【摘要】

海浪是海洋最常见的运动形式，对于航行安全和气候变化有至关重要的影响。高分辨率的全球海浪模型在精确的海浪预测中起着关键作用。MASNUM-WAM（Marine Science and Numerical Modeling - Wave Model)海浪数值模式是我国自主研发的第三代海浪数值模式。神威太湖之光是我国自主研发的超级计算机，其于2016年起登顶世界超算TOP500排行榜，是现在世界上最快的超级计算机。MASNUM-WAM 原版代码的并行效率不高，且并未针对神威太湖之光的SW26010处理器进行优化。本文的主要工作是将MASNUM-WAM原版代码移植上神威太湖之光超算平台，充分利用了SW26010主从核异构协同计算的特点，并且针对神威太湖之光的特性进行了一定的优化。本文首先通过对移植优化的任务瓶颈进行分析确定了工作的核心，然后针对热点和瓶颈进行逐步移植和优化。最终，本文通过用三个不同规模的算例进行测试，证明了移植和优化的代码效率和可拓展性。

【关键词】：海浪数值模拟；神威太湖之光；代码移植；优化

【Abstract】

Surface wave is the most energetic form of motions in the ocean, which is crucial to navigation safety and climate change. High-resolution global wave model plays a key role in accurate surface wave forecasting. However, operational forecasting systems in high-resolution are hard to carry out due to entailed high demand for large computation, as well as low parallel efficiency barrier. Marine Science and Numerical Modeling - Wave Mode (MASNUM-WAM) is a third-generation numerical wave model developed by the Key Laboratory of Marine Science and Numerical Modeling of State Oceanic Administration. Sunway Taihulight supercomputer is homegrown supercomputer in China, and it is the world's first supercomputer with a peak performance higher than 100 PFlops. It claims the top place in the latest TOP500 list since 2016 as the fastest supercomputer in the world. Thus, Sunway Taihulight is an ideal platform to provide high computation power demanded by high resolution MASNUM-WAM. On the one hand, the original code of MASNUM-WAM has poor performance when executed parallelly. On the other hand, it has no dedicated optimization targeted at processor SW26010 on Sunway Taihulight system. This paper mainly proposed the code transplantation dedicated to Sunway Taihulight which utilizes the heterogenous collaborative computing property of SW26010. Besides, this paper illustrates how dedicated optimization for SW26010 is performed. Firstly, this paper analysis the bottleneck of the program on Sunway Taihulight. Then, optimization and dedicated implementation are performed according to hotspots and bottlenecks. Lastly, three workloads are used to test the optimized code, which proves that the code of the paper has high efficiency and good scalability.

【Keywords】Numerical wave model; Sunway Taihulight; Code transplantation; Optimization

目录

第一章 引言

1.1.MASNUM-WAM 应用背景

海浪是最常见的海洋灾害现象之一。对于远洋工程、军事行动、海洋运输和海洋科考等海上活动具有重要影响。通过海浪模式对海浪进行数值预报，一直以来都是海浪预报的主要手段。[9][10]

MASNUM 海浪数值模式是国家海洋局第一海洋研究所发展的第三代全球海浪模式，是基于 LAGFD-WAM 海浪模式建立的球坐标体系下海浪数值模式。[1][6]

近年来，海浪数值模式对于精度和速度的要求越来越高。而分辨率每达到原来的两倍，计算量就要达到原来的四倍。因此，对于海洋数值模式的优化是非常重要的工作。[6][7][8]

1.2.神威太湖之光平台介绍

“神威·太湖之光”计算机系统是国家并行计算机工程技术研究中心在国家 863 计划支持下研制的新一代超级计算机系统。其具有 125PFlops 的理论峰值性能，在 Linpack 测试中取得了 93PFlops 的实测持续运算速度，与 2016 年登顶世界超算 TOP500。2016 年和 2017 年，神威太湖之光上的应用连续两年斩获戈登贝尔奖[3][4],共有五个引用获得戈登贝尔奖提名。

“神威·太湖之光”采用了由国家“核高基”重大专项支持的“申威 26010”众核处理器，该处理器由国家高性能集成电路设计中心采用自主核心技术研制，采用 64 位自主申威指令系统，具有 260 个核心。“神威·太湖之光”超级计算机具有 40960 个申威 26010 处理器，共具有 10649600 个处理器核心。

1.3.MASNUM-WAM 代码针对神威太湖之光的移植问题描述

MASNUM-WAM 原版代码是通用的 FORTRAN 代码，计算效率和可拓展性都不可观，且目前没有公开的针对神威太湖之光超级计算机的实现。

神威太湖之光主要的性能来自于从核的计算能力，从核代码支持 OpenACC 和

Athread 两种从核编程模式。OpenACC 通过编译制导语句实现，具体的从核代码将由编译器生成。Athread 则需要用户将主核代码和从核代码分开编写，显式地对从核线程进行控制。

