Resumo de Computação de Alto Desempenho

Vítor Amorim Fróis

Conteúdo

OpenMP

Introdução

Qual a diferença entre overflow e underflow?

O overflow acontece quando estouramos o limite de representação de números antes da vírgula, isto é, da parte inteira. Já o underflow é um problema relacionado a mantissa, parte decimal.

Processo VS Programa

Processos são programas em execução. Um processo mantém o estado do fluxo de execução (principalmente) em registradores e o estado do fluxo de dados disponíveis. Além destas informações, são mantidas várias outras sobre descritores de arquivos, comunicação, proprietário, tempos.

Processos Paralelos VS Concorrentes

Processos concorrentes são dois ou mais processos que iniciaram e ainda não finalizaram a sua execução e concorrem pelos recursos de um sistema computacional. Processos paralelos são uma especialização de processos concorrentes, que sempre executam em elementos de processamento distintos ao mesmo tempo.

Qual a diferença entre o problema de consistência de memória e o problema de coerência de cache?

A consistência de memória garante que as operações de escrita e leitura em disco sejam realizadas da forma correta, utilizando-se, por exemplo, de regiões críticas. Já a coerência de cache busca garantir que todos os processadores tenham acesso às mesmas informações na cache através de invalidação e sincronização propostos em protocolos como o MESI.

Quais fatores impedem a escalabilidade de um algoritmo paralelo?

O principal fator que impede a escalabilidade é o overhead de comunicação, que adiciona um custo fixo à adição de novos processores. Esse custo fixo corresponde a unificação de informação e comunicação entre processadores. Assim, nem sempre a paralelização irá promover um speedup satisfatório.

Arquiteturas Paralelas

Coerência de Cache com Memória Compartilhada

As caches são muito utilizadas, pois reduzem demanda sobre redes de interconexão e a memórias mais lentas. Os dados replicados podem ficar desatualizados em writes.

- **MESI (Modified, Exclusive, Shared, Invalid)**: é um protocolo para garantir coerência de cache. Funciona marcando blocos que sofreram alterações como inválido ou modificado e especificando blocos exclusivos ou compartilhados.
- **Falso Compartilhamento**: ocorre quando dados fierentes de um bloco compartilhado são atualizados pelos processadores em suas caches. O bloco entã fica inválido mas de fato não precisaria. A solução é manter maior distância entre os dados espalhados nas caches.

Flynn Taxonomy

MIMD com Memória Compartilhada - Multiprocessadores

Diferente módulo de memória são usuais com mesmo endereçamento. As requisições podem ocorrer em ordenes diferentes da esperada, levando a incoerência. Os modelos a seguir organizam R/W nos módulos.

- **Estrita**: escritas são visíveis instantaneamente a todos os processos.
- **Sequencial**: processos veem a mesma ordem de requisições de acesso à memória.
- **de Processador**: escritas de uma processo são observadas por ele na mesma ordem.
- **Fraca**: sincronizações explícitas são consistentes sequencialmente.

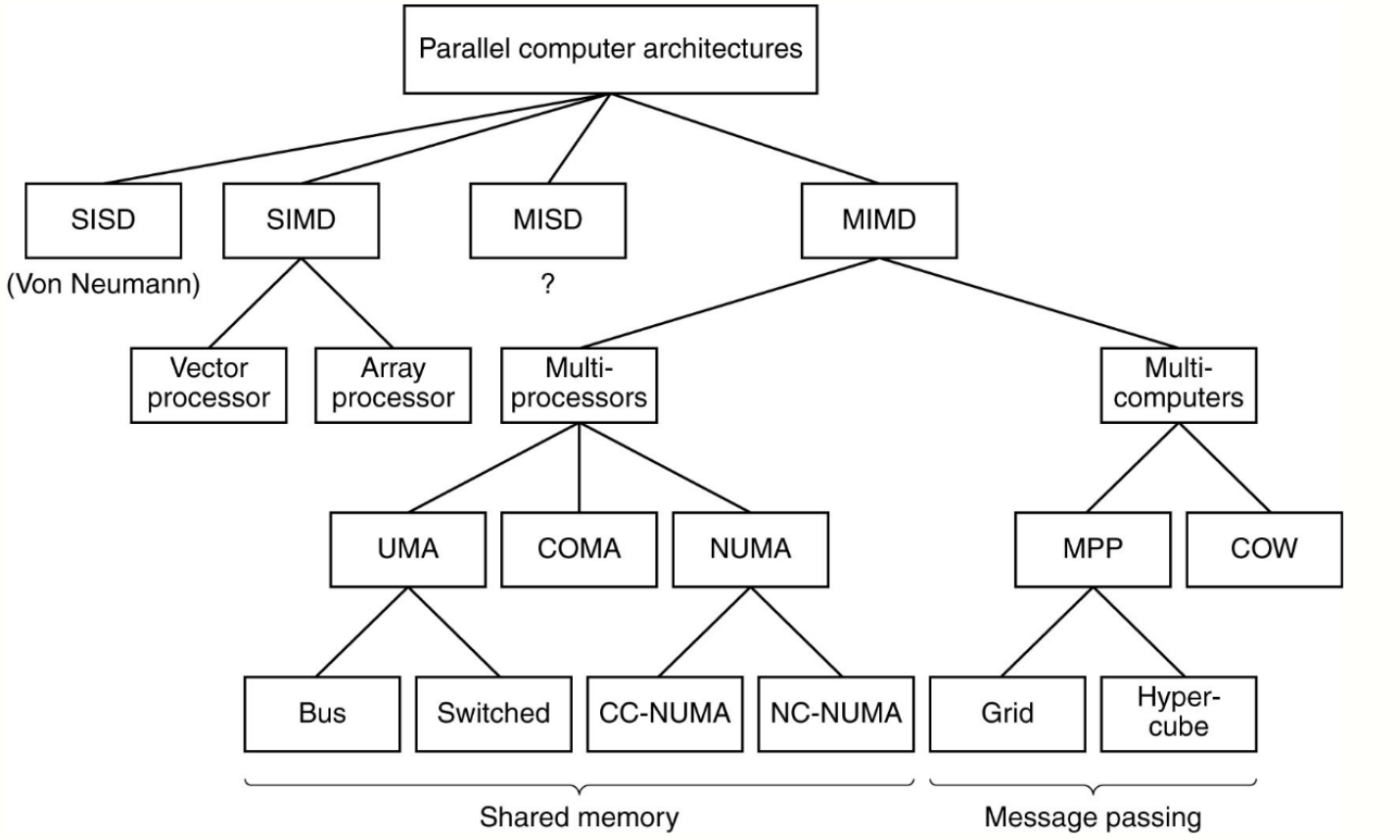

Figura 1: Taxonomia de Flynn

- **de Liberação**: sincronizações explícitas são consistentes ao processador em relação aos demais. São introduzidas operações semelhantes a lock/unlock.

