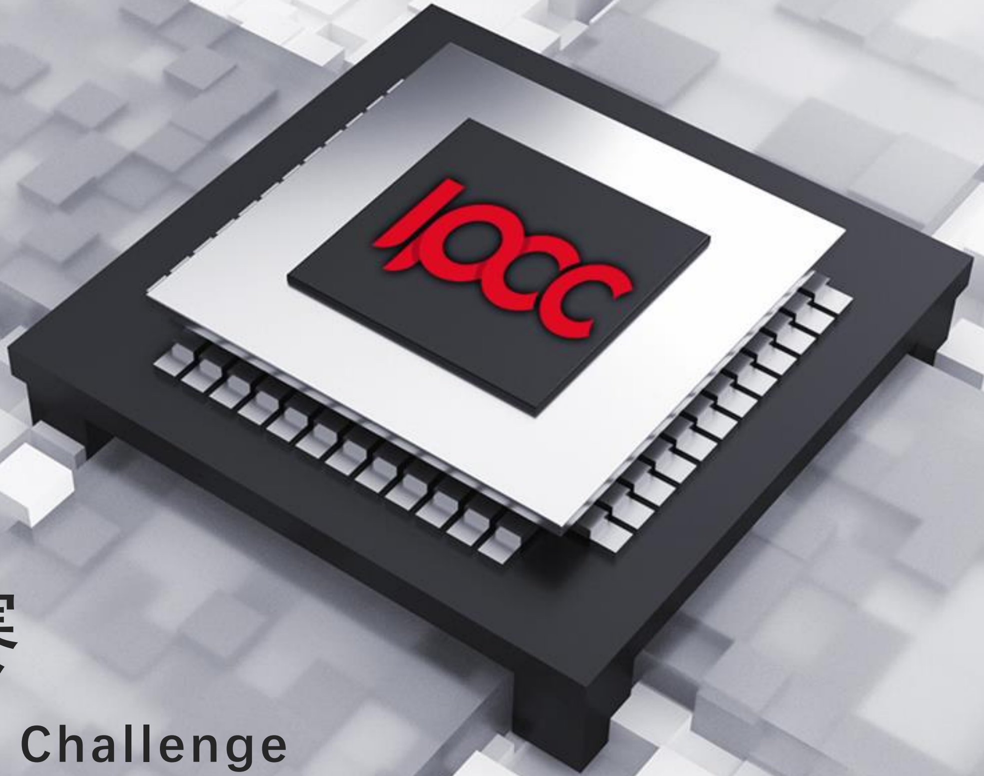
ACM中国—国际并行计算挑战赛

ACM-China International Parallel Computing Challenge



目录 CONTENTS

参赛队简介

参赛队伍编号：IPCC20216734 参赛队伍名称：Result0

参赛队伍学校：武汉大学

指导老师：邓娟 武汉大学计算机学院 副教授

参赛队员：吴智琨 陈洲

应用程序运行的硬件环境和软件环境——硬件信息

Association for Computing Machinery
ACM中国 国际并行计算挑战赛

CPU	Intel 10875H	AMD EPYC 7452	Intel Xeon Glod 8180
Core(s) per socket	8	32	28
Thread(s) per core	2	1	2
Sockets (numa)	1	2	2
Frequency	2.3 ~ 5.1(4.3 all) GHz	2.35 ~ 3.35 GHz	2.5 ~ 3,8 GHz
L1d/L1i cache	256KB/256KB	32KB/32KB	32KB/32KB
L2 cache	2MB	512KB	1MB
L3 cache	16MB	16MB	38MB
AVX2-GFLOPS	< 700	2406.4*2	2240*2
stream	24.1GB/s	244.9GB/s	137.4GB/s
Max bandwidth	45.8 GB/s	400 GB/s	250 GB/s
	开发/编译平台	运行平台	参考参数

开发/编译平台：INTEL i7-10875H
信息简述：8核16线程，开发及初步调优使用

运行平台：AMD EPYC 7452
信息简述：NUMA架构，32核x2，无超线程，支持fma, avx2，比赛实际运行平台

应用程序运行的硬件环境和软件环境——软件信息

```
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal
```

```
LSB Version:    :core-4.1-amd64:core-4.1-noarch:cxx-4.1-amd64:cxx-4.1-noarch:desktop-4.1-amd64:desktop-4.1-noarch:languages-4.1-amd64:languages-4.1-noarch:printing-4.1-amd64:printing-4.1-noarch
Distributor ID: CentOS
Description:    CentOS Linux release 7.9.2009 (Core)
Release:        7.9.2009
Codename:       Core
```

开发/编译平台:
系统：WSL2 Ubuntu 20.04 LTS
Gcc： 9.3.0
Glibc：2.31
ICC: 2021.3.0
Open MPI: 4.0.3

