

# 第二次作业 实验报告

本次作业的参考文献有： [1](#) [2](#) [3](#)。

## 实验目的

1. 实现高效的模板计算
2. 掌握主要的体系结构优化手段
3. 掌握常见的并行计算工具

## 工程说明

本项目实现了模板计算的多种优化的方式，存放于若干个git branch上：

```
$ git branch -v
master  # 主分支, 与mpi(本项目的sota)一致
openmp  # openmp优化
mpi      # mpi优化
```

## 实验过程

### 0 集群基本概况介绍

为了充分利用集群中的资源，我们需要查看集群的配置。集群CPU为双路Xeon Gold 6342，每路CPU共有24物理核心，每个物理核心支持2超线程；每个节点共有96个核心，整个集群共有384个核心。

### 1 baseline分析及其编译优化

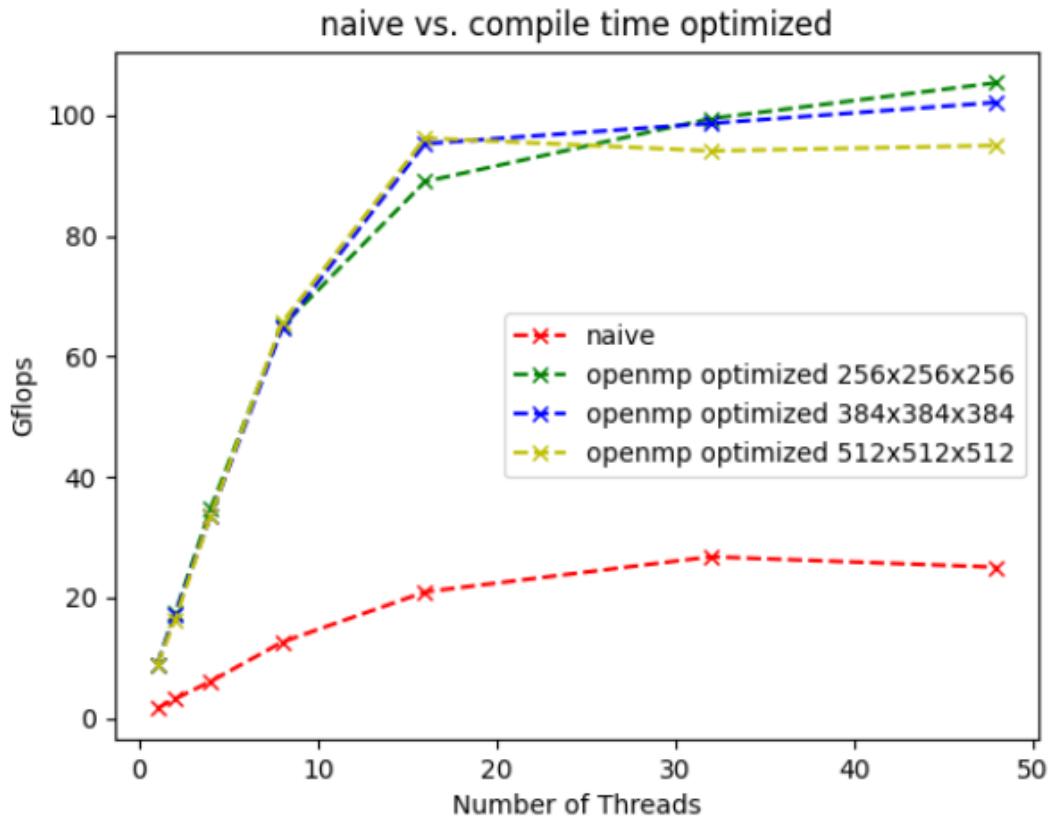
注：为节约时间起见，在比较不同方法时暂时将 `benchmark.sh` 中的 `Timestep` 设置为16。

