PA2 report - stencil 7

Optimization with OpenMP

source code

```
ptr_t stencil_7(ptr_t grid, ptr_t aux, const dist_grid_info_t *grid_info, int nt) {
  ptr_t buffer[2] = {grid, aux};
  int x_start = grid_info->halo_size_x, x_end = grid_info->local_size_x + grid_info->halo_size_x;
  int y_start = grid_info->halo_size_y, y_end = grid_info->local_size_y + grid_info->halo_size_y;
  int z_start = grid_info->halo_size_z, z_end = grid_info->local_size_z + grid_info->halo_size_z;
  int ldx = grid_info->local_size_x + 2 * grid_info->halo_size_x;
  int ldy = grid_info->local_size_y + 2 * grid_info->halo_size_y;
  int ldz = grid_info->local_size_z + 2 * grid_info->halo_size_z;
  int ldxy = ldx * ldy;

  // 针对较大的数据点，采用三维都 timeskew 的方法，这样 bx*by*bz 较小，可以减小 L3 cache miss 的概率
  if (z_end - z_start >= 768) {
    int x_blocksz = 288;
    int y_blocksz = 16;
    int z_blocksz = 16 * 28;

    int x_upper = x_start + (x_end - x_start + nt + x_blocksz - 1) / x_blocksz * x_blocksz;
    int y_upper = y_start + (y_end - y_start + nt + y_blocksz - 1) / y_blocksz * y_blocksz;
    int z_upper = z_start + (z_end - z_start + nt + z_blocksz - 1) / z_blocksz * z_blocksz;
    if (x_blocksz == x_end - x_start)
      x_upper = x_end;
    if (z_blocksz == z_end - z_start)
      z_upper = z_end;

    for (int xb = x_start; xb < x_upper; xb += x_blocksz)
      for (int yb = y_start; yb < y_upper; yb += y_blocksz)
        for (int zb = z_start; zb < z_upper; zb += z_blocksz) {
          // 枚举当前处于哪个块中
                    for (int t = 0; t < nt; ++ t) {
                      int x_low  = min(x_end, max(x_start, xb - t));
                      int x_high = min(x_end, max(x_start, xb + x_blocksz - t));
                      if (x_blocksz == x_end - x_start)
                        x_high = x_end;
                      int y_low  = min(y_end, max(y_start, yb - t));
                      int y_high = min(y_end, max(y_start, yb + y_blocksz - t));
                      int z_low  = min(z_end, max(z_start, zb - t));
                      int z_high = min(z_end, max(z_start, zb + z_blocksz - t));
                      if (z_blocksz == z_end - z_start)
                        z_high = z_end;

                      if (x_low >= x_high || y_low >= y_high) continue;
                      cptr_t a0 = buffer[t % 2];
                      ptr_t  a1 = buffer[(t + 1) % 2];
                      // 处理一块中的数据
#pragma omp parallel for num_threads(28) proc_bind(close)
                      for (int z = z_low; z < z_high; z ++)
                        for (int y = y_low; y < y_high; y ++) {
                          cptr_t g0 = a0 + INDEX(x_low, y, z, ldx, ldy);
                          ptr_t  g1 = a1 + INDEX(x_low, y, z, ldx, ldy);
#pragma omp simd
                          for (int x = x_low; x < x_high; x ++) {
                            *g1 = ALPHA_ZZZ * *g0
                              + ALPHA_NZZ * *(g0 - 1)
                              + ALPHA_PZZ * *(g0 + 1)
                              + ALPHA_ZNZ * *(g0 - ldx)
```

```
                              + ALPHA_ZPZ * *(g0 + ldx)
                              + ALPHA_ZZN * *(g0 - ldxy)
                              + ALPHA_ZZP * *(g0 + ldxy);
                            g1 ++;
                            g0 ++;
                          }
                        }
                    }
        }
  } else {
    // 对于较小的数据，只需要沿着 y 轴进行 timeskew 即可
    int y_blocksz = 24;
    if (z_end - z_start >= 512) y_blocksz = 10;
    if (z_end - z_start >= 768) y_blocksz = 4;
    int upper = y_start + (y_end - y_start + nt + y_blocksz - 1) / y_blocksz * y_blocksz;
    for (int yb = y_start; yb < upper; yb += y_blocksz) {
      for (int t = 0; t < nt; ++ t) {
        cptr_t a0 = buffer[t % 2];
        ptr_t a1 = buffer[(t + 1) % 2];
        int y_low  = min(y_end, max(y_start, yb - t));
        int y_high = min(y_end, max(y_start, yb + y_blocksz - t));
        if (y_low >= y_high) continue;
#pragma omp parallel for num_threads(28) proc_bind(close) schedule(static)
        for (int z = z_start; z < z_end; z ++)
          for (int y = y_low; y < y_high; y ++) {
            cptr_t g0 = a0 + INDEX(x_start, y, z, ldx, ldy);
            ptr_t  g1 = a1 + INDEX(x_start, y, z, ldx, ldy);
            for (int x = x_start; x < x_end; x ++) {
              *g1 = ALPHA_ZZZ * *g0
                + ALPHA_NZZ * *(g0 - 1)
                + ALPHA_PZZ * *(g0 + 1)
                + ALPHA_ZNZ * *(g0 - ldx)
                + ALPHA_ZPZ * *(g0 + ldx)
                + ALPHA_ZZN * *(g0 - ldxy)
                + ALPHA_ZZP * *(g0 + ldxy);
              g1 ++;
              g0 ++;
            }
          }
      }
    }
  }
  return buffer[nt % 2];
}
```