运行平台:
系统：CentOS Linux 7.9
Gcc： 4.8.5
Glibc：2.17
ICC: 19.1.3.304
Open MPI: 3.1.6

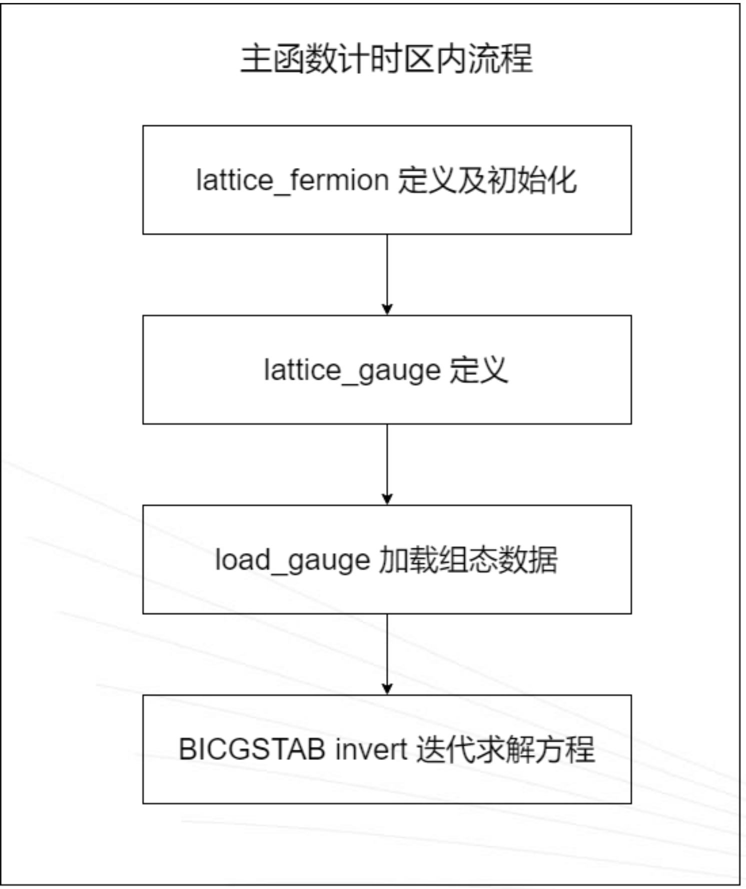
应用程序的代码结构

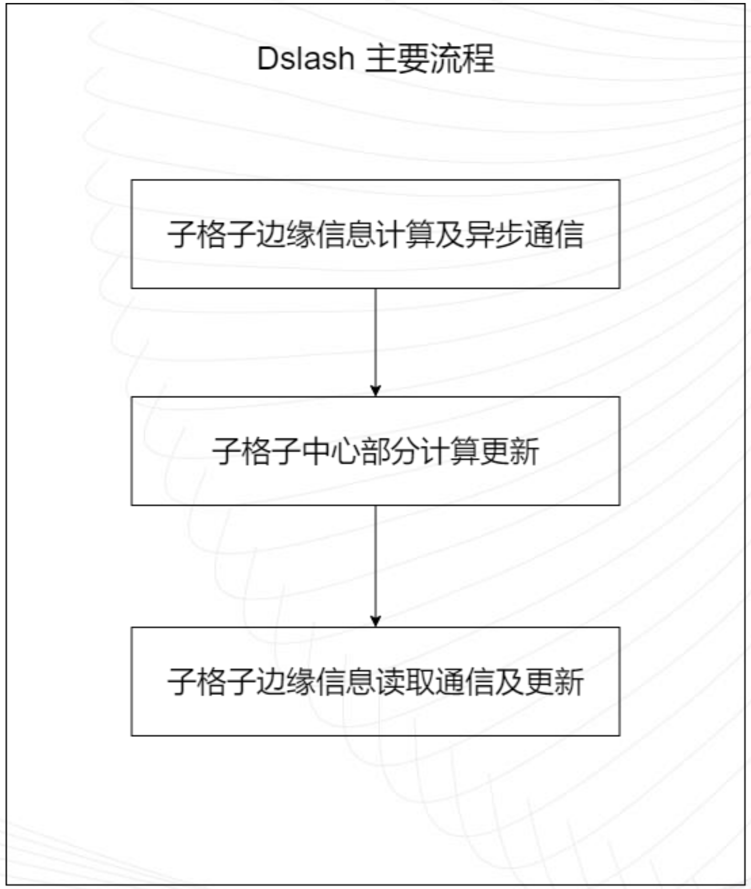
src/	主要源码目录
check.cpp	校验函数文件
dslash.cpp	Dslash 算法主要实现
invert.cpp	迭代求解主要实现
lattice_fermion.cpp	费米子类实现
lattice_gauge.cpp	组态类实现
load_gauge.cpp	加载组态函数文件
main.cpp	主函数文件
Makefile	编译脚本
sub.sh	用于 sbatch 提交

include/	主要源码目录
check.	校验函数头文件
dslash.h	Dslash 算法头文件
invert.h	迭代求解头文件
lattice_fermion.h	费米子类定义
lattice_gauge.h	组态类定义
load_gauge.h	加载组态函数头文件
operator.h	类操作符重载头文件
operator_mpi.h	类操作符重载头文件
utils.h	辅助内容头文件

data/	数据目录（不在提交中）
ipcc_gauge_24_72	24x24x24x72 的组态数据
ipcc_gauge_32_64	32x32x32x64 的组态数据
ipcc_gauge_48_96	48x48x48x96 的组态数据

