# SUNTANS代码结构

## 主程序

main(){

StartMpi() // 启动MPI

ParseFlags() // 读取命令参数 -t -g -s -vvv

if(GRID){

GetGrid() // -t 使用Triangle库生成三角网格

}else{

ReadGrid() //读入Triangle或Gambit生成的三角网格

}

if(SOLVE){ // -s (形参)

ReadProperties() //从suntans.dat读取参数到求解器

InitializeVerticalGrid //初始化垂向分层，dzf, dzz, dzzold

AllocatePhysicalVariables() //分配物理变量的内存空间

AllocateTransferArrays() //分配MPI通信的数组内存空间

OpenFiles() // 打开计算结果输出文件 fopen

if(RESTART){ // -r 热启动

ReadPhysicalVariables() // 读取热启动的中间结果文件

}else{

InitializePhysicalVariables() //初始化物理变量的值

Solve() // 核心计算程序，包含时间迭代

}

}

EndMpi() // 结束MPI

}

## 网格计算(grid.c)

Np, Ne, Nc // 节点、边和单元的个数

【1】读取网格(GetGrid)的程序流程包括以下步骤：

ReadFileName(myproc); //读取suntans.dat中的网格文件名

InitMainGrid // 初始化主网格（没有分解的整体网格），形成(\*grid)->Np等，在该子程序中，分配整体网格相关数组的内存空间，调用SunMalloc

主网格变量的形式为：(\*grid)->Np

然后，读取全局网格的数据：ReadMainGrid(maingrid,myproc);或者读取Triangulate生成的三角网格(GetTriangulation)，其中：ReadMainGrid读取网格的节点坐标等数据。

然后，统计Voronoi图（VoronoiStats）

然后，计算网格的连接关系（Connectivity）

然后，执行水平网格的区域分解（Partition），包括：

初始化局部网格(InitLocalGrid)

创建单元的图（CreateCellGraph）

使用ParMETIS执行区域分解(GetPartitioning)

生成拓扑关系(Topology)

将全局网格数据转移到局部网格(TransferData)

创建网格边的标记(EdgeMarkers)

读取垂向网格(VertGrid)

计算边长、Voronoi距离和面积等(Geometry)

创建进程间通信用的指针(MakePointers)

信息报告(ReportConnectivity, ReportPartition)

其中：

注意TransferData中，包含了很多分解后的数组和映射关系等。例如：局部映射关系指针（lcptr、leptr）、(\*localgrid)->Ne、……

【2】还可以直接读取分解后的网格：ReadGrid，必须要启用正确的进程个数。

## 读取属性文件(ReadProperties)

ReadProperties(&prop,grid,myproc);

调用MPI\_GetValue函数，读取suntans.dat中的计算参数。

## 初始化垂向网格(InitializeVerticalGrid)

InitializeVerticalGrid(&grid,myproc);

// give space and initialize dzf(edge) dzz(center) dzzold(center)

dzf—边上存储的分层厚度；

dzz和dzzold---新旧时刻的节点上存储的分层厚度。

变量还有：

(\*grid)->stairstep, fixdzz, dzsmall, smoothbot, dzfB, dzbot,…

## 分配使用的物理变量的内存空间(AllocatePhysicalVariables)

// allocate physical structure

\*phys = (physT \*)SunMalloc(sizeof(physT),"AllocatePhysicalVariables");

// allocate variables in plan

(\*phys)->u = (REAL \*\*)SunMalloc(Ne\*sizeof(REAL \*),"AllocatePhysicalVariables");

如何在GPU上分配物理变量的内存空间？

## 分配全局到局部的转移数组空间(AllocateTransferArrays)

发送和接收进程间的数据。

在sendrecv.c程序中。

(\*grid)->status = (MPI\_Status \*)SunMalloc(2\*(\*grid)->Nneighs\*sizeof(MPI\_Status),"AllocateTransferrays");

(\*grid)->request = (MPI\_Request \*)SunMalloc(2\*(\*grid)->Nneighs\*sizeof(MPI\_Request),"AllocateTransferrays");

