MPI 实验

> Name: 梁鑫嵘；ID: 200110619

MPI 相关函数介绍

MPI 标准为多进程编程提供了许多 API，如数据的发送、接收等简单数据处理函数，数据的发射和接收等集合数据处理函数。

```
// 归约：从每个进程收集数据到一个进程的单个值
int MPI_Reduce (void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op op, // 操作符：MPI_MAX, MPI_SUM...
  int root,
  MPI_Comm comm)
// 先归约得到值然后分发结果到每一个进程：
int MPI_Allreduce (void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op op,
  MPI_Comm comm)
// 广播：将相同数据分发到各个进程，内存同步
int MPI_Bcast (void *buffer,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm comm)
// 散射相同长度的数据：
int MPI_Scatter (void *sendbuf,
  int sendcnt,  // 指的是单个数据的长度，不是发多少个线程
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm)
// 散射不同长度数据
// 与MPI_Scatter类似，但允许sendbuf中每个数据块的长度不同并且可以按任意的顺序排放。
// sendbuf、sendtype、sendcnts和displs仅对根进程有意义。
// 数组sendcnts和displs的元素个数等于comm中的进程数，
// 它们分别给出发送给每个进程的数据长度和位移，均以sendtype为单位。
int MPI_Scatterv (void *sendbuf,
  int *sendcnts,
```

```
  int *displs,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm)
// 数据聚焦：收集相同长度的数据块。
// 以root为根进程，所有进程(包括根进程自己) 将sendbuf中的数据块发送给根进程，
// 根进程将这些数据块按进程号的顺序依次放到recvbuf中。
int MPI_Gather (void *sendbuf,
  int sendcnt,
  MPI_Datatype sendtype,
  void *recvbuf,
  int recvcnt,
  MPI_Datatype recvtype,
  int root,
  MPI_Comm comm)
```

划分方案

$A_{n\times n}, B_{n\times n}, C_{i,j} = \sum_{k=0}^{n} A_{i,k} \times B_{k,j}$。

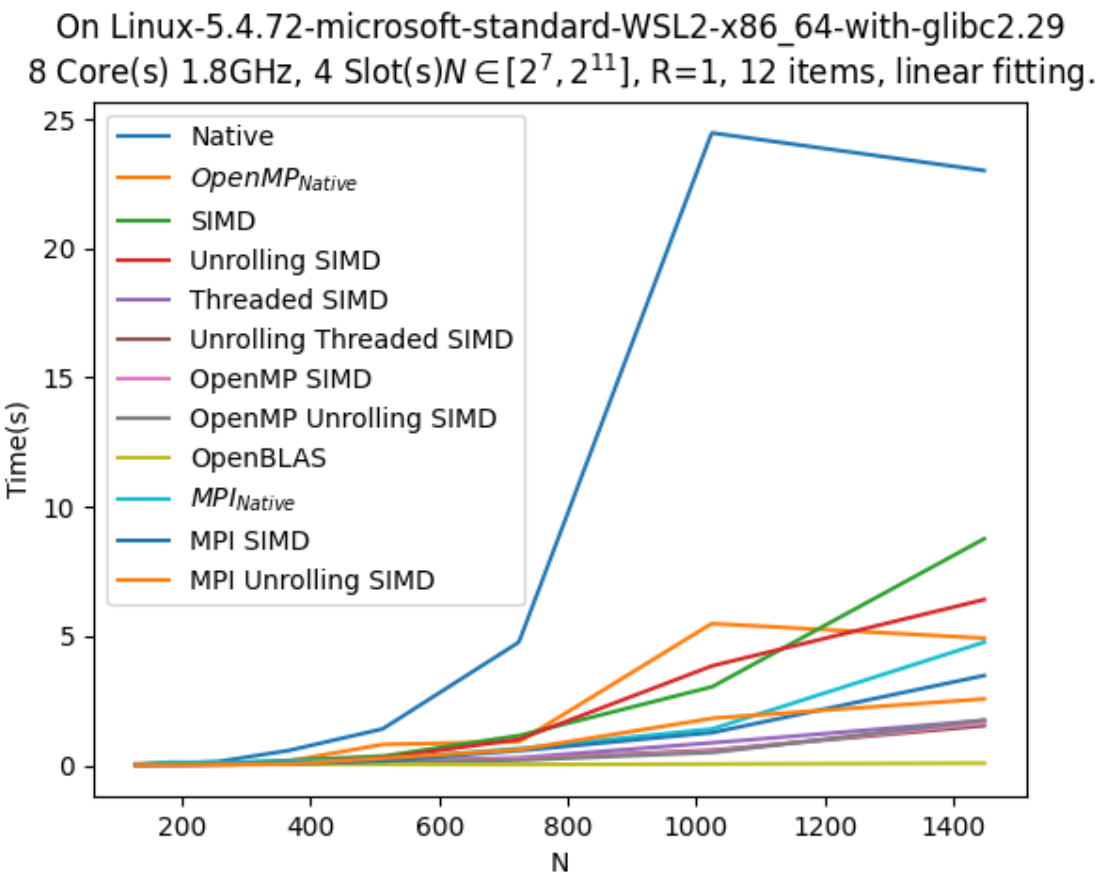
在前几次实验中，划分方案为以k为单位划分，将$A_{*,k}, B_{k,*}$数据传送到对应进程/线程进行计算；而在 MPI 实验中，经过测试$N = 1024$时这样数据传输使用时间为总使用时间的约93%，所以在 MPI 实验中，虽然仍然以k为单位划分，但是在进程启动之初就传输整个矩阵的数据，减少计算中产生的数据传输耗时。