首先看 `stencil-naive.c` 中模板计算的循环实现：

```
for (int t = 0; t < nt; ++t)
{
    cptr_t a0 = buffer[t % 2];
    ptr_t a1 = buffer[(t + 1) % 2];
#pragma omp parallel for schedule(dynamic)
    for (int z = z_start; z < z_end; ++z)
    {
        for (int y = y_start; y < y_end; ++y)
        {
            for (int x = x_start; x < x_end; ++x)
            {
                ...
            }
        }
    }
}
```

我们对其采取编译优化，添加如下编译选项： `-O3 -fomit-frame-pointer -ffast-math -march=native`

性能提升如下：



## 2 串行优化

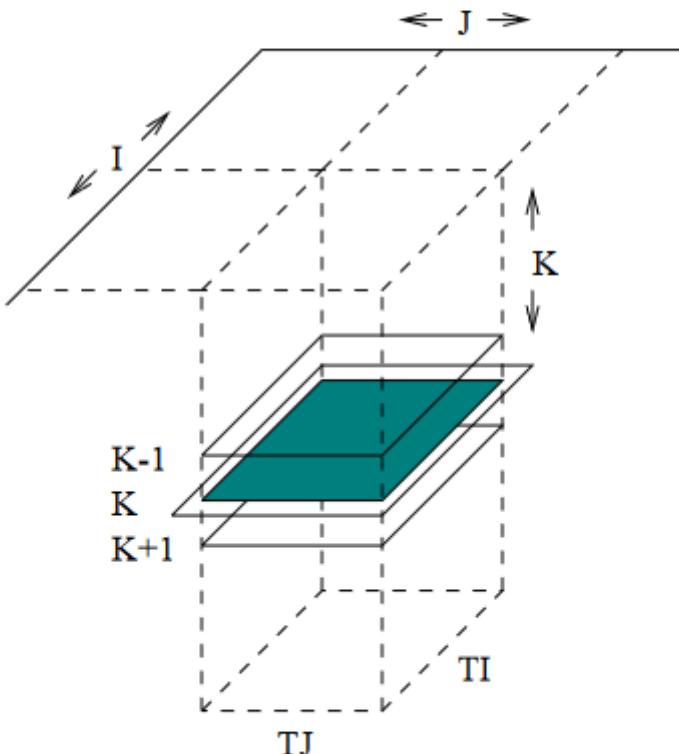
相比于传统的2D stencil模板计算，3D stencil模板计算中对相同数据的访问通常相隔太远，需要在每次阵列扫描时将阵列元素多次带入缓存。对此，我们借鉴...一文中tiling的思路，对数据进行分块重排。

```

do JJ=2,N-1,TJ
do II=2,N-1,TI
do K=2,N-1
  do J=JJ,min(JJ+TJ-1,N-1)
    do I=II,min(II+TI-1,N-1)
      A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K) +
                    B(I,J-1,K)+B(I,J+1,K) +
                    B(I,J,K-1)+B(I,J,K+1))

```

**Figure 6** Tiled 3D Jacobi iteration



**Figure 7** Access pattern of tiled 3D Jacobi

如图所示， $X$  ( $I$ ) 方向和  $Y$  ( $J$ ) 方向被划分为大小为  $xx$  ( $II$ ) 和  $yy$  ( $JJ$ ) 的小块。阴影区域代表在单个  $K$  循环迭代中需要进行修改的点，而周围三个没有阴影的区域代表单个  $K$  循环迭代中需要访问的点（注意如图所示展示的是  $7$ -stencil 的计算，而本文中需要进行  $27$ -stencil 的计算，因此边界的访问情况略有不同，详见代码）。

```

for (int yy = y_start; yy < y_end; yy += BLOCK_Y)
{
  int FIXED_BLOCK_Y = min(BLOCK_Y, y_end - yy); // consider the edge situation
  for (int xx = x_start; xx < x_end; xx += BLOCK_X)
  {
    int FIXED_BLOCK_X = min(BLOCK_X, x_end - xx);

    // get the small block value to write
    ptr_t a1_block = a1 + z_start * ldx * ldy + yy * ldx + xx;

    // get the small block value to read
    ptr_t a0_block_Z = a0 + z_start * ldx * ldy + yy * ldx + xx;
    ptr_t a0_block_P = a0 + (z_start + 1) * ldx * ldy + yy * ldx + xx;

