天元数学西南中心短期课程
Matlab/Fortran 语言编程以及高性能计算

主讲人: 李瑜 (liyu@tjufe.edu.cn)

第一次课–2021.06.11

课前预习

- 通过`doc matlab`了解 Matlab 的概貌
- 通过`doc cell`了解元胞数组的使用
- 通过`doc struct`了解结构体数组的使用
- 通过`doc fminbnd`了解一元函数在给定区间上求最小值

课后操作

- 使用`matlab -nodesktop`打开 Matlab 命令行窗口
- 使用`edit your_script`编辑 Matlab 脚本
- 使用`matlab -batch your_script`运行 Matlab 脚本
- 使用`submit.bat`后台提交 Matlab 脚本
- 使用`save`和`load`保存和读取 Matlab 变量
- 使用匿名函数 (Anoymous Functions) 调用`fminbnd`求解一元函数的极值问题

挑战练习

- 利用元胞数组进行数据可视化

天元数学西南中心短期课程
Matlab/Fortran 语言编程以及高性能计算

主讲人: 李瑜 (liyu@tjufe.edu.cn)

第二次课–2021.06.16

课前预习

- 通过`doc for`了解循环的使用
- 通过`doc if`了解判断的使用
- 通过`doc meshgrid`了解网格坐标的使用
- 通过`doc find`了解查找索引的使用
- 通过`doc fsolve`了解非线性方程组求解的使用
- 通过`doc parfor`了解并行循环的使用
- 通过`doc gpuArray`了解 GPU 计算
- 通过`https://www.vim.org/`了解 Vim

课后操作

- 使用`profile`分析探查函数的执行时间
- 使用`find`对满足条件的数组进行赋值操作
- 使用`meshgrid`代替双层循环
- 使用`parfor`并行调用`fminbnd`
- 使用`gpuArray`, `gather`, `arrayfun`进行 GPU 计算
- 使用`vimrc`文件配置 Vim

挑战练习

- 利用蒙特卡罗方法求解高维数值积分

天元数学西南中心短期课程
Matlab/Fortran 语言编程以及高性能计算

主讲人: 李瑜 (liyu@tjufe.edu.cn)

第三次课–2021.06.18

课前预习

- 通过`https://www.gnu.org/home.en.html`了解 GNU
- 通过`https://gcc.gnu.org/`了解 GCC 项目 (GNU Compiler Collection)
- 通过`http://www.mingw-w64.org/doku.php/`了解 MinGW-w64 项目 (Minimalist GNU for Windows)
- 通过`https://sourceforge.net/projects/mingw-w64/`下载 MinGW-w64
- 通过`https://www.tutorialspoint.com/fortran/index.htm`了解 Fortran 语言 (Formula Translation)
- 通过`https://www.gnu.org/software/gdb/`了解 GDB 项目 (GNU Project Debugger)

课后操作

- 使用 Vim 编辑 Fortran 源代码
- 使用 Windows PowerShell 或者命令行提示符编译 Fortran 源代码
- 使用 GDB 调试可执行文件

挑战练习

- 利用牛顿法求解一元非线性方程的解

天元数学西南中心短期课程
Matlab/Fortran 语言编程以及高性能计算

主讲人: 李瑜 (liyu@tjufe.edu.cn)

第四次课–2021.06.24

课前预习

- 通过`https://ubuntu.com/`了解 Ubuntu
- 通过`http://www.netlib.org/lapack/`和`http://www.netlib.org/blas/`了解 LAPACK 和 BLAS
- 通过`https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html/`了解 Intel MKL (Math Kernel Library)
- 通过`https://hpc.llnl.gov/training/tutorials/openmp-tutorial/`了解 OpenMP (Open Multi-Processing)
- 通过`https://hpc-tutorials.llnl.gov/mpi/`了解 MPI (Message Passing Interface)
- 通过`https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html/`了解 CUDA (Compute Unified Device Architecture)
- 通过`https://www.mcs.anl.gov/petsc/`了解 PETSc/TAO (Portable, Extensible Toolkit for Scientific Computation/Toolkit for Advanced Optimization)
- 通过`https://slepc.upv.es/`了解 SLEPc (Scalable Library for Eigenvalue Problem Computations)
- 通过`http://www.fftw.org/`了解 FFTW (Fastest Fourier Transform in the West)

课后操作

- 使用 DevC++ “编辑–编译–链接–运行–调试” OpenMP+C 程序
- 使用 Xmanager 远程登录服务器, 使用 Vim 编辑代码, 使用 GCC 编译代码, 使用 GDB 调试代码

挑战练习

- 体会 “Linux 的哲学思想”: 分享, 单一功能, 连接, 一切皆文件, 避免令人困惑的用户界面, . . .
- 学习正则表达式和 Shell 编程
- 利用 Vim+Makefile+GDB 在 Linux 下搭建项目环境
- 在 Linux 下安装 PETSc, 并运行示例程序

Q&A

问: 可否比较 Intel MIC 与 NVIDIA GPU 在高性能计算中的优劣?

答: 我着实不了解 Intel MIC, 也没有使用过 Intel Xeon Phi. 中国科学技术大学超级计算中心有这样的计算资源.

问: Matlab 在 Linux 下如何启动?

答: 如果是远程访问服务器, 可视化窗口可能无法启动. 推荐使用

`matlab -nodesktop -nosplash`

启动命令行窗口. 利用

`matlab -batch your_script > output.log &`

后台提交 Matlab 脚本. 具体参数请使用`doc matlab linux`查询.

问: 使用`-nodesktop`选项打开 Matlab 命令行窗口后, 在命令行窗口中不能 TAB 键进行补全, 该怎么办?

答: 目前我也无法解决这个问题, 但使用`edit`通过 Matlab 内置编辑器编写脚本文件时依然可以使用 TAB 键进行补全.

问: 如何保存 Matlab 中的变量到文件, 以及如何读取文件中的变量到 Matlab 的工作空间?

答: 通过`doc save`和`doc load`查询详情.

问: 是否可以对所有函数使用`vpa`, 进而修改 Matlab 变量的默认精度?

答: 可能不行, 必须显示调用`vpa(x,d)`, 但可以利用 Vim 让编写更方便.

```
1 nnoremap <silent> <Leader>vp 0f=wvt;xivpa()<Esc>P0
2 vnoremap <silent> <Leader>vp svpa()<Esc>P<Right>%
3 let @v="f=wvt;xivpa()^[P0j"
```

注意这里`^[`代表 Esc 键.

问: Vim 应该怎么学, 怎么用?

答: 在 Windows 下, 安装 gVim (`https://www.vim.org/download.php`), 使用`vim tutor`迈出学习的第一步. 尝试弄明白 vimrc 中的配置信息. 尝试安装使用一些插件. 使用`:help xxx`查看内置说明文档. 尝试使用 Vim 进行所有的编辑工作, TeX, Matlab, C, Fortran, Python, R, ... 推荐我的知乎小文章– 分享 gVim 的配置文件

问: 如何使用 Vim 编写 TeX, 以及如何进行 PDF 的实时预览和正反向搜索?

答: 推荐我的知乎小文章– Win 10+TeX Live 2020+Vim+LaTex-Suite+SumatraPDF 如果大家使用 WinEdt 编写 TeX, 推荐我的另一个小文章– 你好 TeX

问: 我习惯用 Windows 了, 对于高性能计算, Linux 是必须的吗?

答: 我们可以一步步来, 先熟悉 Windows 下使用命令行窗口的各类操作 (不排斥它), 然后熟悉 gVim 的使用, 最后使用 Xmanager 或者其它远程登录软件, 远程访问 Linux 服务器, 并在命令行窗口下进行操作. 如有必要, 再尝试在个人电脑上安装 Linux 系统. (从虚拟机开始)

问: 在 Matlab 里输入`gpuDevice`报错怎么办?

答: 首先, 明确自己的显卡型号 "此电脑–右键–属性–设备管理器–显示适配器"; 然后, 网上查询显卡所对应的架构 (Architecture); 之后, 对照 Matlab 的 GPU 支持列表 明确你的显卡是否被支持; 如果依然报错, 尝试重新安装 显卡驱动以及 CUDA Toolkit. 在 Ubuntu 下, 可以通过如下命令进行显卡驱动和 CUDA Toolkit 的安装

```
ubuntu-drivers devices
sudo apt-get install nvidia-driver-xxx
sudo reboot
nvidia-smi
sudo apt-get install nvidia-cuda-toolkit
```

问: 在 Matlab 里输入`mex`报错怎么办?

答: 首先, 查阅 Matlab 外部编译环境支持列表. 对于 Linux, 直接使用`mex -setup`进行配置. 对于 Windows, "App-Ons–Get App-Ons–搜索 MinGW-w64", 下载并安装. 过程中需要登录 MathWorks 账号. 如果没有账号的话, 一定要注册一下, 并登录. 如有问题, 先查阅网页评论区中的留言.

自问: 通过这个短课程我能知道点什么?

自答:

1. 使用 Vim 编辑一切 (TeX, Matlab, C, Fortran, Python, R, . . .)
2. 使用`matlab -batch`运行脚本文件
3. 使用批处理文件`submit.bat`后台提交作业
4. 使用 Profile 寻找 Matlab 程序的计算瓶颈
5. 使用 Matlab 中的`parfor`进行多核并行计算
6. 使用 Matlab 支持的 GPU 进行并行加速
7. 使用 MinGW-w64 在 Windows 下安装 GCC (GNU Compiler Collection), 并配置环境变量
8. 了解 Fortran 的基本知识
9. 了解 Fortran, C, C++ 语言的 “编辑–编译–链接–运行–调试” 整个过程
10. 了解 Fortran, C, C++, Matlab, CUDA 混合编程
11. 了解正则表达式和 Shell 编程
12. 了解 “Linux 的哲学理念”
13. 了解 OpenMP, MPI, CUDA 并行编程技术
14. 了解 SSH 远程登录服务器

An Brief Introduction of HPC

李瑜*

June 18, 2021

Abstract

在生命科学、地球科学、地图学和地理学、油气工业建模、气候建模、电子设计自动化、媒体和娱乐等诸多学科领域中,经常会处理各种计算问题,而且时常遇到这样的情况: 由于需要大量的运算,一台通用的计算机无法在合理的时间内完成工作,或者由于所需的数据量过大而可用的资源有限,导致根本无法执行计算. 高性能计算 (HPC) 方法通过使用专门的硬件,或是将多个单元的计算能力进行整合,将数据和运算相应地分布到多个单元中,从而能够有效地克服这些限制.

本课程简要介绍高性能计算的基础知识、Linux 操作系统的基本使用、Matlab 高性能编程、OpenMP 并行程序设计模式、MPI 消息传递并行编程、CUDA 编程、Linux 集群的并行计算平台的使用. 通过一些典型科学计算问题的并行算法与程序设计实例,介绍一些当前国际上流行的科学计算软件工具及平台.

