MPI 和分布式计算

Crazyfish*
CC98

MPI，全称为 Message Passing Interface，是一种消息传递编程模型，用于设计并行计算应用。MPI 为并行计算中的任务分配和进程间通信提供了一套标准和一致的接口，这使得开发者可以在多处理器或多计算机的环境中进行高效的并行编程。

- **并行模型**：MPI 支持多种并行模型，包括点对点通信、集合操作和并行 I/O 等。
- **进程通信**：MPI 提供了一套丰富的通信函数，如发送和接收消息、广播、归约、扫描和分布等。
- **语言独立性**：MPI 标准定义了一套接口，这些接口可以用 C，C++，Fortran 等多种语言实现。
- **可移植性**：MPI 程序可以在各种硬件和操作系统上运行，包括单处理器系统，多处理器系统，集群和超级计算机。
- **性能**：MPI 设计的目标之一就是高性能，它提供了一种机制，允许程序员控制数据的分布和通信，从而优化程序的性能。
- **可扩展性**：MPI 支持大规模并行，可以在数千甚至数十万个进程上运行。

MPI 作为一个标准是由一个跨国的研究团队在 1990 年代初建立的，该团队由来自工业界和学术界的专家组成。这个团队被称为 MPI Forum。

MPI Forum 的目标是创建一个可移植、高效且灵活的并行编程接口标准，以满足当时日益增长的并行计算需求。他们的工作成果就是我们现在所知的 MPI 标准。

*wang_heyu@msn.com

MPI 标准的第一个版本（MPI-1）于 1994 年发布，包含了基本的点对点通信和集合操作。后续的版本（如 MPI-2 和 MPI-3）进一步扩展了标准，增加了并行 I/O、一致性内存访问（RMA）、动态进程管理和其他特性。

MPI Forum 依然活跃，并继续开发和维护 MPI 标准。他们定期举行会议，讨论新的特性和改进，并处理来自社区的问题和建议。

MPI Forum 的官方网站是其主要的信息来源，可以找到最新的 MPI 标准文档、会议记录、工作组信息以及其他相关资源。以下是一些相关链接：

- **MPI Forum 主页**：`https://www.mpi-forum.org/`

  这是 MPI Forum 的主页，提供了 MPI 标准的最新信息，包括最新版本的标准文档、即将举行的会议以及其他 MPI 相关的新闻。

- **MPI 标准文档**：`https://www.mpi-forum.org/docs/`

  这个页面提供了 MPI 标准的所有版本的文档，包括最新的 MPI-4.0 版本。

- **MPI Forum 会议**：`https://www.mpi-forum.org/meetings/`

  这个页面列出了 MPI Forum 的会议日程，包括即将举行的会议和过去的会议的记录。

通过这些资源，你可以获取 MPI 标准的最新信息，了解 MPI Forum 的工作，以及参与到 MPI 社区的活动中。

MPI 的主要实现有以下几种：

- **MPICH**：MPICH 是最早的 MPI 实现之一，由阿贡国家实验室的数学和计算机科学部门开发。MPICH 被设计为高性能和可移植，并且被广泛用作其他 MPI 实现的基础。MPICH 提供了对 MPI-3.1 标准的完全支持，并且在各种硬件和操作系统上进行了测试。MPICH 的网站是：`https://www.mpich.org/`。

- **Open MPI**：Open MPI 是另一种广泛使用的 MPI 实现，由一组合作的研究、学术和商业团队开发。Open MPI 旨在实现 MPI 标准的最新特性，同时提供优秀的性能和可扩展性。Open MPI 支持各种操作系统和硬件平台，包括共享内存系统、分布式内存系统和异构系统。Open MPI 的网站是：`https://www.open-mpi.org/`。以上两种实现都是开源的，可以免费下载和使用。

- **Intel MPI**：Intel MPI 是英特尔公司开发的商业 MPI 实现。它专为英特尔硬件优化，包括英特尔的多核和多处理器系统。Intel MPI 提供了对 MPI-3.1 标准的支持，并提供了一些扩展功能，如高级错误检查和调试支持。Intel MPI 的网站是：`https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html`。这是一种商业软件，需要购买许可证才能使用。

MPICH 和 Open MPI 两种实现在功能和性能上都非常接近，因此在选择时可以根据具体的需求和偏好来决定。Intel MPI 则专注于英特尔硬件的优化，适合在英特尔平台上进行高性能计算。我们以下的讨论以 Open MPI（这纯粹是一种个人喜好）为例，但是大部分的内容同样适用于 MPICH 和 Intel MPI。