主要代码

```
Mat *mat_mul_mpi_all(Mat *a, Mat *b, Mat *c, int unrolling, int native) {
  int rank = 0, size = 0;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  Mat *t = NULL;
  // 同步内存信息
  if (rank == 0) {
    // 首先对 b 进行一个置的转
    t = mat_transpose(b);
    // 然后发送 a, b 数据到其他 slot
    for (int i = 1; i < size; i++) {
      // 按行发送数据，防止因为content_real不一致造成数据错误
      for (int x = 0; x < a->h; x++) {
        MPI_Send(a->data[x], a->w, MPI_DOUBLE, i, MPI_TAG_MAT_A,
                 MPI_COMM_WORLD);
        MPI_Send(t->data[x], a->w, MPI_DOUBLE, i, MPI_TAG_MAT_B,
                 MPI_COMM_WORLD);
      }
    }
  } else {
    t = b;
```

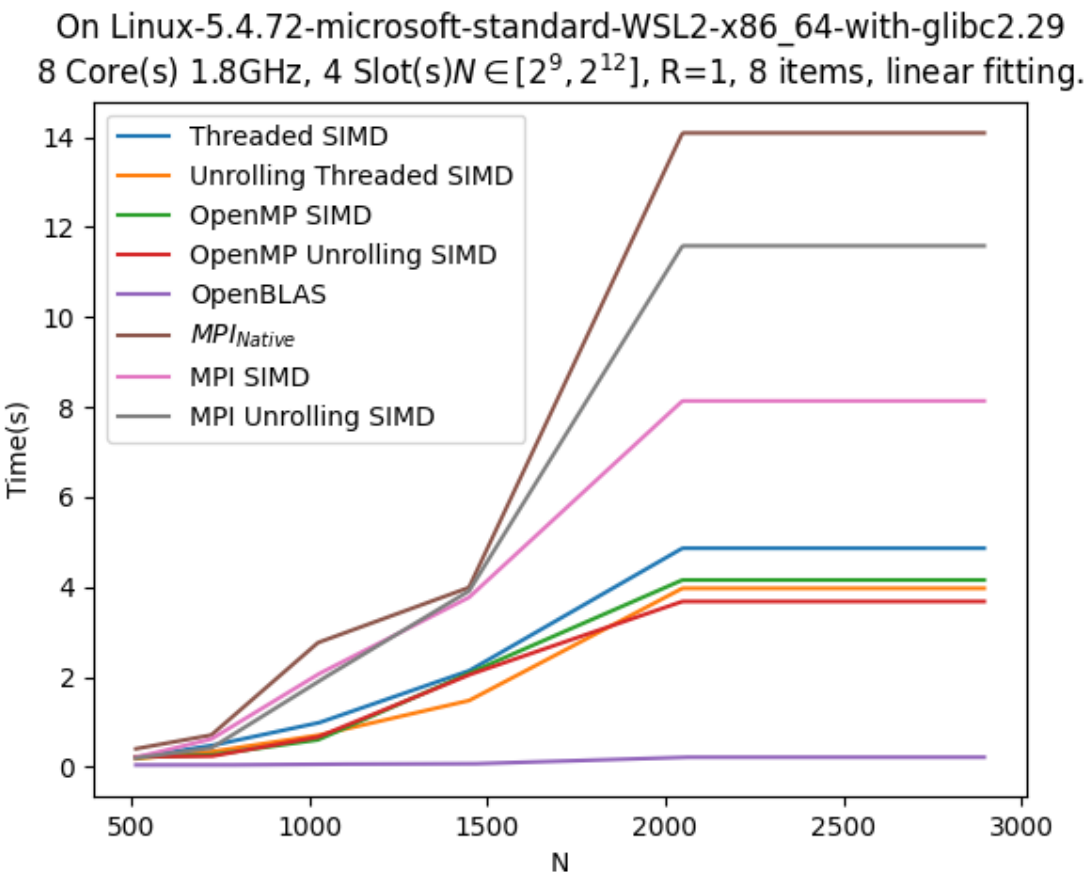
```
    for (int x = 0; x < a->h; x++) {
      MPI_Recv(a->data[x], a->w, MPI_DOUBLE, 0, MPI_TAG_MAT_A, MPI_COMM_WORLD,
               MPI_STATUSES_IGNORE);
      MPI_Recv(t->data[x], a->w, MPI_DOUBLE, 0, MPI_TAG_MAT_B, MPI_COMM_WORLD,
               MPI_STATUSES_IGNORE);
    }
  }
  // 初始化任务数据，进行数据分发
  double *sum_part = malloc(sizeof(double) * size);
  for (int x = rank; x < a->w - a->w % size; x += size) {
    for (int ys = 0; ys < b->h; ys += 1) {
      double sum =
          mat_cell_do_mul(a->data[x], t->data[ys], a->w, unrolling, native);
      MPI_Gather(&sum, 1, MPI_DOUBLE, sum_part, 1, MPI_DOUBLE, 0,
                 MPI_COMM_WORLD);
      if (rank == 0) {
        for (int i = 0; i < size; i++) {
          c->data[x + i][ys] = sum_part[i];
        }
      }
    }
  }
  if (rank == 0)
    for (int xr = a->w - a->w % size; xr < a->w; xr++) {
      for (int ys = 0; ys < b->h; ys += 1) {
        c->data[xr][ys] =
            mat_cell_do_mul(a->data[xr], t->data[ys], a->w, unrolling, native);
      }
    }
  if (rank == 0) {
    mat_free(t);
  }
  free(sum_part);
  return c;
}
```

