GPU architectures, including future GPU architectures yet to be invented. While new versions of the CUDA platform often add native support for a new GPU architecture by supporting the compute capability version of that architecture, new versions of the CUDA platform typically also include software features that are independent of hardware generation.

The Tesla and Fermi architectures are no longer supported starting with CUDA 7.0 and CUDA 9.0, respectively.

Erlang

É uma linguagem de programação de uso geral e um sistema para execução. Foi desenvolvida pela Ericsson para suportar aplicações distribuídas e tolerantes a falhas a serem executadas em um ambiente de tempo real e ininterrupto. Ela suporta nativamente hot swapping, de forma que o código pode ser modificado sem a parada do sistema. A implementação da Ericsson executa um código interpretado em uma máquina virtual, mas também inclui um compilador para código nativo (ainda que não suportado para todas as plataformas).

Criar e gerenciar processos é uma tarefa trivial em Erlang; de forma geral, threadssão consideradas complicadas e fontes de muitos erros de programação em linguagens. A comunicação entre processos é feita por troca de mensagens ao invés de variáveis compartilhadas, o que remove a necessidade de mecanismos explícitos de exclusão mútua.

A linguagem Erlang possui oito tipos de dados primitivos[8]

- Inteiros: Inteiros são escritos como sequências de dígitos decimais, por exemplo, 12, 12375 e -23427 são inteiros. A aritmética de inteiros é exata e de precisão arbitrária, limitada apenas pela memória disponível ao programa.
- Átomos: Átomos são usados nos programas para denotar valores distintos. São escritos como sequências de caracteres alfanuméricos, sendo o primeiro uma letra minúscula. Átomos podem conter qualquer caractere se forem fechados entre apóstrofos.
- Ponto flutuante: Números de ponto flutuante na representação IEEE 754 em 64 bits.
- Referências: Referências são símbolos globalmente únicos cuja única propriedade é a comparação para igualdade. São criadas avaliando a primitiva make_ref().
- Binários: Um binário é uma sequência de bytes. Binários proveem um meio que utiliza o espaço de maneira eficaz para armazenar dados binários. Há primitivas em Erlang para compor e decompor binários bem como fazer a entrada e saída deles.
- Pids: Pid é uma abreviatura de Process Identifier (identificador de processo). São criados pela primitiva spawn(...). Pids são referências para processos Erlang.
- Portas: Portas são utilizadas para se comunicar com o mundo externo. São criadas pela sub-rotina open_port. A comunicação é realizada por mensagens enviadas e recebidas pelas portas segundo o protocolo de portas do Erlang.
- Funs: Funs são fechamento de funções. Funs são criadas por expressões da forma: fun(...) -> ... end.

Há dois tipos de dados compostos:

- Tuplas: Tuplas são recipientes para um número fixo de tipos de dados Erlang, descritas pela sintaxe {D1, D2, ... , Dn}, que denota uma tupla cujos argumentos são D1, D2, ... Dn. Os argumentos podem ser tipos de dados

primitivos ou compostos. Os elementos das tuplas podem ser acessados em tempo constante.

- Listas: Listas são recipientes para um número variável de tipos de dados Erlang. A sintaxe [Dh|Dt] denota uma lista cujo primeiro elemento é Dh, e os demais elementos são a lista Dt. O primeiro elemento da lista e chamado head(cabeça) da lista. O restante da lista quando sua head foi removida é chamada tail (cauda) da lista.

Há dois açúcares sintáticos:

- Strings: Strings são escritos como uma lista de caracteres fechados entre aspas, como açúcar sintático para uma lista de inteiros com o código ASCII dos caracteres. Por exemplo, a string "gato" é na verdade a lista [103, 97, 116, 111]. Há suporte incompleto para strings Unicode[9].
- Records: Records (registros) fornecem um meio conveniente para associar um nome a cada elemento de uma tupla. Isso permite se referir a um elemento de uma tupla por nome e não por posição. O pré-compilador substitui a definição dos registros e o substitui com as referências de tupla apropriadas.

Abaixo está a implementação de um algoritmo Quicksort

```
%% quicksort:qsort(List)
%% Classificar uma lista de itens

 -module(quicksort).
 -export([qsort/1]).

 qsort([]) -> [];
 qsort([Pivot|Rest]) ->
    qsort([X || X <- Rest, X < Pivot]) ++ [Pivot] ++ qsort([Y || Y <- Rest, Y >= Pivot]).
```

O exemplo acima invoca a função recursiva qsort até que não reste nada a ser classificado. A expressão [X || X <- Rest, X < Pivot] é uma lista de abrangência, o que significa “construir uma lista de elementos X tal que X pertence ao Resto e X é menor que Pivot“.

Uma função de comparação pode ser usada, no entanto, a ordem em que se baseia o retorno do código Erlang (verdadeiro ou falso) precisa ser alterada. Se, por exemplo, queremos uma lista ordenada onde a < 1 seja verdadeiro (true).

O seguinte código pode classificar listas de acordo com o tamanho:

```
-module(listsort).
 -export([by_length/1]).

  by_length(Lists) ->
    qsort(Lists, fun(A,B) when is_list(A), is_list(B) -> length(A) < length(B) end).

  qsort([], _)-> [];
  qsort([Pivot|Rest], Smaller) ->
```

```
    qsort([X || X <- Rest, Smaller(X,Pivot)], Smaller)
    ++ [Pivot] ++
    qsort([Y ||Y <- Rest, not(Smaller(Y, Pivot))], Smaller).
```

A principal vantagem de Erlang é suporte a concorrência. Tem um pequeno, mas poderoso, conjunto de funções primitivas para criar processos e fazer com que eles se comuniquem. Processos são o principal meio de criar uma aplicação em Erlang.

Processos, como o sistema operacional (ao contrário de "Green threads" e threads do sistema operacional), não são compartilhadas entre si.

A sobrecarga mínima estimada para cada um é de 300 palavras (4 bytes por palavra sobre plataformas 32-bit, 8 bytes por palavra sobre plataformas 64-bit), e por isso muitas delas podem ser criadas sem degradar o desempenho (foi possível executar 20 milhões de processos. Erlang suporta processamento simétrico na versão R11B desde maio de 2006.

O processo de comunicação é feito através de uma mensagem não-compartilhada em um sistema assíncrono: cada processo tem uma "caixa postal", uma fila de mensagens enviadas por outros processos, que ainda não foram consumidos.

Um processo utiliza um meio primitivo para recuperar mensagens que correspondam aos padrões desejados.