变量还有：

recv, send, total\_cells\_send, total\_cells\_recv, total\_cells\_sendW, total\_cells\_recvW, total\_edges\_send, total\_edges\_recv

for(neigh=0;neigh<(\*grid)->Nneighs;neigh++) {

// 还有一些数组，函数中没有调用，看来可以删掉

}

FreeTransferArrays // 释放相关数组的空间

sendrecv.c的其他一些用于并行分区信息交互的函数，有：

SendRecvCellData2D // 进程间转移2D单元数据

ISendRecvCellData2D // SendRecvCellData2D的非阻塞版本

ISendRecvCellData3D //进程间转移3D单元数据

ISendRecvWData //非阻塞发送/接收3D 垂向流速(w)数据

ISendRecvEdgeData3D //非阻塞发送/接收3D边的数据

CheckCommunicateCells //检查合适的进程间单元的发送和接收

CheckCommunicateEdges

## 打开输出文件(OpenFiles)

在physio.c：

if(prop->readSalinity && prop->readinitialnc == 0) {

MPI\_GetFile(filename,DATAFILE,"InitSalinityFile","OpenFiles",myproc);

prop->InitSalinityFID = MPI\_FOpen(filename,"r","OpenFiles",myproc);

}

。。。。。。

## 冷启动

InitializePhysicalVariables

初始化NetCDF文件；

UpdateDz;

ReturnFreeSurfaceNC

SetDensity

## 核心求解(Solve)

Solve()

{

ComputeConservatives() // 确保动量和质量守恒

MemoryStats() // -vv 屏幕输出使用的内存量 (Mb)

// 设置边界流速、开边界通量、开边界标量

// 初始化气象条件输出数据文件

// 。。。。。。

for(n=nstart;n<nsteps;n++){ // 时间层迭代

StoreVariables() // 存储旧时刻的变量值

SetFluxHeight() // 获取自由水位波动带来的通量值

HorizontalSource() // 计算水平源项（旧时刻的静水压力梯度、Coriolis力

UPredictor() // 计算预测的水平流速（静水压力）-> call CGSolver()计算自由水位

blowup = CheckDZ // 保证垂向分层间距>=0，且当wetdry!=0时自由水位不穿过单元

Continuity() //求解连续方程，得到垂向流速w： Eq.82

EddyViscosity() //计算涡粘性系数 (MY25湍流封闭模型)

UpdateAge() // if calcage>0, 更新age(passive)标量浓度

if(gamma){ // gamma>0，更新计算温度场

getTsurf() //获取水面温度

HeatSource() //计算热源

UpdateScalars() //更新计算标量

getchangeT() //计算水面温度的变化

}

updateAirSeaFluxes() // 更新大气-海面通量

if(beta) { } // 计算盐度，跟温度计算差不多，考虑了大气通量

if(computeSediments) { } // 计算泥沙输移

if(nonhydrostatic && !blowup){ // 计算垂向动量和非静水压力

WPredictor() // 计算预测的垂向流速: phys->w

ComputeQSource() // 计算Poisson压力方程的源项, physc->stmp

CGSolveQ() // 求解非静水压力，使用CG法求解

Corrector() // 使用非静水压力修正非静水压力场下的速度场

}

Continuity() // 代入垂向流速，检验是否满足连续方程

BoundaryScalars() // 设置新时刻(n), 标量和风应力的边界值

WindStress() // 风应力

SetDragCoefficients() // 设置阻力系数

if(gamma || beta) SetDensity() //如果计算了温度或盐度，就重新计算密度

ComputeUC() // 计算新时刻所有网格边上的流速: uc, vc

if(newcells) NewCells() //考虑水位波动后表层单元的动量变化

if(calcaverage) UpdateAverageVariables(); UpdateAverageScalars()

Check() // 检查程序是否blowup

OutputPhysicalVariables() or WriteOutputNcmerge() // 输出结果(二进制格式或netCDF格式：二进制为合并的结果，nc格式可合并也可单独输出)

if(calcaverage) WriteAverageNCmerge() //平均后的数组输出(只有nc格式)

InterpData() // 先确定各进程上给定输入点的最近点，然后以一定的输出频率(suntans.dat)插值到输出结果的数组上

Progress() // 输出计算进度信息

} //时间层迭代

}

### StoreVariables

存储旧时刻的变量值。

for(i=0;i<grid->Nc;i++)

for(k=0;k<grid->Nk[i];k++) {

phys->stmp3[i][k]=phys->s[i][k];

phys->wtmp2[i][k]=phys->w[i][k];

}

for(j=0;j<grid->Ne;j++) {

phys->D[j]=0;

for(k=0;k<grid->Nke[j];k++)

phys->utmp[j][k]=phys->utmp2[j][k]=phys->u[j][k];

}

## 泥沙输移(ComputeSediments)

计算泥沙输移和河床演变的模块，被phys.c调用

ComputeSediments()