为了方便起见，我们略过 MPI 的安装和部署，大家可以直接使用 apt 源安装 Open MPI，或者从官网下载源码编译安装。但注意真正的网络环境下，MPI 的部署可能会比较复杂，需要一些专业知识。包括硬件，操作系统，网络，MPI 实现等等都会影响 MPI 的性能和稳定性。

MPI 正确安装之后，在终端输入 `mpirun --version` 可以查看当前安装的 MPI 版本。

1 MPI 的基本框架

在 MPI 的环境中，`mpirun`和`mpicc`是两个非常重要的命令：

- **mpirun**：`mpirun`是一个用于启动 MPI 程序的命令行工具。它启动指定数量的进程，并将它们分布到可用的处理器上。`mpirun`还处理所有必要的初始化和关闭过程，并提供了一些选项来控制进程的布局和环境。例如，`mpirun -np 4 ./myprog`命令会启动四个`myprog`的进程。
- **mpicc**：`mpicc`是一个用于编译 MPI 程序的编译器包装器。它在调用底层的 C 编译器（如 gcc）时，自动添加了需要的头文件和库。这使得开发者可以像编译普通的 C 程序一样来编译 MPI 程序。例如，`mpicc -o myprog myprog.c`命令会编译源文件`myprog.c`并生成可执行文件`myprog`。

请注意，`mpirun`和`mpicc`的具体名称和行为可能会根据你的 MPI 实现

有所不同。例如，在某些 MPI 实现中，mpirun可能被称为mpiexec，mpicc可能被称为mpicc或mpiCC。

即便是用 gcc 编译的单进程程序，也可以用 mpirun 来运行，比如：

```
$ mpirun -np 4 ./hello
```

你会看到四个进程同时输出 Hello, world!。

mpicc和gcc都是编译器，它们的主要区别在于它们的应用领域和编译的程序类型。

- gcc是 GNU 编译器套件的一部分，是一个通用的 C 编译器，用于编译 C 语言程序。gcc可以在各种硬件平台和操作系统上编译和优化 C 语言源代码。
- mpicc是 OpenMPI 提供的一个包装器编译器，它是为了编译使用 MPI（消息传递接口）标准的并行程序。mpicc实际上在背后调用了一个 C 编译器（如gcc），并自动添加了需要的 MPI 头文件和链接库。这意味着，当你使用mpicc编译一个 MPI 程序时，你实际上是在使用gcc（或其他 C 编译器）加上一些额外的 MPI 特定的参数。

所以，简单来说，mpicc和gcc的关系是：mpicc是一个特化的编译器，它在gcc的基础上添加了对 MPI 并行编程的支持。

因此，如果你运行：

```
$ mpicc -v
```

你实际上会看到 gcc 的版本信息。

下面是我们的第一个 MPI 程序：

```
// 引入MPI库
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // 初始化MPI环境
    MPI_Init(NULL, NULL);

    // 定义一个变量来保存通信域的大小（即总的进程数）
```

```
    int world_size;
    // 获取通信域的大小
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // 定义一个变量来保存当前进程的 rank
    int world_rank;
    // 获取当前进程的 rank
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // 打印出 "Hello world, from rank" 和进程的 rank
    printf("Hello world, from rank %d\n", world_rank);

    // 结束MPI环境
    MPI_Finalize();
    return 0;
}
```

很显然，通讯域的大小是 `mpirun` 中的 `-np` 参数，而当前进程的 rank 就是 `mpirun` 中的进程编号。这是一个局部的自定义编号，一般从 0 开始。这个例子我们可以看到 MPI 如何组织协调各个进程的工作。

在这个认知基础上，我们来尝试做一点简单的计算：

以下是一个 MPI 实现的从 1 加到 N 的并行程序。假设我们有 P 个进程，这个程序将把 1 到 N 的求和任务分割成 P 个部分，每个进程计算一个部分的和，然后所有的进程将它们的部分和收集到一个进程（通常是 root 进程，即进程 0）中，由这个进程完成最终的求和。如果我们需要求和的数非常大，那么这样的并行计算将比单进程的计算更快。

