

pnohs-alphs 参数率定开发者指南

北京科技大学 高性能计算与数据工程实验室
高性能计算课题组

编写: 储根深、吴佳豪

2018 年 9 月 17 日

目录

1	参数率定算法	1
1.1	基本概念及思想	1
1.1.1	子空间	1
1.1.2	自上向下率定	1
1.2	主从模式的参数率定算法	2
2	API	5
2.1	图: Graph	5
2.2	遍历: Traversing	6
2.3	图的元数据: Graph Meta [new]	7
2.4	水循环模拟 API	8
2.4.1	多次水循环模拟	8
2.5	参数率定 API	10
2.6	模拟结果的收集 [new]	11

1 参数率定算法

1.1 基本概念及思想

1.1.1 子空间

当前的参数率定算法采用主从模式，其中包含一个主进程和多个从进程。其中，我们将从进程分为 n 组，每个组称为一个子空间。在每个子空间上可以独立运行流域上的水循环模拟，这个子空间上执行的水循环模拟也是并行的，设各个子空间内都使用相等的进程数 m 来执行水循环模拟。

我们可以很容易知道，在参数率定时，使用的总进程数为： $n \times m + 1$ ，其中从进程数为 $n \times m$ 。主进程主要负责调用相关算法进行参数生成、参数率定的收敛性判断及主要的流程控制。

目前的方案中，在 n 个子空间中同时执行独立的流域水循环模拟，利用子空间的并行方式可以加速参数寻优。各个子空间上的区别只是传入模型的参数不同。

1.1.2 自上向下率定

由于流域的上下游依赖关系，上游的径流量汇入下游后，会影响下游的结果。所以，一般地，在进行参数率定时，采用自上游子流域到下游子流域率定的方式，即：先率定从最上游的子流域开始，率定上游子流域参数，率定好了之后，固定上游子流域的参数，然后再逐层率定下一层的子流域参数。

基于这种率定方式，我们每次只率定一个子流域（各个子空间同时率定该子流域），只要任意一个子空间上的参数率定达到收敛，其他子空间上的关于该子流域的率定过程即可终止。然后固定该子流域上的参数，所有子空间选择另一个下游的子流域进行率定，知道流域出口。

基于这种思想，基于主从模式的并行参数率定的大致算法将在1.2中进行阐述。

1.2 主从模式的参数率定算法

Algorithm 1 参数率定算法 (主进程)

INPUT: n - the count of sub-space

INPUT: SM - set of MASTER processors in all sub-space.

```

1: var  $id \leftarrow \text{null}$ 
2: var  $id\_next \leftarrow \text{null}$ 
3:  $\text{REC}(data = id\_next, from = SM)$ 
4: while  $id\_next \neq \text{null}$  do ▷ 1-1
5:    $id \leftarrow id\_next$ 
6:   while true do
7:     var  $Q_1, Q_2, \dots, Q_n \leftarrow 0$ 
8:     var  $i \leftarrow 0$ 
9:     for all  $sub\_spaces$  do
10:       $i \leftarrow i + 1$ 
11:      var  $params \leftarrow \text{GENPARAMS}(i, id)$  ▷ 1-2
12:       $\text{ISEND}(data = id, to = SM_i)$  ▷ 1-3
13:       $\text{ISEND}(data = params, to = SM_i)$  ▷ 1-4
14:       $\text{IREC}(data = Q_i, from = SM_i)$ 
15:       $\text{IREC}(data = id\_next, from = SM_i)$ 
16:    end for
17:     $\text{WAITALL}$  ▷ 1-5
18:    if  $\text{CONVERGENCE}(Q_1, Q_2, \dots, Q_n)$  then
19:      break
20:    end if
21:  end while
22: end while

```

Algorithm 2 参数率定算法 (从进程)

INPUT: m - the count of processors in this sub-space.

INPUT: $MASTER$ - global MASTER processor.

INPUT: SM - MASTER processor in this sub-space.

```

1: var  $id\_pre \leftarrow \text{null}$ 
2: var  $id \leftarrow \text{null}$ 
3: var  $id\_next \leftarrow \text{NEXTNODE}$ 
4: if IsSlaveMaster then
5:   SEND( $data = id\_next, from = MASTER$ )  $\triangleright$  initialize next_id on
   master. 2-1
6: end if
7: while  $id\_next \neq \text{null}$  do
8:   if IsSlaveMaster then
9:     REC( $data = id, from = MASTER$ )  $\triangleright$  2-2
10:    REC( $data = params, from = MASTER$ )  $\triangleright$  2-3
11:   end if
12:   BCAST( $data = id, root = SM, target = sub\_space$ )
13:   SCATTER( $data = params, root = SM, target = sub\_space$ )
14:   var  $Q = \text{SIMULATE}(id, params)$   $\triangleright$  2-4
15:   if IsSlaveMaster then
16:     SEND( $data = Q, to = MASTER$ )
17:   end if
18:   if  $id\_pre \neq id$  then
19:      $id\_next = \text{NEXTNODE}$   $\triangleright$  2-5
20:   end if
21:   if IsSlaveMaster then
22:     SEND( $data = id\_next, to = MASTER$ )
23:   end if
24:    $id\_pre \leftarrow id$ 
25: end while