Uma mensagem de rotina examina mensagens em cada turno, até que um deles seja correspondente. Quando a mensagem é consumida (retirada da caixa postal), o processo recomeça sua execução.Uma mensagem pode incluir qualquer estrutura de Erlang, incluindo funções primitivas (inteiros, ponto flutuante, caracteres,átomos), tuplas, listas e funções.

Exemplo de código:

```
% cria um processo e chama a função web:start_server(Port, MaxConnections)
 ServerProcess = spawn (web, start_server, [Port, MaxConnections]),

 % cria um processo remoto e chama a função web:start_server(Port,
MaxConnections) na máquina RemoteNode
 RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

 % envia para {pause, 10} a mensagem (uma tupla com o átomo "pause" e um número
"10") para ServerProcess (assincronamente)
 ServerProcess ! {pause, 10},

 % recebe as mensagens enviadas para este processo
 receive
        a_message -> do_something;
        {data, DataContent} -> handle(DataContent);
        {hello, Text} -> io:format("Ola tenho mensagem: ~s", [Text]);
        {goodbye, Text} -> io:format("Tenho mensagem de saídae: ~s", [Text])
 end.
```

Tal como mostra o exemplo, existe um suporte de processos distribuídos. Processos podem ser criados em nós (nodes, em inglês) remotos, e a comunicação com eles é transparente (ou seja, a comunicação com os processos remotos é feita exatamente como a comunicação entre os processos locais).

A Concorrência suporta o método primário para tratamento de erro em Erlang. Quando um processo falha, ele é fechado e envia uma mensagem para o processo controlador tomar alguma decisão.

Esta forma de controle de erros pode aumentar a durabilidade e reduzir a complexidade do código.

4- Pattern matching

Pattern matching em Erlang é usado para associar valores a variáveis, controlar fluxo de execução de programas, extrair valores de estruturas de dados compostas e lidar com argumentos de funções. Mais um ponto para código declarativo e expressividade. Como mostra o código abaixo:

```
-module(sample2).
-export([convert_length/1]).

convert_length({_, 0}) ->
     {error, unconvertable_value},

convert_length(Length) ->
     case Length of
          {centimeter, X} ->
               {inch, X / 2.54};
          {inch, Y} ->
               {centimeter, Y * 2.54}
     end.
```

Fala por si, não fala? Pattern matching para selecionar a função – o que acaba validando a entrada – e para conversão da unidade de medida.

5- Concorrência baseada em passagem de mensagens (a.k.a. Actors) Acho que concorrência baseada em passagem de mensagem entre atores é uma das features mais populares de Erlang. Vejamos o porque com o famoso exemplo do Ping-Pong:

Neste pequeno snippet podemos observar algumas características de Erlang já citadas neste post, tal como pattern matching na captura das mensagens e recursividade no controle das iterações. Agora, falando do aspecto concorrente em si, algumas coisas são particularmente interessantes aqui:

```
-module(sample3).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
     Pong_PID ! finished,
     io:format("Ping finished~n", []);

ping(N, Pong_PID) ->
     Pong_PID ! {ping, self()},
     receive
          pong ->
               io:format("Ping received pong~n", [])
     end,
     ping(N - 1, Pong_PID).

pong() ->
     receive
          finished ->
               io:format("Pong finished~n", []);
          {ping, Ping_PID} ->
               io:format("Pong received ping~n", []),
               Ping_PID ! pong,
               pong()
     end.

start() ->
     Pong_PID = spawn(sample3, pong, []),
     spawn(sample3, ping, [3, Pong_PID]).
```

- Em Erlang, a concorrência acontece entre processos leves, diferente de linguagens como C++ e Java, que baseiam sua concorrência em threads nativas de sistema operacional, que são caríssimas;
- Em Erlang, há um tipo de dado chamado PID, o qual é o identificador do processo paralelo (mais conhecido como Actor) e para o qual as mensagens podem ser enviadas.

Releia o código acima com estas informações em mente e veja como concorrência em Erlang é algo completamente descomplicado e natural

Códigos Entregues

Aula 05 - OpenMP - C/OpenMP - Encontrar Qtde de Primos - ENG COMP

```
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <math.h>

int main ( int argc, char *argv[] );
void prime_number_sweep ( int n_lo, int n_hi, int n_factor );
int prime_number ( int n );

int main ( int argc, char *argv[] )  {
        int n_lo, n_hi, n_factor;

        n_lo = 5;
        n_hi = 500000;
        n_factor = 10;

        omp_set_nested(1);

        prime_number_sweep ( n_lo, n_hi, n_factor );

        printf ( "  Normal end of execution.\n" );


        return 0;
}// end main

void prime_number_sweep ( int n_lo, int n_hi, int n_factor )
/*  Purpose: PRIME_NUMBER_SWEEP does repeated calls to PRIME_NUMBER.
  Parameters:
    Input, int N_LO, the first value of N.
    Input, int N_HI, the last value of N.
    Input, int N_FACTOR, the factor by which to increase N after
    each iteration.
  Example:      N  PRIME_NUMBER
                1          0
               10          4
              100         25
            1,000        168
           10,000      1,229
          100,000      9,592
        1,000,000     78,498
       10,000,000    664,579 */
{
        int i, n, primes;
        struct timeval t0, t1;
        long elapsed;
```

```
        int max;
        max = log(n_hi/n_lo)/log(n_factor);

        printf ( "  Call PRIME_NUMBER to count the primes from %d to %d.\n", n_lo, n_hi );
        printf ( "         N        Pi    Time\n" );
        n = n_lo;

        while ( n <= n_hi ){

                gettimeofday(&t0, 0);

        primes = prime_number ( n );
        gettimeofday(&t1, 0);
        elapsed = (t1.tv_sec-t0.tv_sec)*1000000 + t1.tv_usec-t0.tv_usec;

        printf ( "  %8d  %8d  %d\n", n, primes, elapsed); //wtime );

        n *= n_factor;

        }// end while

        return;
}// end prime_number_sweep

int prime_number ( int n )
/*   PRIME_NUMBER returns the number of primes between 1 and N.
  Author: John Burkardt
  Parameters:
    Input, int N, the maximum number to check.
    Output, int PRIME_NUMBER, the number of prime numbers up to N.  */
{
        int i, j, prime, total = 0;

        #pragma omp parallel for schedule(guided, 50) private (i, j, prime) firstprivate(n)
reduction(+: total) num_threads(8)

            //#pragma omp for schedule(guided)
                        for ( i = 2; i <= n; i++ ){
                            prime = 1;

                            for ( j = 2; j < i; j++ ){
                                if ( i % j == 0 ){
                                        prime = 0;
                                break;
                                } //end if
                            } // end for
                            total = total + prime;
                        } // end for

  return total;
} // end prime_number
```

Pensamento do PCAM:

Particionamento:

O particionamento foi feito pensando em separar o arquivo de entrada em um número definido de blocos, em que cada tarefa destinada a isso irá ler cada um desses blocos. Além dessas tarefas, há também a separação de outras tarefas para analisar cada um desses blocos e obter o resultado esperado. Por fim, há a separação de tarefas para pegar cada um dos resultados e escrever no arquivo de saída.