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

int main(int argc, char* argv[]) {
    // 定义进程的排名、通信域的大小和求和的上限
    int rank, size, N;
    // 定义各个进程的求和结果、求和的开始和结束位置，
    // 以及总的求和结果
```

```
long long sum = 0, start, end, total_sum;

// 初始化MPI环境，这里把命令行参数传递给MPI
MPI_Init(&argc, &argv);
// 获取当前进程的排名
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// 获取通信域的大小
MPI_Comm_size(MPI_COMM_WORLD, &size);

// 设置求和的上限N
N = 1000000000;  // replace with your N

// 计算每个进程的求和开始和结束位置
start = rank*(N/size) + 1;
end
= (rank == size - 1) ? N : (rank + 1)*(N/size);

    // 每个进程进行求和操作
    for (long long i = start; i <= end; i++) {
        sum += i;
    }

    // 使用MPI的归约操作，
    // 将所有进程的求和结果汇总到总的求和结果中
    MPI_Reduce(&sum,
               &total_sum,
               1,
               MPI_LONG_LONG_INT,
               MPI_SUM, 0, MPI_COMM_WORLD);

    // 如果是主进程（排名为0的进程），
    // 则打印出总的求和结果
    if (rank == 0) {
```

```
        printf("Sum from 1 to %d is %lld\n",
               N, total_sum);
    }

    // 结束MPI环境
    MPI_Finalize();

    return 0;
}
```

注意:

```
// 计算每个进程的求和开始和结束位置
start = rank*(N/size) + 1;
end = (rank == size - 1) ? N : (rank + 1)*(N/size);
```

每一个进程，根据自己的局部 rank 值，来确定自己求和的范围。比如，如果有 4 个进程，那么进程 0 的求和范围是 1 到 250000000，进程 1 的求和范围是 250000001 到 500000000，以此类推。注意最后一个进程的求和范围可能会比其他进程的范围要大。因为 N 未必能整除 size。所以剩下的部分就由最后一个进程来计算。

在这个程序中，我们使用 `MPI_Reduce` 函数来收集所有进程的部分和，并在 root 进程中完成最终的求和。

`MPI_Reduce` 是一个 MPI 函数，用于将所有进程中的数据通过某种操作（如求和、求最大值、求最小值等）合并到一个进程中。以下是这个函数的参数解释:

1. `&sum`: 这是一个指向本地（发送）缓冲区的指针，该缓冲区包含此进程要发送给接收进程的数据。
2. `&total_sum`: 这是一个指向全局（接收）缓冲区的指针，该缓冲区用于接收所有进程的归约结果。注意，只有根进程（在这个例子中是进程 0）的接收缓冲区会被填充，其他进程的接收缓冲区不会被修改。
3. `1`: 这是发送和接收的数据的数量。在这个例子中，每个进程发送和接收 1 个 `long long int` 类型的数据。

4. `MPI_LONG_LONG_INT`：这是发送和接收的数据的类型。在这个例子中，数据的类型是 `long long int`。

5. `MPI_SUM`：这是一个预定义的归约操作，表示所有进程的数据将被求和。MPI 提供了一系列的预定义归约操作，如 `MPI_MAX`（求最大值）、`MPI_MIN`（求最小值）、`MPI_PROD`（求乘积）等。

6. `0`：这是根进程的排名，即接收归约结果的进程。在这个例子中，进程 0 是根进程。

7. `MPI_COMM_WORLD`：这是通信器，即进程组。在这个例子中，使用的是 `MPI_COMM_WORLD`，它表示所有的 MPI 进程。

我们在实际环境中测试这个例子，我们会发现，当我们增加进程数时，程序的运行时间会减少。这是因为我们将计算任务分配给了多个进程。但是，计算时间的建设并不是线性的，也就是说，四个进程并不一定比一个进程快四倍。

这里的原因可能很复杂，比如维持一个并行环境是需要开销的，而且进程数越多，开销相应也会增加。但最根本的原因，一般有两个：进程间通讯的开销和任务分配的不均匀。比如在这个例子中，我们注意到四个进程的工作量并不一致，除了最后一个进程的工作量可能会比其他进程大（无法整除时）。这就是任务不平衡。而所有进程把自己的求和结果发送给根进程，也会有一些通讯开销。这也是相对单进程而言的一个额外开销。

不难想到，上述两个问题是无法避免的，而且并行的进程越是多，系统结构越是复杂，这两个问题会越加严重。我们可以通过一些方法来减轻这些问题。比如，我们可以尽量减少通讯的次数，减少通讯的数据量，或者尽量让任务分配均匀。但这就说明我们在并行计算时，能够追求的效率提升极限也不会超过一个常数倍数。并且实际上做不到。也就是说，随着进程数的增加，效率的提升是递减的。

这里还要注意，并不是所以的问题都适合并行计算。一个常见的无法并行的任务是具有强依赖性的任务，也就是说，每一步的输出都依赖于前一步的输出。这种类型的任务通常需要顺序执行，无法并行化。

例如，斐波那契数列的生成就是一个无法并行的任务。斐波那契数列定义为：

$$F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2), n > 1$$

在这个例子中，为了得到 $F(n)$，我们必须首先计算出 $F(n-1)$ 和 $F(n-2)$。这个过程无法并行化，因为每一步都依赖于前面两步的结果。

虽然这个任务无法并行化，但我们可以通过使用动态规划等技术来优化它，避免重复计算。这个是下学期算法课程的内容了。

总结一下，并行算法设计的主要思路就是先把问题分解成互相独立的子问题，然后把这些子问题分配给多个进程并行计算，最后将结果合并。似乎也不难，但实际情况往往比较复杂。比如无法整除时如何分配任务，如何减少通讯开销，如何避免任务不平衡等等。这些都是需要深入研究，并作出针对性修改的问题。

来看一个复杂一些的例子：

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

#define N 4  // 矩阵的行数和列数

// 从文件中读取矩阵和向量
void read_data(double matrix[N][N],
               double vector[N]) {
    FILE *file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            fscanf(file, "%lf", &matrix[i][j]);
        }
        fscanf(file, "%lf", &vector[i]);
    }

    fclose(file);
}
```

```
int main(int argc, char* argv[]) {
    int rank, size;
    double matrix[N][N],
           vector[N],
           result[N],
           local_result[N / size];

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

    // 在根进程中从文件中读取矩阵和向量
    if (rank == 0) {
        read_data(matrix, vector);
    }

    // 将向量广播到所有进程
    MPI_Bcast(vector,
              N,
              MPI_DOUBLE,
              0,
              MPI_COMM_WORLD);

    // 将矩阵的行分散到所有进程
    MPI_Scatter(matrix,
                N * N / size,
                MPI_DOUBLE,
                matrix[rank],
                N * N / size,
                MPI_DOUBLE,
                0,
                MPI_COMM_WORLD);
```

```
    // 每个进程计算其行与向量的点积
    for (int i = 0; i < N / size; i++) {
        local_result[i] = 0;
        for (int j = 0; j < N; j++) {
            local_result[i]
                += matrix[rank][j] * vector[j];
        }
    }

    // 将局部结果收集到结果向量中
    MPI_Gather(local_result,
               N / size,
               MPI_DOUBLE,
               result,
               N / size,
               MPI_DOUBLE,
               0,
               MPI_COMM_WORLD);

    // 在根进程中打印结果向量
    if (rank == 0) {
        printf("Result vector:\n");
        for (int i = 0; i < N; i++) {
            printf("%f\n", result[i]);
        }
    }

    MPI_Finalize();

    return 0;
}
```

