MPI 集合通信函数 MPI_Reduce()，MPI_Allreduce()，MPI_Bcast()，MPI_Scatter()，MPI_Gather()，MPI_Allgather()，MPI_Scan()，MPI_Reduce_Scatter()

▶ 八个常用的集合通信函数

▶ 规约函数 MPI_Reduce()，将通信子内各进程的同一个变量参与规约计算，并向指定的进程输出计算结果

- 函数原型

```
1 MPI_METHOD MPI_Reduce(
2     _In_range_(!= , recvbuf) _In_opt_ const void* sendbuf,  // 指向输入数据的指针
3     _When_(root != MPI_PROC_NULL, _Out_opt_) void* recvbuf, // 指向输出数据的指针，即计算结果存放的地方
4     _In_range_(>= , 0) int count,                           // 数据尺寸，可以进行多个标量或多个向量的规约
5     _In_ MPI_Datatype datatype,                             // 数据类型
6     _In_ MPI_Op op,                                         // 规约操作类型
7     _mpi_coll_rank_(root) int root,                         // 目标进程号，存放计算结果的进程
8     _In_ MPI_Comm comm                                      // 通信子
9 );
```

- 使用范例

```
1 {
2     int size, rank, data, dataCollect;
3     MPI_Init(NULL, NULL);
4     MPI_Comm_size(MPI_COMM_WORLD, &size);
5     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7     data = rank;// 参与计算的数据
8     MPI_Reduce((void *)&data, (void *)&dataCollect, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);// 所有的进程都要调用，而不是只在目标进程中调用
9
10    MPI_Finalize();
11 }
```

- 操作类型，定义于 mpi.h

```
1 #define MPI_OP_NULL ((MPI_Op)0x18000000)
2
3 #define MPI_MAX     ((MPI_Op)0x58000001)
4 #define MPI_MIN     ((MPI_Op)0x58000002)
5 #define MPI_SUM     ((MPI_Op)0x58000003)
6 #define MPI_PROD    ((MPI_Op)0x58000004)
7 #define MPI_LAND    ((MPI_Op)0x58000005)// 逻辑与
```

```
#define MPI_BAND    ((MPI_Op)0x58000006)// 按位与
#define MPI_LOR     ((MPI_Op)0x58000007)
#define MPI_BOR     ((MPI_Op)0x58000008)
#define MPI_LXOR    ((MPI_Op)0x58000009)
#define MPI_BXOR    ((MPI_Op)0x5800000a)
#define MPI_MINLOC  ((MPI_Op)0x5800000b)// 求最小值所在位置
#define MPI_MAXLOC  ((MPI_Op)0x5800000c)// 求最大值所在位置
#define MPI_REPLACE ((MPI_Op)0x5800000d)
```

▶ 规约并广播函数 MPI_Allreduce()，在计算规约的基础上，将计算结果分发到每一个进程中，相比于 MPI_Reduce()，只是少了一个 root 参数。除了简单的先规约再广播的方法，书中介绍了蝶形结构全局求和的方法。

- 函数原型

```
_Pre_satisfies_(recvbuf != MPI_IN_PLACE) MPI_METHOD MPI_Allreduce(
    _In_range_(!= , recvbuf) _In_opt_ const void* sendbuf,
    _Out_opt_ void* recvbuf,
    _In_range_(>= , 0) int count,
    _In_ MPI_Datatype datatype,
    _In_ MPI_Op op,
    _In_ MPI_Comm comm
);
```

- 使用范例

```
{
    int size, rank, data, dataCollect;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    data = rank;
    MPI_Reduce((void *)&data, (void *)&dataCollect, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);// 所有的进程都要调用

    MPI_Finalize();
}
```

▶ 广播函数 MPI_Bcast()，将某个进程的某个变量的值广播到该通信子中所有进程的同名变量中

- 函数原型

```
MPI_METHOD MPI_Bcast(
    _Pre_opt_valid_ void* buffer,    // 指向输入 / 输出数据的指针
    _In_range_(>= , 0) int count,    // 数据尺寸
```

```
4     _In_ MPI_Datatype datatype,      // 数据类型
5     _mpi_coll_rank_(root) int root, // 广播源进程号
6     _In_ MPI_Comm comm               // 通信子
7 );
```