MIMD com Memória Distribuída - Multicomputadores

Nós são unidades assíncronas e independentes com CPU, memória e I/O.

- **Clusters:** buscam redundância para garantir retorno de requisições não retornadas.
- **MPPs**: clusters de milhões de dólares.
- **Grids**: sistemas utilizados para conectar computadores isolados geograficamente.

SIMD - Supercomputadores

São ótimas para executar operações matriciais e vetoriais. Tradicionalmente processadores vetoriais são ULAs com pipeline cujos estágios operam números de ponto flutuante. Já processadores matricias são ULAs paralelas.

Por que a GPU não é uma SIMD pura?

Uma máquina SIMD pura possui uma única thread com múltiplos dados. A GPU possui múltiplos threads, e dessa forma, se configura comoo um misto de MIMD e SIMD. (falar sobre os dois níveis de multithread (primeiro sobre streaming multiprocessors que processam mesmas instrucoes em multiplas threads para multiplos dados e segundo sobre o conjunto de SMs, onde cada SM executa diferentes tarefas)

Compare a rede Ômega (Multiestágio) com crossbar e barramento em relação a desempenho e custos.

As redes de barramento são bem simples, apresentando um baixo custo de implementação, em troca de um desempenho lento, dado que apenas uma comunicação pode acontecer por vez. Já a Crossbar supera esse problema ligando direamente os nós da rede em pares. A principal desvantagem dessa abordagem é o alto custo envolvido. As redes Ômega representam um meio termo entre as vantagem e desvantagens apresentadas nas primeiras redes. As entradas são conectadas às saídas através de vários estágios, garantindo uma comunicação robusta e eficiente sem altos custos presentes na Crossbar.

Avaliação de Desempenho

Redes de Interconexão podem afetar a Granulação de uma região paralela?

Sim. As redes de Interconexão podem afetar a granularidade de uma região paralela ao influenciar o custo de comunicação e sincronização entre os processos. Assim, uma computação de granulação fina e comunicação de elevado custo pode ser convertida em granulação grossa com baixa comunicação caso o desempenho da rede de Interconexão seja ruim.

Redes de Interconexão afetam diretamente o tempo de resposta ou o tempo de execução?

O tempo de execução é o tempo gasto por recursos do processador. As redes de interconexão interferem no tempo de comunicação, e não de execução. Dessa forma, podemos afirmar que apenas o tempo de resposta é afetado.

Redes de Interconexão podem produzir speedup superlinear?

Não. O speedup superlinear é um evento muito específico que acontece quando a paralelização alivia a memória cache dos processaodre, garantindo menos acessos a memória principal e consequentemente tempo de resposta menor.

Speedup

Determina o ganho de desempenho, relacionando os tempos de execução da versão sequencial e paralela. $S_p = T_{seq}/T_{parp}$

Eficiência

Determina a eficiência no uso de p processadores, relacionando o speedup e o número de processadores. $E = S_{pp}/p$

Exercícios

Particionamento e a Comunicação do PCAM Prova 1 de 2023

Desenvolva o Particionamento e a Comunicação da metodologia PCAM para o projeto paralelo do problema a seguir: considere uma matriz quadrada A de ordem Ne com números inteiros não negativos, cujos valores são determinados pseudo-aleatoriamente no código-fonte entre 0 e 99. Calcule a média de todos os valores da matriz A e então determine dois outros

valores: 1. quantos elementos menores que esta média existem em cada coluna j de A; 2. o desvio padrão populacional de cada coluna j de A Insira os resultados de (1) nas posições j de vetor medial com N posições. Insira os resultados de (2) nas posições j do vetor `dsvpdr[]` com N posições. As saidas da aplicação, como exemplificado abaixo, devem ser o vetor `medial[]` e depois o vetor `dsvpdr[]`. Sem perda de generalidade e exclusivamente para esta, considere $N = 4$ para fazer o Particionamento e a Comunicação do PCAM.