优化方法

1. 参考 `stencilprobe`，使用 timeskew 优化，在一块较小的内存上接连着进行多次运算，保证了每次访存都能在 L1 或者 L2 cache 中找到，大大优化了访存效率，这一优化可以在 16 steps 下将效率提升到 `80 Gflops`。
2. 参阅了 `numactl` 的手册后，我添加了 `numactl --interleave=all` 的指令。由于 openMP 中线程很可能需要访问另外一个节点上的数据，而 `interleave` 使用 `round robin` 的策略将数据均匀分布在各个节点上，因此能够使每个线程的访存开销大致相同，由于需要强制 OpenMP 采用 static 的任务分配策略（保证每个线程处理的数据是上一轮处理过的），所以这一改动提高了运行效率、减小了运行时间的波动。加上这一优化可以在 16 steps 下将效率提升到 `101 Gflops`。

Performance

testing with $256 \times 256 \times 256 (step\ 100)$

N threads	time consumption	speedup
1	2.596925	1.000

N threads	time consumption	speedup
2	1.346224	1.929
4	0.694499	3.739
7	0.416366	6.237
14	0.241487	10.753
28	0.149265	17.398

Optimization with MPI

```
ptr_t stencil_7(ptr_t grid, ptr_t aux, const dist_grid_info_t *grid_info, int nt) {
  ptr_t buffer[] = { grid, aux };
  int x_start = grid_info->halo_size_x, x_end = grid_info->local_size_x + grid_info->halo_size_x;
  int y_start = grid_info->halo_size_y, y_end = grid_info->local_size_y + grid_info->halo_size_y;
  int z_start = grid_info->halo_size_z, z_end = grid_info->local_size_z + grid_info->halo_size_z;
  const int ldx = grid_info->local_size_x + 2 * grid_info->halo_size_x;
  const int ldy = grid_info->local_size_y + 2 * grid_info->halo_size_y;
  const int ldxy = ldx * ldy;
  const int pid = grid_info->p_id;
  if (x_start == x_end || y_start == y_end || z_start == z_end) return grid;
  const int havepred = (grid_info->offset_z > 0);
  const int haverear = (grid_info->offset_z + grid_info->local_size_z < grid_info->global_size_z);

  for (int t = 0; t < nt; t += TIME_BLOCK) {
    int tlen = min(TIME_BLOCK, nt - t);
    MPI_Request req[6];
    int nreq = 0;
    ptr_t a0 = buffer[t % 2];
    if (havepred) {
      // communicate with p_id - 1
      MPI_Isend(a0 + INDEX(0, 0, z_start, ldx, ldy), ldxy * TIME_BLOCK, MPI_DOUBLE, pid - 1, 1, MPI_COMM_WORLD
      MPI_Irecv(a0,                                  ldxy * TIME_BLOCK, MPI_DOUBLE, pid - 1, 1, MPI_COMM_WORLD
    }
    if (haverear) {
      // communicate with p_id + 1
      MPI_Isend(a0 + INDEX(0, 0, z_end - TIME_BLOCK, ldx, ldy), ldxy * TIME_BLOCK, MPI_DOUBLE, pid + 1, 1, MPI
      MPI_Irecv(a0 + INDEX(0, 0, z_end,              ldx, ldy), ldxy * TIME_BLOCK, MPI_DOUBLE, pid + 1, 1, MPI
    }