- 使用范例

```
1 {
2     int size, rank, data;
3     MPI_Init(NULL, NULL);
4     MPI_Comm_size(MPI_COMM_WORLD, &size);
5     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7     data = rank;
8     MPI_Bcast((void *)&data, 1, MPI_INT, 0, MPI_COMM_WORLD);// 所有的进程都要调用，调用后所有data 均被广播源进程的值覆盖
9
10    MPI_Finalize();
11 }
```

▶ 散射函数 MPI_Scatter()，将向量数据分段发送到各进程中

- 函数原型和宏定义

```
1 _Pre_satisfies_(sendbuf != MPI_IN_PLACE) MPI_METHOD MPI_Scatter(
2     _In_range_(!= , recvbuf) _In_opt_ const void* sendbuf,  // 指向需要分发的数据的指针
3     _In_range_(>= , 0) int sendcount,                       // 分发到每一个进程的数据量，注意不是分发的数据总量
4     _In_ MPI_Datatype sendtype,                             // 分发数据类型
5     _When_(root != MPI_PROC_NULL, _Out_opt_) void* recvbuf, // 指向接收的数据的指针
6     _In_range_(>= , 0) int recvcount,                       // 接受数据量，不小于上面分发到每一个进程的数据量
7     _In_ MPI_Datatype recvtype,                             // 接收数据类型
8     _mpi_coll_rank_(root) int root,                         // 分发数据源进程号
9     _In_ MPI_Comm comm                                      // 通信子
10 );
11
12 // 宏定义，mpi.h
13 #define MPI_IN_PLACE ((void*)(MPI_Aint)-1 // MPI_Aint 为 __int64 类型，表示地址
```

▶ 聚集函数 MPI_Gather()，将各进程中的向量数据分段聚集到一个进程的大向量中

- 函数原型

```
1 _Pre_satisfies_(recvbuf != MPI_IN_PLACE) MPI_METHOD MPI_Gather(
2     _In_opt_ _When_(sendtype == recvtype, _In_range_(!= , recvbuf)) const void*
```

```
sendbuf,// 指向需要聚集的数据的指针
    _In_range_(>= , 0) int sendcount,
// 每个进程中进行聚集的数据量，不是聚集的数据总量
    _In_ MPI_Datatype sendtype,
// 发送数据类型
    _When_(root != MPI_PROC_NULL, _Out_opt_) void* recvbuf,
// 指向接收数据的指针
    _In_range_(>= , 0) int recvcount,
// 从每个进程接收的接收数据量，不是聚集的数据总量
    _In_ MPI_Datatype recvtype,
// 接收数据类型
    _mpi_coll_rank_(root) int root,
// 聚集数据汇进程号
    _In_ MPI_Comm comm
// 通信子
);
```

- 函数 MPI_Scatter() 和 MPI_Gather() 的范例

```
{
    const int dataSize = 8 * 8;
    const int localSize = 8;
    int globalData[dataSize], localData[localSize], globalSum, i, comSize, comRank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &comRank);

    if (comRank == 0)                                          // 初始化
        for (i = 0; i < dataSize; globalData[i] = i, i++);
    for (i = 0; i < localSize; localData[i++] = 0);

    MPI_Scatter((void *)&globalData, localSize, MPI_INT, (void *)&localData, localSize,
MPI_INT, 0, MPI_COMM_WORLD);     // 分发数据
    for (i = 0; i < localSize; localData[i++]++);
    MPI_Barrier(MPI_COMM_WORLD);
// 进程同步
    MPI_Gather((void *)&localData, localSize, MPI_INT, (void *)&globalData, localSize,
MPI_INT, 0, MPI_COMM_WORLD);      // 聚集数据
    for (i = globalSum = 0; i < dataSize; globalSum += globalData[i++]);

    if (comRank == 0)
        printf("\nSize = %d, Rank = %d, result = %d\n", comSize, comRank, globalSum);
    MPI_Finalize();

    return 0;// 输出结果: Size = 8, Rank = 0, result = 2080, 表示 0 + 1 + 2 + …… + 63
}
```