OpenACC 具有编程简单的优点，只需要在代码中作少量修改，就可以取得成倍的加速效果。同时，OpenACC 也具有明显的缺点，即无法发挥设备的性能。此外，由于神威太湖之光处理器 SW26010 的从核 LDM（局部数据内存，即片上内存）较小，仅有 64KB，使用 OpenACC 的话将无法实现对 LDM 的高效使用，程序会陷入主从核之间大量内存拷贝的瓶颈。

Athread 线程库需要在主核代码显式创建从核线程，然后需要一份独立的从核代码，用于实现从核的计算、传输和控制。Athread 的缺点是，需要显式地控制数据的传输和存储，因此编程代码量将会非常大。

本文选用了 Athread 线程库来编写从核代码，从核计算部分通过 C 语言进行编写。

1.4. 本文的工作

本文主要的工作是实现了 MASNUM-WAM 代码在神威太湖之光上的高效迁移实现。基于原版的 MASNUM-WAM 代码，工作顺序是：首先实现算法上的优化，之后将计算部分的代码改写为 C 语言，然后将逐个函数通过 Athread 线程库迁移上从核，最后再代码的各方面进行优化，包括计算流程、负载均衡、计算等。

本文主要的的创新点主要在于实现了网格区域的划分，将计算步骤的依赖根据网格位置进行了解耦，从而实现对计算流程的拆分和重组，进而实现了高效的计算和通信的流水线，从而掩盖了通信的时间，实现了线性加速比。

最终，本文优化后的代码相比 MASNUM-WAM 原版代码在神威太湖之光上，在使用相同节点数量的情况下最多取得接近 500 倍的加速，优化后的代码也具有良好的可拓展性。

第二章 问题综述

2.1. MASNUM-WAM 问题描述

MASNUM-WAM 的核心等式是海浪能谱平衡方程[1]：

$$\frac{\partial E}{\partial \mathrm{t}}+\left(\frac{C_{g\lambda}+U_{\lambda}}{R\cos\varphi}\right)\frac{\partial E}{\partial \lambda}+\left(\frac{C_{g\varphi}+U_{\varphi}}{R}\right)\frac{\partial E}{\partial \varphi}-\frac{(C_{g\varphi}+U_{\varphi})\tan\varphi}{R}E=SS(E)$$

其中， $E=E(K,\ \lambda,\ \varphi,\ t)$ 是平均波动能量的波谱函数，φ 是纬度，λ 是经度。向量 U代表背景波流速度，向量 Cg 代表群速度。

方程的右侧由五个源函数组成：

$$SS=S_{in}+S_{ds}+S_{bo}+S_{nl}+S_{cu}$$

其中 S_{in}为风输入源函数，S_{ds}为破碎耗散原函数，S_{bo}为底摩擦耗散源函数，S_{nl}为非线性波波相互作用源函数，S_{cu}为波流相互作用源函数。

2.2. 神威太湖之光平台分析

神威太湖之光使用了 SW26010 众核处理器作为计算节点。SW26010 的架构[5]如图所示：

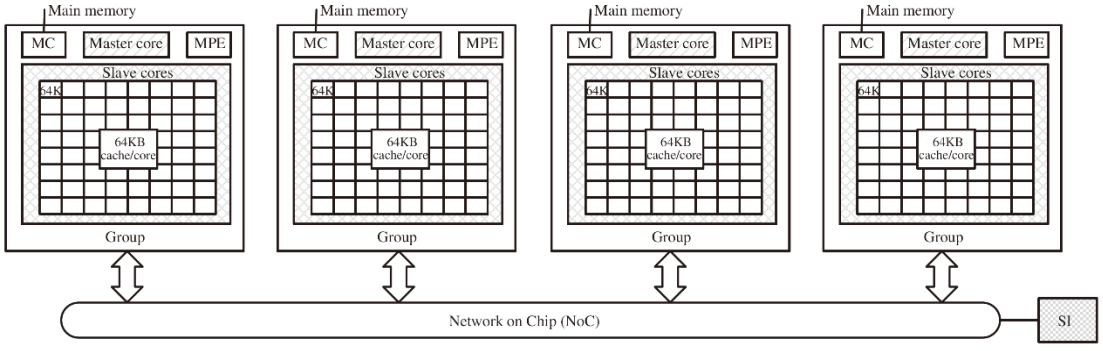

图 1 SW26010 处理器架构

每个 SW26010 处理器包含了 4 个核组。每个核组具有 1 个主核和 64 个从核。每个核组具有独立的 8GB 内存，以及一个内存控制器。四个核组通过片上网络进行互连。