{

if(n==1+nstart){ // 第一个时间步

sediments = (sedimentsT \*)SunMalloc(); //泥沙结构体

ReadSediProperties(); // 读取泥沙属性数据

OpenSediFiles(); // 打开存储泥沙计算结果文件

} // 泥沙初始化

CalculateErosion(); // 计算侵蚀量

for(k=0;k<size;k++){ // 分组计算悬沙

SedimentSource(); // 计算悬移质泥沙浓度场

SedimentVerticalVelocity(); //计算悬沙垂向运动速度： w + ws

CalculateSediDiffusivity(); // 计算悬沙的扩散系数

UpdateScalars(); //跟其他标量一样，更新浓度值

SedimentVerticalVelocity(); //计算悬沙垂向运动速度：w - ws

}

if(WSconstant==0) SettlingVelocity(); // 计算絮凝条件下的泥沙沉速

if(bedInterval) BedChange(); // 河床演变计算

BoundarySediment() ; // 获取下一时刻悬沙的边界条件值

OutputSediment(); // 输出泥沙计算结果到文件

}

CalculateErosion(){

// 分为soft erosion hard erosion

}

CalculateDeposition(){

// 基于泥沙沉速和悬沙浓度

}

## SUNTANS的一些网格变量

使用参数-t运行suntans，将产生3个文件：points, cells ,edges

edges文件包含5列数：



Point1和Point2是节点(points)的标记

Marker是边的标记（Marker=0表示内部计算边；=其他，表示不同类型的边界）

Voronoi1和Voronoi2是Voronoi点的标记，也是Voronoi边的端点，与Delaunay边相交。



Voronoi点对应网格的单元数(cells)

cells文件包含Nc行，8列数字：



xv, yv是存储于Voronoi点上的xy坐标值；

Point1, Point2, Point3对应points文件中的节点标记

Neigh1, Neigh2, Neigh3表示相邻三角单元的标记；对应ghost单元的标记为-1；对于没有计算边界外部的相邻单元有：

在各子进程上输出以下3个文件（区域分解后），共后面读取计算：

nodes包含网格全局节点的xy坐标、节点周围的节点个数（是变化的！）、节点周围的边的个数和边的编号（是变化的！）、节点周围的单元个数和编号（也是变化的！）

celldata包含每个单元的Voronoi点的网格数据，有Nc行数（Nc是各子进程上的单元数，包含intra-processor ghost点）。各行数字是：



xv, yv是存储于Voronoi点上的xy坐标值；

Ac是单元面积；

dv是Voronoi点上的水深；

Nk是垂向分层数；

Edge{1-3}是对应单元(faces)的3条边；

Neigh{1-3}是3个相邻单元的编号；

N{1-3}是各网格边上的外法向；

def{1-3}是Voronoi点到网格边的距离。

edgedata包含边中心处的变量，但热启动没有读取该文件。

网格变量见grid.c的InitMainGrid

注意：在程序中，edge表示Voronoi的边，可认为是连接相邻2个单元中心的连线；而face表示原网格单元的边数。

网格拓扑文件topo如下：



注意：

celldist是边界条件(0, 1, 2, 3)两侧的单元编号

edgedist是边界条件(0, 1, 2, 3)对应的边的编号

vertspace文件包含垂向分层信息，有Nkmax行，Nkmax是z分层数。

存储于边的变量

开辟空间：

(\*grid)->edges = (int \*) SunMalloc (NUMEDGECOLUMNS \* (\*grid)->Ne\* sizeof(int), "InitMainGrid");

(\*grid)->grad = (int \*)SunMalloc(2\*(\*grid)->Ne\*sizeof(int),"InitMainGrid");

(\*grid)->gradf = (int \*)SunMalloc(2\*(\*grid)->Ne\*sizeof(int),"InitMainGrid");

(\*grid)->eneigh = (int \*)SunMalloc(2\*((\*grid)->maxfaces-1)\*(\*grid)->Ne\*sizeof(int),"InitMainGrid");

1条边对应的2个端点（节点）编号: grid->edges

grid->grad 是celldata文件中定义的Voronoi节点的编号（因此此节点编号范围在0~Nc之间），Voronoi节点是原始网格的单元。

grid->gradf是该单元指向边(Edge{1-3})的指针，gradf{1, 2}的编号是0~2，表示Voronoi节点是指向某单元的哪条边，见下图中：



图1 水平网格单元等标记

存储于单元中心的变量

// nfaces array contains the number of faces in each cell 各单元的边数

(\*grid)->nfaces= (int \*)SunMalloc((\*grid)->Nc\*sizeof(int),"InitMainGrid");