▶ 全局聚集函数 MPI_Allgather()，将各进程的向量数据聚集为一个大向量，并分发到每个进程中，相当于各进程同步该大向量的各部分分量。相比于 MPI_Gather()，只是少了一个 root 参数。

- 函数原型

```
_Pre_satisfies_(recvbuf != MPI_IN_PLACE) MPI_METHOD MPI_Allgather(
    _In_opt_ _When_(sendtype == recvtype, _In_range_(!= , recvbuf)) const void* sendbuf,
    _In_range_(>= , 0) int sendcount,
    _In_ MPI_Datatype sendtype,
    _Out_opt_ void* recvbuf,
    _In_range_(>= , 0) int recvcount,
    _In_ MPI_Datatype recvtype,
    _In_ MPI_Comm comm
);
```

- 函数 MPI_Scatter() 和 MPI_Allgather() 的范例，相当于从上面的范例中修改了一部分

```
{
    const int dataSize = 8 * 8;
    const int localSize = 8;
    int globalData[dataSize], localData[localSize], globalSum, i, comSize, comRank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &comRank);

    for (i = 0; i < dataSize; globalData[i] = i, i++);// 改动
    for (i = 0; i < localSize; localData[i++] = 0);

    MPI_Scatter((void *)&globalData, localSize, MPI_INT, (void *)&localData, localSize, MPI_INT, 0, MPI_COMM_WORLD);    // 分发数据
    for (i = 0; i < localSize; localData[i++]++);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Allgather((void *)&localData, localSize, MPI_INT, (void *)&globalData, localSize, MPI_INT, MPI_COMM_WORLD);     // 聚集数据，改动
    for (i = globalSum = 0; i < dataSize; globalSum += globalData[i++]);

    printf("\nSize = %d, rank = %d, result = %d\n", comSize, comRank, globalSum);// 改动
    MPI_Finalize();

    return 0;// 输出结果，八个进程乱序输出 2080
}
```

▶ 前缀和函数 MPI_Scan()，将通信子内各进程的同一个变量参与前缀规约计算，并将得到的结果发送回每个进程，使用与函数 MPI_Reduce() 相同的操作类型

- 函数原型

```
_Pre_satisfies_(recvbuf != MPI_IN_PLACE) MPI_METHOD MPI_Scan(
    _In_opt_ _In_range_(!= , recvbuf) const void* sendbuf,  // 指向参与规约数据的指针
    _Out_opt_ void* recvbuf,                                // 指向接收规约结果的指针
    _In_range_(>= , 0) int count,                           // 每个进程中参与规约的数据量
    _In_ MPI_Datatype datatype,                             // 数据类型
```

```
6     _In_ MPI_Op op,                                    // 规约操作类型
7     _In_ MPI_Comm comm                                 // 通信子
8 );
```

- 范例代码

```
 1 int main(int argc, char **argv)
 2 {
 3     const int nProcess = 8, localSize = 8, globalSize = localSize * nProcess;
 4     int globalData[globalSize], localData[localSize], sumData[localSize];
 5     int comRank, comSize, i;
 6
 7     MPI_Init(&argc, &argv);
 8     MPI_Comm_rank(MPI_COMM_WORLD, &comRank);
 9     MPI_Comm_size(MPI_COMM_WORLD, &comSize);
10
11     if (comRank == 0)
12     for (i = 0; i < globalSize; globalData[i] = i, i++);
13
14     MPI_Scatter(globalData, localSize, MPI_INT, localData, localSize, MPI_INT, 0,
MPI_COMM_WORLD);
15
16     for (i = 0; i < localSize; i++)
17         printf("%2d, ", localData[i]);
18
19     MPI_Scan(localData, sumData, localSize, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
20
21     for (i = 0; i < localSize; i++)
22         printf("%2d, ", sumData[i]);
23
24     MPI_Finalize();
25     return 0;
26 }
```