主核和从核都是 RISC 架构的核心，其工作频率为 1.45GHz，都支持 256-bit

向量化指令和乘加指令。主核具有 64KB 的 L1 cache 和 256KB 的 L2 cache。从核具有 16KB 的 L1 指令 cache 和 64KB 的片上内存。

每个 SW26010 具有 4 个主核和 256 个从核，这 260 个核心提供了 3.06TFlops 的双精度浮点运算峰值性能，其中 96.97%由从核提供。因而，应用在神威太湖之光平台上的优化的核心在于利用从核的计算能力。

每个核组内，从核通过 DMA 指令，访问本核组或片上其余三个核组的 DDR3 主存。每个核组上 64 个从核的峰值 DMA 总带宽约为 30GB/s。这个带宽相对于 64 个从核的计算能力具有明显的差距，因此在将代码计算部分移植到从核之后，内存带宽将会成为瓶颈所在。核组内的 64 个从核组成了 8*8 的从核阵列，在行列方向上能够进行寄存器通信。寄存器通信的带宽非常高，能够达到数 TB/s，且时延非常低，因此充分利用从核阵列的寄存器通信实现从核间重复数据的共享，可以减少 DMA 的总量。

从核的浮点简单运算指令的时延为 7，高于常见的 Intel 处理器[1]，这意味着如果想要接近从核浮点流水线的峰值性能，需要进行循环展开等手段来避免指令等待。

[1] Intel 处理器简单浮点操作指令的时延一般为 1，乘加指令的时延一般是 5

第三章 任务分析

3.1. 计算热点

原版代码函数在 X86 架构[2]上通过 vtune 测得的的热点如图所示：

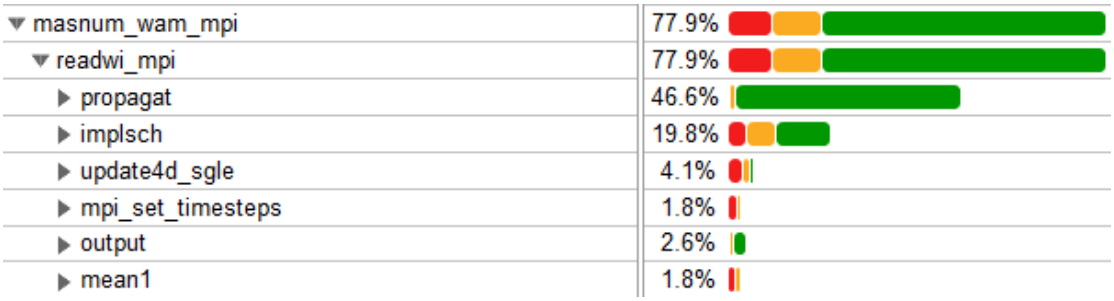

图 2 原版代码热点图

其中，propagat 函数的工作是计算格点坐标，并且对格点进行双线性插值，implsch 函数的工作是计算海浪能谱平衡方程右侧的 5 个源函数。

propagat 函数主要由两部分组成。前面的坐标计算部分主要用于计算待线性插值的波谱的空间位置及相位，后面的线性插值部分则利用前者得到的坐标进行线性插值。

函数 update4d_sgle 的工作是边缘交换。在每次迭代中，需要进行两次边缘交换。

从上图可知，代码的主要热点函数是 propagat 和 implsch。因此，代码计算部分的优化首先需要针对 propagat 和 implsch。此外，当代码的计算部分移植上从核之后，通信所占的时间比例将会大大提升，这时候将有必要进行通信的优化和异步化。

3.2. 代码计算流程分析

程序的核心数组是四维数组 ee。ee 用于存储进程负责的所有格点的波谱数据。原版程序中，还有临时数组 e 和 em，用于缓存各个计算过程中 ee 的计算结果。

原版代码每次迭代的主流程如表 1 所示：

[2] 测试平台为 Intel(R) Xeon(R) CPU E5-2620 v2

表 1 原版代码主流程

过程	意义	对 ee，e，em 的操作
get_wind	从磁盘中读取风力数值	
progaget	线性插值	ee(:, :, ia - 1 : ia + 1, ic - 1 : ic + 1) -> e(:, :, ia, ic)
implsch	海浪能谱平衡方程源函数计算	e(:, :, ia, ic) -> ee(:, :, ia, ic)
updatev	边缘交换	交换边缘的 ee
smooth	数值平滑	每个点的 ee 和其上下左右的 ee 进行平均得出 em，再将 em 拷贝回 ee
updatev	边缘交换	交换边缘的 ee
mean1	求解波浪特征量	ee(:, :, ia, ic) -> ee(:, :, ia, ic)