课程主要内容包括:

- Matlab/Fortran
- Linux 操作系统
- Intel Math Kernel Library
- CUDA
- OpenMP+MPI
- PETSc/TAO
- SLEPc

课程涉及的科学计算问题包括:

- 线性代数数值计算库
- 快速傅里叶变换
- 线性方程组求解
- 非线性方程组求解
- 特征值问题求解
- 约束优化问题求解

*liyu@tjufe.edu.cn, Tianjin University of Finance and Economics, Tianjin 300222, China

Contents

1 HPC

高性能计算 (HPC) 是利用超级计算机和并行处理技术来解决复杂的计算问题. 高性能计算技术的重点是开发并行处理算法和系统.

对于硬件配置而言, 常用的类型有两种：

- 共享内存计算机
- 分布式内存集群

在共享内存计算机上, 所有处理单元都可以访问随机存取存储器 (RAM); 而在分布式内存集群中, 不同的处理单元或节点之间无法访问内存. 在使用分布式内存配置时, 由于不同的处理单元不能访问同一个内存空间, 因此必须存在一个相互连接的网络, 才能在这些单元之间发送接收消息. 鉴于有些单元共享共同的内存空间, 而其他单元又是另一种情况, 现代 HPC 系统通常是融合了这两个概念的混合体.

Table 1: TOP 6 positions of the 56th TOP500 in November 2020

Rank	Rmax (PFLOPS)	Name	Country	Year	CPU cores
1	442.010	Fugaku	Japan	2020	158,976 × 48
2	148.600	Summit	United States	2018	9,216 × 22
3	94.640	Sierra	United States	2018	8,640 × 22
4	93.015	Sunway TaihuLight	China	2016	40,960 × 260
5	63.460	Selene	United States	2020	1,120 × 64
6	61.445	Tianhe-2A	China	2013	35,584 × 12

一般的笔记本大概每秒三亿次浮点指令, 大概相当于 1976 年美国 Cray-1 超级计算机的水平. (1PFLOPS 等于 10^8 亿次浮点指令/秒)

$$1 \times 365 \times 24 \times 60 \times 60 = 0.31536 \times 10^8$$

截至 2021 年 1 月, 中国共建成或正在建设 8 座超算中心, 分别为

- 国家超级计算天津中心 – 天河一号、天河三号
- 国家超级计算广州中心 – 天河二号
- 国家超级计算深圳中心 – 曙光 6000
- 国家超级计算长沙中心 – 天河

- 国家超级计算济南中心 – 神威
- 国家超级计算无锡中心 – 神威 • 太湖之光
- 国家超级计算郑州中心 – 浪潮
- 国家超级计算昆山中心 – 曙光

```
# 总核数 = 物理CPU个数 X 每颗物理CPU的核数
# 总逻辑CPU数 = 物理CPU个数 X 每颗物理CPU的核数 X 超线程数
# 查看物理CPU个数
cat /proc/cpuinfo| grep "physical id" | sort | uniq | wc -l
# 查看每个物理CPU中core的个数(即核数)
cat /proc/cpuinfo| grep "cpu cores" | uniq
# 查看逻辑CPU的个数
cat /proc/cpuinfo| grep "processor" | wc -l
# 查看CPU型号
cat /proc/cpuinfo| grep "model name" | sort -u
```

比如 physical id 共 2 个, cpu cores 为 24, processor 为 96. CPU 型号 Intel Xeon(R) Gold 6248R CPU @ 3.00GHz

推荐中文书籍:

- 《Fortran95 程序设计》, 彭国伦, 2002 年, 中国电力出版社.
- 《并行计算导论》, 张林波等, 2006 年, 清华大学出版社.
- 《Linux 就该这么学》, 刘遄, 2017 年, 人民邮电出版社.

2 Matlab

Stop Trying to Reinvent the Wheel.

百分之一的代码解决百分之九十九的问题.

全世界数以百万计的工程师和科学家都在使用 MATLAB® 分析和设计改变着我们的世界的系统和产品。基于矩阵的 MATLAB 语言是世界上表示计算数学最自然的方式。可以使用内置图形轻松可视化数据和深入了解数据。欢迎您使用桌面环境进行试验、探索和发现。这些 MATLAB 工具和功能全部进行了严格测试，可彼此配合工作。

MATLAB 可帮助您不仅仅将自己的创意停留在桌面。您可以对大型数据集运行分析，并扩展到群集和云。MATLAB 代码可以与其他语言集成，使您能够在 Web、企业和生产系统中部署算法和应用程序。

- `matlab -nodesktop -nosplash`
- `cd, ls, mkdir, delete`
- `doc, which`
- `edit, workspace, desktop`
- `!dos_cmd`

```
@echo off
if "%1"=="h" goto begin
    mshta vbscript:createobject("wscript.shell").^
    run("%~nx0 h",0)(window.close)^
    &&exit
:begin
matlab -batch "your_script" -logfile output.log
```

^表示折行.

2.1 基础知识

- 基本数据类型: 数值数组, 元胞数组, 结构体数组, 函数句柄
- 文件类型: script, m-file
- 说明文档: doc
- 结构体数组是使用名为字段的数据容器. 每个字段都可以包含任意类型的数据. 使用 structName.fieldName 来访问字段中的数据.
- 元胞数组是包含元胞的索引数据容器的数据类型. 每个元胞可以包含任意类型的数据. 对于元胞数组, 使用圆括号 () 进行索引来引用元胞, 使用花括号 {} 进行索引来访问元胞的内容.
- 函数句柄是一种表示函数的数据类型. 函数句柄是将一个函数传递给另一个函数. 使用 @ 运算符创建函数句柄. 匿名函数是不存储在程序文件中, 数据类型是 function_handle 的变量. 匿名函数可以接受输入并返回输出, 就像标准函数一样. 但是, 它们可能只包含一个可执行语句.

脚本文件 test_submit.m

```
clear
A = rand(3); B = rand(5); C = rand(7);
matrix{1}=A; matrix{2}=B; matrix{3}=C;
for i=1:3
    [eigenvector{i}, eigenvalue{i}] = eig(matrix{i});
end
for i=1:3
    struct_data(i).matrix=matrix{i};
    struct_data(i).eigenvalue=diag(eigenvalue{i});
    struct_data(i).eigenvector=eigenvector{i};
end
save mat_eigenpair.mat struct_data
figure
hold on
for i=1:3
    plot(struct_data(i).eigenvalue,'o')
end
```

```
hold off
box on
%saveas(gca,'eigenvalue','epsc')
saveas(gca,'eigenvalue','jpeg')
%%
myfunc2=@(x, y, z)((x-y*z).^2);
x=zeros(3,1);
for i = 1:3
    y = i; z = y.^0.5;
    myfunc1=@(x)(myfunc2(x, y, z));
    x(i)=fminbnd(myfunc1, -10, 10);
end
save min_point.mat coord
whos
```

数组可以进行分块操作. 元胞数组和结构体数组的最大区别在于索引. 二者在很多应用场景中可以互相替换.

可以通过将函数句柄收集到一个元胞数组或结构体数组中, 来创建由这些函数句柄组成的数组. 这样生成一个操作集, 便于之后的使用以及代码的移植.

```
fun=cell(4,1);
fun{1}=@(x)(sin(x)); fun{2}=@(x)(asin(x));
fun{3}=@(x)(tan(x)); fun{4}=@(x)(atan(x));
```

脚本和函数都可以通过将命令序列存储在程序文件中来重用它们. 脚本是最简单的程序类型, 因为它们存储命令的方式与您在命令行中键入命令完全相同. 函数更灵活, 更容易扩展.

m 文件 test_function.m

```
function [x,fval] = test_function(myfunc2,y,z)
    n = length(y(:)); x = zeros(size(y)); fval = zeros(size(y));
    for i = 1:n
        myfunc1 = @(x)(myfunc2(x,y(i),z(i)));
        [x(i),fval(i)] = fminbnd(myfunc1,-10,10);
    end
end
```

脚本文件

```
clear
myfunc2 = @(x, y, z)((x-y*z).^2);
y = [3, 2; 1, 0]; z = [.2, .4; .6, .8];
[x,fval] = test_function(myfunc2,y,z);
```

输入参数和返回参数也可以是结构体数组或元胞数组或函数句柄

Remark 2.1. 人的影响短暂而微弱, 视频的影响则广泛而深远.

- *Introducing Structures and Cell Arrays*

2.2 串行程序

遵循 Matlab 编程原则, 进行编程.

- 利用 Profile 性能分析器, 剖析代码, 寻找运算瓶颈 (`profile on`, `profile viewer`)
- 内存预分配 (`zeros(n,m)`)
- 向量化操作 (替换`for`循环)
- 使用逻辑索引`find` (替换`if`判断)
- 使用`meshgrid` (替换`for for`循环)
- 注意数据的列存储

初始代码:

```
nrows=2000; ncols=2000;
for row=1:nrows
    for col=1:ncols
        if (row+col) <= (nrows+ncols)/2
            A(row,col) = sin(row)*cos(col);
        else
            B(row,col) = log(row)*tan(1/col);
        end
        C(row,col) = A(row,col)*B(row,col);
```

```
    end
end
```

改进代码:

```
nrows=2000; ncols=2000;
A = zeros(nrows, ncols); B = zeros(nrows, ncols);
x = linspace(1,nrows,nrows); y = linspace(1,ncols,ncols);
[cols, rows] = meshgrid(y, x);
logic = (rows+cols) <= (nrows+ncols)/2;
idx_A = find(logic); idx_B = find(~logic);
A(idx_A) = sin(rows(idx_A)).*cos(cols(idx_A));
B(idx_B) = log(rows(idx_B)).*tan(1./cols(idx_B));
C = zeros(nrows, ncols);
C = A.*B;
```

2.3 并行程序

Parallel Computing Toolbox: Perform parallel computations on multicore computers, GPUs, and clusters

在文档中列举了支持并行运算的函数和工具箱, 如, 优化工具包, 统计工具包.