```

注：该部分的伪代码仅做作为说明，实现的时候，可能有所不同 (例如算法中的点对点通信，在实现时会全局通信，如广播)。

2 API

本小节将围绕并行水循环程序 **pnohs-alpha**¹与并行水文模拟框架 **pnohs**², 介绍参数率定算法中可能会用到的 api。

为方便表述, 我们称除进行参数率定控制的进程称为**控制进程**或主进程, 执行水循环模拟的进程为**模拟进程**或子空间从进程。

2.1 图: Graph

```
#include<graph/graph.h>
```

在图相关的 api (如图的初始化、图中结点的获取等) 均只能是模拟进程进行调用。

1. 获取本地子图的结点 id

```
void Graph::getLocalGraphNodesIds(_type_node_id *ids)
```

用于得到该进程上的子图的所有结点 id, 结果存在数组 **ids** 中。

2. 获取本地子图的结点 id

```
std::vector<_type_node_id> Graph::getLocalGraphNodesIds()
```

和上面类似, 只是结点 id 的结果存储在向量中, 而非数组。

3. 获取各进程上的结点数 (MPI communication)

```
void Graph::globalNodesCount(_type_nodes_count *counts)
```

采用 **MPI_Allgather** 的方式将模拟域 (即子空间, 下同) 的所有进程上的结点收集到所有进程上的数组 **counts** 中。务必确保数组 **counts** 长度至少为模拟域内的进程数。

4. 获取各进程上的结点数 (MPI communication)

```
void Graph::globalNodesCount(_type_nodes_count *counts, kiwi  
::RID root)
```

¹ **Repository:** <https://git.hpcer.io/HPCer/hydrology/pnohs-alpha> & <https://git.gensh.me/HPCer/hydrology/pnohs-alpha>

² **Repository:** <https://git.hpcer.io/HPCer/hydrology/pnohs> & <https://git.gensh.me/HPCer/hydrology/pnohs>

采用 **MPI_Gather** 的方式将模拟域的所有进程上的结点收集到 root 进程的数组 counts 中。务必确保数组 counts 长度至少为模拟域内的进程数。除 root 进程外的其他进程的 count 数组可以为空。

5. 获取各进程上的结点 id 列表 (MPI communication)

```
void Graph::gatherNodesIds(_type_node_id *ids,
                           _type_nodes_count *counts)
```

以 **MPI_Allgatherv** 的方式，将模拟域内各个进程内所有结点的 id 收集到所有进程上的数组 ids 中，其中 counts 指定各个进程上的结点数。确保数组 ids 长度至少为模拟域内的全图的结点数。

6. 获取各进程上的结点 id 列表 (MPI communication)

```
void Graph::gatherNodesIds(_type_node_id *ids,
                           _type_nodes_count *counts, kiwi::RID root)
```

以 **MPI_Gatherv** 的方式，将模拟域内各个进程内所有结点的 id 收集到 root 进程上的数组 ids 中，其中 counts 指定各个进程上的结点数。确保数组 ids 长度至少为模拟域内的全图的结点数。除 root 进程外，其他进程的 ids 数组和 counts 数组可以为空。

2.2 遍历: Traversing

遍历过程主要是在全图（指分布在所有进程上的子图的拼接）中，每次返回一个入度为 0 的结点，返回该结点后，即将该结点以及与该结点相连的边删除。

这种遍历方式，与参数率定过程中的自上而下率定的思想恰好完全一致。需要注意的是，遍历相关的 api（如图的初始化、图中结点的获取等）均只能是模拟进程进行调用。如果控制进程需要相关数据，现有的方案只能是从进程调用 api，随后将结果通信发送给控制进程。

头文件：

```
#include<graph/graph_traversing.h>
```

1. 获取全图遍历中的下一个结点 id(MPI communication)

```
_type_node_id Traversing::nextNodeId()
```


该方法需要模拟域内所有的进程同时调用，不能是某个进程单独调用。该方法会优先从缓存中读取下一个结点 (减少通信开销)。该方法是类似于广播方式的，调用后，模拟域内的各个进程均能得到下一个结点的 id。

遍历完成后，继续调用该方法会返回空结点 id。

2.3 图的元数据: Graph Meta [\[new\]](#)

为了方便图的 api 的使用，我们在 graph api 的接口上进一步封装，形成 graph meta api。

```
#include "params/api_graph_meta.h"
```