    int ez = 1 + (z_end - z_start + 2 * TIME_BLOCK + TIME_BLOCK + Z_BLOCK - 1) / Z_BLOCK * Z_BLOCK;
    int ey = y_start + (y_end - y_start + TIME_BLOCK + Y_BLOCK - 1) / Y_BLOCK * Y_BLOCK;
    MPI_Waitall(nreq, req, MPI_STATUSES_IGNORE);
    // 强制在计算前完成全部传输，每次传输足够多的层数，使通信效率达到较高水平

    for (int zz = 1; zz < ez; zz += Z_BLOCK)
      for (int yy = y_start; yy < ey; yy += Y_BLOCK) {
        // timeskew
        for (int tt = 0; tt < tlen; tt ++) {
          cptr_t a0 = buffer[(t + tt) % 2];
          ptr_t  a1 = buffer[(t + tt + 1) % 2];
          int zb = min(z_end + TIME_BLOCK - 1, max(1, zz - tt));
          int ze = min(z_end + TIME_BLOCK - 1, max(1, zz + Z_BLOCK - tt));
          int yb = min(y_end, max(y_start, yy - tt));
          int ye = min(y_end, max(y_start, yy + Y_BLOCK - tt));
          // t-z 图形呈现为梯形，此处避免部分不需要的计算
          if (!havepred) zb = max(zb, z_start);
          else           zb = max(zb, 1 + tt);
          if (!haverear) ze = min(ze, z_end);
          else           ze = min(ze, z_end + TIME_BLOCK - tt - 1);
          if (zb >= ze || yb >= ye) continue;
```

```
        for (int z = zb; z < ze; z ++)
          for (int y = yb; y < ye; y ++) {
            int x = x_start;
            cptr_t g0 = a0 + INDEX(x, y, z, ldx, ldy);
            ptr_t  g1 = a1 + INDEX(x, y, z, ldx, ldy);
            for (; x < x_end; x ++) {
              *g1 = ALPHA_ZZZ * *g0
                + ALPHA_NZZ * *(g0 - 1)
                + ALPHA_PZZ * *(g0 + 1)
                + ALPHA_ZNZ * *(g0 - ldx)
                + ALPHA_ZPZ * *(g0 + ldx)
                + ALPHA_ZZN * *(g0 - ldxy)
                + ALPHA_ZZP * *(g0 + ldxy);
              g1 ++;
              g0 ++;
            }
          }
      }
    }
  }
  return buffer[nt % 2];
}
```

Performance

testing with $256 \times 256 \times 256 (step 100)$

N threads	time consumption	speedup
1	2.456174	1.000
2	1.314131	1.869
4	0.793889	3.271
7	0.539412	4.814
14	0.363937	7.135
28	0.308678	8.143

一点分析和总结

- 对于 MPI 版本，如果相邻进程单次交换的数据较少，则同步的代价更高，且带宽更小，并且无法发挥好 timeskew 的访存优化效果；
  但如果交换的数据较多，则意味着额外的工作量较大（因为这些接收到的数据也要迭代计算），这一问题在数据规模较小的情况下更加显著，因为交换的数据相对于节点应当处理的数据的比例更大。这制约了 MPI 版本的最终效果。
- 通过一系列调参，OpenMP 版本在 $(512 \times 512 \times 512, 100 steps)$ 的数据上达到了 `170Gflops`，而 MPI 在 $(512 \times 512 \times 512, 100 steps)$ 上达到了 `87Gflops`。