```

```

ptr_t a0_block_N = a0 + (z_start - 1) * ldx * ldy + yy * ldx + xx;

// loop inside block
for (int z = z_start; z < z_end; ++z)
{
    for (int y = 0; y < FIXED_BLOCK_Y; ++y)
    {
        for (int x = 0; x < FIXED_BLOCK_X; ++x)
        {
            a1_block[INDEX_NEW(x, y, ldx)] = \
                ALPHA_ZZZ * a0_block_Z[INDEX_NEW(x, y, ldx)] \
                + ALPHA_NZZ * a0_block_Z[INDEX_NEW(x - 1, y, ldx)] + \
ALPHA_PZZ * a0_block_Z[INDEX_NEW(x + 1, y, ldx)] \
                + ALPHA_ZNZ * a0_block_Z[INDEX_NEW(x, y - 1, ldx)] + \
ALPHA_ZPZ * a0_block_Z[INDEX_NEW(x, y + 1, ldx)] \
                + ALPHA_ZZN * a0_block_N[INDEX_NEW(x, y, ldx)] + ALPHA_ZZP * \
a0_block_P[INDEX_NEW(x, y, ldx)] \
                + ALPHA_NNZ * a0_block_Z[INDEX_NEW(x - 1, y - 1, ldx)] + \
ALPHA_PNZ * a0_block_Z[INDEX_NEW(x + 1, y - 1, ldx)] \
                + ALPHA_NPZ * a0_block_Z[INDEX_NEW(x - 1, y + 1, ldx)] + \
ALPHA_PPZ * a0_block_Z[INDEX_NEW(x + 1, y + 1, ldx)] \
                + ALPHA_NZN * a0_block_N[INDEX_NEW(x - 1, y, ldx)] + \
ALPHA_PZN * a0_block_N[INDEX_NEW(x + 1, y, ldx)] \
                + ALPHA_NZP * a0_block_P[INDEX_NEW(x - 1, y, ldx)] + \
ALPHA_PZP * a0_block_P[INDEX_NEW(x + 1, y, ldx)] \
                + ALPHA_ZNN * a0_block_N[INDEX_NEW(x, y - 1, ldx)] + \
ALPHA_ZPN * a0_block_N[INDEX_NEW(x, y + 1, ldx)] \
                + ALPHA_ZNP * a0_block_P[INDEX_NEW(x, y - 1, ldx)] + \
ALPHA_ZPP * a0_block_P[INDEX_NEW(x, y + 1, ldx)] \
                + ALPHA_NNN * a0_block_N[INDEX_NEW(x - 1, y - 1, ldx)] + \
ALPHA_PNN * a0_block_N[INDEX_NEW(x + 1, y - 1, ldx)] \
                + ALPHA_NPN * a0_block_N[INDEX_NEW(x - 1, y + 1, ldx)] + \
ALPHA_PPN * a0_block_N[INDEX_NEW(x + 1, y + 1, ldx)] \
                + ALPHA_NNP * a0_block_P[INDEX_NEW(x - 1, y - 1, ldx)] + \
ALPHA_PNP * a0_block_P[INDEX_NEW(x + 1, y - 1, ldx)] \
                + ALPHA_NPP * a0_block_P[INDEX_NEW(x - 1, y + 1, ldx)] + \
ALPHA_PPP * a0_block_P[INDEX_NEW(x + 1, y + 1, ldx)];
        }
    }
    // update the pointer of block
    a1_block = a1_block + ldx * ldy;
    a0_block_N = a0_block_Z;
    a0_block_Z = a0_block_P;
    a0_block_P = a0_block_P + ldx * ldy;
}
}
}