应用程序的代码结构





该应用解决的核心问题是求解 Mx=b

其中，x 和 b 为 fermion 向量，

M 由于稀疏不直接存储，而是存储

组态数据 Gauge U

为求解该大规模稀疏线性方程组，

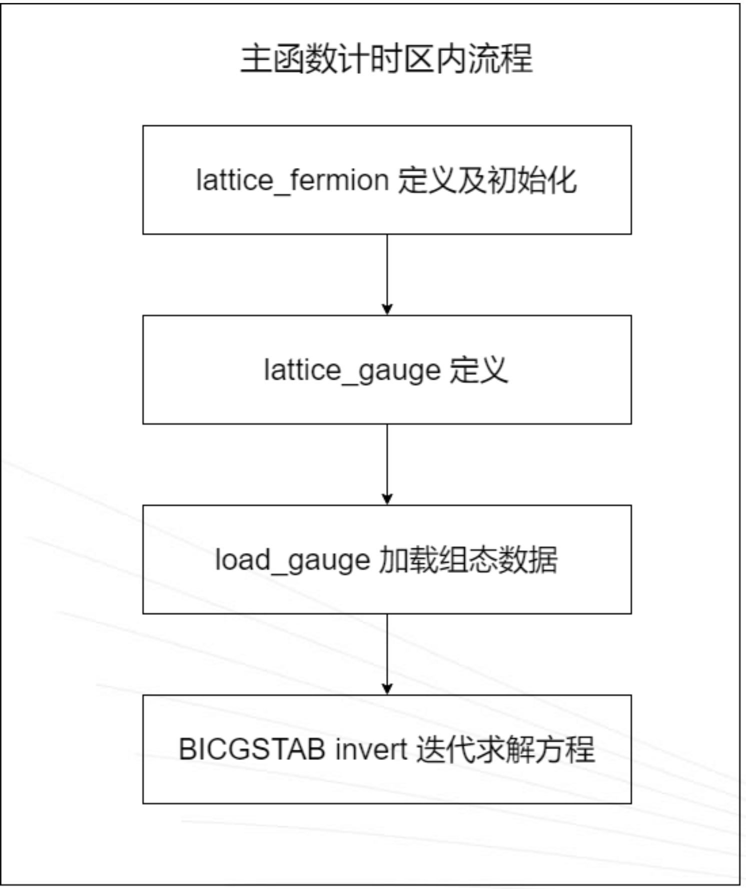
使用共轭梯度法及其改进，即

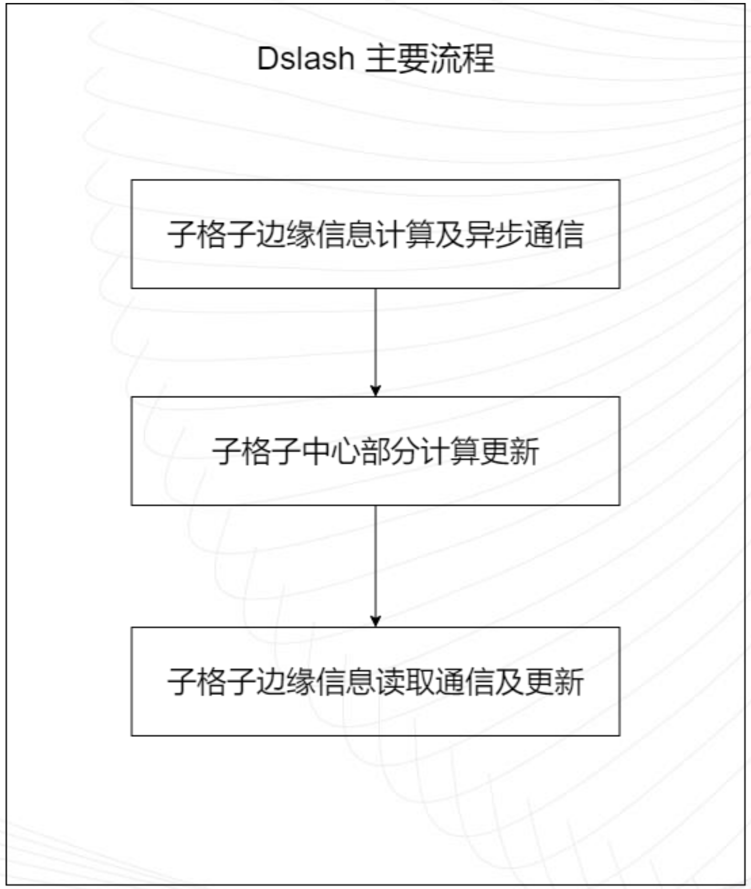
CGinvert 来实现，其中，

利用 U 计算 Mx 的方法为 Dslash，

迭代求解直至满足精度要求退出

优化方法

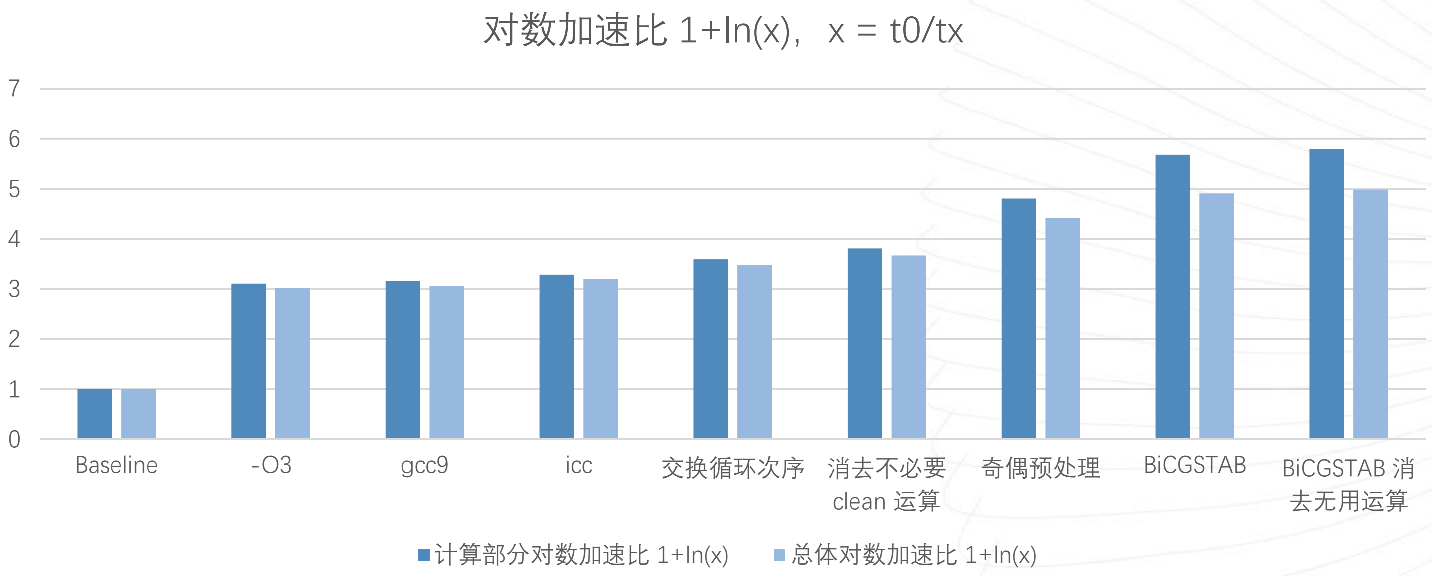


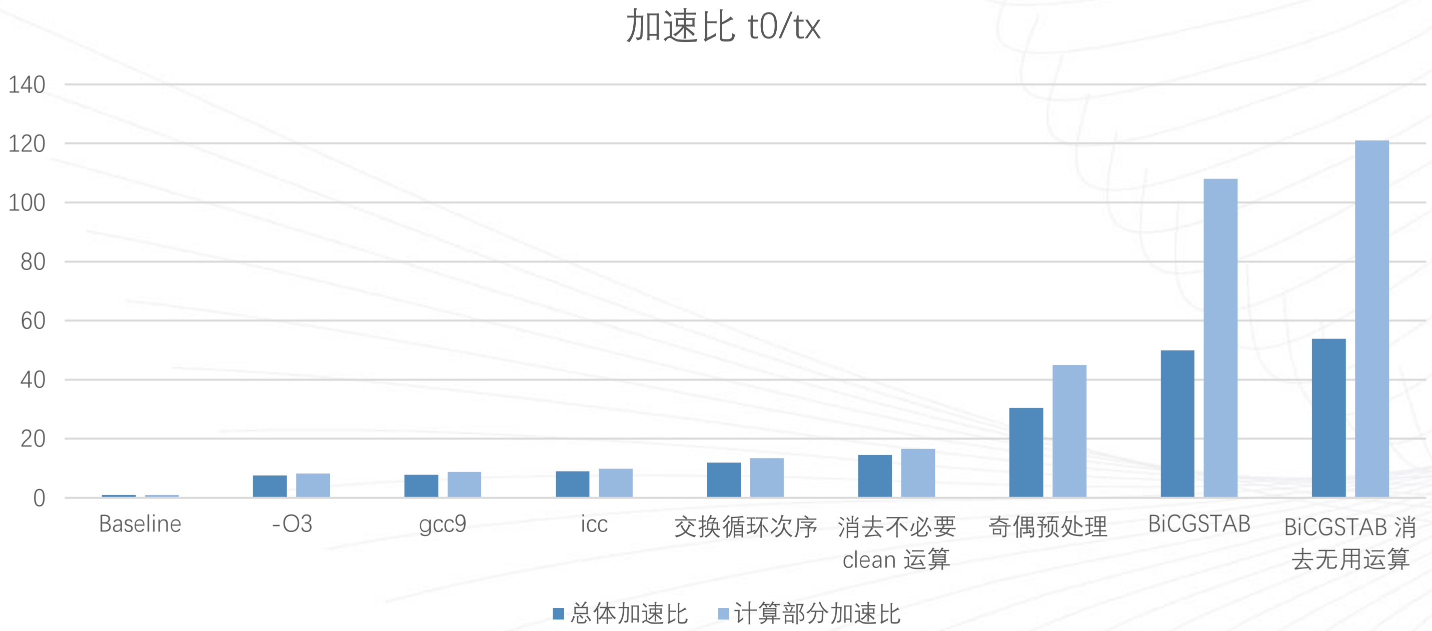


1. Dslash 初步简化，清晰化及模块化
2. Dslash 访存优化
3. 奇偶预处理实现及优化
4. 双稳定共轭梯度法实现

预分析：MPI 异步通信已经充分利用延迟，核心数学运算部分编译器已有良好的向量化操作不作为本次优化重点。

效果预览





1. Dslash 初步简化，清晰化及模块化
2. Dslash 访存优化
3. 奇偶预处理实现及优化
4. 双稳定共轭梯度法实现

取决赛赛题文档和源文件 README 默认的编译运行方式为 Baseline，逐次优化，效果如左图。由于加载组态文件部分依赖于存储速度，最后优化结果计算时间小于读取文件时间，故总体时间不能真实反映优化效果，单独列出计算部分加速比。

优化方法　Dslash 初步简化，清晰化及模块化


```
int z_u = (N_sub[2] == 1) ? subgrid[2] : subgrid[2] - 1;
for (int x = 0; x < subgrid[0]; x++) {
    for (int y = 0; y < subgrid[1]; y++) {
        for (int z = 0; z < z_u; z++) {
            for (int t = 0; t < subgrid[3]; t++) {
                int f_z = (z + 1) % subgrid[2];
                complex<double> tmp;
                complex<double> *srcO =
                    src.A +
                    (subgrid[0] * subgrid[1] * subgrid[2] * t + subgr
                     subgrid[0] * y + x + (1 - cb) * subgrid_vol_cb) *
                        12;
                complex<double> *destE = dest.A + (subgrid[0] * subgr
                                                   subgrid[0] * subgr
                                                   subgrid[0] * y + x
                                                      12;
                complex<double> *AE = U.A[2] + (subgrid[0] * subgrid[
                                                subgrid[0] * subgrid[
                                                x + cb * subgrid_vol_
                                                   9;
                for (int c1 = 0; c1 < 3; c1++) {