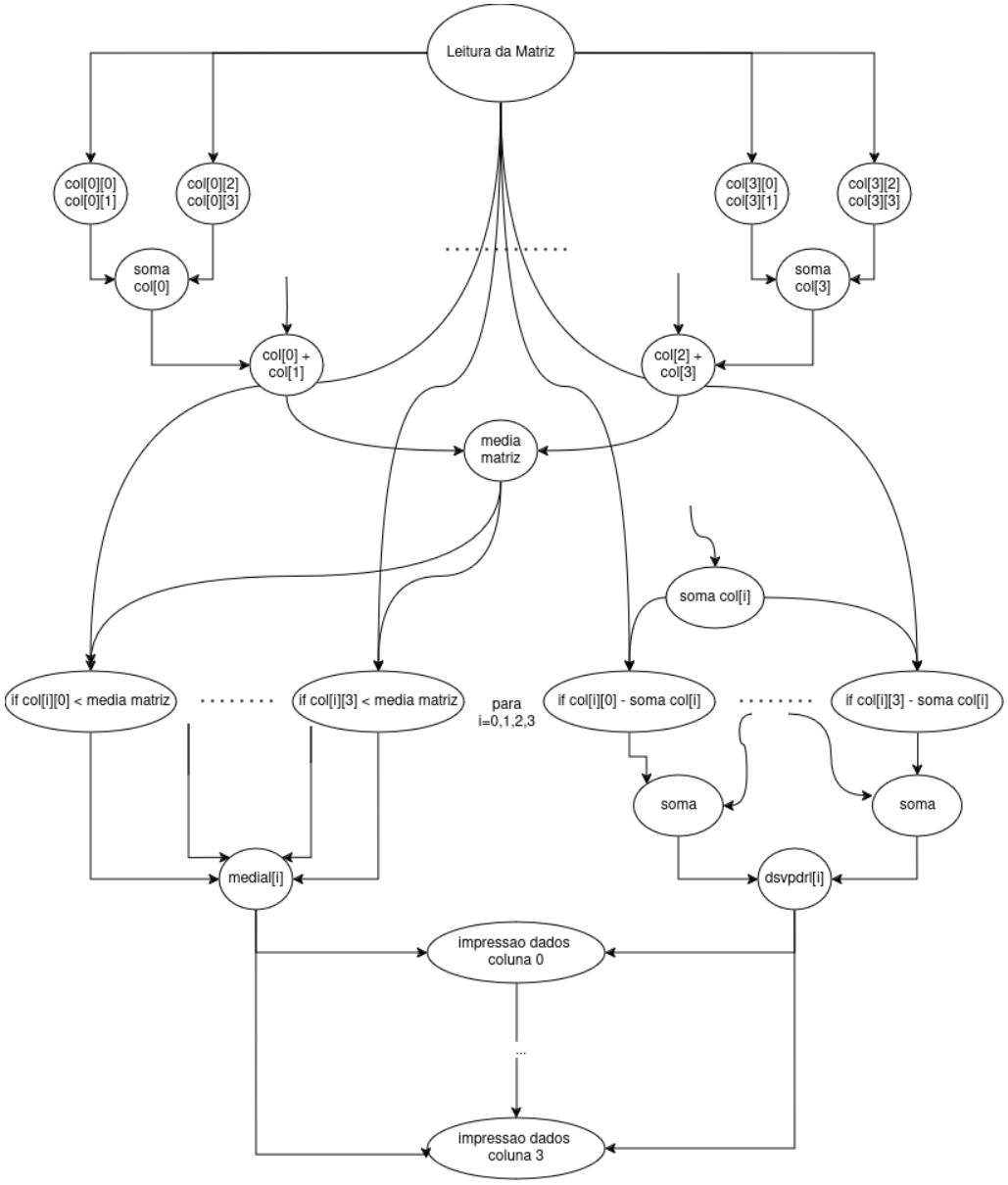

Figura 2: Grafo de Dependências

Particionamento: é realizado um particionamento em duas etapas, tanto por dados quanto por tarefas. A leitura da matriz é feita de forma sequencial. Logo após, os valores de cada coluna `i` são somados utilizando redução e armazenados em `somacoluna[i]`. O vetor é então reduzido para encontrar a média dos valores da matriz. A seguir, o cálculo de `medial[]` e `dsvpdr[]` são realizados em paralelo. Um bloco contendo tarefas é criado para cada $i = 0, 1, 2, 3$ e `medial[i]` e `dsvpdr[i]`. Com os vetores calculados, as colunas são impressas em ordem, de forma sequencial.

Comunicação: o cálculo de `medial[i]` utiliza o vetor `somacoluna[i]` além da média da matriz e os resultados são unificados utilizando região crítica em caso de memória compartilhada e redução em caso de memória distribuída. Já o cálculo de `dsvpdr[i]` utiliza `somacoluna[i]` e os valores obtidos durante a leitura da matriz, além de unificação semelhante ao caso passado.

Código OpenMP da questão acima

A ordem N da matriz A e o número T de threads devem estar indos por diretivas `#define` no começo do código. Use-os nas diretivas OMP adequadamente.

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

#define N 10
#define T 10
// Esse código foi feito para rodar com T = N

int main(){
    char linebreak;
    int A[N][N];
    omp_set_num_threads(T);

    // Leitura da matriz
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            scanf("%d", &A[i][j]);
        }
    }

    // Soma das colunas
    int soma_colunas[N], i, j;
    #pragma omp parallel shared(soma_colunas) private(i, j)
    {
        j = omp_get_thread_num();
        soma_colunas[j] = 0;
        for(i = 0; i < N; i++)
            soma_colunas[j] += A[i][j];
    }

    // Média da matriz
    double media = 0;
    #pragma omp parallel for reduction(+:media) private(i)
    for(i = 0; i < N; i++)
        media += soma_colunas[i];

    media = media / (N*N);

    // Média dos valores da matriz
    int medial[N], soma_diferenca[N];
    double dsvpdr[N];
    #pragma omp parallel
    {
        #pragma omp single
        {
            for(j = 0; j < N; j++){
                // Task medial
                #pragma omp task private(i)
                {
                    medial[j] = 0;
                    for(i = 0; i < N; i++)
                        if (A[i][j] < media)
                            medial[j] += 1;
                }
                // Task dsvpdr
```

```
                #pragma omp task private(i)
                {
                    soma_diferenca[j] = 0;
                    for(i = 0; i < N; i++)
                        soma_diferenca[j] += pow(A[i][j] - soma_colunas[j], 2);
                    dsvpdr[j] = sqrt(soma_diferenca[j] / N);
                }
            }
        }
    }

    for(i = 0; i < N; i++)
        printf("(%d) dsvpdr = %f; medial = %d\n", i, dsvpdr[i], medial[i]);

    return 0;
}
```

PCAM Trabalho 1 2024

Elabore um projeto de algoritmo paralelo que seja capaz de realizar Convolução.

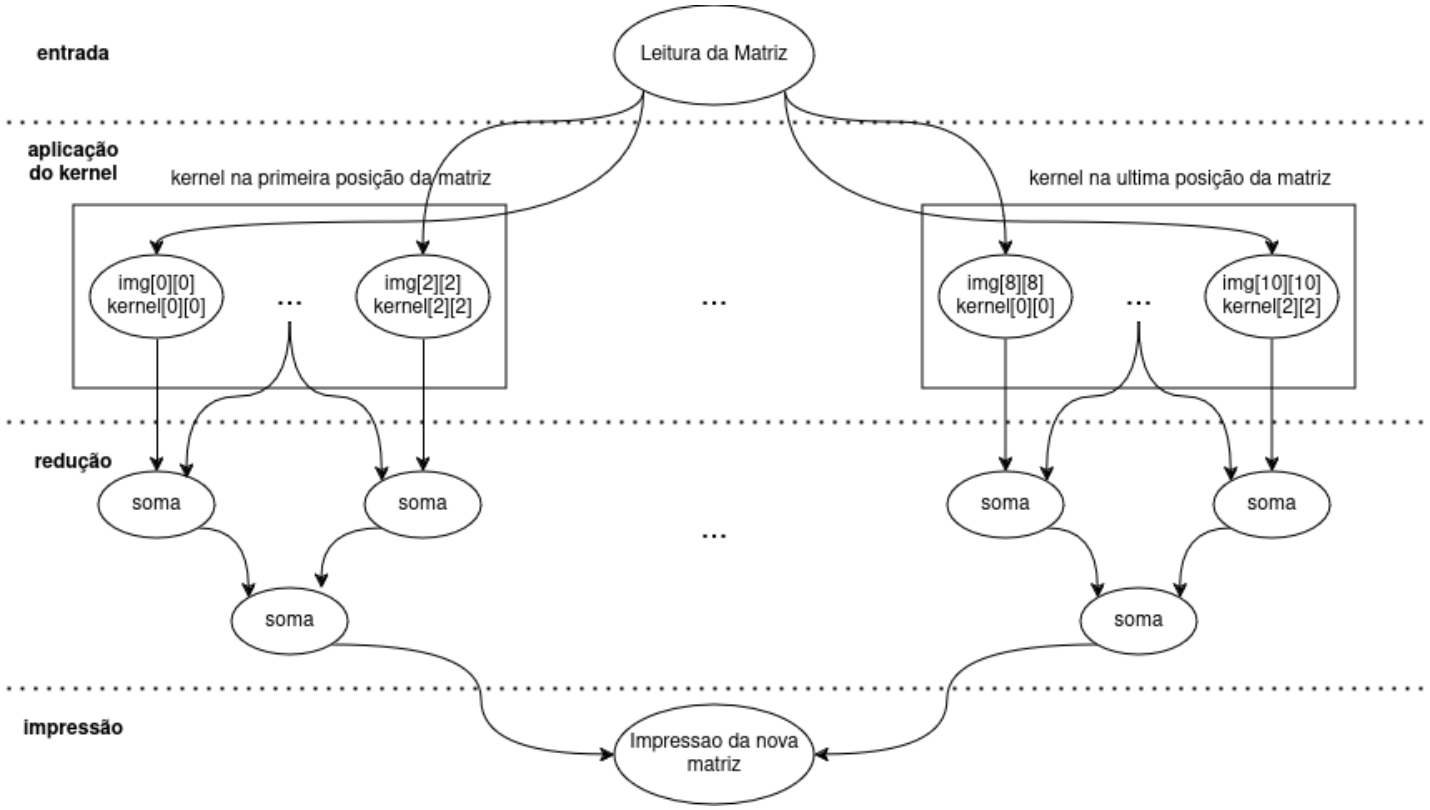

Figura 3: Grafo de Dependências

Particionamento: é realizado um particionamento por dados. A aplicação do kernel em cada posicção da matriz pode ser feita de forma paralela e as multiplicações (cada posição do kernel pelo valor correspondente) também. Uma tarefa lê ambas matriz de forma sequencial. Em seguida, são criados diversos blocos de tarefas, de forma que cada bloco irá cuidar de uma combinação *kernel* × *posição da matriz*. Serão criadas tarefas para multiplicar os elementos vizinhos ao kernel e tarefas dependentes para realizar a redução desses valores. Por fim, é realizada a impressão da nova matriz.

Comunicação: o grafo de comunicação é semelhante ao grafo de dependências. Devemos ressaltar, porém, que para cada ponto i, j da imagem deve ser realizadas comunicação com os $m \times m$ vizinhos ao redor, de acordo com o kernel. Após as multiplicações em cada ponto, é realizada uma redução de forma que as tarefas se conversam em pares até encontrar o valor final de cada ponto da nova matriz.

Aglomeração: a etapa de aglomeração serve para planejar como as tarefas serão divididas entre os processadores. É importante notar que para cada aplicação do kernel na matriz, deve-se realizar comunicação com os valores vizinhos. Dessa

forma, uma granulação mais fina pode aumentar a comunicação entre diferentes processos, aumentar a quantidade de dados replicados e deixar o processamento mais lento. Assim, é ideal utilizar uma granulação mais grossa na divisão de processos em threads e buscar jogar tarefas relativas a localidades próximas na mesma thread.

Mapeamento: é dinâmico, de forma que cada elemento irá receber um processo. Esperamos que esse processo seja feito de maneira uniforme.

Correção de Código OpenMP

Indique os erros no código abaixo. Logo após isso indique a solução