3.3. 负载均衡

代码的问题空间是以经度为横坐标，纬度为纵坐标的二维网格。网格中分为海洋格点和非海洋格点，其中海洋格点需要进行计算。因而，进程间的网格划分的依据是尽可能确保海洋格点均分，从而达到计算负载均衡。

代码的网格划分方式是，首先在 x 方向上进行划分得到若干矩形，然后在每个矩形内，在 y 方向上进行划分。这样，每个进程的计算区域将是一个矩形区域，在水平边界至多与一个进程通信，而垂直边界可能和若干个进程通信，如图：

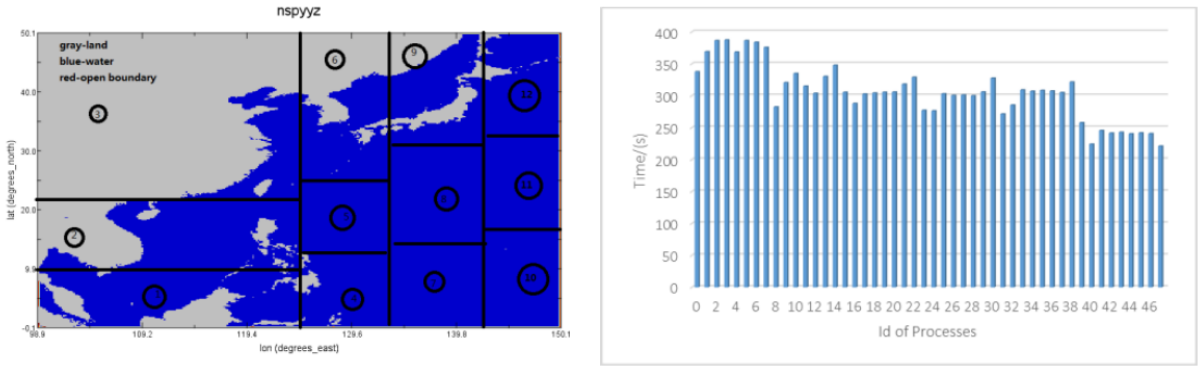

图 3 原版代码划分示例及结果

如图 3 所示，原版代码的划分效果并不好。

第四章 优化

4.1. 减少重复坐标计算

原版 MASNUM-WAM 的 propogat 函数主要完成两个工作：计算出将要进行线性插值的坐标，对目标坐标的点进行线性插值。在迭代中，线性插值的坐标是不会改变的。因此，坐标计算只需要在程序的开始进行一次。在删除了重复的坐标计算之后，propagat 函数的时间占比显著下降，如图 4 所示：

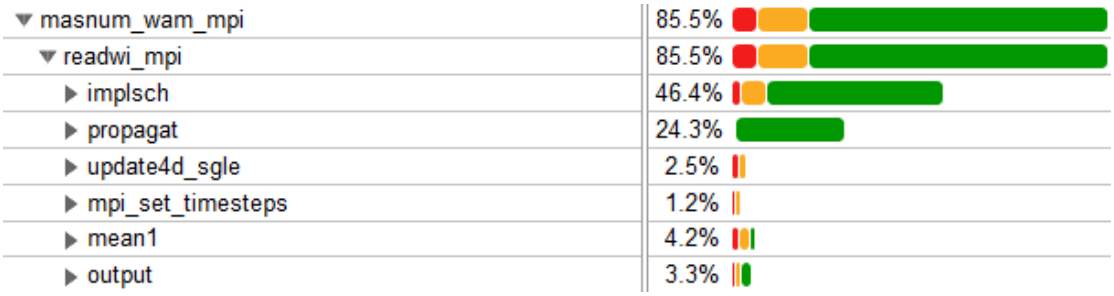

图 4 删除重复坐标计算后代码热点图

在删除了重复的坐标计算之后，implsch 变成了耗时最大的热点。在五个源函数中，非线性波波相互作用源函数 S_{nl} 耗时最长。

4.2. 删除多余的内存拷贝和数值检查

在上面的代码流程中，ee，e 和 em 数组的读入和拷贝次数非常多。在程序中，ee 数组的大小远大于 cache 的大小，这意味着每次从头读取 ee，e 和 em 都需要从 RAM 中加载，消耗了大量的内存带宽。

因此，通过重组计算过程，一方面去除了临时数组 em 的使用，另一方面通过交替使用 e 和 ee 来减少内存读取次数。此外，propaget 这个函数在删除重复的坐标计算后，其性能限制在于内存带宽，而 implsch 函数的限制在于计算量，通过将 progagat 和 implsch 过程进行结合，实现了访存和计算的 overlap，并且减少了 ee 的内存读取。