2.3.1 本地多核并行

通过设置选项, 直接开启并行操作, 无需更改代码. 比如,

```
fun = @(x)(sin(10*x).*exp(-0.1*x.^2));
x0 = 3; options = optimoptions('UseParallel',true);
[x,fval] = fsolve(fun,x0,options);
```

对于参数扫描, 蒙特卡罗方法, 使用parfor对串行代码并行化. 比如,

```
dx = 0.001; bnd = -10:dx:10; n = length(bnd)-1;
x = zeros(n,1); fval = zeros(n,1);
fun = @(x)(sin(10*x).*exp(-0.1*x.^2));
options = optimset('TolX',1e-12);
parfor i = 1:n
```

```
    [x(i),fval(i)] = fminbnd(fun,bnd(i),bnd(i+1),options);
end
[~,i] = min(fval);
```

详见doc parfor, doc UseParallel, doc parpool

2.3.2 本地 GPU 并行

- 大规模适合 GPU 并行化
- 使用 Matlab 内建的 GPU 数据类型和函数
- 直接在 GPU 中创建变量
- 调用 arrayfun, 把多个逐元素运算组合成单个运算
- 连续调用 GPU 计算, 减少 CPU 和 GPU 之间的数据传输

```
nrows=7000;
A=rand(nrows,nrows)+rand(nrows,nrows)*1i;
B=rand(nrows,nrows)+rand(nrows,nrows)*1i;
A=fft2(A); B=fft2(B); C=ifft2(A.*B);
%%
gpuDevice
gpuDevice(1)
gpu_A=gpuArray(A); gpu_B=gpuArray(B);
gpu_A=fft2(gpu_A); gpu_B=fft2(gpu_B); gpu_C=ifft2(gpu_A.*gpu_B);
C=gather(gpu_C);
whos
```

```
nrows = 2000; ncols = 2000;
A = zeros(nrows, ncols, 'gpuArray');
B = zeros(nrows, ncols, 'gpuArray');
x = gpuArray.linspace(1,nrows,nrows);
y = gpuArray.linspace(1,ncols,ncols);
[cols,rows] = meshgrid(y,x);
logic = (rows+cols) <= (nrows+ncols)/2;
idx_A = find(logic); idx_B = find(~logic);
```

```
A(idx_A) = sin(rows(idx_A)).*cos(cols(idx_A));
B(idx_B) = log(rows(idx_B)).*tan(1./cols(idx_B));
C = zeros(nrows, ncols, 'gpuArray');
C = A.*B;
```

```
doc Functions on a GPU
```

2.3.3 远程多节点分布式并行

Matlab Distributed Computing Server

- 内存不足, 执行时间过长
- batch, matlabpool
- submit, wait, load
- Job Monitor

2.4 MEX

- Windows 下推荐 Matlab 与 C 混合编程
- 推荐在 Linux 下进行混合编程

```
#include    <stdio.h>
#include    <math.h>
#include    "mex.h"
#include    "matrix.h"
void C_function(double *A,double *B,double *C,int nrows,int ncols);
void mexFunction(int nlhs,mxArray *plhs[],
        int nrhs,const mxArray *prhs[])
{
    /* Input */
    mwSize nrows, ncols;
    nrows = (mwSize)mxGetScalar(prhs[0]);
    ncols = (mwSize)mxGetScalar(prhs[1]);
    /* Output */
```

```
    plhs[0] = mxCreateDoubleMatrix(nrows, ncols, mxREAL);
    plhs[1] = mxCreateDoubleMatrix(nrows, ncols, mxREAL);
    plhs[2] = mxCreateDoubleMatrix(nrows, ncols, mxREAL);
    mxDouble *A = mxGetPr(plhs[0]);
    mxDouble *B = mxGetPr(plhs[1]);
    mxDouble *C = mxGetPr(plhs[2]);
    C_function((double*)A,(double*)B,(double*)C,(int)nrows,(int)ncols);
    return;
}
void C_function(double *A,double *B,double *C,int nrows,int ncols)
{
    int row, col, idx;
    for(col = 0; col < ncols; ++col) {
        for(row = 0; row < nrows; ++row) {
            idx = row+col*nrows;
            if ((row+col+2) <= (nrows+ncols)/2) {
                A[idx] = sin(row+1)*cos(col+1);
            }
            else {
                B[idx] = log(row+1)*tan(1.0/(col+1));
            }
            C[idx] = A[idx]*B[idx];
        }
    }
    return;
}
```

Remark 2.2. 视频是人类进步的阶梯, 终生的伴侣, 最诚挚的朋友.

- *MATLAB* 高级编程之性能加速: 从代码优化到并行计算
- *GPU* 加速和集群计算
- *MATLAB* 并行计算–从个人桌面到远程集群和云

3 Fortran

3.1 Windows 下的环境配置

MinGW-w64 (Minimalist GNU for Windows) is an advancement of the original mingw.org project, created to support the GCC compiler on Windows systems. It has forked it in 2007 in order to provide support for 64 bits and new APIs. It has since then gained widespread use and distribution.

`http://www.mingw-w64.org/doku.php/`

`https://sourceforge.net/projects/mingw-w64/`

编辑系统环境变量 → 环境变量 →Path→ 新建 → 填入bin文件的路径

3.2 基本数据类型

- 整数
- 浮点数
- 复数
- 字符
- 逻辑判断

3.3 流程控制

- 判断
- 循环

3.4 数组与函数

- `real(8) matrix(3,4)`
- `subroutine`
- `function`

3.5 编辑-编译-链接

- nm
- ldd
- make

3.6 GDB

- gdb a.exe
- run
- quit
- start
- finish
- break function_name
- break line_name
- delete breakpoints number
- info breakpoints
- next
- step
- contimue
- list function_name
- list line_name
- print variable_name
- display/format expression
- undisplay

4 Linux

GNU (GNU is Not Unix) 是一个免费使用和自由传播的类 UNIX 操作系统. GNU 所用的典型内核是 Linux, 该组合叫做 GNU/Linux 操作系统.

Ubuntu/Federal/Redhat/CentOS

4.1 Basic Commands

- `cd`, `ls`, `mv`, `rm`, `cat`, `more`
- `grep`, `sort`, `find`, `tree`
- `awk`, `sed`
- `which`, `locate`, `chmod`
- `xargs`, `>`, `>>`, `|`, `&`
- 正则表达式 (`$`, `^`, `*`)
- `vim` (`vimtutor`, `https://vim-adventures.com/`)
- `ssh`, `sftp`, `scp` (Xmanager for Windows)
- `man`

```
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
function cdls()
{
    builtin cd "$1" && ls -CFh
}
alias cd='cdls'
alias rm='rm -i'
alias cp='cp -i'
```

4.2 GCC 编译链接选项

常用参数

- `-c` : Compile or assemble the source files, but do not link
- `-o file` : If `-o` is not specified, the default is to put an executable file in `a.out`
- `-Idir` : Add directory dir to the list of directories to be searched for header files
- `-Ldir` : Add directory dir to the list of directories to be searched for `-l`
- `-l` : `libm.a` ⇔ `-lm`

Linux 下安装软件:

- `./configure --help`
- `./configure --prefix=/install/directory/`
- `make`
- `make install`

4.3 C+Matlab 混合编程

mex

https://ww2.mathworks.cn/support/requirements/supported-compilers.html

```
clear
% header files
INCS = ['-I' './your_C_code'];
% mex -setup C
% int in C may be replaced by size_t, which is 8 bytes,
% when calling BLAS and LAPACK. ('-Dint="long long int"')
mex('-v','-R2017b', './call_C.c',...
    './your_C_code/hi.c',INCS)
%%
clear;clc;
```

```
N = 2000000;
alpha = 1.0; beta = 1.0;
x = rand(1,N); y = rand(1,N);
% z = alpha*x + beta*y
tic;
for i=1:N
    z(i)=alpha*x(i)+beta*y(i);
end
% w = x'*y
w = 0;
for i=1:N
    w=w+x(i)*y(i);
end
toc;
tic;
z=alpha*x+beta*y;
w=x'*y;
toc;
tic; % not use BLAS yet
[z, w] = call_C(alpha, x, beta, y);
toc;
```

4.4 C+Cpp+Fortran 混合编程

- ldd, nm
- ctags, cscope

```
#!/bin/sh
ctags -R --c++-kinds=+p --fields=+iaS \
    --extra=+q --exclude="backup" --exclude="doc"
```

4.5 GPU 与 CUDA

显卡的架构对显卡的性能有很大的影响. 目前, 显卡架构性能的排序如下:

Table 2: NVIDIA Architecture

Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere
2006	2010	2012	2014	2016	2018	2018	2020

4.5.1 安装

Ubuntu 18.04 安装显卡驱动与 CUDA:

```
ubuntu-drivers devices
sudo apt-get install nvidia-driver-455
sudo reboot
nvidia-smi
sudo apt-get install nvidia-cuda-toolkit
```

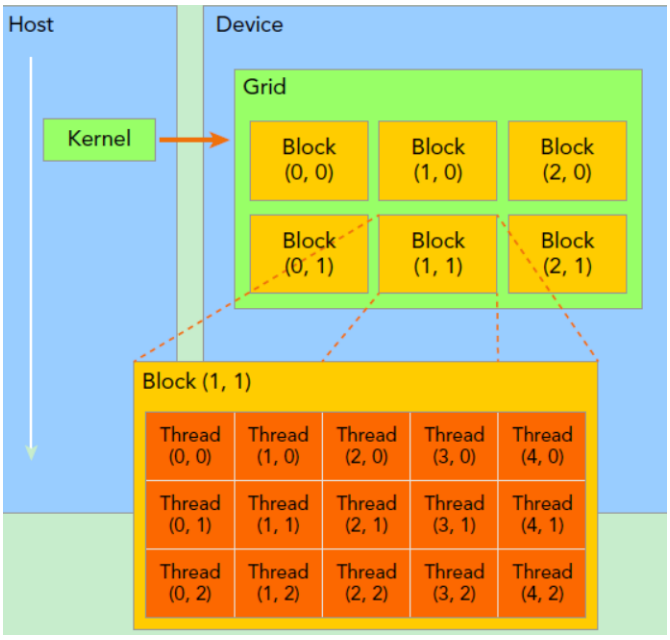
4.5.2 基本概念

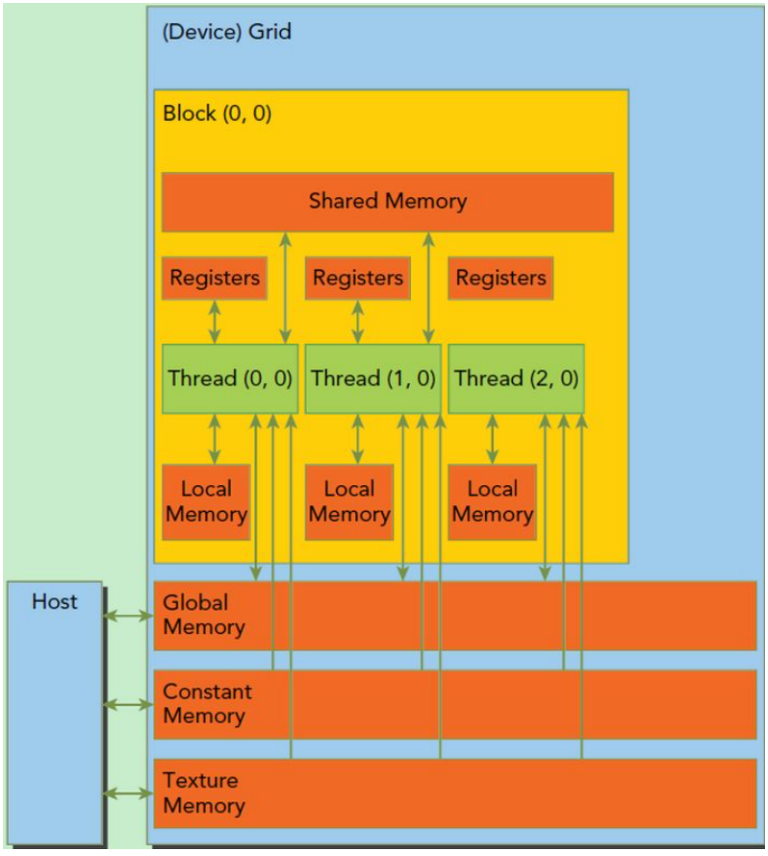
SP (streaming Process), SM (streaming multiprocessor) 是硬件 (GPU) 概念, 而 thread, block, grid, warp 是软件上的 (CUDA) 概念. 需要指出, 每个 SM 包含的 SP 数量依据 GPU 架构而不同.