Comunicação:

A comunicação é feita em relação a cada bloco do arquivo de entrada. A tarefa que leu o bloco i se comunicará com a tarefa que irá analisar e obter o resultado do bloco i, e, por fim, essa irá se comunicar com a tarefa que irá escrever o resultado da pesquisa no bloco i no arquivo de saída.

Aglomeração:

Houve aglomeração de todas as tarefas de leitura do arquivo de entrada em um único processo, em que esse processo irá enviar uma cópia de um bloco do vetor de entrada para cada um dos outros processos. Cada bloco terá tamanho igual ao tamanho da entrada dividido pelo número de processos, exceto ao último processo.

Para o último processo, o tamanho será igual a TAM - (npes-1)*nlocal, em que TAM é o tamanho total do vetor de entrada, npes o número de processos e nlocal o tamanho do bloco para os demais processos.

Houve também aglomeração no processo da escrita no arquivo de saída, em que um só processo recebe e escreve todos os resultados, em ordem.

Mapeamento:

A aglomeração de leitura de dados será destinada a um processador só, enquanto cada uma das tarefas de leitura dos blocos do arquivo de entrada, posterior à leitura, é feita em cada processador

disponível. Por fim, a cada vez que a procura em um bloco i é concluída, ela já se torna disponível para ser escrita no arquivo de saída, processo que também será destinado a um único processador. */

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define TAM 4000000

int main(int argc,char **argv){
   if(argc!=2){
      printf("Informe apenas o valor de k");
      exit(1);
   }

   //Início do MPI
   MPI_Init(&argc, &argv);

   //Variáveis de informação sobre a paralelização e tamanho do bloco
   int npes;
   int myrank;
   int nlocal;

   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   MPI_Status status;
   MPI_Request requestVet;
   MPI_Request requestElem;
```

```
nlocal = TAM/npes;

//Variáveis utilizadas durante a lógica da busca em cada bloco
int i, j, l;
int contador = 0;
int elemento = 0;
int *vet;
int *indices = (int*)malloc(sizeof(int));
int *vet_msg = (int*)malloc(nlocal*sizeof(int));

//Apenas o processo de rank 0 irá ler o arquivo de entrada
//e distribuir os blocos para cada processo
if(myrank == 0){
   //Receberá a entrada completa
   vet = ((int*)malloc(TAM*sizeof(int)));

   FILE *arquivo_entrada;
   arquivo_entrada = fopen("numeros.txt", "r");

   for(i = 0; i < TAM; i++){
      fscanf(arquivo_entrada, "%d,", &vet[i]);
   }

   int k = atoi(argv[1]);
   if(k < 0 || k > (TAM-1)){
      printf("Elemento fora do indice do vetor");
      exit(1);
   }

   elemento = vet[k];

   //Enviamos os blocos e o elemento escolhido para cada processo
   for(i = 1; i < npes; i++){
      MPI_Send(&vet[i*nlocal], nlocal, MPI_INT, i, 0, MPI_COMM_WORLD);
      MPI_Send(&elemento, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
   }

   //Também colocamos as informações no vetor do processo com rank 0
   for(i = 0; i < nlocal; i++){
      vet_msg[i] = vet[i];
   }

   fclose(arquivo_entrada);
}
else{
   //Outros processos que não possuem rank 0 recebem as informações
   //do processo de rank 0
   MPI_Irecv(vet_msg, nlocal, MPI_INT, 0, 0, MPI_COMM_WORLD, &requestVet);
   MPI_Irecv(&elemento, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &requestElem);

   MPI_Wait(&requestVet, &status);
   MPI_Wait(&requestElem, &status);
}

//Tratamento da situação em que TAM não é divisível por nlocal
```

```
int max;
if(myrank == npes-1)
    max = TAM - (npes-1)*nlocal;
else
    max = nlocal;

//Verificação dos números maiores que o escolhido em cada bloco
j = 0;
for(i = 0; i < max; i++){
    if(vet_msg[i] > elemento){
        contador++;
        indices = (int*)realloc(indices,contador*sizeof(int));
        indices[j] = myrank*nlocal + i;
        j++;
    }
}

//Após a verificação, todos os processos, exceto o de rank 0,
//enviam seus resultados para o processo de rank 0
if(myrank != 0){
    MPI_Send(&contador, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Send(indices, contador, MPI_INT, 0, 1, MPI_COMM_WORLD);
}

//Compilação de todos os resultados
if(myrank == 0){
    //Auxiliares para contador e índices, os quais terão, de
    //início, os valores encontrados pelo processo de rank 0
    int auxcontador;
    int *auxindices;

    auxcontador = contador;
    auxindices = (int*)malloc(contador*sizeof(int));
    int c = 0;
    while(c < contador){
        auxindices[c] = indices[c];
        c++;
    }

    //Para cada processo
    for(i = 1; i < npes; i++){
        //Recebemos o seu contador e seu vetor de índices
        MPI_Irecv(&contador, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &requestElem);
        MPI  Wait(&requestElem, &status);
        if(contador != 0){
            indices = (int*)realloc(indices, contador*sizeof(int));
            MPI_Irecv(indices, contador, MPI_INT, i, 1, MPI_COMM_WORLD, &requestVet);
            MPI_Wait(&requestVet, &status);
        }
        c = 0;

        //Somamos ao contador global (auxcontador) e concatenamos
        //ao vetor de índices global (auxindices)
        while(contador > 0){
            auxcontador++;
            auxindices = (int*)realloc(auxindices, auxcontador*sizeof(int));
```

```
            auxindices[j] = indices[c];

            j++;
            c++;
            contador--;
        }
    }

    //Impressão no arquivo de saída
    FILE *arquivo_saida;
    arquivo_saida = fopen("saida.txt", "w+");

    fprintf(arquivo  saida, "%d\n", auxcontador);
    if(auxcontador > 0) {
        for(i = 0; i < auxcontador; i++){
            fprintf(arquivo_saida,"%d ", auxindices[i]);
        }
    }

    fclose(arquivo_saida);
    free(vet);
  }

  free(vet_msg);
  free(indices);