此外，smooth 和 mean1 过程的结合，简化了主流程，同时减少 ee 的读取，也允许了 e 和 ee 的交替使用。

修改后流程对于 e 和 ee 的操作如表 2 所示：

表 2 减少重复读写后流程

过程	对 ee，e 的操作
get_wind	
progaget + implsch	ee(:, :, ia - 1 : ia + 1, ic - 1 : ic + 1) -> e(:, :, ia, ic)
updatev	交换边缘的 e
smooth + mean1	每个点的 ee 和其上下左右的 ee 进行平均得出 e
updatev	交换边缘的 ee

此外，下面的一段代码在一次迭代中将会被执行 3 次：

```
do ic=iys,iyl
  do ia=ixs,ixl
    do k=1,kl
      do j=1,jl
        if(ee(k,j,ia,ic).eq.0.0)ee(k,j,ia,ic) = small
      enddo
    enddo
  enddo
enddo
```

这段代码会检测 ee 中每一个数值是否小于 0，如果是，则将其置为 small。上面的代码将会在内存中从头到尾将 ee 读取一遍，对于内存带宽的开销非常大。

在不改变计算结果的前提下，为了减少访存的开销，需要考虑当 ee 在 cache 中的时候，就完成逻辑判断操作。因此，在 propagat + implsch 和 smooth + mean1 计算出结果之后，就可以进行上面的逻辑判断操作。

4.3. 代码移植

为了高效使用从核上的 64KB 片上内存，从核代码最好用 C 语言实现，而 MASNUM-WAM 的源代码语言是 FORTRAN。为了方便代码向从核上的移植，首先进行的工作是将计算部分的代码从 FORTRAN 改写为 C。

此外，由于 SW26010 的计算能力中 96.97%由从核提供，故论文代码将 propagat，implsch，smooth 和 mean1 的所有计算部分都完全移植到从核上去，主核不进行计算。

从核代码在编写完成后并不会自动执行，而是需要主核代码进行调用。主核代码需要通过 athread_spawn 函数创建从核线程组，以调用从核代码，开始协同计算。

创建从核线程组需要数万个 clock cycle 的时间。为了减少创建线程组的开销，程序将会只使用一次 athread_spawn 函数用于创建从核线程组，在程序的最后通过 athread_join 来等待从核线程组运行结束。

此外，由于每个从核都使用自身的 64KB 片上内存，所以为了方便地修改从核计算的参数和指针，代码使用 FLAG 值中夹杂参数的方法。主核给从核传递的 flag 值包含计算所需的参数，从核写回主核的 flag 包含各种 profile 数据。主从核之间的 FLAG 是长度为 32 的长整数数组。数组的第 0 位是从 0 开始自增的数值，用于主核和从核之间鉴别对方的进度。从核 0 通过 DMA 指令从主核 flag，当 flag[0]数值符合要求的时候，则会通过寄存器通信将 flag 数组广播到 64 个从核。当所有从核完成计算过程之后，从核 0 则会将从核 flag 通过 DMA 写到主核，以告知主核计算已完成。

4.3.1. 寄存器通信实现

SW26010 的 8*8 从核阵列支持寄存器通信，包括行方向上和列方向上，具有广播模式和点对点模式两种寄存器通信模式。本文代码中仅使用了广播通信模式，用来实现从从核 0 到所有从核的广播。算法如图所示：