这个程序的思路是在 rank 为 0 的进程中读取一个矩阵和一个向量，考

虑到矩阵乘以向量其实就是每一行各自乘以向量，然后是一个求和。这里我们将不同的行分配给不同的进程，然后每个进程计算自己的行与向量的点积，最后将结果收集到根进程中。当然这里做了简化设计，在实际计算中应该考虑每个进程分担若干行。同时如果行数不能整除进程数，还需要考虑如何分配最后若干行的任务。这一点和求和很像，我们这里就不再考虑了。我们现在集中讨论并行计算的另一个问题，这里被乘的向量显然是每个进程都需要的信息。这种情况下，我们用到广播：

```
// 将向量广播到所有进程
MPI_Bcast(vector, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

这个含义很明显，将 vector 广播到所有进程。这样每个进程都可以得到 vector 的值。

下面是这个函数的各个参数的含义：

- `vector`: 这是要广播的数据，也就是源数据。在这个例子中，这是一个包含 N 个元素的向量。
- N: 这是要广播的数据的数量。在这个例子中，我们广播 N 个 `double` 类型的数据。
- `MPI_DOUBLE`: 这是要广播的数据的类型。在这个例子中，我们广播的是 `double` 类型的数据。
- 0: 这是数据的来源，也就是进行广播的进程的编号。在 MPI 中，我们通常把进行广播的进程称为根进程。在这个例子中，根进程的编号是 0。
- `MPI_COMM_WORLD`: 这是一个通信器，用于确定广播的范围。`MPI_COMM_WORLD` 是一个预定义的通信器，包含了所有的进程。也就是说，广播会将数据发送给所有的进程。

而矩阵是一个按行分配的过程。这里我们没有像求和时那样自己写分配，而是利用了 `MPI_Scatter` 函数：

这段代码使用了 MPI（Message Passing Interface）库的 `MPI_Scatter` 函数，用于在并行计算环境中进行数据的分散。

下面是这个函数的各个参数的含义：

- `matrix`: 这是要分散的数据，也就是源数据。在这个例子中，这是一个 $N \times N$ 的矩阵。
- $N \times N$/size: 这是每个进程接收的数据数量。在这个例子中，我们将矩阵的每 N/size 行分散到一个进程。
- `MPI_DOUBLE`: 这是要分散的数据的类型。在这个例子中，我们分散的是 `double` 类型的数据。
- `matrix[rank]`: 这是每个进程接收数据的缓冲区。在这个例子中，每个进程将数据存储在其对应的 `matrix[rank]` 中。
- $N \times N$/size: 这是接收缓冲区的大小。在这个例子中，接收缓冲区的大小与发送的数据数量相同。
- `MPI_DOUBLE`: 这是接收缓冲区中数据的类型。在这个例子中，接收的数据类型是 `double`。
- 0: 这是数据的来源，也就是进行数据分散的进程的编号。在 MPI 中，我们通常把进行分散的进程称为根进程。在这个例子中，根进程的编号是 0。
- `MPI_COMM_WORLD`: 这是一个通信器，用于确定分散的范围。`MPI_COMM_WORLD` 是一个预定义的通信器，包含了所有的进程。也就是说，分散会将数据发送给所有的进程。

总的来说，这句代码的作用是将根进程（编号为 0）的矩阵 `matrix` 分散给所有的进程。然后 rank 进程将接收到的数据存储在自己的 `matrix[rank]` 中。这里由于 rank 0 是发送源，所以它的 `matrix` 不会被修改。

然后各自的进程会计算自己的行与向量的点积，最后我们使用 `MPI_Gather` 函数将各个进程的结果收集到根进程中。

以下是 `MPI_Gather` 函数的各个参数的含义：

- `local_result`: 这是发送缓冲区，也就是每个进程需要发送给根进程的数据。在这个例子中，每个进程都有一个名为 `local_result` 的数组，其中包含了该进程计算的局部结果。
- N/size: 这是每个进程发送的数据数量。在这个例子中，每个进程发送了 N/size 个 `double` 类型的数据。

- `MPI_DOUBLE`: 这是发送的数据的类型。在这个例子中，发送的数据是 `double` 类型的。
- `result`: 这是接收缓冲区，也就是根进程接收数据的地方。在这个例子中，所有的局部结果都收集到了名为 `result` 的数组中。
- N/size: 这是接收缓冲区的大小。在这个例子中，接收缓冲区的大小与发送的数据数量相同。
- `MPI_DOUBLE`: 这是接收的数据的类型。在这个例子中，接收的数据是 `double` 类型的。
- 0: 这是数据的目的地，也就是进行数据收集的进程的编号。在 MPI 中，我们通常把进行收集的进程称为根进程。在这个例子中，根进程的编号是 0。