                    for (int c2 = 0; c2 < 3; c2++) {
                        tmp = -(srcO[0 * 3 + c2] - flag * I * srcO[2 * 3 + c2]) * half *
                              AE[c1 * 3 + c2];
                        destE[0 * 3 + c1] += tmp;
                        destE[2 * 3 + c1] += flag * (I * tmp);
                        tmp = -(srcO[1 * 3 + c2] + flag * I * srcO[3 * 3 + c2]) * half *
                              AE[c1 * 3 + c2];
                        destE[1 * 3 + c1] += tmp;
                        destE[3 * 3 + c1] += flag * (-I * tmp);
                    }
                }
            }
        }
    }
}
```

```
for (int z = 1; z < subgrid[2] - 1; z++) {
    // int tzoffset = toffset + subgrid[0] * subgrid[1] * z;
    for (int y = 0; y < subgrid[1]; y++) {
        int tyoffset = toffset + subgrid[0] * y;
        for (int x = 0; x < subgrid[0]; x++) {
            int tyx_1_cb_offset = tyoffset + x + (1 - cb) * subgrid_vol_cb;
            int tyx_0_cb_offset = tyoffset + x + cb * subgrid_vol_cb;
            int f_z = (z + 1) % subgrid[2];
            int b_z = (z - 1 + subgrid[2]) % subgrid[2];
            complex<double> tmp;
            complex<double> *srcO_f_z = src.A + (tyx_1_cb_offset + subgrid[0] * subgrid[1] * f_z) * 12;
            complex<double> *srcO_b_z = src.A + (tyx_1_cb_offset + subgrid[0] * subgrid[1] * b_z) * 12;
            complex<double> *destE = dest.A + (tyx_0_cb_offset + subgrid[0] * subgrid[1] * z) * 12;
            complex<double> *AE_f_z = U.A[2] + (tyx_0_cb_offset + subgrid[0] * subgrid[1] * z) * 9;
            complex<double> *AO_b_z = U.A[2] + (tyx_1_cb_offset + subgrid[0] * subgrid[1] * b_z) * 9;
            cal_z_f(srcO_f_z, AE_f_z, destE, flag, I);
            cal_z_b(srcO_b_z, AO_b_z, destE, flag, I);
        }
    }
}
```