为了方便程序员软件设计、组织线程, CUDA 的软件架构由网格 (Grid)、线程块 (Block) 和线程 (Thread) 组成. 相当于把 GPU 上的计算单元分为若干 (2 或 3) 个网格, 每个网格内包含若干个线程块, 每个线程块包含若干个线程.

- thread: 一个 CUDA 的并行程序会被以许多个 threads 来执行
- block: 数个 threads 会被组成一个 block, 同一个 block 中的 threads 可以同步, 也可以通过 shared memory 通信
- grid: 多个 blocks 则会再构成 grid
- warp: 把 32 个 threads 组成一个 warp. warp 是调度和运行的基本单元. warp 中所有 threads 并行的执行相同的指令. 一个 warp 需要占用一个 SM 运行. 所以, 一个 GPU 上 resident thread 最多只有 SM*warp 个.

CUDA 通过 block 这个概念, 提供了细粒度的通信手段, 因为 block 是加载在 SM 上运行的, 所以可以利用 SM 提供的 shared memory 和 `__syncthreads()` 功能实现线程同步和通信. 而 block 之间, 除了结束 kernel 之外是无法同步的, 一般也





不保证运行先后顺序, 这是因为 CUDA 程序要保证在不同规模 (不同 SM 数量) 的 GPU 上都可以运行, 必须具备规模的可扩展性, 因此 block 之间不能有依赖. 这就是

CUDA 的两级并行结构.

一个 block 只会由一个 SM 调度, block 一旦被分配到某个 SM, 该 block 就会一直驻留在该 SM 中, 直到执行结束. 一个 SM 可以同时拥有多个 blocks, 但需要序列执行.

大部分 threads 只是逻辑上并行, 并不是所有的 thread 可以在物理上同时执行. 这就导致同一个 block 中的线程可能会有不同步调. 另外, 并行 thread 之间的共享数据会导致竞态, 即多个线程请求同一个数据会导致未定义行为. CUDA 提供了`cudaThreadSynchronize()` 来同步同一个 block 的 thread 以保证在进行下一步处理之前, 所有 thread 都到达某个时间点.

同一个 warp 中的 thread 可以以任意顺序执行, active warps 被 SM 资源限制. 当一个 warp 空闲时, SM 就可以调度驻留在该 SM 中另一个可用 warp. 在并发的 warp 之间切换是没什么消耗的, 因为硬件资源早就被分配到所有 thread 和 block, 所以该新调度的 warp 的状态已经存储在 SM 中了. CPU 切换线程需要保存/读取线程上下文 (register 内容), 这是非常耗时的, 而 GPU 为每个 threads 提供物理 register, 无需保存/读取上下文.

4.5.3 CUDA 编程

CUDA 的操作概括来说包含 5 个步骤:

- CPU 在 GPU 上分配内存: `cudaMalloc`
- CPU 把数据发送到 GPU: `cudaMemcpy`
- CPU 在 GPU 上启动内核 (kernel), 它是自己写的一段程序, 在每个线程上运行
- CPU 把数据从 GPU 取回: `cudaMemcpy`
- CPU 释放 GPU 上的内存: `cudaFree`

一个 kernel 的调用为:

```
1 Kernel<<<dimGrid, dimBlock>>>(param1, param2, ...)
```

并通过线程编号进行编程.

三种前缀分别用于在定义函数时限定该函数的调用和执行方式:

- `__host__ int foo(int a){}`与 C 或者 C++ 中的 foo(int a) 相同, 是由 CPU 调用, 由 CPU 执行的函数

- `__global__ int foo(int a){}`表示一个内核函数, 是一组由 GPU 执行的并行计算任务, 以`foo<<<>>>(a)`的形式或者 driver API 的形式调用. 目前`__global__`函数必须由 CPU 调用, 并将并行计算任务发射到 GPU 的任务调用单元.

- `__device__ int foo(int a){}`则表示一个由 GPU 中一个线程调用的函数. 实际上是将`__device__`函数以`__inline`形式展开后直接编译到二进制代码中实现的, 并不是真正的函数.

```
const int N = 3000;
const int threadsPerBlock = 1024;
const int blocksPerGrid = 1;
__device__
int getGlobalIdx_1D_1D(){
    return blockIdx.x * blockDim.x + threadIdx.x;
}
__global__
void dot(float *a, float *b, float *c){
    __shared__ float cache[threadsPerBlock];
    int tid = getGlobalIdx_1D_1D();
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while(tid < N){
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp;
    //对线程块中的线程进行同步
    __syncthreads();
    int i = blockDim.x/2;
    while(i != 0){
        if(cacheIndex < i){
            cache[cacheIndex] += cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
```

```
28      }
29      if (cacheIndex == 0){
30          c[blockIdx.x] = cache[0];
31      }
32  }
33  int main(int argc, char *argv[]){
34      int i;
35      float *a, *b, c, *partial_c;
36      float *dev_a, *dev_b, *dev_partial_c;
37      //在CPU上面分配内存
38      a = (float*)malloc(N*sizeof(float));
39      b = (float*)malloc(N*sizeof(float));
40      partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
41      //在GPU上分配内存
42      cudaMalloc((void**)&dev_a,N*sizeof(float));
43      cudaMalloc((void**)&dev_b,N*sizeof(float));
44      cudaMalloc((void**)&dev_partial_c,blocksPerGrid*sizeof(float));
45      //填充主机内存
46      for(i = 0; i < N; i++){
47          a[i] = 1; b[i] = 0;
48      }
49      //将数组 a 和 数组 b 复制到GPU
50      cudaMemcpy(dev_a,a,N*sizeof(float),cudaMemcpyHostToDevice);
51      cudaMemcpy(dev_b,b,N*sizeof(float),cudaMemcpyHostToDevice);
52      dot<<<blocksPerGrid, threadsPerBlock>>>(
53          dev_a, dev_b, dev_partial_c);
54      //将数组 dev_partial_c 从 GPU 复制到 CPU
55      cudaMemcpy(partial_c,dev_partial_c,
56          blocksPerGrid*sizeof(float),cudaMemcpyDeviceToHost);
57      //在CPU上完成最终的求和运算
58      c = 0.0;
59  #pragma omp parallel for reduction(+:c) num_threads(2)
60      for(i = 0; i < blocksPerGrid; i++){
61          c += partial_c[i];
62      }
63      printf("%s\n", "==================================");
```

```
    for(i = 0; i < N; i++){
        printf("%f  %f \n", a[i], b[i]);
    }
    printf("c = %f \n", c);
    printf("blocksPerGrid %d \n", blocksPerGrid);
    // 释放 GPU 上的内存
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_partial_c);
    // 释放 CPU 上的内存
    free(a); free(b); free(partial_c);
    return 0;
}
```

```
nvcc -Xcompiler -fopenmp main.cu （切勿复制）
```

多 GPU 时, 考虑使用 OpenMP+CUDA. 每一个线程控制一个 GPU. 如果需要在不同 GPU 上进行通信, 则需要在 CPU 上操作.

Remark 4.1. *float* 的运行时间为 *double* 的四分之一.

4.6 CUDA+Matlab 混合编程

- 以谱方法求解薛定谔方程为例, 进行演示
- 串行程序要体现 matlab 编程原则
- 使用`parfor`在不同参数下求解
- 调用`fft2`和`ifft2`, 体现 cpu 计算和 gpu 计算的区别
- 调用`fmincon`进行求解一个优化问题, 体现串行与并行的区别

利用 C+Matlab 混合编程以及`mexcuda`, 可以实现 CUDA+Matlab 混合编程. 所以 C+CUDA+Matlab 可以极大程度地提升 Matlab 中一些操作的计算效率. 比如, 不得不进行循环遍历的操作或者是 FFT. 更方便的是, 直接利用 gpuArray 以及支持 GPU 的操作进行运算.

下面是利用 gpuArray 调用 GPU, 对程序进行加速的 Matlab 代码.[1]

[1] 感谢张少波博士提供源代码

```
clear
format short e
tic
a=-4;b=4;
c=-4;d=4; % interval
beta=-2;  % coefficient of nonlinearity
T=0.5;    % terminal time
tau=1e-4; % time step
Num=T/tau;% iteration of time
N1=1024;N2=N1;
x=gpuArray((b-a)/N1*[0:N1-1]'+a);% discretization on x
y=gpuArray((d-c)/N2*[0:N2-1]'+c);% discretization on y
[X,Y]=meshgrid(x,y);
psi=exp(-2*(X.^2+Y.^2));% initial condition
psi(:,1)=0;             % homogeneous boundary conditions
psi(1,:)=0;             % homogeneous boundary conditions
V=(X.^2+Y.^2)/2;        % external potential
%some factors in second step
M_x=gpuArray(repmat(-4*pi^2/(b-a)^2*[0:N1/2 -N1/2+1:-1].^2,N2,1));
M_y=gpuArray(repmat(-4*pi^2/(d-c)^2*[0:N2/2 -N2/2+1:-1]'.^2,1,N1));
M_hat=exp((M_x+M_y)*tau*1i/2);
% numerical solution
% first step - tau/2   part: V(x)+nonlinearity
psi=exp(-0.5i*tau*(V+beta*abs(psi).^2)).*psi;
for m=1:round(Num)-1
    %second step - tau   part: Laplacian
    psi_hat=fft2(psi);
    w_hat=M_hat.*psi_hat;
    psi=ifft2(w_hat);
    %third step and first step in next iteration
    psi=exp(-1i*tau*(V+beta*abs(psi).^2)).*psi;
end
%second step
psi_hat=fft2(psi);
w_hat=M_hat.*psi_hat;
```

```
psi=ifft2(w_hat);
% first step
psi=exp(-0.5i*tau*(V+beta*abs(psi).^2)).*psi;
toc
```

CPU 版本[2]耗时 68.9 秒, GPU 版本耗时 7.8 秒.