- `MPI_COMM_WORLD`: 这是一个通信器，用于确定收集的范围。`MPI_COMM_WORLD` 是一个预定义的通信器，包含了所有的进程。也就是说，收集会将数据从所有的进程收集到根进程。

`MPI_Gather` 和之前在求和时用到的 `MPI_Reduce` 都是 MPI（Message Passing Interface）库中的函数，用于在并行计算环境中收集数据。然而，它们的工作方式和用途有所不同。

- `MPI_Gather` 是一种简单的数据收集操作。它将每个进程的数据收集到根进程中，形成一个数组。在 `MPI_Gather` 操作结束后，根进程将拥有所有进程的数据，而其他进程的数据保持不变。这对于需要在一个地方收集所有数据进行进一步处理的情况非常有用。
- `MPI_Reduce` 不仅收集数据，而且在收集过程中对数据进行某种特定的操作（如求和、求最大值、求最小值等）。例如，如果使用 `MPI_SUM` 作为 `MPI_Reduce` 的操作，那么在 `MPI_Reduce` 操作结束后，根进程将得到所有进程数据的总和，而其他进程的数据保持不变。这对于需要进行全局计算的并行算法非常有用，例如求解线性系统的迭代方法（将在数值代数这门课中讨论）。

所以我们可以看到，这是一个设计并不精巧的并行策略，有很大的改进空间。但这里我们首先关注一个要点，广播真的必要么？显然，联系我们上

一节课的内容，要确保数据正确广播到每一个进程，在大规模网络并行时，是非常消耗资源的。对于这种必须要同步的信息，除了广播，那么另外一个可以考虑的方式，就是在每个进程中同步读取数据。但是这个涉及到分布式的文件读写。注意在大规模的分布式计算中，我们应该认为不同的进程是运行在不同的计算机上的，如何跨越网络进行读写，可能比广播通讯更加复杂，同时涉及硬件设计。这个已经超出了我们的讨论范围。或者，我们事先将每个进程都需要的文件拷贝到每个计算机上，这样就可以避免广播。但这样又会带来另外的问题，比如数据的一致性，数据的更新等等。

我们这里只是见识一下分布式计算的场景和基本手段，不再纠结于具体的技术问题。在真正应用分布式计算的场景，我们专业的主要任务一般也不是去设计分布式系统本身，而是去设计分布式算法。当然，对分布式系统在硬件上的了解，对设计高效率的算法本身也是重要的。

2 基于分布式计算的科学计算工具包

既然分布式系统本身的设计过于复杂，甚至某种程度上说，MPI 规范对于设计科学计算问题也太技术化了。为此，出现了一些基于分布式计算的科学计算工具包。使得我们可以在 “相信分布式系统是良好设计” 的基础上，专注于算法设计和科学问题的求解。以下是一些常用的基于分布式计算的科学计算工具包：

- **PETSc**: PETSc 是一个用于求解偏微分方程和相关问题的可扩展科学计算工具包，它利用 MPI 实现并行计算。支持 C, C++, Fortran 77 和 Fortran 90 语言。官方网站：https://www.mcs.anl.gov/petsc/
- **Trilinos**: 这是一套基于 C++ 的并行线性和非线性方程求解器库，它使用 MPI 进行并行计算。官方网站：https://trilinos.github.io/
- **Hypre**: 这是一套专门用于求解大规模并行线性代数方程的库，使用 MPI 进行并行计算。官方网站：https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods
- **deal.II**: 这是一个用于有限元分析的 C++ 库，它也支持 MPI 并行计算。官方网站：https://www.dealii.org/
- **LAMMPS**: 这是一款大规模原子/分子并行模拟器，它使用 MPI 来处理并行计算。官方网站：https://lammps.sandia.gov/

- **GROMACS**: 这是一款分子动力学模拟软件，也使用 MPI 进行并行计算。官方网站：http://www.gromacs.org/
- **OpenFOAM**: 这是一个用于处理流体动力学问题的 C++ 库，它支持 MPI 并行计算。官方网站：https://openfoam.org/
- **Quantum ESPRESSO**: 这是一款从第一原理进行量子力学计算和材料建模的软件包，使用 MPI 进行并行计算。官方网站：https://www.quantum-espresso.org/
- **NAMD**: 这是一款用于进行大规模分子动力学模拟的并行计算软件，也使用 MPI 进行并行计算。官方网站：https://www.ks.uiuc.edu/Research/namd/

下面我们以 PETSc 为例，介绍一下这个工具包的基本使用方法。

