机群系统上三维傅里叶变换

主讲人: 孙广中

2020 年 4 月 21 日

目录

目录

物理描述

在现代物理研究中，获取材料的物性可通过实验和采用第一性原理。第一性原理主要以密度泛函理论为基础。

在密度泛函理论中，多电子体系的基态能量是一个电子密度泛函的极小值:

$$E = E[n_0(r)] = min\{E[n(r)]\} \tag{1}$$

可以将其转化为求解如下 Kohn-Sham 方程

$$[-\frac{\hbar^2}{2m}\nabla^2 + V_{eff}(r)]\varphi_i(r) = \epsilon_i\varphi_i(r) \tag{2}$$

其中 $-\frac{\hbar^2}{2m}\nabla^2$ 为动能部分,$V_{eff}(r)$ 为势能部分。我们用 $\varphi_i(r) = \sum_j c_j q_j$ 表示 $\varphi_i(r)$，则可得到倒空间中的形式。

物理描述

$$\sum_k q_k^T[-\frac{\hbar^2}{2m}\nabla^2 + V_{eff}(r)]\sum_j c_j q_j = \epsilon_i \sum_k q_k^T \sum_j c_j q_j = \epsilon_i \sum_h c_j \tag{3}$$

其中动能部分是对角的，势能部分是非对角的。为了避免非对角矩阵乘法，做如下变换。

$$\sum_k q_k^T[V_{eff}(r)]\sum_j c_j q_j = \sum_k q_k^T e^{-iqr}[V_{eff}(r)]e^{iqr}\sum_j c_j q_j \tag{4}$$

这样，我们先做一次逆方向的傅里叶变换将波函数变换到实空间中，与 $V_{eff}(r)$ 的对角形式做了乘积之后，再做一次正方向的傅里叶变换回到倒空间之中。因此需要连续做两次快速傅里叶变换。

数学描述

考虑一串序列 $\{x_h\}, 0 < k \leqslant n$ 对其离散傅里叶变换后。

$$X_k = \sum_{j=0}^{n-1} \omega_n^{kj} x_j \tag{5}$$

其中 $\omega_n = exp(-2\pi i/n)$，可以将 $\{X_k\}$ 看成一个矩阵 F 乘 $\{x_k\}$ 的结果，$F_{kj} = \omega_n^{kj}$，当 $n = 2^m$ 时

$$X_k = \sum_{j=0}^{2^{m-1}} \omega_n^{kj} x_j^e + \omega_n^k \sum_{j=0}^{2^{m-1}} \omega_n^{kj} x_j^o \tag{6}$$

其中 $\omega_n^{2k} = \omega_{n/2}^k, x_k^e = x_{2k}$，$x_k^o = x_{2k-1}$

数学描述

在三维情况下，离散傅里叶变换变为:

$$X_{k_x k_y k_z} = \sum_{j_x=0}^{n_x-1} \omega_{n_x}^{k_x j_x} \sum_{j_y=0}^{n_y-1} \omega_{n_y}^{k_y j_y} \sum_{j_z=0}^{n_z-1} \omega_{n_z}^{k_z j_z} x_{j_x j_y j_z} \tag{7}$$

下文为了讨论便利，将向量 $x_{j_x j_y j_z}$ 称为三维傅里叶变换格子。

目录

机群系统

机群系统通过连接一组松散集成的计算机软件和硬件来高度紧密地协作完成计算工作。在某种意义上，它们可以被看作是一台计算机。机群中的单个计算机通常称为节点，节点之间通过局域网连接，但也有可能有其他的连接方式。机群计算机通常用来改进单个计算机的计算速度和可靠性。

机群具有以下特征:1. 机群各节点都是一个完整的系统，节点可以是工作站，也可以是 PC 机；2. 互连网络通常使用商品化网络，如以太网，FDDI 等，部分商用机群也采用专用网络互连；3. 网络接口和节点的 I/O 总线松耦合；4. 各节点有自己完整的操作系统。