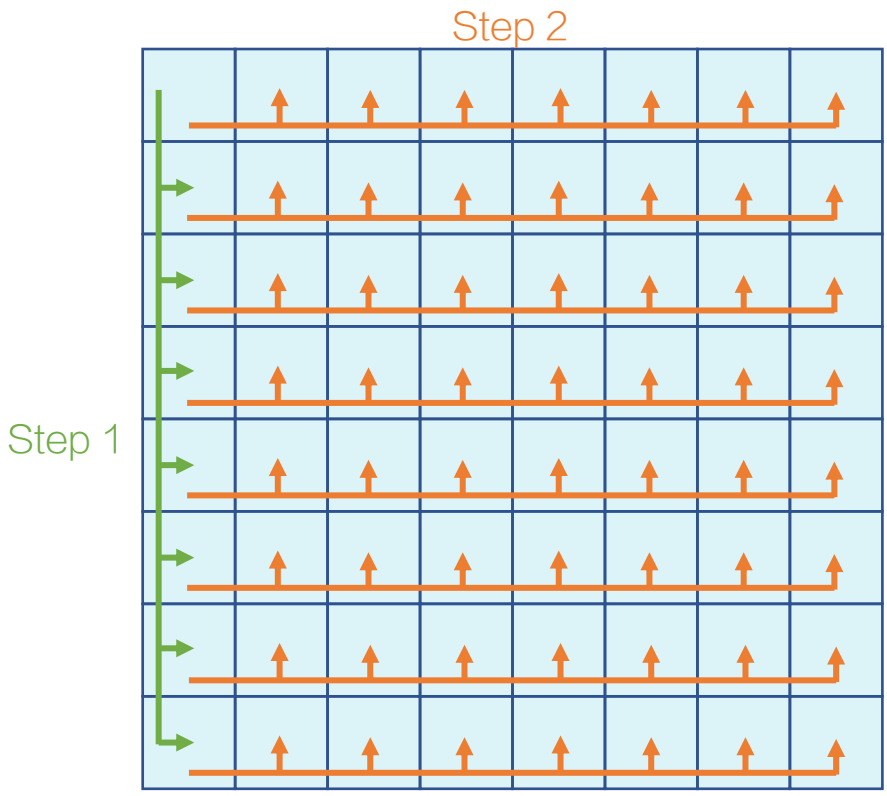

图 5 寄存器通信广播实现

下面的代码实现了从从核 0 到所有从核的 256bit 数据的广播：

```
void cpe0_bcast_256(int my_core, volatile intv8* bcast_data)
{
    bcast_tmp256 = *bcast_data;

    if (my_core / 8 == 0)
    {
        __builtin_sw64_putC(bcast_tmp256, 0x0000000F);
    } else {
        bcast_tmp256 = __builtin_sw64_getxC(bcast_tmp256);
    }
    if (my_core % 8 == 0)
    {
        asm volatile ("nop":::"memory");
        __builtin_sw64_putR(bcast_tmp256, 0x0000000F);
        *bcast_data = bcast_tmp256;
    } else {
        asm volatile ("nop":::"memory");
        *bcast_data = __builtin_sw64_getxR(bcast_tmp256);
    }
}
```

4.3.2. FLAG 值传递与等待

通过 athread 编程的一大难点是如何方便地控制从核的工作流程。如果计算流程一改变，整个从核代码就需要重构，那么代码工作量显然非常大。

由于程序只执行了一次 athread_spawn，从核上的线程的生命周期将持续到程序结束前。这意味着从核上将有一个函数贯穿整个迭代的过程。一种直观的编程方法是，在从核上同步跟进主核的计算过程，从核代码有一个和主核代码等长的迭代循环。然而，这样的话会大大提高编程开销和维护难度，如果要重构计算过程的话将会非常困难。

因此，本文程序的从核代码将仅通过 flag 值来获得主核指派的任务详情。这样，在修改主核代码流程的时候，将完全不需要修改从核代码，只需要在主核代码修改传入的 flag。

对于传入的 flag 各个位置的意义如下宏定义所示：

```
#define FLAG_SIZE        32
#define MPI_RANK         1
#define KERNEL_ACTION    2
#define GROUP_SIZE       3
#define REMAIN_POINT     4
#define PACKED_PTR       5
#define OUT_DATA_PTR     6
#define INDEXS_PTR       7
#define IAS_PTR          8
#define ICS_PTR          9
#define NSPS_PTR         10
#define TOTAL_NSP_PTR    11
```

除去用于指示执行进程的第 0 位，flag 数组的第 1 位到第 11 位都被用于传递不同的信息。比如，flag 数组的第 2 位被用来指示从核将要进行什么操作。

主核端有以下全局变量：

```
volatile long host_flag[FLAG_SIZE];
volatile long slave_flag[FLAG_SIZE];
volatile long flag_to_wait;
```

其中，host_flag 是在主核端进行修改的代码，从核 0 会通过 DMA 反复读取直到获取到目标 flag 值。slave_flag 是提供给从核 0 用于 DMA 输出 flag 的数组，主核会检查 slave_flag[0]以获得从核代码执行情况。slave_flag[0]和 host_flag[0]都会被初始化为 0，而 flag_to_wait 会被初始化为 1。主核端等待

从核 flag 的函数如下：

```
void wait_slave_flag()
{
    while(slave_flag[0] < flag_to_wait);
    flag_to_wait++;
}
```

主核向从核传递 flag 信息，则使用以下方式：

```
host_flag[MPI_RANK] = my_rank;
host_flag[KERNEL_ACTION] = MEAN1_FLAG;
host_flag[GROUP_SIZE] = mean1_group_count;
host_flag[REMAIN_POINT] = mean1_group_remain;
host_flag[PACKED_PTR] = (long)mean1_packed_float;
host_flag[INDEXS_PTR] = (long)mean1_packed_int;
host_flag[OUT_DATA_PTR] = (long)mean1_out_buffer;
host_flag[IAS_PTR] = (long)mean1_ia_set;
host_flag[ICS_PTR] = (long)mean1_ic_set;
host_flag[NSPS_PTR] = (long)mean1_nsp_set;
host_flag[TOTAL_NSP_PTR] = (long)_nsp;
asm volatile ("#nop":::"memory");
host_flag[0] = host_flag[0] + 1;
```

