Utilização de RMA com MPI 2.0 para acesso de memória remota

Descrição da solução paralela

Para essa solução paralela foram usados os conceitos de window do MPI e a função MPI_Get, de forma a fazer a comunicação unilateral entre os processos, podendo cada um deles acessar a memória dos outros conforme a necessidade.

Cada vez que se calcula um novo ind[i] checa-se a qual processo esse valor pertence, caso seja local, basta acessar o array A no displacement correto, caso seja remoto, basta usar a função MPI_Get para acessar o processo correto no displacement calculado.

Após calcular cada soma local, utiliza-se a função MPI_Reduce para colocar a soma total no processo 0.

Parte modificada de SomaPar.c

```
// calcula a soma
int i;
long localSum=0;

for (i = 0; i < localSizeInd; i++) {
  int ind = VetorInd[i];
  int indRank = ind/localSizeA;
  int indDisp = ind % localSizeA;
  int A = 0;

  MPI_Win_fence(0, win);

  if (indRank == rank) { // A local contém o indice
    A = VetorA[indDisp];
  } else { // O indice está em outro local
    MPI_Get(
        &A, 1, MPI_INT,
        indRank,
        indDisp, 1, MPI_INT,
        win);
  }

  MPI_Win_fence(0, win);
  localSum += A;
}
```

```
// Fazer um reduce no processo de rank 0

long totalSum=0;
MPI_Reduce(&localSum, &totalSum, 1, MPI_LONG,
            MPI_SUM, 0, MPI_COMM_WORLD);
```

Arquivos de saída

- SomaPar_1MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 1 ranks MPI
  ```

- SomaPar_2MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 2 ranks MPI
  ```

- SomaPar_4MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 4 ranks MPI
  ```

- SomaPar_8MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 8 ranks MPI
  ```

- SomaPar_16MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 16 ranks MPI
  ```

- SomaPar_32MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 32 ranks MPI
  ```

- SomaPar_64MPI.out

  ```
  **RESULTADO CERTO** (48820191429) com 64 ranks MPI
  ```

Avaliação

Eu achei esse um dos melhores exercicios em nível de dificuldade, especialmente considerando a dificuldade que há em debugar os programas feitos com MPI. Bastava entender os conceitos de MPI_Get e das janelas e utiliza-los de forma correta para conseguir resolver o exercicio, e, portanto, não há muita margem para pequenos erros bobos em diferentes etapas. Ao mesmo tempo não era um exercicío muito simples de apenas modificar uma ou duas linhas de código, tendo algum desafio envolvido na resolução.