在机群上实现三维傅里叶变换，我们需要使用消息传递的方法来设计并行算法。

目录

三维傅里叶变换一般并行算法

传统傅里叶变换的主要思想是块划分和面划分。

算法 D.1　传统三维傅里叶变换并行算法一

输入：需要计算的 FFT 格子$\{a_{x,y,z}\}$

输出：计算后的 FFT 格子$\{a_{x,y,z}\}$

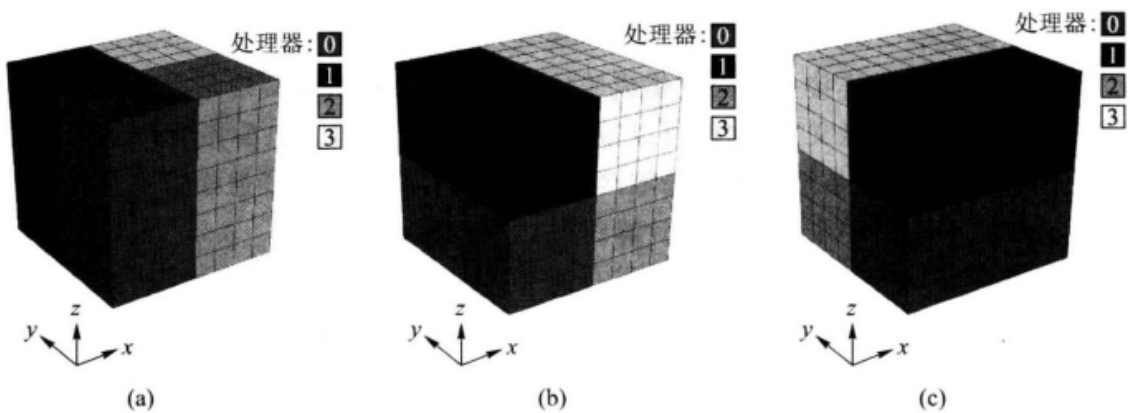
Begin

(1) 将 FFT 格子分成 4 份。每个进程负责一份$\{a_{x',y',z}\}$，进行 z 方向的一维快速傅里叶变换，如图 D.1(a)所示。

(2) 进行一次通信，每个进程得到更新后的$\{a_{x',y,z'}\}$，再进行 y 方向的一维快速傅里叶变换，如图 D.1(b)所示。

(3) 再进行一次通信，每个进程得到新的$\{a_{x,y',z'}\}$，进行 x 方向的一维快速傅里叶变换，如图 D.1(c)所示。

End



三维傅里叶变换一般并行算法

算法 D.2　传统三维傅里叶变换并行算法二

输入：需要计算的 FFT 格子$\{a_{x,y,z}\}$

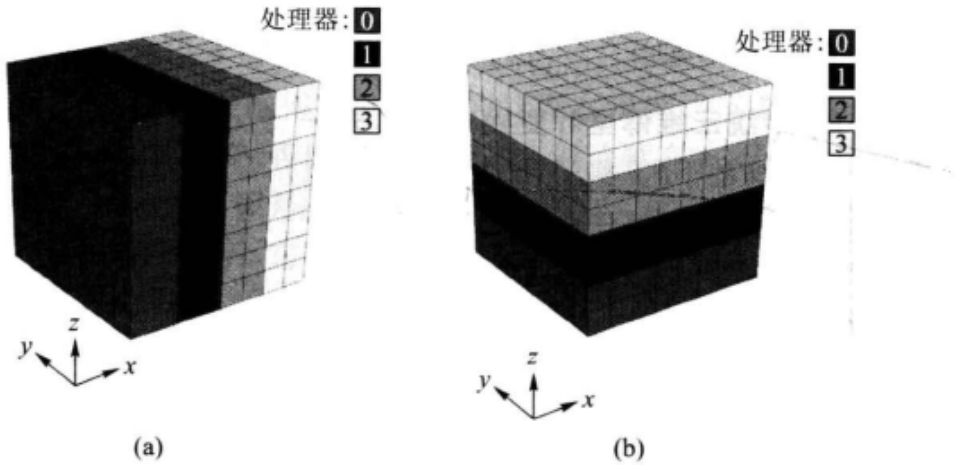
输出：计算后的 FFT 格子$\{a_{x,y,z}\}$

Begin

(1) 将 FFT 格子按 y-z 面分为四份，每个进程负责一份$\{a_{x',y,z}\}$，这样可以进行 z 方向和 y 方向的一维傅里叶变换，如图 D.2(a)所示。