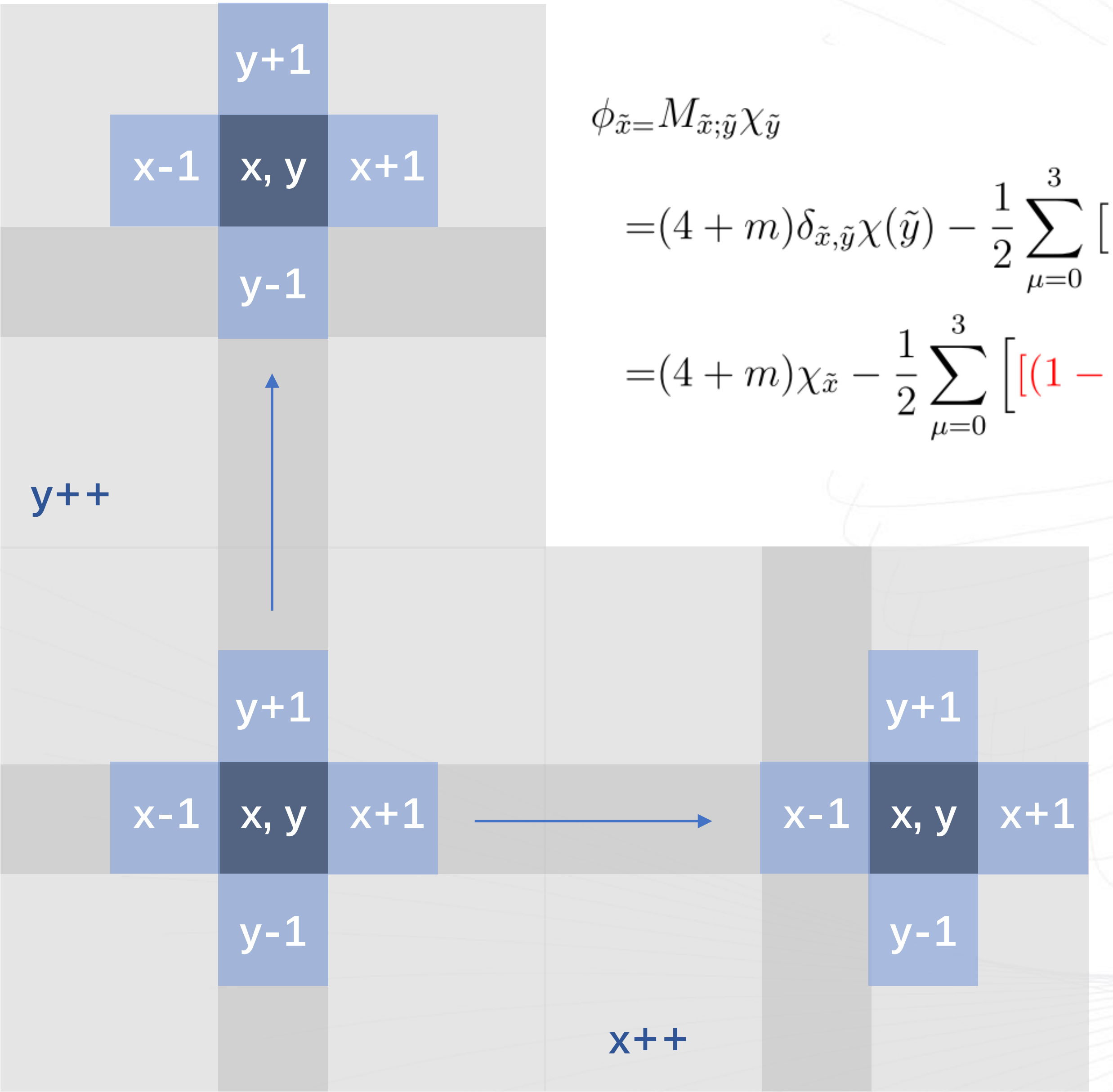
原函数代码较为繁琐，且存在一定的重复计算，多层 for 循环不利于观察。

通过抽取计算共同部分，抽离 for 循环，使得修改后 Dslash 行数大幅减少，代码清晰，模块形式易于后续修改。

同时，合并更新同一维度的两个循环。

注：本部分基本不优化实际执行效率。

Dslash 访存优化



$$\phi_{\tilde{x}} = M_{\tilde{x};\tilde{y}}\chi_{\tilde{y}}$$
$$= (4+m)\delta_{\tilde{x},\tilde{y}}\chi(\tilde{y}) - \frac{1}{2}\sum_{\mu=0}^{3}\left[[(1-\gamma_\mu)\otimes U_\mu(\tilde{x})]\delta_{\tilde{x}+\mu,\tilde{y}} + [(1+\gamma_\mu)\otimes U_\mu^\dagger(\tilde{x}-\mu)]\delta_{\tilde{x}-\mu,\tilde{y}}\right]\underline{\chi(\tilde{y})}$$
$$= (4+m)\chi_{\tilde{x}} - \frac{1}{2}\sum_{\mu=0}^{3}\left[[(1-\gamma_\mu)\otimes U_\mu(\tilde{x})]\underline{\chi(\tilde{x}+\mu)} + [(1+\gamma_\mu)\otimes U_\mu^\dagger(\tilde{x}-\mu)]\underline{\chi(\tilde{x}-\mu)}\right]$$

由公式可知，更新 (x, y, z, t) 1个向量需要访问自身和四个维度共计八个方向上的相邻向量，这时候典型的 Stencil 运算，鉴于源码实现时拆开了各个维度，适当选取遍历次序对访存效率至关重要。

以二维空间为例，更新一点需要周围四点，沿 x 方向更新时毫无疑问将 x 置于 for 循环最内层最高效，但沿 y 方向更新时则存在两种选择：沿 x 方向更新能换得循环间元素相邻，访存连续；沿 y 方向更新则能重复访问相同元素，各有利弊。