4.7 Makefile

《跟我一起学 Makefile》

4.8 DevC++ for Windows

[2]Matlab 默认 OpenMP 加速

5 Intel MKL

Use the Intel Math Kernel Library (Intel MKL) when you need to perform computations with high performance. Intel MKL offers highly-optimized and extensively threaded routines which implement many types of operations.

5.1 Linear Algebra

- BLAS:

  Basic Linear Algebra Subprograms are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations.

- LAPACK:

  Linear Algebra PACKage provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

- ScaLAPACK:

  Scalable LAPACK includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers.

- PARDISO:

  PARallel DIrect SOlver is a thread-safe, high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared-memory and distributed-memory multiprocessors.

- ...

5.2 Fast Fourier Transforms

The Intel MKL provides an interface for computing a discrete Fourier transform through the fast Fourier transform algorithm.

- Multi-dimensional (up to 7D) FFTs
- FFTW interfaces
- Cluster FFT

For $k_1 = 0, \ldots, n_1 - 1$ and $k_2 = 0, \ldots, n_2 - 1$,

$$z[k_1][k_2] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} w[j_1][j_2] \exp\left(\delta 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2})\right)$$

where $\delta = -1$ for the forward transform, and $\delta = +1$ for the inverse (backward) transform.

测试环境:

Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz, 逻辑核数 2*24*2, 内存 500GB

GeForce RTX 2080 Ti, 多处理器 (SM) 数量 68, 显存 11016MB

FFT 3D 1024*1024*1024, 数据类型是复的双精度浮点型. 第一列是使用的线程数或进程数. 每个行块的第一行是 plan 的时间, 第二行是执行一次正变换和一次逆变换的时间.

Table 3: 性能测试

	fftw+gcc+OpenMP	fftw+mpicc	fttw+icc+OpenMP	Intel MKL
4	251.1294	223.8174	276.0089	0.0603
	15.6085	22.8294	16.0727	13.4966
8	246.0492	169.1315	237.5262	0.0556
	8.3121	13.7400	10.3057	7.7221
16	209.0238	188.6919	272.4807	0.0564
	7.8849	10.8663	8.4581	5.2425
32	215.0523	125.7797	224.9777	0.0579
	8.5432	11.4789	8.4955	4.4279

利用 GPU, 测试 CUFFT 的 plan 时间为 0.1527, 执行时间为 3.4662. 这里的测试规模是 512*1024*1024, 而且是复的单精度浮点型.

注意, CUFFT 的执行时间是包括从 CPU 拷贝输入数据到 GPU 中, 以及从 GPU 拷贝计算结果到 CPU 中. 执行时间几乎都花费在了数据传入传出.

5.3 编译链接选项

https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl/link-line-advisor.html

```
/* compiler options */
-I${MKLROOT}/include -qopenmp
-mkl=parallel(seqential|cluster)
/* linker options */
-L${MKLROOT}/lib/intel64
-lmkl_scalapack_lp64.a
-lmkl_lapack95_lp64
-lmkl_blas95_lp64
-Wl,--start-group
${MKLROOT}/lib/intel64/libmkl_cdft_core.a
${MKLROOT}/lib/intel64/libmkl_blacs_intelmpi_lp64.a
${MKLROOT}/lib/intel64/libmkl_intel_thread.a
${MKLROOT}/lib/intel64/libmkl_intel_lp64.a
${MKLROOT}/lib/intel64/libmkl_core.a
-Wl,--end-group
-liomp5 -lpthread -lm -ldl
```

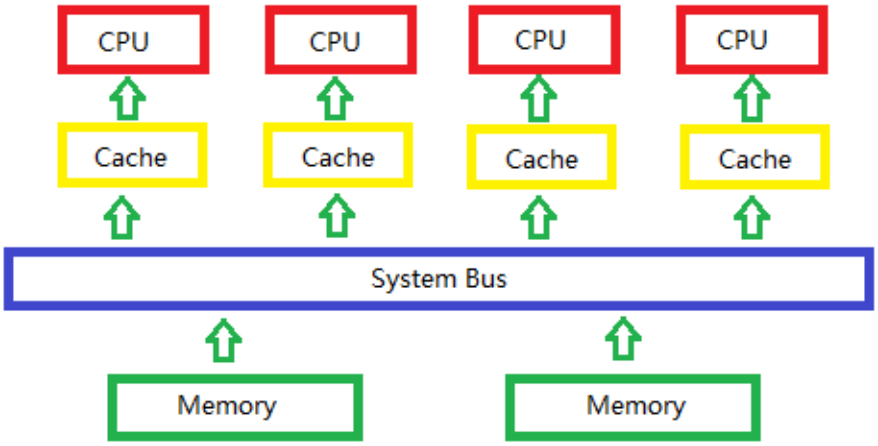
`-Wl,--start-group ... -Wl,--end-group` 用于解决几个库的循环依赖关系.

Remark 5.1. 强烈推荐使用 *Intel* 编译器进行编译, 并使用 *Intel MKL* 进行链接.

6 OpenMP

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures.

`https://computing.llnl.gov/tutorials/openMP/`



高效利用 Cache 是关键.

6.1 Hello World

```
#include    <stdio.h>
#include    <omp.h>
int main(int argc, char *argv[])
{
    printf("Hello World! Number of processors: %d\n",
        omp_get_num_procs());
    printf("Hello World! Thread: %d\n",
        omp_get_thread_num());
    int i, sum = 0;
#pragma omp parallel for reduction(+:sum) num_threads(5)
    for (i = 0; i < 11; ++i) {
        sum += i;
```

```
13          printf("Hello World! Thread: %d, i = %d, sum = %d\n",
14              omp_get_thread_num(), i, sum);
15      }
16      return 0;
17  }
```

需要头文件include <omp.h>, 由编译指令控制将代码分为串行区和并行区. 串行区只有一个 master 线程存在. 通过gcc编译链接时均加入-fopenmp.

6.2 常用环境函数

- omp_get_num_procs : 返回运行本线程的多处理机的处理器个数
- omp_get_num_threads: 返回当前并行区域中的活动线程个数
- omp_get_thread_num : 返回线程号
- omp_set_nested : 设置嵌套并行深度
- omp_set_num_threads: 设置并行执行代码时的线程个数

6.3 编译指令

基本格式为:

```
1  #pragma omp <编译关键字> [ 子句[ [,]子句] ]
```

常用编译关键词:

- parallel: 创建线程组, 并行执行
- for: 将for循环分配给各个线程并行化执行, 循环变量只能是整型
- sections: 非迭代式共享任务并行化
- ordered: 指定在接下来的代码块中, 被并行化的for循环将依序运行

常用子句:

- private: 其列出来的变量对于线程私有

- firstprivate: private的功能且对于线程局部存储的变量,其初值是进入并行区之前的值
- lastprivate: private的功能且并行区里的值在最后会赋值给并行区前面的变量
- default: 自定义一个并行区的默认的变量的作用范围
- shared: 其列出来的变量对于所有线程共享
- reduction: 对于各个线程私有的变量,在并行区结束时通过某种运算归一(* + max min,以及逻辑运算和位运算)
- schedule: 线程调度,有 dynamic、guided、runtime、static四种方法
- num_threads: 设置线程数量的数量.默认值为当前计算机硬件支持的最大并发数

用private说明的变量,其在并行块外的值不能传入并行块,在并行块内计算出的值也不能传出并行块.为解决这个问题,可使用firstprivate和lastprivate.

7 MPI

MPICH is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard (MPI-1, MPI-2 and MPI-3).

https://computing.llnl.gov/tutorials/mpi/

高效进行消息传递是关键.

7.1 Hello World

```
#include    <stdio.h>
#include    <omp.h>
#include    <mpi.h>
int main(int argc, char *argv[]) {
    int nprocs,rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank == 0)
        printf("Hello World! Number of procs is %d\n",nprocs);
    printf("Hello World! This is %d processor\n",rank);
    MPI_Barrier(MPI_COMM_WORLD);
    /* allocating task */
    int nlocal, nglobal = 100, surplus;
    nlocal  = nglobal/nprocs;
    start   = nlocal*rank;
    surplus = nglobal-nlocal*nprocs;
    if (rank < surplus) {
        nlocal++;
        start += rank;
    }
    else {
        start += surplus;
    }
    /* start from 1 */
    start += 1;
```

```
27      end = start+nlocal;
28      printf("[%d]: nlocal=%d, nglobal=%d, start=%d, end=%d\n",
29              rank, nlocal, nglobal, start, end);
30      MPI_Barrier(MPI_COMM_WORLD);
31      int i, sum = 0;
32  #pragma omp parallel for reduction(+:sum) num_threads(2)
33      for (i = start; i <= end; ++i) {
34          sum += i;
35      }
36      printf("[%d]: before Allreduce, sum = %d\n", rank, sum);
37      MPI_Barrier(MPI_COMM_WORLD);
38      MPI_Allreduce(MPI_IN_PLACE, &sum, 1, MPI_INT,
39          MPI_SUM, MPI_COMM_WORLD);
40      printf("[%d]: after Allreduce, sum = %d\n", rank, sum);
41      MPI_Finalize();
42      return 0;
43  }
```

7.2 Sending and Receiving Messages

- `MPI_Bcast`
- `MPI_Scatter`, `MPI_Gather`
- `MPI_Reduce`
- `MPI_Allgather`, `MPI_Allreduce`
- `MPI_Allgatherv`
- `MPI_Iallreduce`, `MPI_Wait`
- `MPI_IN_PLACE`

7.3 Intel MPI Library

Intel MPI Library is a multifabric message-passing library that implements the open-source MPICH specification. Use the library to create, maintain, and test advanced, com-

plex applications that perform better on high-performance computing (HPC) clusters based on Intel processors.

- `icc`, `icpc`, `ifort`
- `mpiicc`, `mpiicpc`, `mpiifort`
- `mpiexec`

Remark 7.1. 值得注意的是, `mpiicc`, `mpiicpc`, `mpiifort`是收费的! 另外, 非常不建议, *intel* 编译器与 *GNU* 编译器混用. 推荐使用各自的`mpiexec`执行各自的 *MPI* 程序.

8 FFTW

FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms or DCT/DST).

In order to use FFTW effectively, you need to learn one basic concept of FFTW's internal structure: FFTW does not use a fixed algorithm for computing the transform, but instead it adapts the DFT algorithm to details of the underlying hardware in order to maximize performance. Hence, the computation of the transform is split into two phases. First, FFTW's planner "learns" the fastest way to compute the transform on your machine. The planner produces a data structure called a plan that contains this information. Subsequently, the plan is executed to transform the array of input data as dictated by the plan. The plan can be reused as many times as needed. In typical high-performance applications, many transforms of the same size are computed and, consequently, a relatively expensive initialization of this sort is acceptable. On the other hand, if you need a single transform of a given size, the one-time cost of the planner becomes significant. For this case, FFTW provides fast planners based on heuristics or on previously computed plans.