(2) 进行 Alltoall 操作，每个进程得到更新后的$\{a_{x,y,z'}\}$，再进行 x 方向的一维傅里叶变换，如图 D.2(b)所示。

End



目录

改进的三维傅里叶变换并行算法

我们将 $a_{x,y,z}$ 简记为 (x, y, z)，并引入记号"$x_i y_j-$ 柱" 表示傅里叶格子中的一列 z 方向的数据。

设集合 $N = \{1, 2, ..., N\}$，其中 n 为向量维数。集合 Ω 是 $N \times N \times N$ 的一个子集。定义相关映射如下。

$$f_z : \Omega \to N \times N \quad (x, y, z) \to (x, y) \tag{8}$$

$$f_{x,z} : \Omega \to N \quad (x, y, z) \to (y) \tag{9}$$

改进的三维傅里叶变换并行算法

我们的算法如下:

算法 D.3　串行计算三维 FFT

输入:需要计算的 FFT 格子,在$(x_i,y_j,z_k),(i,j,k)\in\Omega$ 位置上的数据不为零

输出:计算后的 FFT 格子

Begin

(1) 对所有的$(i,j)\in f_z(\Omega)$,"x_iy_j-柱"施行一维 FFT 计算。

(2) 对所有的$(i,j)\in N\times f_{x,z}(\Omega)$,"$y_iz_j$-柱"施行一维 FFT 计算。

(3) 对所有的$(i,j)\in N\times N$,"x_iz_j-柱"施行一维 FFT 计算。

End

算法 D.4　并行计算三维 FFT

输入:需要计算的 FFT 格子,在集合 Ω 上数据不为零

输出:计算后的 FFT 格子

Begin

(1) 将所有含有非零元素的"x_iy_i-柱"平均分配给 p 个进程。

(2) 各进程计算自己负责的"x_iy_i-柱"。

(3) 进行 Alltoall 操作,只需传递计算过的"x_iy_i-柱"的数据,下面的操作中每个进程只负责 N/p 个 xy 面的数据。

(4) 每个进程对所有的$(i,j)\in\mathbf{N}\times f_{x,z}(\Omega)$,"$y_iz_j$-柱"施行一维 FFT 计算。

(5) 对所有的$(i,j)\in\mathbf{N}\times\mathbf{N}$,"$x_iz_j$-柱"施行一维 FFT 计算。

End

目录

应用实例

重新回到量子计算的例子中，其一般计算流程如下

① psi 数组$\{a_i\}$的元素a_i在 FFT 格子里的坐标为(x_i, y_i, z_i)。将元素a_i按坐标(x_i, y_i, z_i)映射到 FFT 格子里。

② 对 FFT 格子进行一次三维 FFT 计算(将向量从倒格子空间映到实空间)，进行一次矩阵乘法操作后，再反方向进行一次三维 FFT 计算(将向量从实空间映到倒格子空间)。