Dslash 访存优化

```
int z_u = (N_sub[2] == 1) ? subgrid[2] : subgrid[2] - 1;
for (int x = 0; x < subgrid[0]; x++) {
    for (int y = 0; y < subgrid[1]; y++) {
        for (int z = 0; z < z_u; z++) {
            for (int t = 0; t < subgrid[3]; t++) {
                int f_z = (z + 1) % subgrid[2];
                complex<double> tmp;
                complex<double> *srcO =
                    src.A +
                    (subgrid[0] * subgrid[1] * subgrid[2] * t + subgr
                     subgrid[0] * y + x + (1 - cb) * subgrid_vol_cb)
                        12;
                complex<double> *destE = dest.A + (subgrid[0] * subgr
                                                   subgrid[0] * subgr
                                                   subgrid[0] * y + x
                                                       12;
                complex<double> *AE = U.A[2] + (subgrid[0] * subgrid[
                                                subgrid[0] * subgrid[
                                                x + cb * subgrid_vol_
                                                    9;
                for (int c1 = 0; c1 < 3; c1++) {
                    for (int c2 = 0; c2 < 3; c2++) {
                        tmp = -(srcO[0 * 3 + c2] - flag * I * srcO[2 * 3 + c2]) * half *
                              AE[c1 * 3 + c2];
                        destE[0 * 3 + c1] += tmp;
                        destE[2 * 3 + c1] += flag * (I * tmp);
                        tmp = -(srcO[1 * 3 + c2] + flag * I * srcO[3 * 3 + c2]) * half *
                              AE[c1 * 3 + c2];
                        destE[1 * 3 + c1] += tmp;
                        destE[3 * 3 + c1] += flag * (-I * tmp);
                    }
                }
            }
        }
    }
}
```

```
for (int z = 1; z < subgrid[2] - 1; z++) {
    // int tzoffset = toffset + subgrid[0] * subgrid[1] * z;
    for (int y = 0; y < subgrid[1]; y++) {
        int tyoffset = toffset + subgrid[0] * y;
        for (int x = 0; x < subgrid[0]; x++) {
            int tyx_1_cb_offset = tyoffset + x + (1 - cb) * subgrid_vol_cb;
            int tyx_0_cb_offset = tyoffset + x + cb * subgrid_vol_cb;
            int f_z = (z + 1) % subgrid[2];
            int b_z = (z - 1 + subgrid[2]) % subgrid[2];
            complex<double> tmp;
            complex<double> *srcO_f_z = src.A + (tyx_1_cb_offset + subgrid[0] * subgrid[1] * f_z) * 12;
            complex<double> *srcO_b_z = src.A + (tyx_1_cb_offset + subgrid[0] * subgrid[1] * b_z) * 12;
            complex<double> *destE = dest.A + (tyx_0_cb_offset + subgrid[0] * subgrid[1] * z) * 12;
            complex<double> *AE_f_z = U.A[2] + (tyx_0_cb_offset + subgrid[0] * subgrid[1] * z) * 9;
            complex<double> *AO_b_z = U.A[2] + (tyx_1_cb_offset + subgrid[0] * subgrid[1] * b_z) * 9;
            cal_z_f(srcO_f_z, AE_f_z, destE, flag, I);
            cal_z_b(srcO_b_z, AO_b_z, destE, flag, I);
        }
    }
}
```

原代码中除了沿 X 轴更新部分，其余方向上的遍历次序均为 X Y Z T，而数组的存储形式均为 T Z Y X，访存局部性差。

对此，存在前述两种可能的优化方式，总体上均按照 T Z Y X 的形式访问，经检验得沿 x 方向遍历效率较高，故选用。

奇偶预处理

设目标函数的条件数为 $\kappa(G) = ||G||_2||G^{-1}||_2$ ，则共轭梯度法产生的点列误差估计为:

$$\frac{||x_{k+1} - x^*||_G}{||x_k - x^*||_G} \le \frac{\sqrt{\kappa(G)} - 1}{\sqrt{\kappa(G)} + 1}$$

所以可以看出来，共轭梯度法是线性收敛的，而且与梯度下降法相比有更快的收敛速率（与梯度下降法比起来，共轭梯度的收敛速率就是把 κ 替换成了 $\sqrt{\kappa}$ ）。通过改善 G 的条件数，可以加快收敛速度。

为加速共轭梯度法的收敛，我们选取补充文档中描述的奇偶预处理方法分解矩阵，改善了矩阵的条件数，加速了收敛，同时还利用了对角信息，简化了计算，以下为使用的具体方法的推导。

$$\begin{aligned}
M\vec{x} &= \vec{b} \\
M &= \begin{bmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{bmatrix} \\
&= \begin{bmatrix} I & 0 \\ M_{oe}M_{ee}^{-1} & I \end{bmatrix} \begin{bmatrix} M_{ee} & 0 \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{bmatrix} \begin{bmatrix} I & M_{ee}^{-1}M_{eo} \\ 0 & I \end{bmatrix} \\
&= \begin{bmatrix} I & 0 \\ \frac{1}{a+mass}M_{oe} & I \end{bmatrix} \begin{bmatrix} M_{ee} & 0 \\ 0 & (a+mass)I - \frac{1}{a+mass}M_{oe}M_{eo} \end{bmatrix} \begin{bmatrix} I & \frac{1}{a+mass}M_{eo} \\ 0 & I \end{bmatrix} \\
&= L\tilde{M}U
\end{aligned}$$

$$L^{-1} = \begin{bmatrix} I & 0 \\ -M_{oe}M_{ee}^{-1} & I \end{bmatrix} = \begin{bmatrix} I & 0 \\ -\frac{1}{a+mass}M_{oe} & I \end{bmatrix}, \quad U^{-1} = \begin{bmatrix} I & -M_{ee}^{-1}M_{eo} \\ 0 & I \end{bmatrix} = \begin{bmatrix} I & \frac{1}{a+mass}M_{eo} \\ 0 & I \end{bmatrix}$$

$$\begin{aligned}
M\vec{x} &= \vec{b} \\
L\tilde{M}U\vec{x} &= \vec{b} \\
\tilde{M}(U\vec{x}) &= L^{-1}\vec{b} \\
\tilde{M}^\dagger\tilde{M}(U\vec{x}) &= \tilde{M}^\dagger L^{-1}\vec{b}
\end{aligned}$$

将原问题转化上述形式，仅需要在迭代首尾使用 L、U 做一次变换即可

优化方法　奇偶预处理

令 $\vec{x} = \begin{bmatrix} \vec{x}_0 \\ \vec{x}_1 \end{bmatrix}$，记 $Dslashoffd$ 为 $DslashOD$，简写为 OD