  MPI_Finalize();
}
```

Aula 7 - MPI Coletivas e Spawn - Entrega da Aplicação MPI com coletivas e spawn TA

Aula 7 - MPI Coletivas e Spawn - Entrega da Aplicação MPI com coletivas e spawn TB

```
//==master

#include<stdio.h>
#include<stdlib.h>
#include <limits.h>
#include <math.h>
#include <mpi.h>
```

```
#include <string.h>

int main(int argc, char* argv[]){
   MPI_Init(&argc, &argv);
   int npes, myrank;
   FILE *arquivo_entrada;
   arquivo_entrada=fopen("entrada.txt","r");
   int tam;
   int i, j, n, m;
   fscanf(arquivo  entrada, "%d\n", &tam);
   int *matriz=((int*)malloc(tam*tam*sizeof(int)));

   //Leitura da matriz
   for(i = 0; i < tam*tam; i++){
     fscanf(arquivo_entrada, "%d\n", &(matriz[i]));
   }

  int aux_comp;
  int diferenca = INT_MIN;
  int indice_i_maior, indice_j_maior, indice_i_menor, indice_j_menor;
  int valor_maior, valor_menor;

  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  int a[7],linhaum[tam],linhadois[tam],lixo[tam];
  //Criar os Slaves
    char slave[20];
    strcpy(slave,"slave8");
    int N = 4, errcodes[10], root =0;
    MPI_Comm    inter_comm;
    MPI_Comm_spawn(slave, MPI_ARGV_NULL, N, MPI_INFO_NULL, root,
MPI_COMM_WORLD, &inter_comm, errcodes);
  //Mandar tamanho da matriz para Slaves
    MPI_Bcast(&tam,1, MPI_INT, MPI_ROOT, inter_comm);
    MPI_Scatter(&matriz[0], tam, MPI_INT, &linhaum[0] , tam, MPI_INT, MPI_ROOT,
inter_comm);
    MPI_Scatter(&matriz[tam], tam, MPI_INT, &linhadois[0] , tam, MPI_INT, MPI_ROOT,
inter_comm);
  //
int max,lixo;
  MPI_Reduce( &lixo, &max, 1, MPI_INT, MPI_MAX, MPI_ROOT, inter_comm );

  free(matriz);

  //Fechando os arquivos de entrada e saída
  fclose(arquivo_entrada);
  MPI_Finalize();

  return 0;
}


//===== slave
#include<stdio.h>
```

```c
#include<stdlib.h>
#include <limits.h>
#include <math.h>
#include <mpi.h>
#include <string.h>

int main(int argc, char* argv[]){
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    int npes, myrank;
    int tam;
    int i, j, n, m;


  //Auxiliar para comparar o valor da diferenca entre os vizinhos
  int aux_comp;
  //Variável para guardar a maior diferenca encontrada
  int diferenca = INT_MIN;
  //Variáveis para armazenar os indices da maior diferença encontrada
  int indice_i_maior, indice_j_maior, indice_i_menor, indice_j_menor;
  //Variáveis para armazenar os valores da maior diferença encontrada
  //Pderia utilizar apenas os indices armazenados, colocados aqui por questões de clareza
  int valor_maior, valor_menor;

  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);


  int a[7],linhaum[tam],linhadois[tam],lixo[tam];
  MPI_Comm inter_comm;
  MPI_Comm_get_parent(&inter_comm);
  MPI_Bcast(&tam,1, MPI_INT, 0, inter_comm);
  printf("%d\n",tam);MPI_Scatter(&matriz[0], tam, MPI_INT, &linhaum[0] , tam, MPI_INT,
MPI_ROOT, inter_comm);
  MPI_Scatter(&lixo[0], tam, MPI_INT, &linhaum[0] , tam, MPI_INT, 0, inter_comm);
  MPI_Scatter(&lixo[0], tam, MPI_INT, &linhadois[0] , tam, MPI_INT, 0, inter_comm);



  for (i = myrank-1; i < tam; i+=(npes-1)) {
    for (j = 0; j < tam; j++) {
      //Avalia a diferenca entre o elemento matriz[i][j] e seus vizinhos, com exceção dos
visinhos da linha de cima
      for (n = i ; n < i + 2; n++) {
        for (m = j - 1; m < j + 2; m++) {
          //Se não for a borda da matriz, tire a diferenca
          if (!(n < 0 || n >= tam || m < 0 || m >= tam || (n == i && m == j) )){
            //Módulo da diferenca entre o elemento matriz[i][j] e seu vizinho
            aux_comp = abs(matriz[i][j] - matriz[n][m]);
            //Se a comparação atual é maior que a armazenada
            if (aux_comp > diferenca){
              //Diferença global é atualizada
              diferenca = aux_comp;
              //Armazenando as posições da maior diferença encontrada até o momento
              indice_i_maior = i;
              indice_j_maior = j;
```

```
            indice_i_menor = n;
            indice_j_menor = m;
            //Armazenando o valor da maior diferença encontrada até o momento
            valor_maior = matriz[i][j];
            valor_menor = matriz[n][m];
          }
        }
      }
    }
  }
  a[0]=diferenca;
a[1]=indice_i_maior;a[2]=indice_i_menor;a[3]=indice_j_maior;a[4]=indice_j_menor;a[5]=valor
_maior;a[6]=valor_menor;


  MPI_Reduce( &diferenca, &lixo, 1, MPI_INT, MPI_MAX, 0, inter_comm );
}
```

Aula 07 - MPI Coletivas e Spawn - Entrega da Aplicação MPI com coletivas e spawn para ENG COMP