```

### 3 OpenMP并行优化

OpenMP可以用于单个计算节点内的线程并行。在上一节分块策略的基础上，我们考虑对外围的块循环进行多线程并行，使用如下语句：

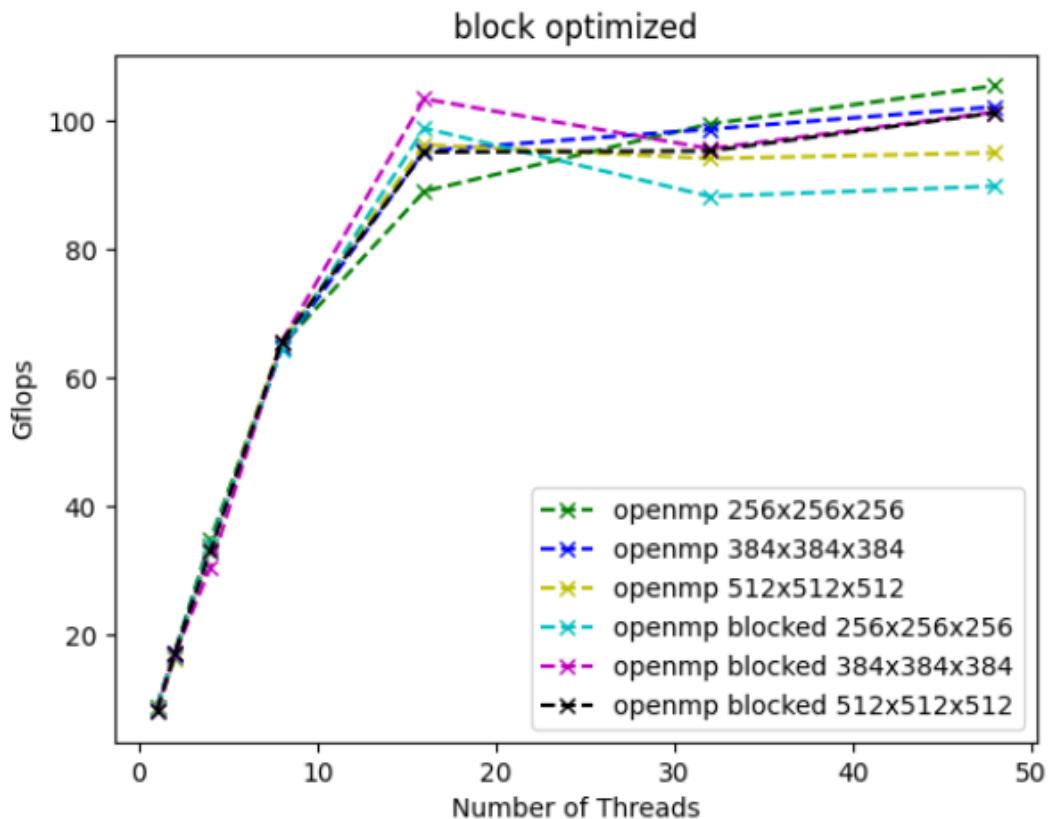
```

#pragma omp parallel
#pragma omp for schedule(dynamic) collapse(2)
for (int yy = y_start; yy < y_end; yy += BLOCK_Y)
{
    for (int xx = x_start; xx < x_end; xx += BLOCK_X)
    {
        int FIXED_BLOCK_Y = min(BLOCK_Y, y_end - yy); // consider the edge
situation
        int FIXED_BLOCK_X = min(BLOCK_X, x_end - xx);
        ...
    }
}

```

这里值得注意的是，考虑到 `BLOCK_X` 和 `BLOCK_Y` 的值通常较大，循环所进行的次数较少，而线程数最多 48，所以使用 `collapse(2)` 语句对两层循环进行展开，以充分利用线程资源。

颇感意外的是，这一优化并没有明显提升效率：



推测可能是编译优化起到了类似的内存优化效果。

## 4 MPI并行优化

在实现并行计算时，我们可以对计算任务在空间上进行划分。

假设我们在  $n$  (in  $x, y, z$ ) 轴上对 stencil 进行划分，那么每一个节点上计算的模板尺寸将发生变化，即设置 `grid_info->local_size_n = grid_info->global_size_n / grid_info->p_num`。与此同时，被划分后的不同子数组在空间上也应在不同的方向上错开一定距离，即设置 `grid_info->offset_n = grid_info->local_size_n * grid_info->p_id`。