上面的代码指示从核进行 smooth + mean1 操作，将从核所需的数值和指针包含在了 flag 数组中，然后修改 flag 的第 0 位，从而结束从核的等待过程。

在每个从核的局部存储，都有以下变量：

```
__thread_local volatile long local_flag[FLAG_SIZE];
__thread_local volatile long out_flag[FLAG_SIZE];
__thread_local long slave_flag_to_wait;
```

同样的，local_flag[0] 和 out_flag[0] 会被初始化为 0，而 slave_flag_to_wait 会被初始化为 1。从核等待主核 flag 的函数如下：

```
void standard_wait_flag(int my_core)
{
    if(my_core == 0)
    {
        while(1)
        {
            get_reply = 0;
            athread_get(
                PE_MODE, (void*)param.host_flag, (void*)&local_flag[0],
                sizeof(long) * FLAG_SIZE, (void*)(&get_reply), 0, 0, 0
            );
            while(get_reply != 1);
            asm volatile ("#nop":::"memory");
```

```
            if(local_flag[0] >= slave_flag_to_wait)
                break;
        }
        slave_flag_to_wait++;
    }

    int i;
    for(i = 0; i < FLAG_SIZE / 4; i++)
    {
        cpe0_bcast_256(my_core, (void*)&local_flag[i * 4]);
    }
}
```

在上面的代码中，从核 0 将会无限循环地通过 DMA 检查主核的 flag 值。当 flag 值达到要求后，则会将 flag 数组通过寄存器通信广播到所有从核。

从核向主核写入 flag 的函数如下：

```
void standard_write_flag(int my_core)
{
    out_flag[0] = out_flag[0] + 1;
    athread_syn(ARRAY_SCOPE, 0x0000FFFF);
    if(my_core == 0)
    {
        put_reply = 0;
        athread_put(
            PE_MODE, (void*)&out_flag[0], (void*)param.slave_flag,
            sizeof(long) * FLAG_SIZE, (void*)(&put_reply), 0, 0
        );
        while(put_reply != 1);
    }
    athread_syn(ARRAY_SCOPE, 0x0000FFFF);
}
```

在上面的代码中，首先进行所有从核的同步，确保没有从核未完成计算，然后从核 0 将自增后的 flag 值写回主核，从而主核可以获知从核已经完成计算。

从核的代码主迭代主体如下：

```
while (1)
{
    standard_wait_flag(my_core);

    if (local_flag[KERNEL_ACTION] == PROPAGAT_FLAG)
    {
        RPCC(stcc);
        cpe_propagat(my_core);
```

```
            RPCC(edcc);
            ppg_cc += edcc - stcc;
            ppg_time++;
        }

        if (local_flag[KERNEL_ACTION] == MEAN1_FLAG)
        {
            RPCC(stcc);
            cpe_mean1(my_core);
            RPCC(edcc);
            mean1_cc += edcc - stcc;
            mean1_time++;
        }

        if (local_flag[KERNEL_ACTION] == EXIT_FLAG)
        {
            standard_write_flag(my_core);
            standard_write_flag(my_core);
            break;
        }
    }
```

可以看出，从核代码的主迭代主体非常简短，且不包含任何和计算过程有关的信息，主核代码的计算流程即使改变，从核代码也不需要进行任何修改，极大地提高了开发效率，降低了优化过程中的代码错误率。

4.4. 从核代码优化

4.4.1. 向量化

SW26010 从核支持 256-bit 的双精度浮点指令和 128-bit 的单精度浮点指令，因而，对于已有的循环，通过增大循环步长的方法，来实现向量化。只要将循环的跨步从 1 改为 4，就可以将循环体内的标量变为向量数据类型填充。

比如对于 smooth 运算的核心计算部分：

```
for(kj = 0; kj < 300; kj++)
{
    cpe_mean1_em[kj] = (a * cpe_mean1_ees[kj] +
                        cpe_mean1_ees[_jl * _kl * 1 + kj] +
                        cpe_mean1_ees[_jl * _kl * 2 + kj] +
                        cpe_mean1_ees[_jl * _kl * 3 + kj] +
```

```
                        cpe_mean1_ees[_jl * _kl * 4 + kj]) / n;
}
```

利用 SW26010 支持的向量类型进行指针转换，然后可以展开成：

```
for(kj = 0; kj < 300; kj += 4)
{
    floatv4* fv4p1 = &cpe_mean1_em[kj];
    floatv4* fv4p2 = &cpe_mean1_ees[kj];
    floatv4* fv4p3 = &cpe_mean1_ees[kj + 600];
    floatv4* fv4p4 = &cpe_mean1_ees[kj + 900];
    floatv4* fv4p5 = &cpe_mean1_ees[kj + 1200];

    *fv4p1 = afv4 * (*fv4p2) + *fv4p1;
    *fv4p1 = *fv4p3 + *fv4p1;
    *fv4p1 = *fv4p4 + *fv4p1;
    *fv4p1 = *fv4p5 + *fv4p1;
    *fv4p1 /= nfv4;
}
```