```
colocar codigo aqui
```

Aula 09 - TA - Multiplicação de Matrizes - OpenMP e MPI - TURMA A

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <omp.h>

#define T 10

int main (int argc, char **argv){
   int rank, n;

   MPI_Init(argc, argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if(rank == 0){  //master
```

```
        FILE *f = fopen("Entrada.txt", "r");
        fscanf(f,"%d", &n);

        int i, j, matrix1[n][n], matrix2[n][n];

        //read matrix1
        for(i = 0; i < n; i++){
            for(j = 0; j < n; j++){
                fscanf(f, "%d", &matrix1[i][j]);
                if (i != 0 && j != 0){
                    MPI_Send(&n, 1, MPI_INT, n * i + j, 0, MPI_COMM_WORLD); //send n
                    MPI_Send(matrix1[i], n, MPI_INT, n * i + j, 0, MPI_COMM_WORLD); //send
rows of matrix 1
                }
            }
        }
        //read matrix2
        for(i = 0; i < n; i++){
            for(j = 0; j < n; j++){
                fscanf(f, "%d", &matrix2[j][i]);// Transposed matrix
                if (i != 0 && j != 0)
                    MPI_Send(matrix2[i], n, MPI_INT, n * i + j, 0, MPI_COMM_WORLD); //send
column of matrix 2
            }
        }

        fclose(f);

        int mult[n][n];

        //multiplication with openmp
        int answer = 0;
        #pragma omp parallel for firstprivate(answer) reduction(+:answer)
num_threads(T)
        for (i = 0; i < n; i++){
            answer+=matrix1[0][i] * matrix2[0][i];
        }

        MPI_Gather(&answer, 1, MPI_INT, mult, n*n, MPI_INT, 0, MPI_COMM_WORLD);

        f = fopen ("Saida.txt", "w");
        for(i = 0; i < n; i++){
            for(j = 0; j < n; j++){
                fprintf(f, "c[%d][%d]= %d", i, j, mult[i][j]);
            }
        }

        fclose(f);

    }else{ //slaves
        MPI_Status status;
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status); //receive n

        int lin[n], col[n];
        MPI_Recv(lin, n, MPI_INT, 0, 0, MPI_COMM_WORLD, status); //receive row of matrix 1
        MPI_Recv(col, n, MPI_INT, 0, 0, MPI_COMM_WORLD, status); //receive column of
```

```
matrix 2

        int answer = 0, *aux;

        #pragma omp parallel for firstprivate(answer) reduction(+:answer)
num_threads(T)
        for (int i = 0; i < n; i++){
            answer+=lin[i] * col[i];
        }

        MPI_Gather(&answer, 1, MPI_INT, aux, n*n, MPI_INT, 0, MPI_COMM_WORLD);
    }

    MPI  Finalize();
    return 0;
}
```

Aula 09 - TB - Produto Escalar - OpenMP e MPI TURMA B -- FUI MAL

```
#include<stdio.h>
#include<stdlib.h>
#include <limits.h>
#include <math.h>
#include <mpi.h>
#include <string.h>


int ** readMatrix(char * fileName, int * size){
  *size = 4;
  int matrix[4][4] = {{92,24,8,62},{97,70,14,65},{26,38,60,90},{79,38,1,25}};
  for()
  return (int**) matrix;
}

void printWithLevel(int rank, char * str){
  printf("%d - %s\n", rank, str);
}

int * fromMatrixToArray(int ** matrix, int size){
  int * array = (int*)malloc(size * size * sizeof(int));
  int i, j, x = 0;
  for(i = 0; i < size; ++i){
    for(j = 0; j < size; ++j){
      array[x] = matrix[i][j];
      x++;
    }
  }
  return array;
}
```

```
int main(int argc, char* argv[]){
   MPI_Init(&argc, &argv);

   int myrank;
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   int worldSize;
   MPI_Comm_size(MPI_COMM_WORLD, &worldSize);

   int size;
   if(myrank == 0){
     printWithLevel(myrank, "master\n");
     int enteredMatrix = readMatrix("matrix.txt", &size);
     printf("entered %d\n", enteredMatrix[0][0]);
     MPI  Bcast(&size,1, MPI  INT, myrank, MPI  COMM_WORLD);
     int recievedTotalElements = size*size/worldSize;
     int recievedList[recievedTotalElements];
     MPI_Scatter(fromMatrixToArray(enteredMatrix, size), size, MPI_INT, &recievedList[0],
size, MPI_INT, 0, MPI_COMM_WORLD);
     printf("%d\n", recievedList[0]);
   }
   else{
     int recievedSize;
     int ** recievedMatrix;
     printWithLevel(myrank, "slave\n");
     MPI_Bcast(&recievedSize,1, MPI_INT, 0, MPI_COMM_WORLD);
     //printf("%d - size: %d\n", myrank, recievedSize);
     int recievedTotalElements = size*size/worldSize;
     int recievedList[recievedTotalElements];
     MPI_Scatter(NULL, 0, NULL, &recievedList[0], recievedTotalElements, MPI_INT, 0,
MPI_COMM_WORLD);
     printf("%d\n", recievedList[0]);
   }

   MPI_Finalize();

  return 0;
}
```

Aula 09 - TB - DESCOBRIR - OpenMP e MPI ENG COMP

```
Compilar: mpicc -o codigo codigo.c

Executar: mpirun -np 4 ./codigo < entrada.txt
*/

#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<omp.h>
#include<math.h>
```

```c
int main(int argc,char **argv){
   if(argc!=2){
      printf("Informe o numero de threads que voce deseja criar no OpenMP\n");
      exit(1);
   }

   int npes;                        // n de processos no openMPI
   int myRank;                        // rank de cada processo no openMPI
   MPI_Status status;

   // Inicializando OpenMPI
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

   // Setando  número de threads do OpenMP com o argumento passado pelo programa
   int omp_nThreads = atoi(argv[1]);
   omp_set_num_threads(omp_nThreads);

   // Cálculo do processador para enviar e recever o token
   int next = (myRank + 1) % npes;
   int previous = (myRank - 1 + npes) % npes;

   // printf("myRank: %d, previous: %d,  next: %d\n", myRank, previous, next);
   int token;

   // Processo 0 inicializa o token e envia-lo para o processo 1
   // recebe do processo (npes - 1) e imprime resultado final
   if (myRank == 0){

      token = 0;
      MPI_Send(&token, 1, MPI_INT, next, 1, MPI_COMM_WORLD);
      // printf("myRank %d enviou para %d\n", myRank, next);

      MPI_Recv(&token, 1, MPI_INT, previous, 1, MPI_COMM_WORLD, &status);
      // printf("myRank %d recebeu de %d\n", myRank, previous);

      printf("RESULTADO FINAL from processo %d = %d\n", myRank, token);

   // Demais processos recebem o token do processo de rank enterior
   // realizam processamento utilizando openMP
   // envia resultado final para processo de rank seguinte
   } else {

      MPI_Recv(&token, 1, MPI_INT, previous, 1, MPI_COMM_WORLD, &status);
      // printf("myRank %d recebeu de %d\n", myRank, previous);

      // Processamento
      // token += 5; // Sequencial

      // Paralelo com OpenMP
      #pragma omp parallel shared(token)
      {
            // int omp_npes = omp_get_num_threads();
            // int omp_myRank = omp_get_thread_num();
```

```
            #pragma omp critical
            {
                token++;
            }

            // printf("thread %d - %d: token final: %d\n", myRank, omp_myRank, token);
        }

        MPI_Send(&token, 1, MPI_INT, next, 1, MPI_COMM_WORLD);
        // printf("myRank %d enviou para %d\n", myRank, next);
    }

    // Fim do uso do OpenMPI
    MPI_Finalize();

    return 0;
}
```