注意到数组的 index 按照如下方式定义：

```
#define INDEX(xx, yy, zz, ldxx, ldyy) ((xx) + (ldxx) * ((yy) + (ldyy) * (zz)))
```

即按照内存顺序访问矩阵中的值时，z的更新最慢。所以，我们需要在z轴方向上划分数组：

```
void create_dist_grid(dist_grid_info_t *grid_info, int stencil_type)
{
    grid_info->local_size_x = grid_info->global_size_x;
    grid_info->local_size_y = grid_info->global_size_y;
    // divide the thread to p_num parts
    grid_info->local_size_z = grid_info->global_size_z / grid_info->p_num;
    if (grid_info->p_id == 0)
    {
        printf("use %d processes to divide the z\n", grid_info->p_num);
    }
    grid_info->offset_x = 0;
    grid_info->offset_y = 0;
    grid_info->offset_z = grid_info->local_size_z * grid_info->p_id;
    grid_info->halo_size_x = 1; //! 一个简单的padding
    grid_info->halo_size_y = 1;
    grid_info->halo_size_z = 1;
}
```

```
void extract_subarrays(dist_grid_info_t *grid_info, MPI_Datatype *recv_from_up,
MPI_Datatype *send_to_up, MPI_Datatype *send_to_down, MPI_Datatype
*recv_from_down)
{
    const int lz = grid_info->local_size_z;
    const int hz = grid_info->halo_size_z;
    const int ly = grid_info->local_size_y;
    const int hy = grid_info->halo_size_y;