求解的问题为：$\begin{bmatrix} M_{ee} & 0 \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{bmatrix} \begin{bmatrix} \vec{x}_0 \\ \vec{x}_1 \end{bmatrix} = \begin{bmatrix} \vec{b}_0 \\ \vec{b}_1 \end{bmatrix}$

$$DslashEE(x) = \begin{bmatrix} M_{ee}\vec{x}_0 \\ 0 \end{bmatrix}, \qquad DslashOD(x,0) = \begin{bmatrix} M_{eo}\vec{x}_1 \\ 0 \end{bmatrix}$$

$$DslashOO(x) = \begin{bmatrix} 0 \\ M_{oo}\vec{x}_1 \end{bmatrix}, \qquad DslashOD(x,1) = \begin{bmatrix} 0 \\ M_{oe}\vec{x}_0 \end{bmatrix}$$

$$Dslash(x) = EE(x) + OO(x) + OD(x,\ 0) + OD(x,\ 1)$$

$$\tilde{M}\vec{x} = \begin{bmatrix} M_{ee} & 0 \\ 0 & M_{oo} - M_{oe}M_{ee}^{-1}M_{eo} \end{bmatrix} \begin{bmatrix} \vec{x}_0 \\ \vec{x}_1 \end{bmatrix}$$

$$= \begin{bmatrix} M_{ee}\vec{x}_0 \\ M_{oo}\vec{x}_1 - M_{oe}M_{ee}^{-1}M_{eo}\vec{x}_1 \end{bmatrix}$$

$$= EE(x) + OO(x) - EE^{-1}\Big(OD\big(OD(x,\ 0),\ 1\big)\Big)$$

此外，$\tilde{M}\vec{x}$ 中 $\vec{x}_0$ 可直接解出，$\vec{x}_0 = M_{ee}^{-1}\vec{b}_0$

```cpp
void Dslash(lattice_fermion &src,
    lattice_fermion &dest,  lattice_gauge &U,
    const double mass,      const bool dagger)
{
    dest.clean();
    lattice_fermion tmp(src.subgs, src.site_vec);
    DslashEE(src, tmp, mass);
    dest = dest + tmp;
    DslashOO(src, tmp, mass);
    dest = dest + tmp;
    Dslashoffd(src, tmp, U, dagger, 0); // cb=0, EO
    dest = dest + tmp;
    Dslashoffd(src, tmp, U, dagger, 1);
    dest = dest + tmp;
}
```

EE(x)

OO(x)

OD(x, 0)

OD(x, 1)

```cpp
void Dslash_middle(
    lattice_fermion &src,    // x
    lattice_fermion &dest,   // y = Mx
    lattice_gauge &U,        // 组态
    const double mass,       // 夸克质量
    const bool dagger        // 是否要取 M^{dagger}
)
{
    dest.clean();
    Dslashoffd_new(src, dest, U, dagger, 0);
    Dslashoffd_new(dest, dest, U, dagger, 1);
    const double a = 4.0;
    int subgrid_vol = (src.subgs[0] * src.subgs[1]
                       * src.subgs[2] * src.subgs[3]);
    int subgrid_vol_cb = (subgrid_vol) >> 1;
    for (int i = subgrid_vol_cb * 3 * 4;
         i < subgrid_vol * 3 * 4; i++) {
        dest.A[i] = (a + mass) * src.A[i] - (1/(a+mass)) * dest.A[i];
    }
}
```

OD(x, 0)

OD(OD(x, 0), 1)

OO(x) – EE^{-1}(OD(OD(x, 0), 1)

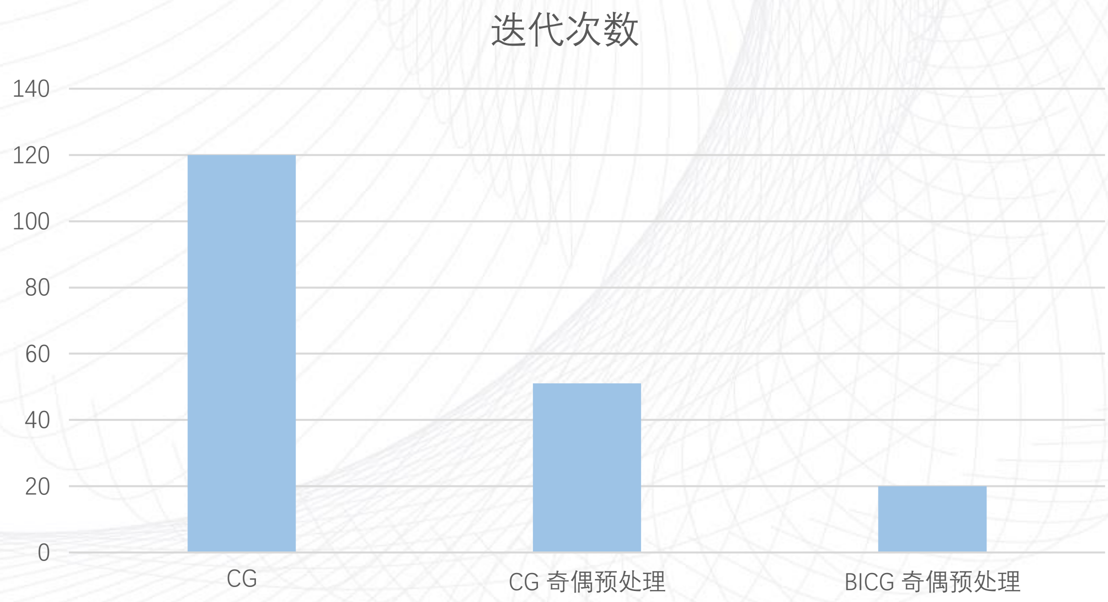
双稳定共轭梯度法

PRECONDITIONED BI-CGSTAB ALGORITHM.

x_0 is an initial guess; $r_0 = b - Ax_0$;
$\bar{r}_0$ is an arbitrary vector, such that
  $(\bar{r}_0, r_0) \neq 0$, e.g., $\bar{r}_0 = r_0$;
$\rho_0 = \alpha = \omega_0 = 1$;
$v_0 = p_0 = 0$;
for $i = 1, 2, 3, \cdots$,
  $\rho_i = (\bar{r}_0, r_{i-1})$; $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$;
  $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1})$;
  Solve y from $Ky = p_i$;
  $v_i = Ay$;
  $\alpha = \rho_i/(\bar{r}_0, v_i)$;
  $s = r_{i-1} - \alpha v_i$;
  Solve z from $Kz = s$;
  $t = Az$;
  $\omega_i = (K_1{}^{-1}t, K_1{}^{-1}s)/(K_1{}^{-1}t, K_1{}^{-1}t)$;
  $x_i = x_{i-1} + \alpha y + \omega_i z$;
  if x_i is accurate enough then quit;
  $r_i = s - \omega_i t$;
end

Vorst H . Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems[J]. SIAM Journal on Scientific and Statistical Computing, 1992, 13:631.