Aula 11 - TA - CUDA 1 - TA

Faça um código em C que, com auxílio do modelo CUDA, seja capaz de, dado um escalar (constante inteira) x e um vetor v, realizar o produto do escalar pelo vetor, de forma que: x=4 v=(2,4,6) x*v=4*(2,4,6)=(8,16,24) Para este problema considere uma entrada do formato: 3 4 2 4 6 Em que, o primeiro elemento da primeira linha indica o tamanho do vetor, o segundo elemento da primeira linha indica o escalar e a segunda linha indica o vetor (todos separados por espaço e quebra de linha entre as variáveis e o vetor) A saída deste código deve ser (saída padrão stdout): 8 16 24

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

__global__ void kernelMul(int scalar, int *v, int n) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if(i < n)
                v[i] = scalar * v[i];
}

int main(int argc, char **argv){

        int i, j, scalar;
        int *v, *v_d, n;

        FILE *arq;

        arq = fopen("entrada.txt", "r");

        if (arq == NULL){
                printf("Problemas na CRIACAO do arquivo\n");
                return;
        }
```

```
        fscanf(arq, "%d %d", &n, &scalar);

        v = (int*) malloc (n * sizeof (int));

        for(i=0; i<n; i++){
                fscanf(arq, "%d", &v[i]);
        }

        cudaMalloc(&v  d, sizeof(int) *n);
        cudaMemcpy(v_d, v, sizeof(int) * n, cudaMemcpyHostToDevice);

        kernelMul<<<ceil(n/32.0), 32>>>(scalar, v_d, n);

        cudaMemcpy(v, v_d, sizeof(int) * n, cudaMemcpyDeviceToHost);

        for(int i = 0; i<n; i++)
                printf("%d ", v[i]);

        printf("/n");
        return 0;

}
```

Aula 11 - TB - CUDA - TB -- Tirou 6.5

Faça um código em C que, com auxílio do modelo CUDA, seja capaz de, dado dois vetores v e w, realize o produto escalar entre eles. Considere o exemplo abaixo: v=(2,4,6) w=(3,2,1) v.w=(2,4,6).(3,2,1)=(2*3+4*2+6*1)=6+8+6=20 Para este problema considere uma entrada do formato: 3 2 4 6 3 2 1 Em que, a primeira linha indica o tamanho dos vetores (quantidade de elementos), a segunda linha indica os componentes do primeiro vetor e a terceira linha indica os componentes do segundo vetor. A saída deste código deve ser (saída padrão stdout): 20

```
#include<stdio.h>
#include<stdlib.h>
#define N 10

__global__ void produto_escalar(int *vet1, int *vet2, int *vet_res, int tam){
        int tid = threadIdx.x;
        if(tid<N){
                vet_res[tid] = vet1[tid]*vet2[tid];
        }

        while(tid < tam) {
                vet_res[tid] = vet1[tid]*vet2[tid];
                tid += blockDim.x;
        }
}
```

```
int main(){
	FILE *arquivo_entrada;

	arquivo_entrada = fopen("entrada.txt", "r");

	int tam, i, *dev_vet1, *dev_vet2, *dev_vet_res, *dev_tam, resultado=0;

	fscanf(arquivo_entrada, "%d\n", &tam);

	//vetor 1
	int *vet1 = (int *) malloc(tam * sizeof(int));

	//vetor 2
	int *vet2 = (int *) malloc(tam * sizeof(int));

	//vetor resultando
	int *vet_res = (int *) malloc(tam * sizeof(int));

	//Ler os vetores
	for(i=0; i<tam; i++){
	fscanf(arquivo_entrada, "%d\n", &vet1[i]);
	}

	for(i=0; i<tam; i++){
	fscanf(arquivo_entrada, "%d\n", &vet2[i]);
	}

	//alocar os vetores na GPU
	cudaMalloc((void **)&dev_vet1, tam*sizeof(int));
	cudaMalloc((void **)&dev_vet2, tam*sizeof(int));
	cudaMalloc((void **)&dev_vet_res,
tam*sizeof(int));
	cudaMalloc((void **)&dev_tam, sizeof(int));

	//copiar os vetores para a GPU
	cudaMemcpy( dev_vet1, vet1, tam*sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy( dev_vet2, vet2, tam*sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy( dev_tam, &tam, sizeof(int), cudaMemcpyHostToDevice);

	//processamento
	produto_escalar<<<1,N>>>(dev_vet1,dev_vet2,dev_vet_res, tam);

	//copiar o vetor resultante
	cudaMemcpy(vet_res, dev_vet_res, tam*sizeof(int),cudaMemcpyDeviceToHost);

	//desalocar os vetores da GPU
	cudaFree(dev_vet1);
	cudaFree(dev_vet2);
	cudaFree(dev_vet_res);

	//escrever os resultados
	for(i=0;i<tam;i++){
	resultado=vet_res[i]+resultado;
	}

	printf("%d", resultado);
```

```
    return 0;
}
```

Aula 11 - TB - CUDA - ENG COMP

```
// EXERCICIO CUDA
// Mutiplicação vetor - escalar
// GRUPO gprec011


#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<cuda.h>

#define BLOCK_SIZE 64

#define checkError(err) ({ \
    if (err != cudaSuccess) { \
        printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__); \
        exit(EXIT_FAILURE); \
    } \
})

// kernel function that runs on the device
// Each thread multiplies a distinct vector element by the given scalar
__global__
void vec_scalar_kernel(int *d_A, int n, int scalar){
    // Thread position identifier
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;

    if (i < n){
        d_A[i] = d_A[i] * scalar;
    }
}

// Function to multily a vector h_a of size n by a scalar
void vec_scalar(int *h_A, int n, int scalar)
{
    int size = n * sizeof(int);
    int *d_A; // vector pointer on the device

    // Memory allocation
    cudaError_t err;
    err = cudaMalloc((void**) &d_A, size);
    checkError(err);

    // Data transfer HOST->DEVICE
    err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    checkError(err);
```

```
    // Kernel configurationa and launch
    vec_scalar_kernel<<<ceil(n/(double)BLOCK_SIZE), BLOCK_SIZE>>> (d_A, n, scalar);

    // Data transfer DEVICE -> HOST
    err = cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);
    checkError(err);

    // Memory free
    err = cudaFree(d_A);
    checkError(err);
}

int main(int argc,char **argv){

    int n, scalar;
    int *vec;
    FILE *input_file;

    input_file = fopen("entrada.txt","r");