// Pointers to xp,yp coordinates of vertices that make up polygons (0<cells<Np) 各单元的3个节点编号

(\*grid)->cells = (int \*)SunMalloc((\*grid)->maxfaces\*(\*grid)->Nc\*sizeof(int),"InitMainGrid");

// Pointers to neighboring cells (0<neigh<Nc) 各单元的相邻单元编号

(\*grid)->neigh = (int \*)SunMalloc((\*grid)->maxfaces\*(\*grid)->Nc\*sizeof(int),"InitMainGrid");

// Dot product of unique normal with outward normal 一个单元的3个face的外法向

(\*grid)->normal = (int \*)SunMalloc((\*grid)->maxfaces\*(\*grid)->Nc\*sizeof(int),"InitMainGrid");

// Indices of voronoi edges to cells 一条Voronoi边连接的2个单元编号

(\*grid)->grad = (int \*)SunMalloc(2\*(\*grid)->Ne\*sizeof(int),"InitMainGrid");

// Indices of pointers to faces of each cell 各单元的3条边(face)的编号

(\*grid)->face = (int \*)SunMalloc((\*grid)->maxfaces\*(\*grid)->Nc\*sizeof(int),"InitMainGrid");

// Indices to edges for momentum control volume 动量控制体的2条边的编号（相邻2个单元上的2条边）

(\*grid)->eneigh = (int \*)SunMalloc(2\*((\*grid)->maxfaces-1)\*(\*grid)->Ne\*sizeof(int),"InitMainGrid");

输出单元中心存储的变量

for(n=0;n<grid->Nc;n++) { // Nc-单元个数

fprintf(ofile,"%d %e %e %e %e %d ,grid->nfaces[n],grid->xv[n],grid->yv[n],grid->Ac[n],grid->dv[n],grid->Nk[n]);

for(nf=0;nf<grid->nfaces[n];nf++)

fprintf(ofile,"%d ",grid->face[grid->maxfaces\*n+nf]); // 各单元的3条边的编号

for(nf=0;nf<grid->nfaces[n];nf++)

fprintf(ofile,"%d ",grid->neigh[grid->maxfaces\*n+nf]); // 各单元的3个相邻单元编号

for(nf=0;nf<grid->nfaces[n];nf++)

fprintf(ofile,"%d ",grid->normal[grid->maxfaces\*n+nf]); // 各单元的3条边的法向量

for(nf=0;nf<grid->nfaces[n];nf++)

fprintf(ofile,"%e ",grid->def[grid->maxfaces\*n+nf]); // 各单元3条边对应的Voronoi点的距离

for(nf=0;nf<grid->nfaces[n];nf++)

fprintf(ofile,"%d ",grid->cells[grid->maxfaces\*n+nf]); // 各子进程上的单元个数(包含ghost cells)

fprintf(ofile,"%d ",grid->mnptr[n]); // global-to-local mapping pointer

fprintf(ofile,"\n");

}

//

for(n=0;n<Nc;n++) {

for(nf=0;nf<(\*grid)->nfaces[n];nf++) {

ne = (\*grid)->face[n\*(\*grid)->maxfaces+nf];

(\*grid)->def[n\*(\*grid)->maxfaces+nf] =

-(((\*grid)->xv[n]-maingrid->xp[(\*grid)->edges[ne\*NUMEDGECOLUMNS]])\*(\*grid)->n1[ne]+

((\*grid)->yv[n]-maingrid->yp[(\*grid)->edges[ne\*NUMEDGECOLUMNS]])\*(\*grid)->n2[ne])\*

(\*grid)->normal[n\*(\*grid)->maxfaces+nf];

// Check for nan

if((\*grid)->def[n\*(\*grid)->maxfaces+nf] != (\*grid)->def[n\*(\*grid)->maxfaces+nf])

printf("Warning: nan computed for edge distance (def)\n");

}

}

## SUNTANS模型中典型的循环计算

for(i=0;i<grid->Nc;i++) {// 对每个单元循环

for(nf=0;nf<grid->nfaces[i];nf++) { // for each face 对每个单元相邻的3个单元循环

。。。。。。

}

}

网格边的法向量：

for(i=0; i<grid->Ne; i++){

// 一条边的2个端点（节点）的编号

nodes[0] = grid->edges[NUMEDGECOLUMNS\*ie];

nodes[1] = grid->edges[NUMEDGECOLUMNS\*ie+1];

grid->n1[i]; // x direction normal

grid->n2[i]; // y direction normal

}

单元面积：Ac[Nc]

for(i=0; i<grid->Nc; i++) {

if((nc=grid->neigh[i\*grid->maxfaces+nf])!=-1) { // 相邻单元不是计算域外

sum+=grid->Ac[nc]; // 单元面积求和

}

}

单元的3个相邻单元编号：grid->neigh[maxfaces\*Nc]