- 输出结果，分别展示了 localSize 取 1 和 8 的结果，每个进程的输出中，前一半（分别为 1 个和 8 个元素）为个进程的原始数据，后一半为进行完前缀求和后的结果。注意到 localSize 取 8 时，程序将各进程保存向量的每一个元素分别进行前缀和，但同一进程中各元素之间不相互影响。

```
D:\Code\MPI\MPIProjectTemp\x64\Debug>mpiexec -n 8 -l MPIProjectTemp.exe
[1]  1,   1,
[6]  6,  21,
[5]  5,  15,
[4]  4,  10,
[3]  3,   6,
[0]  0,   0,
[2]  2,   3,
[7]  7,  28,
D:\Code\MPI\MPIProjectTemp\x64\Debug>mpiexec -n 8 -l MPIProjectTemp.exe
[4] 32,  33,  34,  35,  36,  37,  38,  39,  80,  85,  90,  95, 100, 105, 110, 115,
[1]  8,   9,  10,  11,  12,  13,  14,  15,   8,  10,  12,  14,  16,  18,  20,  22,
[0]  0,   1,   2,   3,   4,   5,   6,   7,   0,   1,   2,   3,   4,   5,   6,   7,
```

```
[7] 56,  57,  58,  59,  60,  61,  62,  63, 224, 232, 240, 248, 256, 264, 272, 280,
[3] 24,  25,  26,  27,  28,  29,  30,  31,  48,  52,  56,  60,  64,  68,  72,  76,
[6] 48,  49,  50,  51,  52,  53,  54,  55, 168, 175, 182, 189, 196, 203, 210, 217,
[5] 40,  41,  42,  43,  44,  45,  46,  47, 120, 126, 132, 138, 144, 150, 156, 162,
[2] 16,  17,  18,  19,  20,  21,  22,  23,  24,  27,  30,  33,  36,  39,  42,  45,
```

▶ 规约分发函数 MPI_Reduce_Scatter()，将数据进行规约计算，结果分段分发到各进程中

- 函数原型

```
1 _Pre_satisfies_(recvbuf != MPI_IN_PLACE) MPI_METHOD MPI_Reduce_scatter(
2     _In_opt_ _In_range_(!= , recvbuf) const void* sendbuf,  // 指向输入数据的指针
3     _Out_opt_ void* recvbuf,                                // 指向接收数据的指针
4     _In_ const int recvcounts[],                            // 各进程接收规约结果的元素个数
5     _In_ MPI_Datatype datatype,                             // 数据类型
6     _In_ MPI_Op op,                                         // 规约操作类型
7     _In_ MPI_Comm comm                                      // 通信子
8 );
```

- 使用范例

```
 1 {
 2     const int nProcess = 8, localSize = 8, globalSize = nProcess * localSize, countValue
= 1;
 3     int globalData[globalSize], localData[localSize], count[localSize],
localSum[countValue], i, comSize, comRank;
 4
 5     MPI_Init(&argc, &argv);
 6     MPI_Comm_size(MPI_COMM_WORLD, &comSize);
 7     MPI_Comm_rank(MPI_COMM_WORLD, &comRank);
 8
 9     if (comRank == 0)

10         for (i = 0; i < globalSize; globalData[i] = i, i++);
11
12     MPI_Scatter(globalData, localSize, MPI_INT, localData, localSize, MPI_INT, 0,
MPI_COMM_WORLD);
13
14     for (i = 0; i < localSize; count[i++] = 1);
15
16     for (i = 0; i < localSize; i++)
17         printf("%3d, ", localData[i]);
18
19     MPI_Reduce_scatter(localData, localSum, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
20
21     for (i = 0; i < countValue; i++)
22         printf("%3d, ", localSum[i]);
23
24     MPI_Finalize();
25     return 0;
26 }
```

● 输出结果，这里取定 localSize 为 8，输出结果的前 8 个元素为分发到各进程中参与规约计算的原始数据，后面元素为规约计算结果。程序将各进程保存向量的每一个元素分别进行前缀和，但同一进程中各元素之间不相互影响，通过修改 countValue（即参数 count 各元素的值），可以将规约计算的结果分发到各进程中