    // Input file read
    fscanf(input_file, "%d", &n);
    fscanf(input_file, "%d", &scalar);

    vec=((int*)malloc(n*sizeof(int)));
    int i;
    for ( i = 0; i < n; i++){
        fscanf(input_file, "%d", &(vec[i]));
    }

    // Vector-scalar multiplication
    vec_scalar(vec, n, scalar);

    // Output
    for (i = 0; i < n; i++){
        printf("%d ", vec[i]);
    }
    printf("\n");

    return 0;
}
```

Aula 12 - CUDA - Soma Matrizes Quadradas BCC - Tirou 10

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

/*
 Dividimos o grid em 2 x 2 x 1 blocos onde cada bloco contém dim_da_matriz/2 x
```

```
dim_da_matriz/2   threads (respeitando a divisão em quadrantes).
 Cada thread fica responsável pela (quadrante = bloco):
 - linha  = y_quadrante*dim_da_matriz/2 + coordenada y do thread

  - coluna = x_quadrante*dim_da_matriz/2 + coordenada x do thread

 Onde o primeiro termo é responsável por "pular" posições pertencentes ao outro quadrante
e o segundo faz a correspondência entre a posição (x, y) do thread dentro do bloco com a
posição (linha, coluna) pela qual o thread fica responsável (transforma mapeamento dentro
do quadrante ->
mapeamento dentro da matriz).
 */

__global__ void soma_matriz(int *a, int *b, int *c, int *n){
  int row, col, tile_width;
  tile_width = (*n)/2;

  row = blockIdx.y*tile_width + threadIdx.y;
  col = blockIdx.x*tile_width + threadIdx.x;

  c[row*(*n)+col] = a[row*(*n)+col] + b[row*(*n)+col];
}

int main(int argc, char **argv){
  int *a, *b, *c, n, *dev_a, *dev_b, *dev_c, *dev_n, i;

  n = atoi(argv[1]);

  // matriz a
  a = (int *) malloc(n * n * sizeof(int));

  // matriz b
  b = (int *) malloc(n * n * sizeof(int));

  // matriz c
  c = (int *) malloc(n * n * sizeof(int));

  // inicializar matrizes
  for(i=0; i<n*n; i++){
    a[i] = i;
    b[i] = i;
  }

  // alocar as matrizes na GPU
  cudaMalloc((void **)&dev_a, n*n*sizeof(int));
  cudaMalloc((void **)&dev_b, n*n*sizeof(int));
  cudaMalloc((void **)&dev_c, n*n*sizeof(int));
  cudaMalloc((void **)&dev_n, sizeof(int));

  // copiar as matrizes para a GPU
  cudaMemcpy(dev_a, a, n*n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, n*n*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(dev_n, &n, sizeof(int), cudaMemcpyHostToDevice);

  // processamento
  dim3 dimGrid(2, 2, 1);
```

```
  dim3 dimBlock(n/2, n/2, 1);
  soma_matriz<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, dev_n);

  // copiar a matriz resultado
  cudaMemcpy(c, dev_c, n*n*sizeof(int), cudaMemcpyDeviceToHost);

  //desalocar as matrizes da GPU
  cudaFree(dev_a);
  cudaFree(dev_b);
  cudaFree(dev_c);
  cudaFree(dev_n);

  // mostrar resultados
  for(i=0;i<n*n;i++){
    if (i%n==0 && i != 0) {
      printf("\n");
    }
    printf("%d ", c[i]);
  }
  printf("\n");

  free(a);
  free(b);
  free(c);

  return 0;
}
```

Aula 12 - CUDA - Adição de 3 matrizes ENG COMP

```
// EXERCICIO CUDA
// Adição de 3 Matrizes
// GRUPO gprec011

#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>

// We'll only work with power 2 matrix dimensions
#define MATRIX_SIZE 32
#define MAX_BLOCK_WIDTH 32

// Macro to handle cuda errors
#define checkError(err) ({ \
    if (err != cudaSuccess) { \
        printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__); \
        exit(EXIT_FAILURE); \
    } \
})

// kernel to perform two matrices addition C = A1 + A2
```

```
__global__
void add_matrix_kernel(int *d_A1, int *d_A2, int *d_C){
    // Thread position identifier
    int col = (blockDim.x * blockIdx.x) + threadIdx.x;
    int row = (blockDim.y * blockIdx.y) + threadIdx.y;

    // Do the addition
    d_C[row*MATRIX_SIZE+col] = d_A1[row*MATRIX_SIZE+col] +
d_A2[row*MATRIX_SIZE+col];
}

// Function to add 3 matrices and store the result on the C matrix
void add_3_matrices(int *h_A1, int *h_A2, int *h_A3, int *h_C)
{
    int size = (MATRIX_SIZE * MATRIX_SIZE) * sizeof(int);
    // Matrix pointers on the DEVICE
    int *d_A1, *d_A2, *d_A3;

    // Memory allocation
    cudaError_t err;
    err = cudaMalloc((void**) &d_A1, size);
    checkError(err);
    err = cudaMalloc((void**) &d_A2, size);
    checkError(err);
    err = cudaMalloc((void**) &d_A3, size);
    checkError(err);

    // Data transfer HOST->DEVICE
    err = cudaMemcpy(d_A1, h_A1, size, cudaMemcpyHostToDevice);
    checkError(err);
    err = cudaMemcpy(d_A2, h_A2, size, cudaMemcpyHostToDevice);
    checkError(err);
    err = cudaMemcpy(d_A3, h_A3, size, cudaMemcpyHostToDevice);
    checkError(err);

    // Kernel configurationa and launch

    // Since the thread number limit per block is 1024 (32X32), a matrix which
    // order is greater than 64 might lead to a launch error
    // To solve this, we divide the matrix recursively by 4 blocks until we reach
    // a block size <= 32X32
    int blockWidth = MATRIX_SIZE;
    int numBlocks = 1;
    do {
        blockWidth /= 2;
        numBlocks *= 2;
    } while (blockWidth > MAX_BLOCK_WIDTH);

    dim3 gridDim(numBlocks,numBlocks,1);
    dim3 blockDim(blockWidth, blockWidth, 1);

    // A2 = A1 + A2
    add_matrix_kernel<<<gridDim, blockDim>>>(d_A1, d_A2, d_A2);
    err = cudaGetLastError();
    checkError(err);
```

```
    // A3 = A2 + A3
    add_matrix_kernel<<<gridDim, blockDim>>>(d_A2, d_A3, d_A3);
    err = cudaGetLastError();
    checkError(err);

    // Data transfer DEVICE -> HOST
    err = cudaMemcpy(h_C, d_A3, size, cudaMemcpyDeviceToHost);
    checkError(err);

    // Memory free
    err = cudaFree(d_A1);
    checkError(err);
    err = cudaFree(d_A2);
    checkError(err);
    err = cudaFree(d_A3);
    checkError(err);
}

int main(int argc,char **argv){

    int *h_A1, *h_A2, *h_A3, *h_C;

    // Matrix allocation and initialization
    h_A1 = (int*) malloc(MATRIX_SIZE*MATRIX_SIZE*sizeof(int));
    h  A2 = (int*) malloc(MATRIX  SIZE*MATRIX  SIZE*sizeof(int));
    h_A3 = (int*) malloc(MATRIX_SIZE*MATRIX_SIZE*sizeof(int));
    h_C = (int*) malloc(MATRIX_SIZE*MATRIX_SIZE*sizeof(int));

    for(int i = 0; i < MATRIX_SIZE; i++){
        for(int j = 0; j < MATRIX_SIZE; j++){
            h_A1[i*MATRIX_SIZE + j] = 1;
            h_A2[i*MATRIX_SIZE + j] = 1;
            h_A3[i*MATRIX_SIZE + j] = 1;
        }
    }

    // Matrix Addition
    add_3_matrices(h_A1, h_A2, h_A3, h_C);

    // Output
    printf("Matrix C:\n");
    for(int i = 0; i < MATRIX_SIZE; i++){
        for(int j = 0; j < MATRIX_SIZE; j++){
            printf("%d ", h_C[i*MATRIX_SIZE+j]);
        }
        printf("\n");
    }

    return 0;
}
```