```
#include <petscvec.h>

int main(int argc,char **args)
{
    Vec            x;  // 向量
    PetscErrorCode ierr;
    PetscInt       i, n = 10;
    PetscScalar    v;

    // 初始化PETSc环境，注意返回值总是出错码.
    ierr = PetscInitialize(&argc,&args,(char*)0,0);
    CHKERRQ(ierr); // 检查是否出错的断点.

    // 创建向量
    ierr = VecCreate(PETSC_COMM_WORLD,&x);
    CHKERRQ(ierr);

    // 创建向量x
    ierr = VecSetSizes(x,PETSC_DECIDE,n);
    CHKERRQ(ierr); // 设置向量的大小
```

```
    ierr = VecSetFromOptions(x);
    CHKERRQ(ierr); // 从命令行选项中设置向量的类型

    // 设置值
    for (i=0; i<n; i++) { // 对向量中的每个元素赋值
        v = i;
        ierr = VecSetValues(x,1,&i,&v,INSERT_VALUES);
        CHKERRQ(ierr);
    }

    // 装配向量
    ierr = VecAssemblyBegin(x);
    CHKERRQ(ierr); // 开始装配向量

    ierr = VecAssemblyEnd(x);
    CHKERRQ(ierr); // 结束装配向量

    // 打印向量
    ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);
    CHKERRQ(ierr); // 在标准输出中查看向量

    // 释放内存
    ierr = VecDestroy(&x);
    CHKERRQ(ierr); // 销毁向量

    ierr = PetscFinalize();
    CHKERRQ(ierr); // 结束PETSc环境
    return 0;
}
```

这个程序很简单，就是产生一个向量，然后给向量赋值，最后打印出来。我们这里可以看到 PETSc 的一些工作特点。

比如它很明显启用了 MPI 的框架。我们可以看到 `PETSC_COMM_WORLD` 这个参数，这是一个 MPI 的通信器，用于确定向量的范围。而且

```
// 初始化PETSc环境，注意返回值总是出错码.
ierr = PetscInitialize(&argc,&args,(char*)0,0);
CHKERRQ(ierr); // 检查是否出错的断点.
```

和

```
MPI_Init(&argc, &argv);
```

也很类似。

但是，我们并没有看见基于 rank 的操作，也没有看见显式的广播和分散操作。这是因为 PETSc 是一个高级的科学计算工具包，它封装了这些细节。所以我们只是做一些简单的操作，而不用关心底层的细节。这也是科学计算工具包的优势所在。它会自行考虑如何在分布式系统上高效运行。

当然有时候这些工具包的使用也会有一些限制，比如我们可能无法直接访问底层的数据，或者我们可能无法直接控制底层的计算。有时这样的封装使我们无法获得在针对性问题上的最优性能。但优点是确实简化了我们的工作，使我们可以专注于算法设计和科学问题的求解。

不论是 MPI 本身，还是这种封装，都使得程序的调试和维护变得更加困难。所以在使用这些工具包时，我们需要更加小心，确保我们的程序是正确的。这也是 PETSc 在每一个步骤都有一个 `CHKERRQ` 的原因。这个函数会检查每一步的操作是否出错，如果出错，就会打印出错误信息，并终止程序。这样一旦错误出现，我们可以快速发现并定位。从某种意义上说，这也暗示了在 MPI 框架下的调试，本质上就是 printf 大法。因为你无法想像继续用 gdb 来调试 MPI 程序。

我们这里再看一个更复杂的例子，用积分来计算 log(2)。具体的算法我们会在数值分析课程再讨论。

多进程并行和多线程并行相比，主要的优势在于多进程并行可以在多个计算机上运行，而多线程并行只能在一个计算机上运行。但这也导致了多进程并行的通讯开销更大。而多线程因为只是在一个计算机上运行，通讯开销小，甚至可以通过共享内存交换数据，所以在单机的情况下，多线程并行可能会比多进程并行效率更高。除了之前我们介绍过的多线程编程，还有一种更加高级的多线程并行工具，叫做 OpenMP。

3 OpenMP 多线程计算

OpenMP（Open Multi-Processing）是一个支持多平台共享内存并行编程的 API，以 C，C++ 和 Fortran 语言为基础。它由一组编译指示、库例程、环境变量以及运行时库组成，用于说明并行的区域、数据的私有性或共享性、同步、用户级运行时例程等。

1. **简单性**：OpenMP 使用编译指示的方式进行并行化，对于程序员来说，只需要在代码中添加适当的编译指示即可。
2. **可移植性**：OpenMP 是跨平台的，并行化后的程序无需修改即可在不同的操作系统和计算机体系结构上运行。
3. **兼容性**：未经并行化的串行程序可以在 OpenMP 环境中运行，并且可以逐步并行化串行程序。
4. **扩展性**：OpenMP 支持并行循环、并行区域、并行任务等多种并行模式。

OpenMP 的编译指示都是以 `#pragma omp` 开始的，例如：