```
#include <stdio.h>

long long num_passos = 1e10;
double passo;

int main(){
    int i;
    double x, pi, soma=0.0;

    #pragma omp parallel for simd firstprivate(soma)
    for(int i = 0; i < num_passos; i++){
        x = (i + 0.5)*passo;
        soma = soma + 4.0/(1.0 + x*x);
    }

    pi = soma * passo;

    printf("O valor de Pi é %f\n", pi);
    return 0;
}
```

R.: *O erro está na cláusula `firstprivate(soma)`. Queremos reduzir o valor de soma entre todos processos para um único valor. Assim, o correto é utilizar a cláusula `reduction(+:soma)`.*

Message Passing Interface (MPI)

MPI visa oferecer uma alternativa a programação sequencial presente nos códigos comuns. Diversas instâncias sequenciais que se comunicam formam o MPI e se comunicam através de mensagens. Se tornaram populares por funcionarem tanto em sistemas distribuídos quanto multiprocessadores. Utilizam o paradigma MPMD (Multiple Program, Multiple Data) ou SPMD (Single Program, Muliple Data).

Para usar, use `#include <mpi.h>` e compile com `mpicc -o programa programa.c -np 2`. A flag `np` indica que 2 processos serão criados.

Programas MPI

- `MPI_Init()`: cada processo deve realizar uma call
- `MPI_COMM_WORLD()`: comunicador definido na inicialização. Cada comunicador possui um grupo, lista de processos.
- `MPI_Finalize()`: finaliza todos processos

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

int main(int argc, char **argv)  {
  int my_rank, num_procs;
  int tag, src, dst, test_flag, i;
  int buffersize, sizemsgsnd, nrmsgs;

  char *buffersnd;
```

```
12    char msgsnd[30], msgrcv[30], msgsync[30];
13
14    MPI_Status  status;
15    MPI_Request mpirequest_mr, mpirequest_mr_ready;
16
17    MPI_Init(&argc, &argv);
18    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
19    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
20
21    ... // codigo
22
23    free(bufrecv);
24      fflush(0);
25      ret = MPI_Finalize();
26      if (ret == MPI_SUCCESS)
27          printf("MPI_Finalize success! \n");
28      return(0);
29  }
```

Comunicação Ponto a Ponto

- **Síncrono**: recebe informação sobre a conclusão da comunicação, ou seja, a operação não completa (mesmo em background) até que o receiver receba a mensagem.
- **Assíncrono**: recebe informação apenas na sáida da mensagem.
- **Bloqueantes**: operações retornam apenas quando a operação foi concluída.
- **Não Bloqueantes**: operações retornam imediatamente e permite o programa realizar outras tarefas. Mais tarde é possível checar se o envio foi concluído.

O send não-bloqueante síncrono do MPI cria uma thread para processar a operação, permitindo que o programa continue a execução e recebe informação sobre a conclusão da comunicação, garantindo que o processo termine apenas quando o receiver receber a mensagem.

No caso que a comunicação bloqueante utiliza buffer, a mensagem está segura assim que for copiada para o buffer, permitindo que o send retorne. O receive bloqueante retorna quando recebe a mensagem diretamente do send ou do buffer de recebimento.

> Bloqueante não é sinônimo de síncrono em aplicações paralelas!

Sends do MPI

- **Padrão** (`MPI_Send`): conclui quando a mensagem tiver sido enviada. Pode ou não ter buffer, a implementação decide.
- **Buffered** (`MPI_Bsend`): o usuário deve criar o buffer com antedência.
- **Síncrono** (`MPI_Ssend`): sincroniza sender e receiver, e sabemos que a mensagem foi recebida.
- **Ready** (`MPI_Rsend`): se um receive existe, completa imediatamente, caso contrário o resultado é indefinido.

As primitivas não bloqueantes possuem um `I` na frente, como por exemplo `MPI_Isend`, o send padrão não bloqueante.