Programa SomaPar.c completo

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

```c
#include <math.h>
#include <mpi.h>


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

  // para desenvolvimento

//#define VA "bin/VectorATeste.bin"
//#define VI "bin/VectorIndTeste.bin"
//#define RES -1544794685
  //const int m=32;
  //const int n=16;

  // para entregar

  #define VA "bin/VectorA.bin"
  #define VI "bin/VectorInd.bin"
  #define RES 48820191429
    const int m=16777216;
    const int n=2048;

  // inicializa MPI

  int size, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // verifica se numero de processos MPI eh potencia de 2

  int pow2=size;
  while ((pow2 >> 1) != 0) {
    if ((pow2 & 01) != 0) {
      if (rank == 0) printf("MPI size (%d) nao eh potencia de 2\n",size);
      MPI_Finalize();
      exit(0);
    }
    pow2 = pow2 >> 1;
  }

  // verifica se ha mais dados que processos MPI

  if (size > m || size > n) {
    if (rank == 0) printf("MPI size (%d) eh maior que m (%d) ou n (%d)\n",
               size, m, n);
```

```
    MPI_Finalize();
    exit(0);
  }

  // cria vetores A e Ind, distribuidos por blocos entre os processos MPI

  const int localSizeA=m/size;
  const int myFirstA=rank*localSizeA;
  const int myLastA=(rank+1)*localSizeA-1;
  int *VetorA;
  VetorA=(int*)malloc(localSizeA*sizeof(int));

  const int localSizeInd=n/size;
  const int myFirstInd=rank*localSizeInd;
  const int myLastInd=(rank+1)*localSizeInd-1;
  int *VetorInd;
  VetorInd=(int*)malloc(localSizeInd*sizeof(int));

  //printf("rank %d armazena A global [%d:%d] em e A local [0:%d] e Ind global [%d:%d] em I
  //     rank, myFirstA, myLastA, localSizeA-1, myFirstInd, myLastInd, localSizeInd-1);

  // popula vetores A e Ind, lendo seu trecho dos respectivos arquivos

#define TAGA 28
#define TAGIND 29
  if (rank==0){

    // rank 0 le seu trecho

    FILE *fp1=NULL;
    fp1 = fopen(VA, "r");
    if (fp1 == NULL) {
      printf("Erro ao abrir o arquivo %s\n", VA);
      exit(0);
    }
    size_t sizeA;
    sizeA=fread(VetorA,sizeof(int),localSizeA,fp1);
    if (sizeA != (size_t) localSizeA) {
      printf("Erro ao ler o arquivo %s\n", VA);
      exit(0);
    }

    FILE *fp2=NULL;
    fp2 = fopen(VI, "r");
    if (fp2 == NULL) {
      printf("Erro ao abrir o arquivo %s\n", VI);
```

```
    exit(0);
  }
  size_t sizeInd;
  sizeInd=fread(VetorInd,sizeof(int),localSizeInd,fp2);
  if (sizeInd != (size_t) localSizeInd) {
    printf("Erro ao ler o arquivo %s\n", VI);
    exit(0);
  }

  // rank 0 le trechos dos demais processos e envia

  int *outroA;
  outroA=(int*)malloc(localSizeA*sizeof(int));
  int *outroInd;
  outroInd=(int*)malloc(localSizeInd*sizeof(int));
  int proc;
  for (proc=1; proc<size; proc++) {
    sizeA=fread(outroA,sizeof(int),localSizeA,fp1);
    if (sizeA != (size_t) localSizeA) {
  printf("Erro ao ler o trecho do rank %d no arquivo %s\n", proc, VA);
  exit(0);
    }
    MPI_Send(outroA, localSizeA, MPI_INT,
         proc, TAGA, MPI_COMM_WORLD);
    sizeInd=fread(outroInd,sizeof(int),localSizeInd,fp2);
    if (sizeInd != (size_t) localSizeInd) {
  printf("Erro ao ler o trecho do rank %d no arquivo %s\n", proc, VI);
  exit(0);
    }
    MPI_Send(outroInd, localSizeInd, MPI_INT,
         proc, TAGIND, MPI_COMM_WORLD);
  }
  free(outroA);
  free(outroInd);
  fclose(fp1);
  fclose(fp2);
} else{

  // demais processos recebem seu trecho

  MPI_Status status;
  MPI_Recv(VetorA, localSizeA, MPI_INT,
       0, TAGA, MPI_COMM_WORLD, &status);
  MPI_Recv(VetorInd, localSizeInd, MPI_INT,
       0, TAGIND, MPI_COMM_WORLD, &status);
}
```

```c
// Criar janelas de comunicacao entre os processos
MPI_Win win;
MPI_Win_create(
    VetorA, localSizeA * sizeof(int), sizeof(int),
    MPI_INFO_NULL, MPI_COMM_WORLD, &win);

// calcula a soma
int i;
long localSum=0;

for (i = 0; i < localSizeInd; i++) {
  int ind = VetorInd[i];
  int indRank = ind/localSizeA;
  int indDisp = ind % localSizeA;
  int A = 0;

  MPI_Win_fence(0, win);

  if (indRank == rank) { // A local contém o indice
    A = VetorA[indDisp];
  } else { // O indice está em outro local
    MPI_Get(
        &A, 1, MPI_INT,
        indRank,
        indDisp, 1, MPI_INT,
        win);
  }

  MPI_Win_fence(0, win);
  localSum += A;
}

// Fazer um reduce no processo de rank 0

long totalSum=0;
MPI_Reduce(&localSum, &totalSum, 1, MPI_LONG,
            MPI_SUM, 0, MPI_COMM_WORLD);

// verifica correcao

if (rank == 0) {
  if (totalSum == RES)
    printf("**RESULTADO CERTO** (%ld) com %d ranks MPI\n", totalSum, size);
  else
    printf("**RESULTADO ERRADO** (%ld, deveria ser %ld) com %d ranks MPI\n", totalSum, RES
```

```
  }


  free(VetorA);
  free(VetorInd);
  MPI_Win_free(&win);
  MPI_Finalize();
  exit(0);
}
```