## 模块计时

计时、函数计算和交换进程间单元数据，以U的预测计算为例：

t0=Timer();

UPredictor(grid,phys,prop,myproc,numprocs,comm);

ISendRecvCellData2D(phys->h,grid,myproc,comm);

t\_predictor+=Timer()-t0;

## 边界条件施加的代码

## MPI交换

某变量值更新了，就得交换单元上的值，例如：

UPredictor(grid,phys,prop,myproc,numprocs,comm);

ISendRecvCellData2D(phys->h,grid,myproc,comm);

# VS2013编译suntans遇到的一些问题及解决

（1）error4703

在VS2012创建项目时，会有一个勾选项，叫做“安全开发生命周期（SDL）检查”，这个东西是微软在VS2012新推出的东西，为了是能更好的监管开发者的代码安全，如果勾选上这一项，那么他将严格按照SDL的规则编译代码，会有一些以前常用的函数无法通过编译，比如在VS2010中的scanf是warning那么在VS2012中就是error了。

查看MSDN中对SDL的描述，SDL检查会视以下告警为错误：

从这些功能来看，有些功能还是不错的，不但方便了程序员使用也增强了程序的健壮性，但是对老版本程序的兼容就会出现问题，以前程序中编译通过的内容也许在VS2012中就没有办法通过。

问题解决方法：

有一个选择打开和关闭SDL检查的位置就是：项目属性->配置属性->C/C++->SDL检查，选测是或者否。

此时我们选择”否”，则上面的那个错误就不会出现了

（2）警告关闭，谨慎使用。

在suntans.h中使用：

#pragma warning(disable:4018 4101 4244) //LJ for VS2013

（3）Visual studio中没有strptime()函数

VS中关于时间的函数time()。使用田开发的代码解决这个问题。

（4）MSVCRTD.lib(crtexew.obj) : error LNK2019: unresolved external symbol[\_WinMain@16](http://wxiaolei.blog.163.com/blog/static/103876012009101791613792/_WinMain@16) referenced in function \_\_\_tmainCRTStartup

在网上搜了一下，做如下修改：  
Project -> xxxx Properties -> Configuration Properties -> Linker -> System ->Console ，原来是/subsystem:windows，这里改为(/SUBSYSTEM:CONSOLE)。  
修改后， 该问题解决。

（5）在VS2008工程中，经常出现warning LNK4099: 未找到 PDB“vc90.pdb等问题，解决办法如下：

1. 在项目属性—C/C++—输出文件—程序数据库文件名后面填写$(IntDir)\vc80.pdb。

2. 在项目属性—连接器—调试—生成调试信息后面改成“否”。（但是之后就不能进行调试了！）

（6）添加triangle库后，运行中断。

可以使用gambit生成网格，读入，不需要使用triangle库。因此，使用triangulate-notriangle.c

# CG法代码实例

/\*

\* Function: linsolve()

\* -----------------------------------

\* Solves a linear system of equations A.x=b

\*

\* A is a square matrix and b is a vector.

\* The solution x is returned in vector b

\*

\* Reference:

\* Golub and Van Loan, "Matrix Computations", 1999, Ch 3

\*/

void linsolve(REAL \*\*A, REAL \*b, int N){

int i,j,k;

REAL sumi;

// Algorithm to find LU decomp - See page 99

for(k=0;k<N-1;k++){

for(i=k+1;i<N;i++){

A[i][k] = A[i][k]/A[k][k];

for(j=k+1;j<N;j++){

A[i][j] = A[i][j] - A[k][j]\*A[i][k];

}

}

}

// Solve L.y=b via forward substitution (Alg 3.1.1);

b[0] = b[0];

for(i=1;i<N;i++){

sumi=0.0;

for(j=0;j<i;j++){

sumi = sumi + A[i][j]\*b[j];

}

b[i]=b[i]-sumi;

}

//Solve U.x=y via backward substitution (Alg 3.1.2)

b[N-1] = b[N-1]/A[N-1][N-1];

for(i=N-2;i>=0;i--){

sumi=0.0;

for(j=i+1;j<N;j++){

sumi = sumi + A[i][j]\*b[j];

}

b[i] = (b[i] - sumi)/A[i][i];

}

} // End of linsolve