重新回顾本题，求解 Mx=b 运用的是共轭梯度法，解决 M 不对称正定的方法为 $M^{\dagger}M$，然而，在矩阵本身不对称正定的情况下，双稳定共轭梯度法是比标准的共轭梯度法更好的选择，它们的核心计算量均为两次 Dslash，但是后者获得更好的收敛效果。



优化结果 case1 24x24x24x72 547MB

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446056.out
CG: 118 iter, rsd |r| = 1.29836e-09
CG: 119 iter, rsd |r| = 1.02325e-09
CG: 120 iterations, convergence residual |r| = 8.0646e-10
Source and Gauge, time: 0.624727
CG invert, time: 41.8487
Total time: 42.4734
Result checking ...
 [ Accuracy = 1e-09 ]
=====================================
|b| = 2821.88     |x| = 819.232
(Mb, Mb) = (1.59543e+08,0)     (b, MMb) = (1.59543e+08,1.2642e-10)
(Mx, Mx) = (7.96303e+06,0)     (x, MMx) = (7.96303e+06,1.7053e-12)
|M x -b| = 3.16441e-10
|M^dagger M x - Mdb| = 8.06578e-10
=====================================
```

默认方式，42.4734s

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446336.out
CG: 118 iter, rsd |r| = 1.29836e-09
CG: 119 iter, rsd |r| = 1.02325e-09
CG: 120 iterations, convergence residual |r| = 8.0646e-10
Source and Gauge, time: 0.482019
CG invert, time: 4.44543
Total time: 4.92745
Result checking ...
 [ Accuracy = 1e-09 ]
=====================================
|b| = 2821.88     |x| = 819.232
(Mb, Mb) = (1.59543e+08,0)     (b, MMb) = (1.59543e+08,1.24146e-10)
(Mx, Mx) = (7.96303e+06,0)     (x, MMx) = (7.96303e+06,9.37916e-13)
|M x -b| = 3.16441e-10
|M^dagger M x - Mdb| = 8.0658e-10
=====================================
```

ICC，-O3，4.9274s

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446291.out
CG: 17 iter, rsd |r| = 2.7489e-09
CG: 18 iter, rsd |r| = 5.18415e-10
CG: 19 iterations, convergence residual |r| = 7.71779e-11
Source and Gauge, time: 0.457947
CG invert, time: 0.353209
Total time: 0.811156
Result checking ...
 [ Accuracy = 1e-10 ]
=====================================
|b| = 2821.88     |x| = 819.407
(Mb, Mb) = (1.59587e+08,-1.95668e-13)     (b, MMb) = (1.59587e+08,-4.50882e-10)
(Mx, Mx) = (7.96303e+06,-4.15371e-14)     (x, MMx) = (7.96303e+06,-7.38964e-12)
|M x -b| = 7.71827e-11
|M^dagger M x - Mdb| = 3.93312e-10
=====================================
```

默认方式，42.4734s

由于原代码已经并行化，本次优化不涉及并行化导致的效率提升。但仍就代码优化部分便实现 13 倍加速，相比于赛题指定初始方式实现了 52 倍提升。由于最终优化后计算时间低于读取数据时间，考虑计算部分加速比则是达到了 120 倍。

注：均舍弃前两次由于不可避免的文件系统不命中导致的较长加载时间。(原代码没有添加后续补充的 xg[0] = xg0[0]，导致结果略有不同)

优化结果

case2

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446874.out
CG: 130 iter, rsd |r| = 1.34243e-09
CG: 131 iter, rsd |r| = 1.07908e-09
CG: 132 iterations, convergence residual |r| = 8.67566e-10
Source and Gauge, time: 6.01738
CG invert, time: 100.931
Total time: 106.948
Result checking ...
 [ Accuracy = 1e-09 ]
=======================================
|b| = 4096.25     |x| = 1192.53
(Mb, Mb) = (3.36361e+08,0)     (b, MMb) = (3.36361e+08,-2.63753e-10)
(Mx, Mx) = (1.67792e+07,0)     (x, MMx) = (1.67792e+07,-4.14957e-12)
|M x -b| = 3.53084e-10
|M^dagger M x - Mdb| = 8.67796e-10
=======================================
```

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446836.out
CG: 18 iter, rsd |r| = 1.99077e-09
CG: 19 iter, rsd |r| = 2.60171e-10
CG: 20 iterations, convergence residual |r| = 4.65409e-11
Source and Gauge, time: 0.477917
CG invert, time: 0.947683
Total time: 1.4256
Result checking ...
 [ Accuracy = 1e-10 ]
=======================================
|b| = 4096.25     |x| = 1193.1
(Mb, Mb) = (3.36349e+08,-1.40294e-13)     (b, MMb) = (3.36349e+08,-5.65706e-1
(Mx, Mx) = (1.67792e+07,-6.9715e-16)     (x, MMx) = (1.67792e+07,-3.86393e-11
|M x -b| = 4.65575e-11
|M^dagger M x - Mdb| = 2.3196e-10
=======================================
```

总体加速比：71

case3

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446736.out
CG: 133 iter, rsd |r| = 1.2637e-09
CG: 134 iter, rsd |r| = 1.01743e-09
CG: 135 iterations, convergence residual |r| = 8.19249e-10
Source and Gauge, time: 1.70347
CG invert, time: 539.569
Total time: 541.273
Result checking ...
 [ Accuracy = 1e-09 ]
=======================================
|b| = 9216.07     |x| = 2666.78
(Mb, Mb) = (1.7016e+09,0)     (b, MMb) = (1.7016e+09,-3.09046e-09)
(Mx, Mx) = (8.4936e+07,0)     (x, MMx) = (8.4936e+07,-4.54747e-12)
|M x -b| = 3.34135e-10
|M^dagger M x - Mdb| = 8.20529e-10
=======================================
```

```
[sc94525@ln112%bscc-a3 src]$ tail -n 15 slurm-1446845.out
CG: 19 iter, rsd |r| = 1.93739e-09
CG: 20 iter, rsd |r| = 1.70786e-10
CG: 21 iterations, convergence residual |r| = 2.88905e-11
Source and Gauge, time: 1.60444
CG invert, time: 7.15981
Total time: 8.76426
Result checking ...
 [ Accuracy = 1e-10 ]
=======================================
|b| = 9216.07     |x| = 2666.66
(Mb, Mb) = (1.70173e+09,1.58751e-12)     (b, MMb) = (1.70173e+09,-3.41061e
(Mx, Mx) = (8.4936e+07,1.39223e-13)     (x, MMx) = (8.4936e+07,-3.3458e-10
|M x -b| = 2.90303e-11
|M^dagger M x - Mdb| = 1.47389e-10
=======================================
```

总体加速比：61

感谢指导

THANKS FOR WATCHING