    const int lx = grid_info->local_size_x;
    const int hx = grid_info->halo_size_x;
    const int array_of_sizes = {lz + 2 * hz, ly + 2 * hy, lx + 2 * hx};
    const int array_of_subsizes = {hz, ly + 2 * hy, lx + 2 * hx};
    int array_of_starts[3];

    // get the recv_from_up
    array_of_starts[0] = 0;
    array_of_starts[1] = 0;
    array_of_starts[2] = 0;
    MPI_Type_create_subarray(3, array_of_sizes, array_of_subsizes,
array_of_starts, MPI_ORDER_C, MPI_DOUBLE, recv_from_up);

    array_of_starts[0] = hz;
    array_of_starts[1] = 0;
    array_of_starts[2] = 0;
    MPI_Type_create_subarray(3, array_of_sizes, array_of_subsizes,
array_of_starts, MPI_ORDER_C, MPI_DOUBLE, send_to_up);

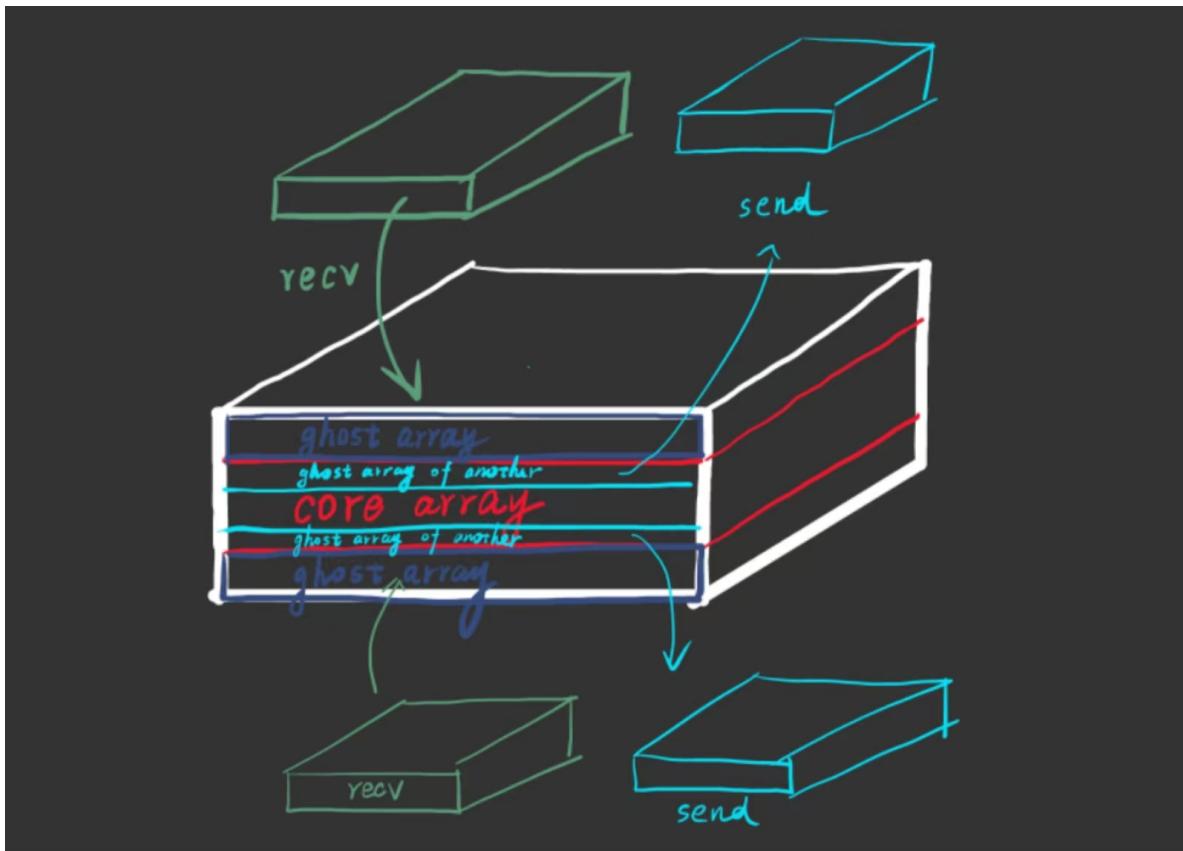
    array_of_starts[0] = lz;
    array_of_starts[1] = 0;
    array_of_starts[2] = 0;
    MPI_Type_create_subarray(3, array_of_sizes, array_of_subsizes,
array_of_starts, MPI_ORDER_C, MPI_DOUBLE, send_to_down);
```

```

array_of_starts[0] = hz + lz;
array_of_starts[1] = 0;
array_of_starts[2] = 0;
MPI_Type_create_subarray(3, array_of_sizes, array_of_subsizes,
array_of_starts, MPI_ORDER_C, MPI_DOUBLE, recv_from_down);
}