Exemplo de comunicação ponto a ponto não bloqueante

No código abaixo, as primitivas send Padrão, Bloqueante e Síncrono são utilizadas

`modo-com.c`

```
29  tag = 0;
30  if (my_rank == 0)
31  {
32      src = 1;  // determina o processo de origem de uma msg recebida como nao bloqueante
33      dst = 1;  // determina o processo de destino das msgs enviadas pelo processo com rank 0
34
35      // recebe msg com primitiva nao bloqueante standard
36      MPI_Irecv(msgrcv, sizeof(msgrcv), MPI_CHAR, src, tag, MPI_COMM_WORLD, &mpirequest_mr);
37      MPI_Test(&mpirequest_mr, &test_flag, &status);
38      if(test_flag)
39          printf("Rank 0: MPI_Test flag = %d, msgrcv=%s. \n", test_flag, msgrcv);
```

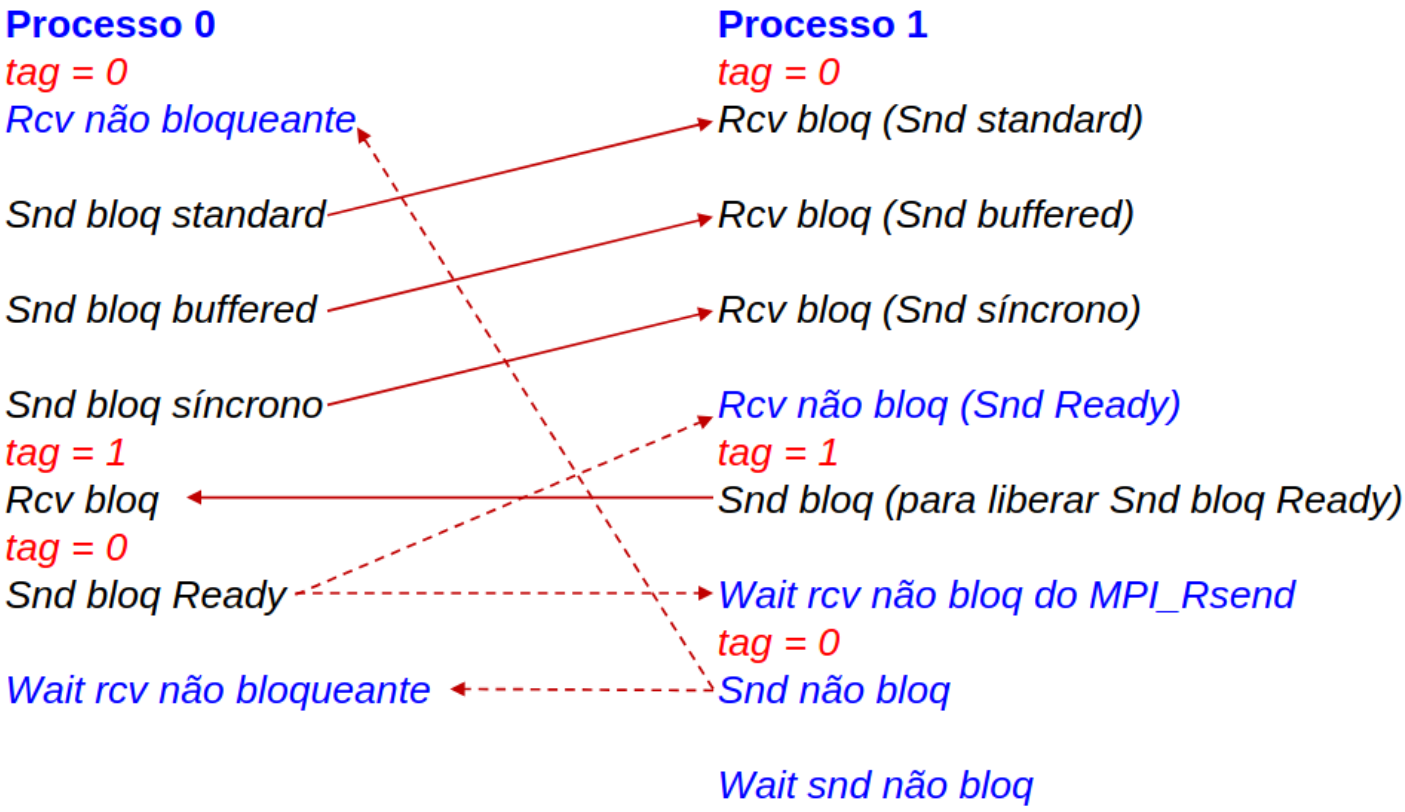

Figura 4: Diferentes modos de comunicação no MPI

```
40      else
41          printf("Rank 0: MPI_Test flag = %d, nao recebeu a msg ainda. \n", test_flag);
42
43      strcpy(msgsnd, "Blocking standard send");
44      MPI_Send(msgsnd, strlen(msgsnd)+1, MPI_CHAR, dst, tag, MPI_COMM_WORLD);
45
46      strcpy(msgsnd, "Blocking buffered send");
47      MPI_Pack_size(strlen(msgsnd)+1, MPI_CHAR, MPI_COMM_WORLD, &sizemsgsnd);
48      nrmsgs = 1; // apenas uma msg bsend sera enviada
49      buffersize = sizemsgsnd + nrmsgs * MPI_BSEND_OVERHEAD;
50      buffersnd = (char*) malloc(buffersize);
51      MPI_Buffer_attach(buffersnd, buffersize);
52      // envia a msg bsend usando o buffer criado pelo usuario
53      MPI_Bsend(msgsnd, strlen(msgsnd)+1, MPI_CHAR, dst, tag, MPI_COMM_WORLD);
54      MPI_Buffer_detach(buffersnd, &buffersize);
55
56      strcpy(msgsnd, "Blocking synchronous send");
57      MPI_Ssend(msgsnd, strlen(msgsnd)+1, MPI_CHAR, dst, tag, MPI_COMM_WORLD);
58
59      tag = 1; // Se este MPI_Recv abaixo usar a tag 0 entao esta msg podera ser recebida pelo MPI_Irecv
60      MPI_Recv(msgsync, sizeof(msgsync), MPI_CHAR, src, tag, MPI_COMM_WORLD, &status);
61
62      tag = 0; // retorna para a tag = 0 usada nos sends e receives
63      strcpy(msgsnd, "Blocking ready send");
64      MPI_Rsend(msgsnd, strlen(msgsnd)+1, MPI_CHAR, dst, tag, MPI_COMM_WORLD);
65
66      if(!test_flag) {
67          MPI_Wait(&mpirequest_mr, &status);
68          printf("Rank 0: MPI_Wait liberou: msgrcv=%s. \n", msgrcv);
```

```
69      }
70  } // fim do if (my_rank == 0)
71  else   // my_rank == 1
72  {
73      dst = 0; // determina o destino da msg a ser enviada
74      src = 0; // determina o processo de origem da msg recebida
75
76      // recebe do send bloqueante standard
77      MPI_Recv(msgrcv, sizeof(msgrcv), MPI_CHAR, src, tag, MPI_COMM_WORLD, &status);
78      printf("Rank 1: msg 01 recebida: %s \n", msgrcv);
79      fflush(0);
80
81      // recebe do send bloqueante com buffer
82      MPI_Recv(msgrcv, sizeof(msgrcv), MPI_CHAR, src, tag, MPI_COMM_WORLD, &status);
83      printf("Rank 1: msg 02 recebida: %s \n", msgrcv);
84      fflush(0);
85
86      // recebe do send bloqueante sincrono
87      MPI_Recv(msgrcv, sizeof(msgrcv), MPI_CHAR, src, tag, MPI_COMM_WORLD, &status);
88      printf("Rank 1: msg 03 recebida: %s \n", msgrcv);
89      fflush(0);
90
91      // dispara previamente o MPI_IRecv para o send bloqueante ready existente no processo 0
92      MPI_Irecv(msgrcv, sizeof(msgrcv), MPI_CHAR, src, tag, MPI_COMM_WORLD, &mpirequest_mr_ready);
93
94      tag = 1;
95      strcpy(msgsync, "unlock ready send");
96      MPI_Send(msgsync, strlen(msgsync)+1, MPI_CHAR, dst, tag, MPI_COMM_WORLD);
97      // aguarda pelo recebimento da msg do MPI_Rsend. Ira finalizar o MPI_Irecv acima.
98      MPI_Wait(&mpirequest_mr_ready, &status);
99
100     printf("Rank 1: msg 04 recebida: %s \n", msgrcv);
101     fflush(0);
102     tag = 0;
103     // send nao bloqueante
104     strcpy(msgsnd, "Nonblocking standard send");
105     MPI_Isend(msgsnd, strlen(msgsnd)+1, MPI_CHAR, dst, tag, MPI_COMM_WORLD, &mpirequest_mr);
106     MPI_Wait(&mpirequest_mr, &status);
107 }
```