8.1 Sequential FFTW

```
#include <fftw3.h>
int main(int argc, char *argv[])
{
    const ptrdiff_t n = ...;
    fftw_complex *in, *out;
    fftw_plan p;
    ...
    in  = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*n);
    out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*n);
    p = fftw_plan_dft_1d(N,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
    ...
    fftw_execute(p); /* repeat as needed */
    ...
    fftw_destroy_plan(p);
    fftw_free(in); fftw_free(out);
```

```
}
```

In 1-dimension DFT, FFTW computes an unnormalized transform, in that there is no coefficient in front of the summation. In other words, applying the forward and then the backward transform will multiply the input by n. Precise mathematical definitions for the transforms that FFTW computes are provideed as follows: for $k = 0, \ldots, n-1$

$$z[k] = \sum_{j=0}^{n-1} w[j] \exp\left(\delta 2\pi i \cdot (\frac{jk}{n})\right)$$

where $\delta = -1$ for the forward transform, and $\delta = +1$ for the inverse (backward) transform.

If you have a C compiler (such as gcc) that supports the C99 revision of the ANSI C standard, you can use C's new native complex type. In particular, if you `#include <complex.h>` before `<fftw3.h>`, then `fftw_complex` is defined to be the native complex type and you can manipulate it with ordinary arithmetic (e.g. `x=y*(3+4*I)`, where `x` and `y` are `fftw_complex` and `I` is the standard symbol for the imaginary unit).

The multi-dimensional arrays passed to `fftw_plan_dft` are expected to be stored as a single contiguous block in row-major order. To be more explicit, let us consider an array of rank 3 whose dimensions are $n_0 \times n_1 \times n_2$. Now, we specify a location in the array by a sequence of 3 (zero-based) indices, one for each dimension: (i_0, i_1, i_2). If the array is stored in row-major order, then this element is located at the position $i_2 + n_2 \times (i_1 + n_0 \times i_0)$.

Planner Flags:

- `FFTW_FORWARD FFTW_BACKWARD`
- `FFTW_ESTIMATE FFTW_MEASURE FFTW_PATIENT FFTW_EXHAUSTIVE`

8.2 Parallel FFTW

```
#include <fftw3-mpi.h>
int main(int argc, char *argv[])
{
    const ptrdiff_t N0 = ..., N1 = ...;
    fftw_plan plan;
    fftw_complex *data;
    ptrdiff_t alloc_local, local_n0, local_0_start, i, j;
```

```
    MPI_Init(&argc,&argv);
    fftw_mpi_init();
    /* get local data size and allocate */
    alloc_local = fftw_mpi_local_size_2d(N0,N1,MPI_COMM_WORLD,
        &local_n0,&local_0_start);
    data = fftw_alloc_complex(alloc_local);
    /* create plan for in-place forward DFT */
    plan = fftw_mpi_plan_dft_2d(N0,N1,data,data,MPI_COMM_WORLD,
        FFTW_FORWARD,FFTW_ESTIMATE);
    /* initialize data to some function my_function(x,y) */
    for (i = 0; i < local_n0; ++i)
        for (j = 0; j < N1; ++j)
            data[i*N1+j] = my_function(local_0_start+i,j);
    /* compute transforms, in-place, as many times as desired */
    fftw_execute(plan);
    fftw_destroy_plan(plan);
    MPI_Finalize();
}
```

You can call `fftw_mpi_local_size_2d` to find out what portion of the array resides on each processor, and how much space to allocate. Here, the portion of the array on each process is a `local_n0` by `N1` slice of the total array, starting at index `local_0_start`. The total number of `fftw_complex` numbers to allocate is given by the `alloc_local` return value, which may be greater than `local_n0*N1` (in case some intermediate calculations require additional storage).

In particular, FFTW uses a 1-dimension block distribution of the data, distributed along the first dimension. For example, if you want to perform a 100×200 complex DFT, distributed over 4 processes, each process will get a 25×200 slice of the data. That is, process 0 will get rows 0 through 24, process 1 will get rows 25 through 49, process 2 will get rows 50 through 74, and process 3 will get rows 75 through 99.

9 CUFFT

In $\mathbb{R}^3$,

$$i\frac{\partial\varphi}{\partial t} = \Big(-\varepsilon\Delta + V + \beta|\varphi|^2 + \lambda\Phi - \omega L_z\Big)\varphi$$

where $\Phi = U_{dip} * |\varphi|^2$ and $L_z = -i(x\partial_y - y\partial_x) = -i\partial_\theta$.

```
OperateLaplace(cufftComplex *laplace_wave,
    cufftComplex *wave, cufftReal *epsilon, int add_insert);
OperateExternalPotential(cufftComplex *ext_pot_wave,
    cufftComplex *wave, int add_insert);
OperateTwoBodyInteraction(cufftComplex *two_body_wave,
    cufftComplex *wave, cufftReal *beta, int add_insert);
OperateDDI(cufftComplex *ddi_wave,
    cufftComplex *wave, cufftReal *lambda, int add_insert);
OperateRotation(cufftComplex *rot_wave,
    cufftComplex *wave, cufftReal *omega, int add_insert);
```

- $\rho_0 \to \hat{\rho}_0 \to \hat{\Phi}_0 = \hat{\rho}_0\hat{U} \to \Phi_0$
- $\varphi_0 \to \varphi_{\frac{1}{2}}$ solving $i\dfrac{\partial\varphi}{\partial t} = V\varphi + \beta|\varphi_0|^2\varphi + \lambda\Phi_0\varphi$
- $\varphi_{\frac{1}{2}} \to \varphi_{\frac{3}{2}}$ solving $i\dfrac{\partial\varphi}{\partial t} = -\varepsilon\Delta\varphi$
- $\rho_{\frac{3}{2}} \to \hat{\rho}_{\frac{3}{2}} \to \hat{\Phi}_{\frac{3}{2}} = \hat{\rho}_{\frac{3}{2}}\hat{U} \to \Phi_{\frac{3}{2}}$
- $\varphi_{\frac{3}{2}} \to \varphi_2$ solving $i\dfrac{\partial\varphi}{\partial t} = V\varphi + \beta|\varphi_{\frac{3}{2}}|^2\varphi + \lambda\Phi_{\frac{3}{2}}\varphi$

计算 Φ 时, 需要对 $|\rho|^2$ 进行零延拓, 每个方向的节点为原来的两倍. 有效部分在中间, 两边是零. 然后做 FFT, 并于 $\hat{U}$ 做点乘, 然后再 $IFFT$. 在访问有效数据时, 原点需要进行平移.

容易验证,

$$\begin{aligned}
\varphi(x,t) &= \int_{\mathbb{R}^3} \psi(\omega,t)\cdot\exp(+i\omega\cdot x)\,d\omega \\
\frac{\partial\varphi}{\partial t}(x,t) &= \int_{\mathbb{R}^3} \frac{\partial\psi}{\partial t}(\omega,t)\cdot\exp(+i\omega\cdot x)\,d\omega \\
\Delta\varphi(x,t) &= \int_{\mathbb{R}^3} -|\omega|^2\psi(\omega,t)\cdot\exp(+i\omega\cdot x)\,d\omega
\end{aligned}$$

For

$$i\frac{\partial\varphi}{\partial t}=V\varphi+\beta|\varphi(x,t_s)|^2\varphi,$$

i.e.,

$$\varphi(x,t)=\varphi(x,t_s)\exp\Big(-i(t-t_s)(V(x)+\beta|\varphi(x,t_s)|^2)\Big).$$

For

$$i\frac{\partial\varphi}{\partial t}=-\varepsilon\Delta\varphi,$$

i.e.,

$$i\frac{\partial\psi}{\partial t}(\omega,t)=\varepsilon|\omega|^2\psi(\omega,t)$$

$$\psi(\omega,t)=\psi(\omega,t_s)\exp\Big(-i\varepsilon|\omega|^2(t-t_s)\Big).$$

9.1 离散形式

设 $(x,y,z)\in[a_1,b_1]\times[a_2,b_2]\times[a_3,b_3]$, 且

$$\begin{aligned}\varphi(x,y,z)=&\sum_{j_1=0}^{n_1/2}\sum_{j_2=0}^{n_2/2}\sum_{j_3=0}^{n_3/2}\psi[j_1][j_2][j_3](t)\exp\Big(+2\pi i\cdot(j_1\frac{x-a_1}{l_1}+j_2\frac{y-a_2}{l_2}+j_3\frac{z-a_3}{l_3})\Big)\\&+\sum_{j_1=-n_1/2+1}^{-1}\sum_{j_2=-n_2/2+1}^{-1}\sum_{j_3=-n_3/2+1}^{-1}\psi[n_1+j_1][n_2+j_2][n_3+j_3](t)\cdot\\&\exp\Big(+2\pi i\cdot(j_1\frac{x-a_1}{l_1}+j_2\frac{y-a_2}{l_2}+j_3\frac{z-a_3}{l_3})\Big)\end{aligned}$$

等式右端第二个求和式为

$$\begin{aligned}&\sum_{j_1=n_1/2+1}^{n_1-1}\sum_{j_2=n_2/2+1}^{n_2-1}\sum_{j_3=n_3/2+1}^{n_3-1}\psi[j_1][j_2][j_3](t)\cdot\\&\exp\Big(+2\pi i\cdot((j_1-n_1)\frac{x-a_1}{l_1}+(j_2-n_2)\frac{y-a_2}{l_2}+(j_3-n_3)\frac{z-a_3}{l_3})\Big)\\=&\sum_{j_1=n_1/2+1}^{n_1-1}\sum_{j_2=n_2/2+1}^{n_2-1}\sum_{j_3=n_3/2+1}^{n_3-1}\psi[j_1][j_2][j_3](t)\exp\Big(+2\pi i\cdot(j_1\frac{x-a_1}{l_1}+j_2\frac{y-a_2}{l_2}+j_3\frac{z-a_3}{l_3})\Big)\cdot\\&\exp\Big(+2\pi i\cdot(-n_1\frac{x-a_1}{l_1}-n_2\frac{y-a_2}{l_2}-n_3\frac{z-a_3}{l_3})\Big)\end{aligned}$$

其中, $l_i=b_i-a_i$, $i=1,2,3$. 取

$$x_{k_1}=a_1+k_1\Delta x,$$

$$y_{k_2}=a_2+k_2\Delta y,$$

$$z_{k_3} = a_3 + k_3 \Delta z,$$

其中, $k_i = 0, 1, \ldots, n_i - 1$, $\Delta x = l_1/n_1$, $\Delta y = l_2/n_2$, $\Delta z = l_3/n_3$, $i = 1, 2, 3$.