```
#pragma omp parallel
{
    // 并行执行的代码
}
```

这段代码会被所有可用线程并行执行。

OpenMP 还提供了一套运行时库，例如获取线程数量、设置线程数量等功能。OpenMP 定义了一些环境变量来控制并行执行的行为，例如 `OMP\_NUM\_THREADS` 用来设置线程数量。总的来说，OpenMP 提供了一种简单、灵活的并行编程方式，极大地降低了并行编程的难度。

下面是一个简单的例子：

```
#include <stdio.h>  // 引入标准输入输出库
#include <omp.h>  // 引入OpenMP库

int main() {
    // 一个预处理指令，
```

```
    // 告诉编译器以下的代码块要并行执行
    #pragma omp parallel
    {
        // 返回当前线程的编号，
        // 这个编号是唯一的，由 OpenMP 运行时系统分配
        int id = omp_get_thread_num();
        // 每个线程打印一条包含其线程编号的消息

        printf("Hello from thread %d\n", id);
    }
    // 程序正常结束，返回0
    return 0;
}
```

注意 OpenMP 程序的编译方式如下：

```
gcc -fopenmp filename.c -o outputfile
```

在 GCC 编译器中，`-fopenmp` 是一个编译器选项，用于启用 OpenMP 并行编程支持。该选项会告诉编译器去识别源代码中的 OpenMP 编译指示（如 `#pragma omp`），并生成相应的并行代码。它还会自动链接 OpenMP 运行时库，这个库提供了线程管理、同步等功能，使得 OpenMP 程序可以运行。此外，还会开启一些与并行编程相关的优化，例如自动循环并行化。如果在编译包含 OpenMP 编译指示的代码时不使用`-fopenmp`选项，那么这些指示将被忽略，代码将以串行方式运行。（或者因为调用了无法识别的函数而报错。）

这里我们用

```
gcc -fopenmp -o hello_mp hello_mp.c
```

编译。可以看到我们每一个核（不论强弱），都参与了运行。

如果你不想启动这么多线程，有一种 OpenMP 独特的控制方式是通过环境变量：`OMP_NUM_THREADS`。

例如我们这里如果在 shell 中设置：

```
export OMP_NUM_THREADS=4
```

那么同样的程序，就只有 4 个线程在运行了。

当然，除了预编译和环境变量以外，OpenMP 也提供了一些运行时函数，用于在程序运行时控制线程的数量，比如 `omp_set_num_threads` 函数。

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

int main() {
    omp_set_num_threads(4);  // 设置线程数为4
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello from thread %d\n", id);
    }
    return 0;
}
```

这段代码能实现同样的效果。但通过环境变量修改不需要重新编译。

下面我们同样用 OpenMP 来实现一下求和的例子:

```
#include <stdio.h>  // 引入标准输入输出库
#include <omp.h>  // 引入OpenMP库

int main() {
    long long N = 1000000000;
    long long sum = 0;

    // #pragma omp parallel for reduction(+:sum)
    // 是一个预处理指令，
    // 告诉编译器以下的for循环要并行执行，
    // 并在所有线程完成后将各线程的sum变量的值加起来
    #pragma omp parallel for reduction(+:sum)
    for (long long i = 1; i <= N; i++) {
        sum += i;
    }
```

```
    printf("Sum = %lld\n", sum);
    return 0;
}
```

.

我们可以看到由于不需要处理通讯，OpenMP 的代码比 MPI 的代码简洁很多。而且预编译指令有一定的自动优化处理能力，在这个例子中，这句风骚的 `#pragma omp parallel for reduction(+:sum)` 就是告诉编译器，这个循环可以并行化，而且最后把每个线程的 sum 变量加起来。甚至已经有一丝人工智能的味道了。未来的编译器可能会更加智能，自动优化我们的代码。只需要我们指出编程的思路和基本逻辑。

如果只是在单机上运行，OpenMP 还有一个显著的优点，就是线性加速比。让我们用环境变量 `OMP_NUM_THREADS` 来测试一下，当启动不同线程数量时的运行时间。

在未来的课程学习中，利用 OpenMP 来提升程序的并行效率是一个加分因素。而相比之下，MPI 的使用场景更多的是在分布式计算中，也就是多 CPU 的情形。需要从硬件到算法的全面调整，相当的苦大仇深。但它确实是目前我们在计算上能够实现的终极手段之一（不考虑 GPU）。