③ 从 FFT 格子里的位置(x_i, y_i, z_i)将a_i'(计算后的数据)逆映射回 psi 数组。

应用实例

其并行算法如下

算法 D.5　量子计算中三维 FFT 并行算法

输入：需要计算的 psi 数组

输出：计算后的 psi 数组

Begin

(1) 将 psi 数组映射到各个进程中，将含有非零元素的“xy-柱”平均分给每个进程，并存放在各进程中的 FFT 格子的前面。其余位置全部补零。

(2) 使用算法 D.4 并行计算 FFT。

/ * 至此完成第一次 FFT 计算，下面参照算法 D.3，逆方向做三维 FFT 并行计算 * /

(3) 对所有的$(i,j)\in \mathbf{N}\times\mathbf{N}$，“$x_iz_j$-柱”施行一维 FFT 计算。

(4) 各进程对所有 y 坐标在球内的“yz-柱”施行一维 FFT 计算。

(5) 各进程对数据进行重排序，将所需要的元素所在的“xy-柱”放到 FFT 格子前面。

(6) 进行 Alltoall 操作，每个进程得到个数大致相同的“xy-柱”。

(7) 各进程对这些“xy-柱”进行一维 FFT 计算。用(1)的逆映射将数据映射回 psi 数组。

End

目录

编程实现

P3DFFT 是并行计算三维傅里叶变换的主函数。参数 *psi* 为输入的向量。当参数 *sign* 为 1 时从倒空间到实空间。*sign* 为 -1 时反之。

```
void FFT::P3DFFT(complex<double> * psi,const int sign)
{
  /* 该变量为此进程中 z 柱的长度 */
  int npps_now=this->npps[rank_use];
  /* 倒空间- -> 实空间 */
  if(sign == 1)
  {
    /* 首先进行 z 方向的 FFT 变换,psi 数组为输入,计算结果在 aux 数组中 */
    this->fftz(psi,sign,this->aux);
    /* 我们将计算结果用 scatter 函数分发,结果仍存放在 aux 中,psi 仅为辅助空间 */
    this->scatter(psi,sign);
    /* 然后将 z 柱重新映射回原来的位置,结果在 psi 中 */
    int ctr=0; /* ctr 为计数器 counter */
```

编程实现

-

```
for(int ip=0;ip < this->nproc_use;ip ++){
  for(int is=0;is < this->nst_per[ip];is ++){
    /* ismap 函数返回 z-柱的 x,y 位置 */
    int locate_xy=this->ismap[is + this->st_start[ip]];
    for(int k=0;k < npps_now;k ++){
      psi[locate_xy * npps_now+k]=this->aux[ctr * npps_now+k];
    }
    ctr ++;
  }
}
/* 最后进行二维 FFT 计算. */
this->fftxy(psi,sign);


 }
 /* 实空间--> 倒空间 */
 else if(sign ==-1){
   /* 首先计算二维 FFT */
   this->fftxy(psi,sign);
   /* 然后将需要进行计算的 z-柱放到 aux 向量的前面 */
   int ctr=0;
```

编程实现

```
    for(int i=0;i < nproc_use;i ++){
      for(int j=0;j < nst_per[i];j ++){
        int ns=this->ismap[j + st_start[i]];
        for(int k=0; k < npps_now;k ++){
          this->aux[ctr * npps_now+k]=psi[ns * npps_now+k];
        }
        ctr ++;
      }
    }
    /* 然后用 scatter 函数分发 aux 中的数据,结果仍然在 aux 中,psi 为辅助空间 */
    this->scatter(psi,sign);
    /* 最后进行 z 方向的 FFT 计算,输入为 aux,结果在 psi 中 */
    this->fftz(aux,sign,psi);
  }
  return;
}
```

编程实现

scatter 函数负责分发各进程的计算结果，输入输出皆为 *aux* 数组,*psi* 为辅助空间。当 sign 为 1 时，需要将每个 z-柱的不同部分分到不同进程中。当 sign 为-1 时需要将不同进程的所计算的同一 z-柱的不同部分合并。

```
void FFT::scatter(complex<double> * psi,int sign)
{
  int ns=nst_per[rank_use]; /* z-柱的数目 */
  int nz=this->plan_nz; /* z-柱的长度 */
  if(sign == 1){
```

编程实现

-

```
for(int i=0;i < nproc_use;i ++){
  int npps_now=this->npps[i]; /* 进程 i 中 z-柱的长度 */
  for(int j=0;j < ns;j ++){
    for(int k=0;k < npps_now;k ++){
      /* start[i]表示 z-柱在进程 i 的起始位置,sdis 为 alltoallv 中发送地址的偏移,

                这里就将连续的 z-柱分割成按进程所需数据排列 */
              psi[sdis[i]+j * npps_now+k]=this->aux[j * nz+start[i]+k];
          }
        }
      }
      /* 然后调用 alltoallv 函数,其中的发送数目,发送偏移,接收数目,接收偏移为 sentc,
      sdis,recvc,rdis,mpicomplex 为向 MPI 注册的复数类型 */
      MPI_Alltoallv(psi,sentc,sdis,mpicomplex,this->aux,recvc,rdis,mpicomplex,MPI_
COMM_WORLD);
      }
      else if(sign==-1){
      /* sign 为-1 时先进行 alltoall 操作,然后再重排数组,限于篇幅,此处从略 */
      }
      return;
    }
```

目录

实验比较

我们在 IBM JS22 刀片服务器，曙光 4000A 以及 KD-50 国产个人高性能计算机上实现了上述算法，并在 KD-50 上做了新旧算法的对比实验。