$$\begin{aligned}
\varphi(x_{k_1}, y_{k_2}, z_{k_3}, t) =& \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
\frac{\partial \varphi}{\partial t}(x_{k_1}, y_{k_2}, z_{k_3}, t) =& \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} \frac{\partial \psi[j_1][j_2][j_3]}{\partial t}(t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
\Delta \varphi(x_{k_1}, y_{k_2}, z_{k_3}, t) =& \sum_{j_1=0}^{n_1/2} \sum_{j_2=0}^{n_2/2} \sum_{j_3=0}^{n_3/2} -4\pi^2 \Big((\frac{j_1}{l_1})^2 + (\frac{j_2}{l_2})^2 + (\frac{j_3}{l_3})^2 \Big) \cdot \\
& \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
& + \sum_{j_1=n_1/2+1}^{n_1-1} \sum_{j_2=n_2/2+1}^{n_2-1} \sum_{j_3=n_3/2+1}^{n_3-1} -4\pi^2 \Big((\frac{j_1 - n_1}{l_1})^2 + (\frac{j_2 - n_2}{l_2})^2 + (\frac{j_3 - n_3}{l_3})^2 \Big) \cdot \\
& \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
\partial_x \varphi(x_{k_1}, y_{k_2}, z_{k_3}, t) =& \sum_{j_1=0}^{n_1/2} \sum_{j_2=0}^{n_2/2} \sum_{j_3=0}^{n_3/2} +2\pi i \Big(\frac{j_1}{l_1} \Big) \cdot \\
& \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
& + \sum_{j_1=n_1/2+1}^{n_1-1} \sum_{j_2=n_2/2+1}^{n_2-1} \sum_{j_3=n_3/2+1}^{n_3-1} +2\pi i \Big(\frac{j_1 - n_1}{l_1} \Big) \cdot \\
& \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
\partial_y \varphi(x_{k_1}, y_{k_2}, z_{k_3}, t) =& \sum_{j_1=0}^{n_1/2} \sum_{j_2=0}^{n_2/2} \sum_{j_3=0}^{n_3/2} +2\pi i \Big(\frac{j_2}{l_2} \Big) \cdot \\
& \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big) \\
& + \sum_{j_1=n_1/2+1}^{n_1-1} \sum_{j_2=n_2/2+1}^{n_2-1} \sum_{j_3=n_3/2+1}^{n_3-1} +2\pi i \Big(\frac{j_2 - n_2}{l_2} \Big) \cdot \\
& \psi[j_1][j_2][j_3](t) \exp\Big(+ 2\pi i \cdot (\frac{j_1 k_1}{n_1} + \frac{j_2 k_2}{n_2} + \frac{j_3 k_3}{n_3})\Big)
\end{aligned}$$

那么, $j_i = 0, 1, \ldots, n_i/2, i = 1, 2, 3,$

$$i\frac{\partial \psi[j_1][j_2][j_3]}{\partial t}(t) = 4\pi^2\varepsilon\Big((\frac{j_1}{l_1})^2 + (\frac{j_2}{l_2})^2 + (\frac{j_3}{l_3})^2\Big)\psi[j_1][j_2][j_3](t)$$

且 $j_i = n_i/2 + 1, \ldots, n_i - 1, i = 1, 2, 3,$

$$i\frac{\partial \psi[j_1][j_2][j_3]}{\partial t}(t) = 4\pi^2\varepsilon\Big((\frac{j_1 - n_1}{l_1})^2 + (\frac{j_2 - n_2}{l_2})^2 + (\frac{j_3 - n_3}{l_3})^2\Big)\psi[j_1][j_2][j_3](t)$$

9.2 程序实现

$512 \times 512 \times 512, \Delta t = 0.0001, T = 1.0$. 单精度浮点型, 耗时 279.66 秒. 双精度浮点型, 耗时 1050.70 秒.

```
int i, N[3] = {2, 2, 2};
int num_time_step = 100;
float T = 1.0;
float dt = T/num_time_step;
dim3 dimGrid (N[0], N[1]);
dim3 dimBlock(N[2]);
cufftComplex *wave, *wave_dev;
cufftReal *density, *density_dev;
cufftReal mass;
//在CPU上面分配内存
wave = (cufftComplex*)malloc(sizeof(cufftComplex)*N[0]*N[1]*N[2]);
density = (cufftReal*)malloc(sizeof(cufftReal)*N[0]*N[1]*N[2]);
//在GPU上分配内存
cudaMalloc((void**)&wave_dev,sizeof(cufftComplex)*N[0]*N[1]*N[2]);
cudaMalloc((void**)&density_dev,sizeof(cufftReal)*N[0]*N[1]*N[2]);
//赋初值
SetInitialCondition(wave,N);
cudaMemcpy(wave_dev,wave,sizeof(cufftComplex)*N[0]*N[1]*N[2],
        cudaMemcpyHostToDevice);
cufftPlan3d(&plan,N[0],N[1],N[2],CUFFT_C2C);
ComputeNonLinear<<<dimGrid,dimBlock>>>(wave_dev,0.5*dt);
for (ts = 0; ts < num_time_step-1; ++ts) {
   cufftExecC2C(plan,wave_dev,wave_dev,CUFFT_FORWARD);
   ComputeLinear<<<dimGrid,dimBlock>>>(wave_dev,1.0*dt);
```

```
25      cufftExecC2C(plan,wave_dev,wave_dev,CUFFT_INVERSE);
26      ComputeNonLinear<<<dimGrid,dimBlock>>>(wave_dev,1.0*dt);
27  }
28  cufftExecC2C(plan,wave_dev,wave_dev,CUFFT_FORWARD);
29  ComputeLinear<<<dimGrid,dimBlock>>>(wave_dev,1.0*dt);
30  cufftExecC2C(plan, wave_dev, wave_dev, CUFFT_INVERSE);
31  ComputeNonLinear<<<dimGrid,dimBlock>>>(wave_dev,0.5*dt);
32  mass = ComputeDensity<<<dimGrid,dimBlock>>>(density_dev,wave_dev);
33  cudaMemcpy(density,density_dev,sizeof(cufftReal)*N[0]*N[1]*N[2],
34          cudaMemcpyDeviceToHost);
35  cudaMemcpy(wave,wave_dev,sizeof(cufftComplex)*N[0]*N[1]*N[2],
36          cudaMemcpyDeviceToHost);
37  //释放 GPU 上的内存
38  cudaFree(wave_dev); cudaFree(density_dev);
39  //释放 CPU 上的内存
40  free(wave); free(density);
```

```
1   __device__
2   int GetGlobalIdx3D()
3   {
4       return (blockIdx.x*gridDim.y+blockIdx.y)*blockDim.x+threadIdx.x;
5   }
6   __device__
7   void GetCoordi3D(float coordi[3])
8   {
9       float dx = (X_UPPER-X_LOWER)/gridDim.x;
10      float dy = (Y_UPPER-Y_LOWER)/gridDim.y;
11      float dz = (Z_UPPER-Z_LOWER)/blockDim.x;
12      coordi[0] = X_LOWER+ blockIdx.x*dx;
13      coordi[1] = Y_LOWER+ blockIdx.y*dy;
14      coordi[2] = Z_LOWER+threadIdx.x*dz;
15      return;
16  }
17  __global__
18  void ComputeNonLinear(cufftComplex *data, float dt)
19  {
```

```
20      int globalIndex = GetGlobalIdx3D();
21      /* 每个线程指向属于自己的部分 */
22      cufftComplex *cache = data+globalIndex;
23      float coordi[3];
24      GetCoordi3D(coordi);
25      float v = ExternalPotential(coordi);
26      v += BETA*(cache->x*cache->x+cache->y*cache->y);
27      v *= -dt;
28      /* cache[cacheIndex] *= exp(v*1.0I); */
29      *cache = cuCmulf(*cache,make_cuFloatComplex(cosf(v),sinf(v)));
30      return;
31  }
32  __global__
33  void ComputeLinear(cufftComplex *data, float dt)
34  {
35      int globalIndex = GetGlobalIdx3D();
36      /* 每个线程指向属于自己的部分 */
37      cufftComplex *cache = data+globalIndex;
38      int m[3];
39      m[0] = ( blockIdx.x> gridDim.x/2)?
40              ( blockIdx.x- gridDim.x): blockIdx.x;
41      m[1] = ( blockIdx.y> gridDim.y/2)?
42              ( blockIdx.y- gridDim.y): blockIdx.y;
43      m[2] = (threadIdx.x>blockDim.x/2)?
44              (threadIdx.x-blockDim.x):threadIdx.x;
45      float v = m[0]*m[0]/(X_UPPER-X_LOWER)/(X_UPPER-X_LOWER)
46              +m[1]*m[1]/(Y_UPPER-Y_LOWER)/(Y_UPPER-Y_LOWER)
47              +m[2]*m[2]/(Z_UPPER-Z_LOWER)/(Z_UPPER-Z_LOWER);
48      v *= -4*M_PI*M_PI*EPSILON*dt;
49      /* cache[cacheIndex] *= exp(v*1.0I); */
50      cache->x /= (gridDim.x*gridDim.y*blockDim.x);
51      cache->y /= (gridDim.x*gridDim.y*blockDim.x);
52      *cache = cuCmulf(*cache,make_cuFloatComplex(cosf(v),sinf(v)));
53      return;
54  }
55  __global__
```

```
56 void ComputeDensity(cufftReal *density, cufftComplex *wave)
57 {
58     int globalIndex = GetGlobalIdx3D();
59     density[globalIndex] =
60             +wave[globalIndex].x*wave[globalIndex].x
61             +wave[globalIndex].y*wave[globalIndex].y;
62     return;
63 }
```

考虑将一些乘数先算出来,直接调用.

10 PETSc

Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It supports MPI, and GPUs through CUDA or OpenCL, as well as hybrid MPI-GPU parallelism.

PETSc includes a large suite of parallel linear solvers, nonlinear solvers, and time integrators that may be used in application codes written in Fortran, C, C++, and Python. PETSc provides many of the mechanisms needed within parallel application codes, such as parallel matrix and vector assembly routines. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem.

10.1 Linear Solver

$$Ax = b$$

10.2 Nonlinear Solver

$$F(x) = b$$

10.3 Time Integrator

The TS library provides a framework for the scalable solution of ordinary differential equations (ODEs) and differential algebraic equations (DAEs) arising from the discretization of time-dependent PDEs.