Saída As mensagens são recebidas na ordem esperada

```
$~ mpirun -np 2 --oversubscribe CommPontoAPonto
Rank 0: MPI_Test flag = 0, nao recebeu a msg ainda.
Rank 1: msg 01 recebida: Blocking standard send
Rank 1: msg 02 recebida: Blocking buffered send
Rank 1: msg 03 recebida: Blocking synchronous send
Rank 1: msg 04 recebida: Blocking ready send
Rank 0: MPI_Wait liberou: msgrcv=Nonblocking standard send.
```

Grupos e Comunicação Coletiva

`MPI_COMM_WORLD` é o Intracomunicador padrão definido pelo MPI. Novos intracomunicadores especificam subgrupos de `MPI_COMM_WORLD`. Intercomunicadores permitem comunicações entre processos com intracomunicadores diferentes.

- **Barrier** (`MPI_Barrier`): N-to-N, sincroniza todos os processos.
- **Broadcast** (`MPI_Bcast`): 1-to-N, envia a mesma mensagem para todos processos no contexto.
- **Gather** (`MPI_Gather`): n-to-1, coleta dados dispersos entre todos os processos.
- **Scatter** (`MPI_Scatter`): 1-to-n, espalha informações de um processo sobre todos os processos de um domínio de comunicação.
- **Reduce** (`MPI_Reduce`): associa uma computação a comunicação, considerando todos os processos de um domínio.

Todas essas primitivas tem versões não bloqueantes. Há uma barreira não bloqueante, tal que processos retornam mesmo que os demais não tenham chegado na barreira.

Exemplo de uso de primitivas coletivas (reduce)

Nesse código os processos somam os dados localmente e enviam para o root realizar a redução via soma.

06-reduce.c

```
20 num_elements_per_proc = 4;
21
22 rand_nums = create_rand_nums(num_elements_per_proc);
23
24 for (i = 0; i < num_elements_per_proc; i++) {
25   local_sum += rand_nums[i];
26 }
27
28 // Print the random numbers on each process
29 printf("Local sum for process %d - %f, avg = %f\n", world_rank, local_sum, local_sum / num_elements_per_proc);
30
31 // Reduce all of the local sums into the global sum
32 MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, root, MPI_COMM_WORLD);
33
34 // Print the result
35 if (world_rank == root) {
36   printf("Total sum = %f, avg = %f\n", global_sum, global_sum / (world_size * num_elements_per_proc));
37 }
```

Saída

```
$~ mpirun -np 4 06-reduce 4
Local sum for process 2 - 2.332896, avg = 0.583224
Local sum for process 3 - 2.419048, avg = 0.604762
Local sum for process 0 - 2.816110, avg = 0.704027
Local sum for process 1 - 2.999350, avg = 0.749838
Total sum = 10.567404, avg = 0.660463
```

Intercomunicadores

Primitivas coletivas têm comportamentos diferentes com intercomunicadores. No caso do `MPI_Gather`, o processo root não fornece uma porção dos dados. De forma semelhante, no `MPI_Scatter`, o root não recebe uma porção dos dados.

A geração de novos processos é feita com `MPI_Comm_spawn()`

Exemplo de spawn de processos e intercomunicadores

São criados `NUM_SPAWNS` processos filhos a partir do master.

03-spawn-collective-master.c

```
65 #define NUM_SPAWNS 4
66
67 char msg_0[] = "hello worker, i'm your master";
68 char msg_1[50], master_data[] = "workers to work", worker[40];
69
70 MPI_Get_processor_name(processor_name, &name_len);
71 for (i = 0; i < NUM_SPAWNS; i++)
72     vet[ i ] = i;
73 src = dst = root = 0;
74 MPI_Comm_spawn(worker, MPI_ARGV_NULL, NUM_SPAWNS, MPI_INFO_NULL, root, MPI_COMM_WORLD, &inter_comm, errcodes);
75
76 MPI_Send(msg_0, 50, MPI_CHAR, dst, tag, inter_comm);
```

```
77 MPI_Irecv(msg_1, 50, MPI_CHAR, src, tag, inter_comm, &mpirequest_mr);
78
79 MPI_Send(master_data, 50, MPI_CHAR, dst, tag, inter_comm);
80
81 MPI_Scatter(vet, 1, MPI_INT, &buf_rcv, 1, MPI_INT, MPI_ROOT, inter_comm);
82 MPI_Gather(vet, 1, MPI_INT, &vet_master, 1, MPI_INT, MPI_ROOT, inter_comm);
83
84 MPI_Wait(&mpirequest_mr, &status);
85
86 for (i = 0; i < NUM_SPAWNS; i++)
87 printf("Master (%s): msg_1=%s,vet[%d]=%d,buf_rcv=%d,vet_master[%d]=%d\n",processor_name,msg_1,i,vet[i],buf_rcv
```

03-spawn-collective-worker.c

```
34 vet = (int*) malloc(num_proc*sizeof(int));
35
36 src = dst = root = 0;
37 if ( my_rank == 0 )
38 {
39   MPI_Recv(msg_0, 50, MPI_CHAR, src, tag, inter_comm, &status);
40   MPI_Send(msg_1, 50, MPI_CHAR, dst, tag, inter_comm);
41
42   MPI_Recv(master_data, 50, MPI_CHAR, src, tag, inter_comm, &status);
43   strcpy(workers_data, master_data);
44 }
45 MPI_Bcast(workers_data, 50, MPI_CHAR, root, MPI_COMM_WORLD);
46
47 MPI_Scatter(vet, 1, MPI_INT, &buf_rcv, 1, MPI_INT, root, inter_comm);
48
49 for (i = 0; i < num_proc; i++){
50   printf("SLV nr %d (%s):workers_data=%s,vet[%d]=%d,buf_rcv=%d\n", my_rank, processor_name, workers_data, i, v
51 }
52
53 MPI_Gather(&buf_rcv, 1, MPI_INT, vet, 1, MPI_INT, root, inter_comm);
```