GraphMeta 成员变量

1. 本地子图对象指针

```
Graph *graph
```

2. 全局结点数列表

```
_type_nodes_count *g_nodes_counts
```

这里，全局指的是一个子空间内的所有进程 (下同)，该指针指向一个一维数组的首元素。GraphMeta 对象初始化后，数组长度一般为子空间的进程数。数组的各个元素表示各个进程上的结点数，按照进程 rank id 顺序排列。

3. 全局结点总数

```
_type_nodes_count g_total_nodes_count
```

该成员值为数组 g_nodes_counts 种的各个元素值之和，表示全局的结点总数。

4. 本地结点总数

```
_type_nodes_count local_nodes_count
```

本地的结点总数，即当前进程上的结点总数。

5. 全局结点 id 列表

```
_type_node_id *g_nodes_id_lists
```

该指针指向一个一维数组的首元素，数组长度为全局各进程结点总数（见成员 `g_total_nodes_count`）。该数组存储全局的各个进程上的所有结点的 `id`。同一个进程上的所有结点 `id` 是位于连续的内存空间上的；不同进程的 `id` 列表按照进程 `rank id` 顺序排列。

<code>id list</code> of rank 0	<code>id list</code> of rank 1	...	<code>id list</code> of rank n-1
--------------------------------	--------------------------------	-----	----------------------------------

注意：目前的实现中，仅 `root` 进程可以访问该数组，其他进程上该指针值为 `nullptr`。

6. 本地结点 `id` 列表

```
std::vector<_type_node_id> *local_nodes_id_lists
```

当前进程上的结点 `id` 列表，结点排列顺序的未定义的。

GraphMeta 方法

1. 查询结点所在进程

```
kiwi::RID locateRank(const _type_node_id id)
```

依据结点 `id`，查询该结点所在的进程 `id`（返回进程在子空间的进程 `id`）；如果结点不存在，则返回 `ApiGraphMeta::RANK_NOT_FOUND`。

注意：目前的实现中，仅 `root` 进程调用该方法有效，其他进程调用则直接返回 `ApiGraphMeta::RANK_NOT_FOUND`。

2.4 水循环模拟 API

结合现有的需求及背景特点，将 `pnohs` 框架中的相关接口进一步封装，形成完整的水循环模拟流程。其中，相关接口将在下文进行描述。

2.4.1 多次水循环模拟

由于在参数率定中，需要重复调用水循环模拟过程，每次传递给模型不同的参数，一次模拟结束后，根据（子）流域出口的流量，进行收敛性判断等操作。

下面的示例中，展示了如何进行多次水循环模拟：

```

1 #include <logs/logs.h>
2 #include <utils/sim_domain.h>
3 #include <scheduler/ring_pickup.h>
4 #include "simulation.h"
5 #include "models/muskingum/muskingum_routing_model.h"
6 #include "models/xaj/xaj3_runoff_model.h"
7
8 int main(int argc, char *argv[]) {
9     // configure
10    ConfigValues cv;
11    cv.simulationTimeSteps = 6;
12    cv.pickupStrategy = RingPickup::Key;
13    cv.dispatchFilePath = "example/xaj-dispatch.dis";
14    // set communication domain.
15    domain::mpi_sim_process = kiwi::mpiUtils::global_process;
16
17    Simulation mSimulation(&cv);
18    NodesPool *pool = mSimulation.setupNodes();
19    mSimulation.loadModel(pool);
20    mSimulation.initScheduler(pool);
21    for (int i = 0; i < 10; i++) {
22        double *params = getYourParamsData(); // load your
23        parameters here.
24        // set model params.
25        unsigned long total_params_length = pool->nodes() * (14 + 1)
26        ; // lenght of array params
27        if (!mSimulation.passParams(params, total_params_length)) {
28            mSimulation.teardown();
29            delete params;
30            FAIL() << "error passing parameter count, which is " <<
31            total_params_length;
32        }
33        // ! end of setting models parameters.
34        mSimulation.startMessageLooper();
35        mSimulation.simulate(nullptr);
36        mSimulation.reset();
37        delete params;
38    }
39    mSimulation.teardown();
40    return 0;
41 }

```

Listing 1: 多次水循环模拟例程

在例程 Listing 1中，第9-13行是配置项，其中包括模拟时间步，调度算法，结点（子流域）划分到各个进程的划分文件等；第17-20行为创建水循环

模拟对象，各进程加载结点与模型以及初始化调度器；后面的 for 循环中，执行了 10 次水循环模拟过程（相当于参数率定过程中的 10 组参数优选），每一次水循环模拟结束后，需要进行重置操作 (line 33)，以便进行下一次模拟。