数据对比



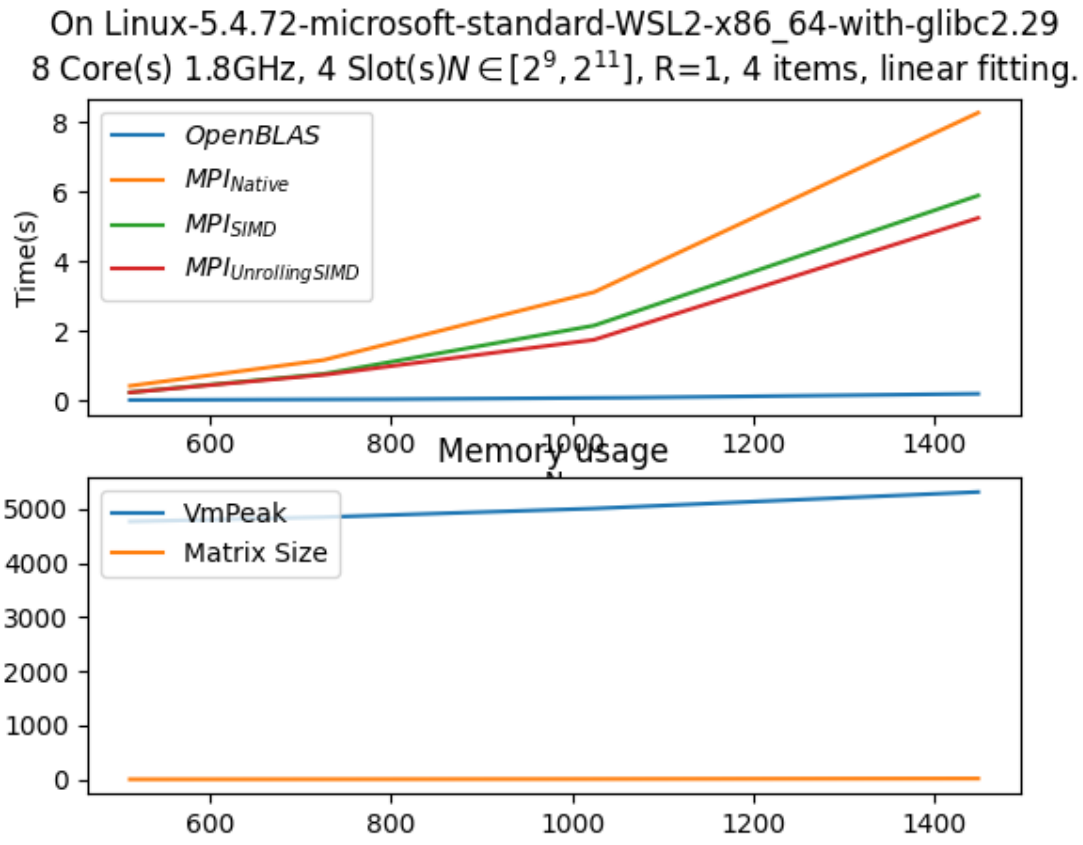
mpi_wsl_s7_m11_r1_linear_sl4.png

各算法效率对比。



mpi_wsl_s9_m12_r1_linear_sl4_t4.png

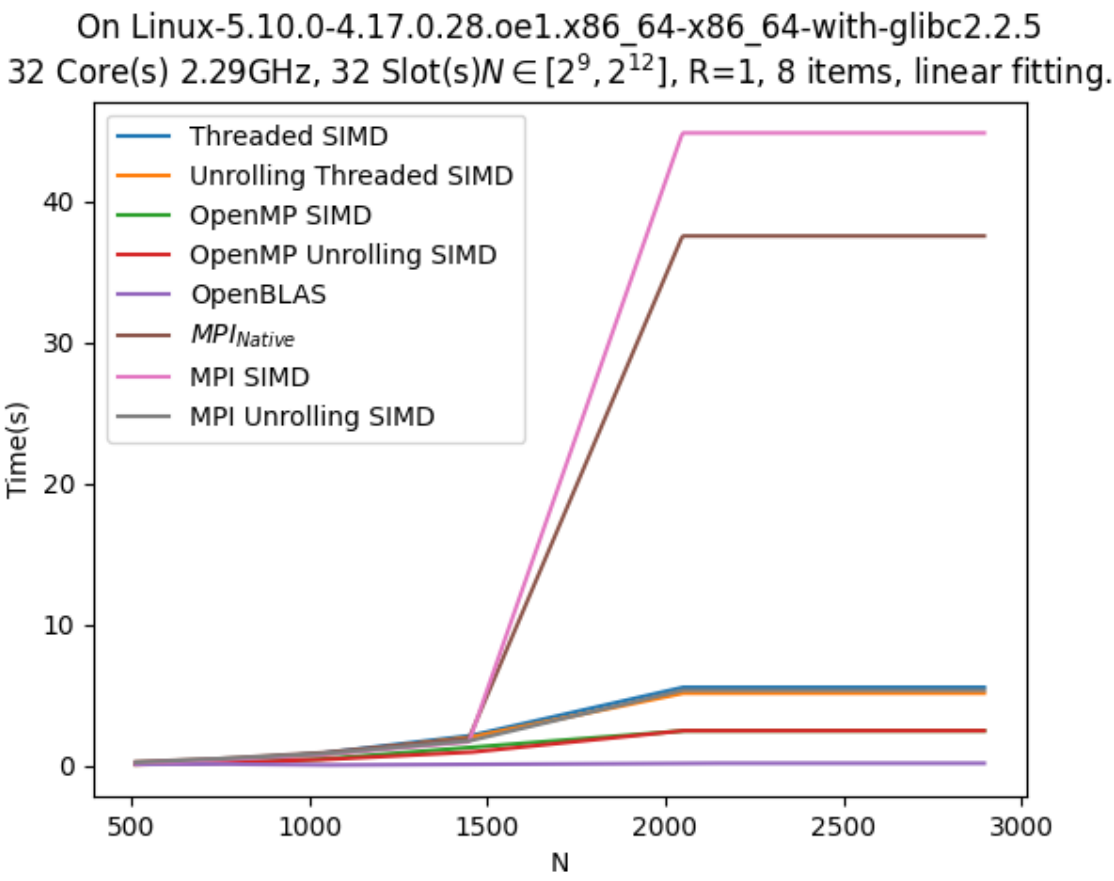
多线程算法对比。（矩阵过大时因为内存过大被系统 Kill 故没有数据。）



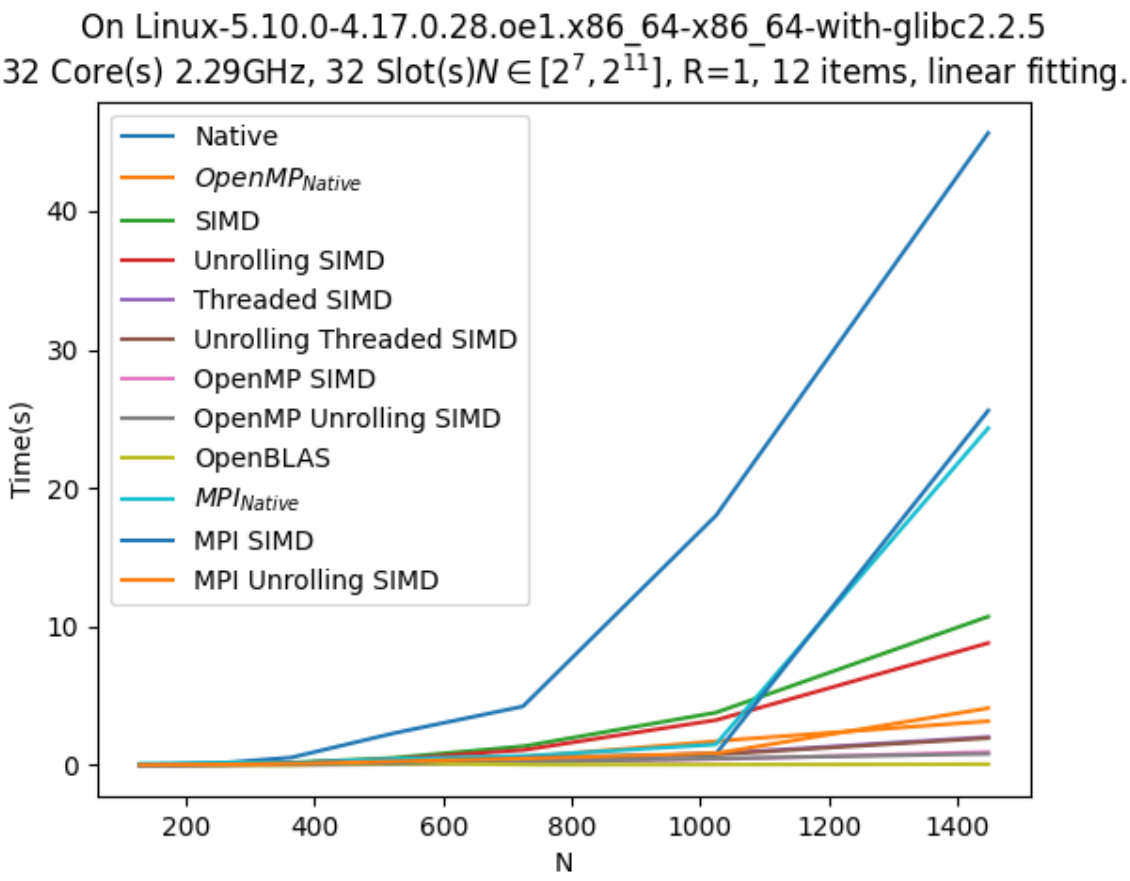
mem_plot_s9_m11_r1_linear_sl4_t8_d0.png

MPI 的内存使用对比。

内存图的“VmPeak”线是 4 个 MPI Slot 同时运行时的 `/proc/${pid}/status` 中的 `VmPeak` 值的和，单位为“MB”。



mpi_server_s9_m12_r1_linear_sl32_t4.png



mpi_server_s7_m11_r1_linear_sl32_t0.png

在多核服务器上的情况。

实验结论

1. 单线程和多线程对比：

   多线程计算相比单线程计算有很大优势。

2. 测量 MPI 多线程计算内存使用：

   在 MPI多线程计算中内存使用比理论值大很多。猜测原因：

   1. 多进程运行时 `VmPeak` 将每个进程使用的 `OpenMPI` 的库在内存中的占用重复计算了
   2. 因为计算的是虚拟内存，有可能内存被压缩或者有内存共享。

3. 特殊情况分析：

   a. 在 N 特别大时，MPI 效率突然变慢：有可能是因为 N 过大，造成 CPU 缓存装不下单行、单列数据，以至于数据访问延迟突然增大。

   b. 数据有时候持平：内存过大，进程被 Linux 系统杀死。