O master preenche `vet=[0,1,2,3]` e realiza o Spawn de 4 processos *worker*. O master realiza um `Send`, recebido pelo Worker 0 e responde com um `Send`, recebido pelo Master com `IRcv`. O Master realiza um `Send master_data`, recebido pelo Worker 0, que faz o broadcast para o grupo de workers. O master realiza um Scatter de `vet` e o worker 0 faz um Scatter de `vet`, que possui lixo. Ambos processos realizar o Gather para receber as informações. Os workers printam lixo, pois o Gather ocorre após o `print`.

Saída

```
$~ mpirun --np 1 --oversubscribe 03-spawn-collective-master
SLV nr 0 (camel):workers_data=workers to work,vet[0]=-1653793569,buf_rcv=0
SLV nr 0 (camel):workers_data=workers to work,vet[1]=22484,buf_rcv=0
SLV nr 0 (camel):workers_data=workers to work,vet[2]=0,buf_rcv=0
SLV nr 0 (camel):workers_data=workers to work,vet[3]=0,buf_rcv=0
SLV nr 2 (camel):workers_data=workers to work,vet[0]=-1151853675,buf_rcv=2
SLV nr 2 (camel):workers_data=workers to work,vet[1]=28878,buf_rcv=2
SLV nr 2 (camel):workers_data=workers to work,vet[2]=0,buf_rcv=2
SLV nr 2 (camel):workers_data=workers to work,vet[3]=0,buf_rcv=2
SLV nr 3 (camel):workers_data=workers to work,vet[0]=754593807,buf_rcv=3
SLV nr 3 (camel):workers_data=workers to work,vet[1]=29987,buf_rcv=3
SLV nr 3 (camel):workers_data=workers to work,vet[2]=0,buf_rcv=3
SLV nr 3 (camel):workers_data=workers to work,vet[3]=0,buf_rcv=3
SLV nr 1 (camel):workers_data=workers to work,vet[0]=597966703,buf_rcv=1
SLV nr 1 (camel):workers_data=workers to work,vet[1]=31262,buf_rcv=1
SLV nr 1 (camel):workers_data=workers to work,vet[2]=0,buf_rcv=1
SLV nr 1 (camel):workers_data=workers to work,vet[3]=0,buf_rcv=1
Master (camel): msg_1=hi master,vet[0]=0,buf_rcv=805306368,vet_master[0]=0
```

```
Master (camel): msg_1=hi master,vet[1]=1,buf_rcv=805306368,vet_master[1]=1
Master (camel): msg_1=hi master,vet[2]=2,buf_rcv=805306368,vet_master[2]=2
Master (camel): msg_1=hi master,vet[3]=3,buf_rcv=805306368,vet_master[3]=3
```

MPI e OMP

MPI com OMP permite executar diferentes processos MPI em máquinas (ou nós ou hosts) distintas, onde cada processo MPI com suas threads compartilhando memória na máquina local. Há um conjunto de funções no MPI para o suporte de threads com OMP, sendo a mais importante `MPI_Init_thread()`.

```
32 if ( myrank == 0)
33 {
34   for (i = 1; i < npes; i++)
35   {
36     MPI_Recv(bufrecv, MAXMESSAGE, MPI_CHAR, MPI_ANY_SOURCE,
37                         MPI_ANY_TAG, MPI_COMM_WORLD, &status);
38     printf("%s\n", bufrecv);
39     fflush(0);
40   }
41 }
42 else
43 {
44   // OMP sequential region
45   #pragma omp parallel private(local_i, total_threads, thread_number) shared(i) num_threads(NT)
46   {
47     // OMP parallel region
48       total_threads = omp_get_num_threads();
49       thread_number = omp_get_thread_num();
50       #pragma omp critical (adding)
51       {
52         local_i = ++i;
53       }
54       printf("Hello from thread %d/%d, from process %d/%d on %s. i = %d\n",
55         thread_number, total_threads, myrank, npes, processor_name, local_i);
56   }
57   // OMP sequential region
58   msgtag = 1;
59   sprintf(bufsend, "Hello World from process %d with i=%d", myrank, i);
60   fflush(0);
61   dest = 0;
62   MPI_Send(bufsend, strlen(bufsend)+1, MPI_CHAR, dest, msgtag, MPI_COMM_WORLD);
63 }
```

O processo 0 realiza o `MPI_Recv` dos processos 1 e 2, os quais criam threads utilizando OpenMP. Cada uma das threads criadas realiza um print. No final, cada processo realiza um `MPI_Send`.

```
$~ mpirun -np 3 --oversubscribe 02-hello-mpi-omp-snd-rcv
Hello from thread 3/4, from process 2/3 on camel. i = 1
Hello from thread 1/4, from process 2/3 on camel. i = 2
Hello from thread 2/4, from process 2/3 on camel. i = 3
Hello from thread 3/4, from process 1/3 on camel. i = 1
Hello from thread 1/4, from process 1/3 on camel. i = 2
Hello from thread 0/4, from process 1/3 on camel. i = 3
Hello from thread 2/4, from process 1/3 on camel. i = 4
Hello from thread 0/4, from process 2/3 on camel. i = 4
Hello World from process 1 with i=4
Hello World from process 2 with i=4
MPI_Finalize success!
MPI_Finalize success!
MPI_Finalize success!
```

Compute Unified Device Architecture (CUDA)

Nos anos 2000 surgiram as GPUs (Graphics Processing Units) para substituir controladores VGA. Essas GPUs possuem características SIMD, porém não são SIMD puras. Implementam o paradigma SIMT (Single Instruction Multiple Thread), onde uma única instrução é executada por múltiplas threads. SIMT flexibiliza máquinas SIMD por não estarem restritas à execução de uma única instrução, mas à execução simultânea de múltiplas threads, majoritariamente executando a mesma instrução.

Modelo de Programação CUDA

CUDA (Compute-Unified Device Architecture) é um modelo de programação paralela que atua como uma extensão de C/C++ e Fortran para programar dispositivos heterogêneos e começou a ser desenvolvido pela NVDIA em 2006.