另外，Simulation::simulate 方法可以传递模拟结果对象的指针作为参数，用于在每完成一个结点的一个时间步的模拟时，进行回调。例如：

```
1 BaseSimResult returnData;
2 mSimulation.simulate(&returnData);
```

Listing 2: 模拟过程回调例程

在进行某个结点的某个时间步模拟的，如果回调函数被调用，则可以将对应结点该时间步的流量暂存起来。

Note: 由于在现有的代码中，进行参数率定时，配置项解析，Simulation 对象创建、加载结点、加载模型、初始调度等过程已经完成了。所以，在参数率定过程中，只需关心例程 Listing 1 for 循环中的代码即可，即只需在适当时候进行 for 循环中几个过程的调用。

为了适用参数率定，我们将水循环的多次模拟进一步封装，相关细节将在下一节中详述。

2.5 参数率定 API

参数率定调用水循环模拟的相关 API 位于 `src/params` 目录。该 API 的大部分内容均在 `ApiParams` 类中。

```
#include "params/api_params.h"
```

1. ApiParams 构造方法 (MPI communication)

```
ApiParams(Simulation *pSim, kiwi::RID root);
```

在该构造方法中，会初始化全局图信息，包括通信每个进程上的结点数和各个进程上的所有节点 id。同时，会初始化 Simulation 对象的引用以及设置该子空间的 root 进程（一般可以设为 rankid 为 0 的进程），以便后续调用水循环模拟。

2. 水循环模拟 (MPI communication) [\[update\]](#)

```
void simThisParams(_type_param params[], const size_t length,
    const size_t size, _type_node_id id, ParamsSimResult *
    paramsSimResult)
```

传递全局的模型参数 (即所有进程上的所有子流域上的 (产汇流) 模型参数), 进行一次水循环模拟。其中从进程中的 root 进程 (SM 进程) 的 params 数组不能为空, 其他进程的 params 数组可以为空, 该函数会根据各个进程上的节点数及其 id, 使用 MPI_Scatterv 的方式将参数分配到各个进程。length 参数是 params 数组长度, size 参数是一个结点上的水文参数个数; Q 是当前正在进行率定的子流域 id, paramsSimResult 在模拟过程的每一个时间步进行回调, 存储率定的子流域中各个时间步的径流量。

其中, 关于 ParamsSimResult 相关的文档请参阅[模拟结果](#)章节。

Note: 该 api 需要且仅限于空间内所有的从进程同时调用。

2.6 模拟结果的收集[\[new\]](#)

ParamsSimResult 类继承子类 **BaseSimResult**, 用于存储参数率定过程中的模拟结果。由于现有的参数率定方案中, 一次只率定一个子流域, 所以一次模拟结束后, 也仅返回一个子流域的所有时间步内的径流量。

对于一次模拟, 在 **ParamsSimResult** 类的实现中, 仅会保存当前正在率定的子流域的各个时间步的模拟结果, 而过滤掉其他子流域的结果。

需要注意的是, 当前率定的子流域只是在子空间内的某一个进程上 (设为进程 p_i), 所以, 仅进程 p_i 上有径流量值, 其他进程上是没有径流量值的。换言之, 使用 ParamsSimResult 的对象来接收模拟结果时, 仅进程 p_i 上的 ParamsSimResult 对象中的结果数据不为空, 而其他进程上的为空。

```
#include "params/params_sim_result.h"
```

1. ParamsSimResult 构造方法

```
ParamsSimResult(const _type_node_id id);
```

在参数率定场景下, 该构造方法中的 id 一般为当前正在率定的子流域的 id。

2. 模拟结果拷贝

```
bool flatNode(T *result_vec, const _type_node_id id, const unsigned long len)
```

将模拟结果拷贝到一维数组, 其中 result_vec 为数组首地址, id 为结点 id, len 为数组长度。如果模拟结果中存在 id 为 @param id 的结点

的模拟结果，则将数据拷贝到数组 `result_vec` 中，并返回 `true`；否则返回 `false`。

在参数率定中，参数 `id` 可以为当前正在率定的结点的 `id`。由于并不是所有的进程上都有当前率定的结点（仅子空间内的一个进程有 p_i ）。而其他非 p_i 进程调用该函数，则不会进行内存的拷贝，并返回 `false`。

结果收集之后

为了最终能够使得主进程得到当前率定的子流域的模拟结果，还需要进行通信将模拟结果发送给主进程。可以有两种通信方式：

1. 正在率定的子流域所在进程和子空间的主进程进行通信，使得子空间的主进程获得正在率定的子流域的模拟结果；随后，子空间主进程和主进程进行通信，使得主进程获得当前正在率定的子流域的模拟结果。
2. 当前正在率定的子流域所在进程直接和主进程进行通信，使得主进程获得当前正在率定的子流域的模拟结果。