■ countValue == 1（count == { 1, 1, 1, 1, 1, 1, 1, 1 }）情况，每个进程分得一个结果（注意与上面的函数 MPI_Scan() 作对比）

■ countValue == 2（count == { 2, 2, 2, 2, 2, 2, 2, 2 }）情况，前 4 个进程每个进程分得 2 个结果，后 4 的进程访问越界，得到无意义的值

■ count == { 2, 0, 2, 0, 2, 0, 2, 0 } 情况，偶数号进程每个进程分得 2 个结果，奇数号进程分得 0 个结果，表现为无意义的值

■ 思考，这列每个 localData 长度为 8，所以规约计算的结果为一个长度为 8 的向量，可以在不同进程中进行分发（注意数据尺寸大小 localSize 与运行程序的进程数 nProcess 没有任何关系，只是在范例中恰好相等），而函数 MPI_Scan() 则相当于在此基础上保留了所有中间结果（部分前缀结果），所以其输出为一个长为 localSize，宽度为 nProcess 的矩阵，并且自动按照进程号均分。

```
D:\Code\MPI\MPIProjectTemp\x64\Debug>mpiexec -n 8 -l MPIProjectTemp.exe // countValue = 1
[6] 48,  49,  50,  51,  52,  53,  54,  55, 272,
[0]  0,   1,   2,   3,   4,   5,   6,   7, 224,
[2] 16,  17,  18,  19,  20,  21,  22,  23, 240,
[4] 32,  33,  34,  35,  36,  37,  38,  39, 256,
[3] 24,  25,  26,  27,  28,  29,  30,  31, 248,
[1]  8,   9,  10,  11,  12,  13,  14,  15, 232,
[5] 40,  41,  42,  43,  44,  45,  46,  47, 264,
[7] 56,  57,  58,  59,  60,  61,  62,  63, 280,

D:\Code\MPI\MPIProjectTemp\x64\Debug>mpiexec -n 8 -l MPIProjectTemp.exe // countValue = 2
[0]  0,   1,   2,   3,   4,   5,   6,   7, 224, 232,
[6] 48,  49,  50,  51,  52,  53,  54,  55, 1717986912, 1717986912,
[1]  8,   9,  10,  11,  12,  13,  14,  15, 240, 248,
[3] 24,  25,  26,  27,  28,  29,  30,  31, 272, 280,
[4] 32,  33,  34,  35,  36,  37,  38,  39, 1717986912, 1717986912,
[5] 40,  41,  42,  43,  44,  45,  46,  47, 1717986912, 1717986912,
[2] 16,  17,  18,  19,  20,  21,  22,  23, 256, 264,
[7] 56,  57,  58,  59,  60,  61,  62,  63, 1717986912, 1717986912,

D:\Code\MPI\MPIProjectTemp\x64\Debug>mpiexec -n 8 -l MPIProjectTemp.exe // countValue = 2,
count[i] = (i + 1) % 2 * 2
[4] 32,  33,  34,  35,  36,  37,  38,  39, 256, 264,
[2] 16,  17,  18,  19,  20,  21,  22,  23, 240, 248,
[3] 24,  25,  26,  27,  28,  29,  30,  31, -858993460, -858993460,
[1]  8,   9,  10,  11,  12,  13,  14,  15, -858993460, -858993460,
[7] 56,  57,  58,  59,  60,  61,  62,  63, -858993460, -858993460,
[0]  0,   1,   2,   3,   4,   5,   6,   7, 224, 232,
[5] 40,  41,  42,  43,  44,  45,  46,  47, -858993460, -858993460,
[6] 48,  49,  50,  51,  52,  53,  54,  55, 272, 280,
```

分类: MPI

标签：《并行程序设计导论》，MPI

曩曩曩好
关注 - 2
粉丝 - 5
+加关注

0 0

« 上一篇：778. Swim in Rising Water
» 下一篇：MPI 派生数据类型 MPI_Type_create_struct(), MPI_Type_contiguous(), MPI_Type_vector(), MPI_Type_create_hvector(), MPI_Type_indexed()

posted on 2018-02-10 13:48 曩曩曩好 阅读(2380) 评论(0) 编辑 收藏