- ODEs
$$y' = f(t, y)$$
- DAEs
$$y' = f(t, y, z),\ 0 = g(t, y, z)$$
- Solving Steady-State Problems with Pseudo-Timestepping
$$G(u) = 0,\ u_t = G(u)$$

10.3.1 An Example

The user first creates a TS object with the command

```
int TSCreate(MPI_Comm comm,TS *ts);
int TSSetProblemType(TS ts,TSProblemType problemtype);
```

The TSProblemType is one of TS_LINEAR or TS_NONLINEAR. To set up TS for solving an ODE, one must set the "initial conditions" for the ODE with

```
TSSetSolution(TS ts, Vec initialsolution);
```

One can set the right hand side function of the ODE with

```
TSSetRHSFunction(TS ts,Vec R,
    PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),
    void *funP);
```

One can set the solution method with the routine

```
TSSetType(TS ts,TSType type);
```

Currently supported types are TSEULER, TSRK (Runge-Kutta), TSBEULER, TSCN (Crank-Nicolson), TSTHETA, TSGLLE (generalized linear), TSPSEUDO, and TSSUNDIALS (only if the Sundials package is installed), or the command line option -ts_type with

- euler,rk,beuler,cn,theta,gl
- pseudo,sundials,eimex,arkimex,rosw

Set the initial time with the command

```
TSSetTime(TS ts,PetscReal time);
```

One can change the timestep with the command

```
TSSetTimeStep(TS ts,PetscReal dt);
```

can determine the current timestep with the routine

```
TSGetTimeStep(TS ts,PetscReal* dt);
```

One sets the total number of timesteps to run or the total time to run (whatever is first) with the commands

```
TSSetMaxSteps(TS ts,PetscInt maxsteps);
TSSetMaxTime(TS ts,PetscReal maxtime);
```

and determines the behavior near the final time with

```
TSSetExactFinalTime(TS ts,TSExactFinalTimeOption eftopt);
```

where `eftopt` is one of

- `TS_EXACTFINALTIME_STEPOVER`
- `TS_EXACTFINALTIME_INTERPOLATE`
- `TS_EXACTFINALTIME_MATCHSTEP`

One performs the requested number of time steps with

```
TSSolve(TS ts,Vec U);
```

The solve call implicitly sets up the timestep context; this can be done explicitly with

```
TSSetUp(TS ts);
```

One destroys the context with

```
TSDestroy(TS *ts);
```

and views it with

```
TSView(TS ts,PetscViewer viewer);
```

In place of `TSSolve()`, a single step can be taken using

```
TSStep(TS ts);
```

11 TAO

The Toolkit for Advanced Optimization (TAO) focuses on the design and implementation of optimization software for solving large-scale optimization applications on high-performance architectures.

TAO includes a variety of optimization algorithms for several classes of problems (unconstrained, bound-constrained, and PDE-constrained minimization, nonlinear least-squares, and complementarity).

The TAO solvers use fundamental PETSc objects to define and solve optimization problems: vectors, matrices, index sets, and linear solvers.

11.1 PDE-Constrained Optimization

$$\begin{aligned} &\min_{u,v} f(u,v) \\ &\text{s.t. } g(u,v) = 0 \end{aligned}$$

where the state variable u is the solution to the discretized partial differential equation defined by g and parametrized by the design variable v, and f is an objective function.

11.2 Nonlinear Least-Squares

$$\min_x \|F(x)\|_2^2$$

11.3 Complementarity

$$\begin{aligned} F_i(x^*) \geq 0 \quad &\text{if } x_i^* = l_i \\ F_i(x^*) = 0 \quad &\text{if } l_i < x_i^* < u_i \\ F_i(x^*) \leq 0 \quad &\text{if } x_i^* = u_i \end{aligned}$$

11.4 Nonlinear Programming

$$\begin{aligned} &\min_x f(x) \\ &\text{s.t. } g(x) = 0 \end{aligned}$$

$$h(x) \geq 0$$
$$x^- \leq x \leq x^+$$

12 SLEPc

The Scalable Library for Eigenvalue Problem Computations (SLEPc) is a software library for the solution of large scale sparse eigenvalue problems on parallel computers. It is an extension of PETSc and can be used for linear eigenvalue problems in either standard or generalized form, with real or complex arithmetic. It can also be used for computing a partial SVD of a large, sparse, rectangular matrix, and to solve nonlinear eigenvalue problems (polynomial or general). Additionally, SLEPc provides solvers for the computation of the action of a matrix function on a vector.

12.1 Eigenvalue Problem

$$Ax = \lambda x,\ Ax = \lambda Bx$$

12.2 Singular Value Decomposition

$$A = U\Sigma V^H$$

12.3 Polynomial Eigenvalue Problems

$$P(\lambda)x = 0,\ (K + \lambda C + \lambda^2 M)x = 0$$

12.4 Nonlinear Eigenvalue Problems

$$T(\lambda)x = 0,\ (\lambda I + A + e^{-\tau\lambda}B)x = 0$$

13 A Journey from Matlab to SLEPc

13.1 Matlab

```
nrows = 100; ncols = 200;
alpha = 1.0; beta  = 1.0;
X  = rand(nrows,ncols);
Y  = rand(nrows,ncols);
Y  = alpha*X+beta*Y;
ip = X(:,2:ncols)' * Y(:,2:ncols);
```

13.2 C

13.2.1 gcc

13.2.2 icc

13.2.3 BLAS+LAPACK

```
subroutine dgemm (character                 TRANSA,
     character                              TRANSB,
     integer                                M,
     integer                                N,
     integer                                K,
     double precision                       ALPHA,
     double precision, dimension(lda,*)     A,
     integer                                LDA,
     double precision, dimension(ldb,*)     B,
     integer                                LDB,
     double precision                       BETA,
     double precision, dimension(ldc,*)     C,
     integer                                LDC
)
```

dgemm 的说明文档

13.3 OpenMP

13.4 MPI

每个进程按列存储矩阵 M, 且

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

将每个进程中的 D 进行归约求和.

```
/* 子矩阵块通讯, 全局变量 */
MPI_Datatype SUBMAT_TYPE;
MPI_Op SUBMAT_OP;
static int SUBMAT_TYPE_NROWS = 0;
static int SUBMAT_TYPE_NCOLS = 0;
static int SUBMAT_TYPE_LDA   = 0;
```

```
CreateMPIDataTypeSubMat(&SUBMAT_TYPE,nrows,ncols,ldIP);
int commute = 1;
MPI_Op_create((MPI_User_function*)user_fn_submat_sum,
    commute,&SUBMAT_OP);
/* 归约求和 1 个 SUBMAT_TYPE */
MPI_Allreduce(MPI_IN_PLACE,inner_prod,
    1,SUBMAT_TYPE,SUBMAT_OP,MPI_COMM_WORLD);
MPI_Op_free(&SUBMAT_OP);
DestroyMPIDataTypeSubMat(&SUBMAT_TYPE);
```

```
/* 子矩阵块创建 */
int CreateMPIDataTypeSubMat(MPI_Datatype *SUBMAT_TYPE,
    int nrows, int ncols, int ldA)
{
    /* int MPI_Type_vector(
            int count, int blocklength, int stride,
            MPI_Datatype oldtype, MPI_Datatype *newtype) */
    MPI_Type_vector(ncols,nrows,ldA,MPI_DOUBLE,SUBMAT_TYPE);
    MPI_Type_commit(SUBMAT_TYPE);
```

```
10      SUBMAT_TYPE_NROWS = nrows; SUBMAT_TYPE_LDA = ldA;
11      SUBMAT_TYPE_NCOLS = ncols;
12      return 0;
13  }
14  /* 子矩阵块销毁 */
15  int DestroyMPIDataTypeSubMat(MPI_Datatype *SUBMAT_TYPE)
16  {
17      MPI_Type_free(SUBMAT_TYPE);
18      SUBMAT_TYPE_NROWS = 0; SUBMAT_TYPE_NCOLS = 0;
19      SUBMAT_TYPE_LDA   = 0;
20      return 0;
21  }
22  /* 子矩阵块求和操作 */
23  void user_fn_submat_sum(double *in, double *inout,
24      int *len, MPI_Datatype *data_type)
25  {
26      int i, j; double *a, *b;
27      double one = 1.0; int inc = 1;
28      for (i = 0; i < *len; ++i) {
29          for (j = 0; j < SUBMAT_TYPE_NCOLS; ++j) {
30              a = in   +SUBMAT_TYPE_LDA*j;
31              b = inout+SUBMAT_TYPE_LDA*j;
32              daxpy(&SUBMAT_TYPE_NROWS,&one,a,&inc,b,&inc);
33          }
34      }
35  }
```

Here `data_type` is `SUBMAT_TYPE` and `len=1`. Moreover, the `data_type` argument is a handle to the data type that was passed into the call to `MPI_Allreduce`. The user reduce function should be written such that the following holds: Let `u[0]`, ..., `u[len-1]` be the `len` elements in the communication buffer described by the arguments `in`, `len`, and `data_type` when the function is invoked; let `v[0]`, ..., `v[len-1]` be `len` elements in the communication buffer described by the arguments `inout`, `len`, and `data_type` when the function is invoked; let `w[0]`, ..., `w[len-1]` be `len` elements in the communication buffer described by the arguments `inout`, `len`, and `data_type` when the function returns; then `w[i] = u[i] o v[i]`, for `i=0 ,..., len-1`,

where o is the reduce operation that the function computes.

Informally, we can think of `in` and `inout` as arrays of `len` elements that function is combining. The result of the reduction over-writes values in `inout`, hence the name.

13.5 MPI+OpenMP

13.6 SLEPc

```
/* set random values */
BVSetActiveColumns(X,0,ncols);
BVSetRandom(X);
BVSetActiveColumns(Y,0,ncols);
BVSetRandom(Y);
/* Y = alpha*X+beta*Y */
BVSetActiveColumns(X,0,ncols);
BVSetActiveColumns(Y,0,ncols);
BVMult(Y,alpha,beta,X,NULL);
/* ip = X^H*Y */
BVSetActiveColumns(X,1,ncols);
BVSetActiveColumns(Y,1,ncols);
BVSetMatrix(X,NULL,PETSC_FALSE);
BVSetMatrix(Y,NULL,PETSC_FALSE);
Mat dense_mat; const double *ip;
MatCreateSeqDense(PETSC_COMM_SELF,ncols,ncols,NULL,&dense_mat);
BVDot(Y,X,dense_mat);
MatDenseGetArrayRead(dense_mat,&ip);
...
MatDenseRestoreArrayRead(dense_mat,&ip);
```

14 FreeFEM

FreeFEM is a popular 2D and 3D partial differential equations (PDE) solver used by thousands of researchers across the world. It allows you to easily implement your own physics modules using the provided FreeFEM language. FreeFEM offers a large list of finite elements, like the Lagrange, Taylor-Hood, etc., usable in the continuous and discontinuous Galerkin method framework.

As a high level multiphysics finite element software, FreeFEM offers a fast interpolation algorithm and a language for the manipulation of data on multiple meshes.

15 PHG

Parallel Hierarchical Grid (PHG) is a toolbox for developing parallel adaptive finite element programs.

PHG deals with conforming tetrahedral meshes and uses bisection for adaptive local mesh refinement and MPI for message passing. PHG has an object oriented design which hides parallelization details and provides common operations on meshes and finite element functions in an abstract way, allowing the users to concentrate on their numerical algorithms.