- **Thread**: Thread CUDA é um vetor de instruções SIMD a serem executadas sequencialmente. Esse vetor é replicado e opera sobre dados diferentes.
- **Warp**: é um conjunto de 32 threads CUDA, de um bloco de threads, que são executadas simultaneamente.
- **Host**: CPU e sua hierarquia de memória (host memory)
- **Device**: GPU e suas memórias (device memory)
- **Kernel**: função que é replicada na GPU como múltiplas threads

Sobre memória, é importante saber que as memórias do host e do device são usualmente separadas. A gerência de memória do *device* é feito utilizando as funções `cudaMalloc(), cudaFree(), cudaMemcpy()`, que são análogas ao C.

Hierarquia de Threads

- **Threads**: possuem memória local à thread e ao bloco;
- **Blocos**: cada bloco pode agrupar até 1024 threads. Threads de um mesmo bloco são executadas em um mesmo Streaming Multiprocessor;
- **Grids**: agrupam blocos, permitem que diferentes blocos utilizem a memória global do *device*.

Lançamento de Threads em Paralelo na GPU

- `__global__`: chamado pelo host e executado no device
- `__device__`: chamado pelo device e executado no device
- `__host__`: chamado pelo host e executado no host (default)

Para lançar um kernel use `funcaoKernel<<<NumeroDeBlocos,NumeroDeThreads>>>(args)`

Memória nas GPUs

Memória global

Memória DRAM acessada por quaisquer threads de qualquer SM. Possui o maior tamanho na GPU mas tem acesso mais lento. São armazenadas na memória global variáveis declarados com `__device__`, como argumento das funções de kernel e alocadas dinâmicamente com o uso do `cudaMalloc()`.

Memória local à thread

Memórias SRAM, privadas às threads. Variáveis declaradas dentro do kernel possuem memória local. **Exemplo**: `tam, i, j` estão na memória local. `matrizA, matrizB e matrizC` estão na global devido ao `cudaMalloc()`

```
#define TAM 100
__global__ void soma(int *matrizA, int *matrizB,int *matrizC, int tam)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;
  if (i < tam && j < tam)
  {
    matrizC[i*TAM+j]=matrizA[i*TAM+j]+matrizB[i*TAM+j];
  }
}
```

Memória compartilhada

Threads de um mesmo bloco compartilham essa memória. Possuem uma latência menor e uma largura de banda maior comparadas à memória global.

- **Alocação estática**: `__shared__`
- **Alocação dinâmica**: `extern __shared__`

A memória compartilhada não permite transferência de dados com o host. Nesse caso, a global deve ser utilizada.

Memória constante

Permite apenas leitura do kernel, com tempo de latência menor que a memória global. São visíveis para todas as threads da *Grid*. Utiliza o modificador `__constant__` no host.

Memória unificada

Permite a cópia implícita de memória entre host e device. Não reduz o tempo de execução de um programa, de forma que as transferências de memória continuam acontecendo, sob demanda, transparentemente.

- **Alocação estática**: variável global no host usando `__managed__`
- **Alocação dinâmica**: `cudaMallocManaged()` no host

Registradores

Localizados nos SPs e armazenam o estado das threads em execução no SM

Memória de textura

Memória DRAM que armazena coordenadas e outros dados referentes à textura das imagens e têm o apoio de caches de textura

Operações Atômicas

Operações atômicas protegem regiões críticas em CUDA. `atomicAdd()` é um exemplo de operação atômica em função `__device__`.

CUDA usa primitivas streams que determinam uma sequência de comandos na GPU. Múltiplos streams podem executar concorrentemente, maximizando o uso de recursos. Para tanto é necessário utilizar memórias não paginàveis (pinned memories ou page-locked) no host, o que habilita a execução concorrente das atividades de uma stream ou memória mapeada (mapped memory ou zero-copy memory), que permite memória não paginável do host ser mapeado na memória do device.

- `malloc()`: aloca memória paginável no host
- `cudaMalloc()`: aloca memória no device
- `cudaMallocManaged()`: aloca memória paginável com ptr para ambos: host e device. memória unificada ou gerenciada (managed memory) `__managed__`
- `cudaMallocHost()`: aloca memória não paginável no host (pinned memory)
- `cudaHostAlloc()`: aloca memória não paginável ou só no host ou em ambos memória mapeada (mapped memory)
- `cudaMemcpy()`:faz cópia da memória entre host e device de maneira síncrona
- `cudaMemcpyAsync()`:faz cópia assíncrona entre host e device e requer pinned memory

Exercícios

Memória CUDA

Analise o código abaixo e responda

- Qual o tipo de memoria esta sendo usado na linha 31? `cudaMallocHost((void**)&vetorA,tam*(sizeof(int)))`. R.: *Memória do host.*
- Que memoria é essa? R.: *Memórias não paginàveis.*
- Qual o tamanho do bloco? *O bloco tem tamanho* `tam*sizeof(int)` *bytes* $= 16 * 4$ *bytes.*

```
29  int blocksPerGrid = (threadsPerGrid + threadsPerBlock - 1) / threadsPerBlock;
30
31  cudaMallocHost((void **)&vetorA, tam * (sizeof(int)));
```

Soma de Matrizes Transpostas em CUDA

Implemente um código em c usando CUDA que some a transposta da Matriz A com a transposta da Matriz B e armazene na Matriz C. A geração dos números das Matrizes A e B tem que ser pseudo-aleatória. Deve-se usar um único Kernel Cuda para fazer esse processo. As matrizes A , B, C devem ser alocadas no Host Device e todas as operações tem que ser feitas em memória compartilhada.

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

#define N 16 // Size of the matrix (N x N)

// CUDA kernel to compute C = A^T + B^T using shared memory
__global__ void transposeAndAdd(float *A, float *B, float *C, int n) {
    __shared__ float tileA[N][N];
    __shared__ float tileB[N][N];

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x < n && y < n) {
        // Load elements into shared memory
        tileA[threadIdx.y][threadIdx.x] = A[y * n + x];
        tileB[threadIdx.y][threadIdx.x] = B[y * n + x];

        // Synchronize to ensure all threads have loaded their data
        __syncthreads();

        // Perform transpose and addition
        C[x * n + y] = tileA[threadIdx.x][threadIdx.y] + tileB[threadIdx.x][threadIdx.y];
    }
}

// Function to initialize a matrix with pseudo-random values
void initializeMatrix(float *matrix, int n) {
    for (int i = 0; i < n * n; i++) {
        matrix[i] = (float)(rand() % 100) / 10.0f; // Random values between 0 and 10
    }
}

int main() {
    int size = N * N * sizeof(float);

    // Allocate memory for matrices on the host
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize matrices A and B with pseudo-random values
    initializeMatrix(h_A, N);
    initializeMatrix(h_B, N);

    // Allocate memory for matrices on the device
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    // Copy matrices A and B to the device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Define block and grid dimensions
    dim3 blockDim(N, N);
    dim3 gridDim(1, 1);

    // Launch the kernel
    transposeAndAdd<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);

    // Copy the result matrix C back to the host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Print the matrices
    printMatrix("A", h_A, N);
    printMatrix("B", h_B, N);
    printMatrix("C", h_C, N);

    // Free memory on the device and host
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```