```

数组之间的通信原理如图所示，每一个local\_array都可以分为 core\_array 部分（这部分会发生更改）和 ghost\_array 部分（这部分需要依赖进程通信来获取数据用于读取，但不作更新）；同时，自身 core\_array 的外层部分也会作为其他local\_array的 ghost\_array 部分，需要进行发送）。



具体的解释如下：在划分时，在每一个local\_array中被划分的方向上，总维度数为 `local_size + 2 * halo_size`。第0维作为receive buffer接收上一个local\_array的值；第 `halo_size` 维作为send buffer发送给上一个local\_array；第 `local_size` 维作为send buffer发送给下一个local\_array；第 `local_size + halo_size` 维作为receive buffer接收下一个local\_array的值。

代码实现如下：

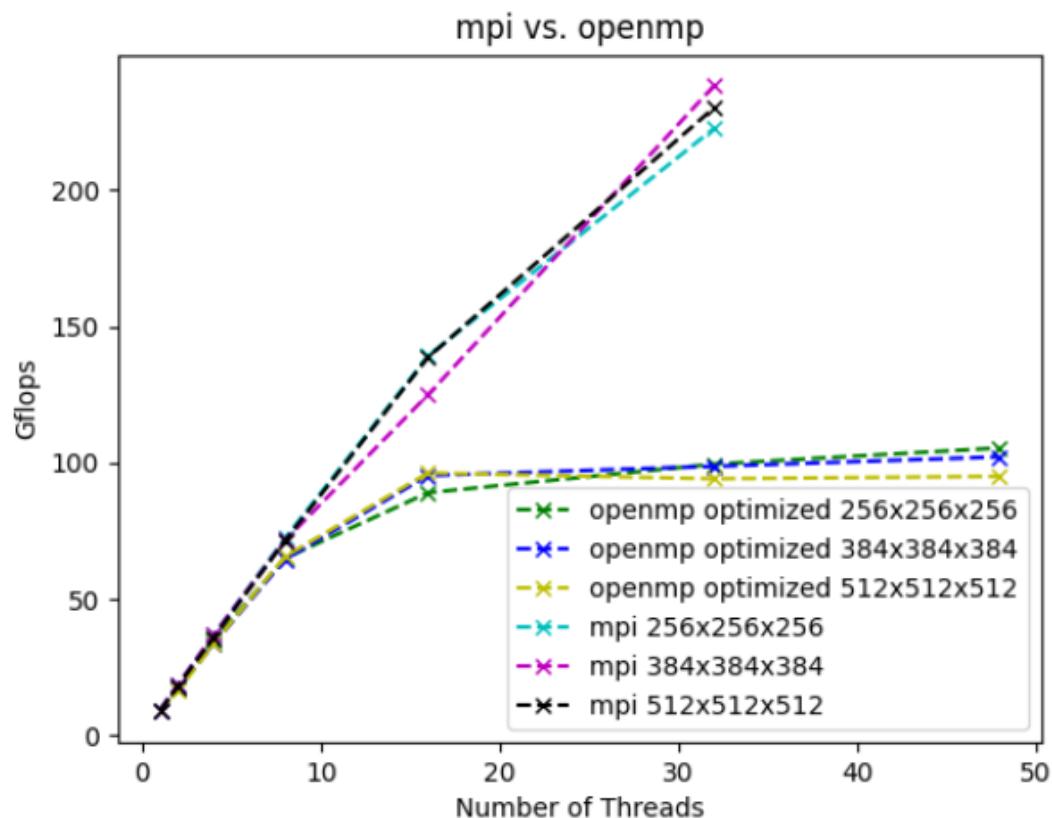
```

MPI_Sendrecv(&a0[INDEX(0, 0, grid_info->local_size_z, ldx, ldy)], ldx * ldy,
MPI_DOUBLE, rank_id < rank_num - 1 ? rank_id + 1 : MPI_PROC_NULL, 0, \
&a0[INDEX(0, 0, 0, ldx, ldy)], ldx * ldy, MPI_DOUBLE, rank_id > 0 \
? rank_id - 1 : MPI_PROC_NULL, 0, MPI_COMM_WORLD, &status);
MPI_Sendrecv(&a0[INDEX(0, 0, grid_info->halo_size_z, ldx, ldy)], ldx * ldy,
MPI_DOUBLE, rank_id > 0 ? rank_id - 1 : MPI_PROC_NULL, 1, \
&a0[INDEX(0, 0, grid_info->halo_size_z + grid_info->local_size_z, \
ldx, ldy)], ldx * ldy, MPI_DOUBLE, rank_id < rank_num - 1 ? rank_id + 1 : \
MPI_PROC_NULL, 1, MPI_COMM_WORLD, &status);

```

注意这里用 `MPI_Sendrecv` 将send逻辑和recv逻辑进行合并，同时使用 `MPI_PROC_NULL` 处理通信的边界条件。

最终结果如图所示：



可以看到，随着线程数的增加，使用MPI进行并行计算的速度增长接近线性。

## 特别说明

为防止在不同线程内打印多次调试信息，特对主代码进行以下更改：

```
+ if (p_id == 0)
+ {
    printf("errors:\n      1-norm = %.16lf\n      2-norm = %.16lf\n      inf-norm =
%.16lf\n",
           result.norm_1, result.norm_2, result.norm_inf);
    if (result.norm_inf > 1e-9)
    {
        printf("Significant numeric error.\n");
    }
+ }
```

## 参考文献

1. [StencilProbe: A Microbenchmark for Stencil Applications](#)
2. [adept-kernel-openmp](#)
3. [Message Passing Interface \(MPI\)](#)