Aula 13 - Erlang - Fibonnaci TA

```
-- % Ele só usa o N como contador, Ele começa botando o 0, Depois 1, Depois 1, 2, E aí vaiE decrescendo o N (contador)

-module(fibs).
-export([main/1]).

fib(N) ->
    fib(N, 0, 1, [0]).

fib(0, Current, Next, Fibs) ->
    lists:reverse(Fibs); % Reverse the list as the order is important

fib(N, Current, Next, Fibs) ->
    fib(N - 1, Next, Current + Next, [Next | Fibs]).
```

Aula 13 - Erlang - Encontrar o vlr mínimo em uma lista TB -7,5

```
% ----Para Rodar digite no terminal linux:
%erl
%c(minimo).
%minimo:mini([10,20,30]).   ---- Aqui digitamos a lista desejada. No exemplo, iremos calcular o minimo dentre 10,20 e 30

-module(minimo).
-export([mini/1]).

mini([]) -> io:format("Nao eh possivel achar o minimo de uma lista vazia ~n");

mini([H|T])  ->
    Z = mini(H, T),
    io:format("O valor minimo eh: ~w~n ", [Z]).

mini(Min, [H|T]) ->
        case Min < H of
                true -> mini(Min, T);
                false -> mini(H, T)
        end;

mini(Min, []) -> Min.
```

Aula 13 - Erlang - QuickSort - ENG COMP

```
% EXERCICIO ERLANG
% QUICK SORT

-module(quick).
-export([start/1]).
```

```
% Duas funções start para tratar caso quando programa é lançado dentro ou não do
interpretador
start([X]) ->
    A = list_to_integer(X),
    quick(A);
start(X) ->
    quick(X).

quick(Order) ->
    A = [7, 3, 5, 10, 11, 2, 1, 15, 14, 7],
    Sorted = sort({Order, A}),
    io:format("Ordered List: ~w ~n", [Sorted]).

% Função que implementa quick sort
% sort(1, Lista) - ascendente
% sort(2, Lista) - descendente

sort({1, [Pivot|T]}) ->
    sort({1, [ X || X <- T, X < Pivot]}) ++
    [Pivot] ++
    sort({1, [ X || X <- T, X >= Pivot]});
sort({2, [Pivot|T]}) ->
    sort({2, [ X || X <- T, X > Pivot]}) ++
    [Pivot] ++
    sort({2, [ X || X <- T, X =< Pivot]});
sort({Order, [_|_]}) ->
    io:format("Invalid order ~w ~n", [Order]),
    [];
sort({_, []}) -> [].

% Outras formas implementar função sort / Criando 2 outras funções e tratando a ordem na
função start
%sort_asc([Pivot|T]) ->
%    sort_asc([ X || X <- T, X < Pivot]) ++
%    [Pivot] ++
%    sort_asc([ X || X <- T, X >= Pivot]);
%sort_asc([]) -> [].
%
%sort_desc([Pivot|T]) ->
%    sort_desc([ X || X <- T, X > Pivot]) ++
%    [Pivot] ++
%    sort_desc([ X || X <- T, X =< Pivot]);
%sort_desc([]) -> [].

% Tratando a ordem na função sort com if
%sort({Order, [Pivot|T]}) ->
%    if
%        Order == 1 ->
%            sort({1, [X || X <- T, X < Pivot]}) ++
%            [Pivot] ++
%            sort({1, [ X || X <- T, X >= Pivot]});
%        Order == 2 ->
%            sort({2, [X || X <- T, X > Pivot]}) ++
%            [Pivot] ++
%            sort({2, [ X || X <- T, X =< Pivot]})
```

```
%    end;
%sort({Order, []}) -> [].
```

Aula 14 - Erlang - ENG COMP

```
% EXERCICIO ERLANG
% TOKEN RING

-module(token).
-export([start/0, recvThenSend/1]).

recvThenSend(Order) ->
    io:format("starting process ~w ~n", [Order]),
    receive
        {SenderAlias, List, Token} ->
            io:format("~w received \"~s\" from ~w ~n", [Order, Token,SenderAlias]),
            send(List, Token, Order),
            io:format("ending process ~w ~n", [Order])
    end.

send([H|Tail], Token, SelfAlias) ->
    H ! {SelfAlias, Tail, Token}.

% Função para gerar demais processos do token ring, com exceção do primeiro
% Pid dos processos são colocados em uma lista
create(1) ->
    [spawn(token, recvThenSend, [1])];

create(N) ->
    create(N-1) ++ [spawn(token, recvThenSend, [N])].

start() ->
    List = create(3) ++ [self()],
    io:format("LIST: ~w ~n", [List]),

    Token = "hello",
    send(List, Token, 0),
    %Pid1 ! {self(), List, Token},
    receive
        {SenderAlias, [], Token} ->
            io:format("~w received \"~s\" from ~w ~n", [0, Token,SenderAlias]),
            io:format("ending process ~w ~n", [0])
    end.
```