4.4.2. 逻辑判断指令优化

神威太湖之光的从核指令集中，有向量化的浮点数判断指令，不仅实现了向量化的浮点判断和选择操作，而且执行的时钟周期数远小于条件分支语句的实现。

比如，max 操作原本的定义如下：

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

该定义是不支持向量化操作的。此外，由于涉及了条件分支语句，在 RISC 架构的神威太湖之光从核上执行效率将会非常低。

因此，可以使用 SW26010 支持的内置逻辑判断指令，来进行快速的向量浮点数判断和选择。比如，对于标量语句：

```
ee = max(small, tmp);
```

嵌汇编函数__builtin_sw64_sllt 的作用是，对第一个参数进行判别，如果该参数小于 0，则返回第二个参数，否则返回第三个参数。对于上面的语句，其向量实现如下：

```
ems = smallv4 - tmpv4;

eev4 = __builtin_sw64_sllt(ems, tmpv4, smallv4);
```

使用了条件判断嵌汇编的向量实现，处理的数据量是前者的四倍，而消耗的

时钟周期远小于前者。

4.4.3. 循环展开

为了更加高效地利用浮点流水线，需要不互相依赖的指令相互 overlap 以掩盖彼此的时延。例子如下：

__builtin_sw64_vmad 是神威太湖之光平台的嵌汇编函数，其对应的操作是 256-bit 双精度浮点乘加。以下面这个循环为例：

```
for(i = 0; i < 105000000; i++)
{
    va = __builtin_sw64_vmad(va, vx, vc);
    vb = __builtin_sw64_vmad(vb, vx, vc);
    vaa = __builtin_sw64_vmad(vaa, vx, vc);
    vbb = __builtin_sw64_vmad(vbb, vx, vc);
    vab = __builtin_sw64_vmad(vab, vx, vc);
    va1 = __builtin_sw64_vmad(va1, vx, vc);
    va = __builtin_sw64_vmad(va, vx, vc);
    vb = __builtin_sw64_vmad(vb, vx, vc);
    vaa = __builtin_sw64_vmad(vaa, vx, vc);
    vbb = __builtin_sw64_vmad(vbb, vx, vc);
    vab = __builtin_sw64_vmad(vab, vx, vc);
    va1 = __builtin_sw64_vmad(va1, vx, vc);
}
```

这个循环中总共将会执行 1260000000 次乘加指令。上面的代码中，最多有 6 条连续的不互相依赖的指令。在神威太湖之光上，上面的循环执行的时钟周期为 1470000322，大约达到了 86.7143%的峰值性能，即 6/7。而下面的代码则可以达到接近 100%的峰值性能：

```
for(i = 0; i < 90000000; i++)
{
    va = __builtin_sw64_vmad(va, vx, vc);
    vb = __builtin_sw64_vmad(vb, vx, vc);
    vaa = __builtin_sw64_vmad(vaa, vx, vc);
    vbb = __builtin_sw64_vmad(vbb, vx, vc);
    vab = __builtin_sw64_vmad(vab, vx, vc);
    va1 = __builtin_sw64_vmad(va1, vx, vc);
    va2 = __builtin_sw64_vmad(va2, vx, vc);
    va = __builtin_sw64_vmad(va, vx, vc);
    vb = __builtin_sw64_vmad(vb, vx, vc);
    vaa = __builtin_sw64_vmad(vaa, vx, vc);
```

```
        vbb = __builtin_sw64_vmad(vbb, vx, vc);
        vab = __builtin_sw64_vmad(vab, vx, vc);
        va1 = __builtin_sw64_vmad(va1, vx, vc);
        va2 = __builtin_sw64_vmad(va2, vx, vc);
    }
```

同样也是 1260000000 条乘加指令，这个循环的执行时钟周期为 1260000333。这个循环中，连续的不互相依赖的指令数为 7。

在一般程序的循环体中，很难保证总是有 7 条不互相依赖的连续指令。因此可以通过加大循环步长，以高效利用指令流水线，如以下循环：

```
    for(i = 0; i < 10000; i++)
    {
        a = arr[i] * 10.0;
        b = a * arr[i] + 1.0;
        arr2[i] = b * 23.33;
    }
```

可以进行如下展开：

```
    for(i = 0; i < 10000; i += 2)
    {
        a0 = arr[i] * 10.0;
        b0 = a0 * arr[i] + 1.0;
        arr2[i] = b0 * 23.33;
        a1 = arr[i + 1] * 10.0;
        b1 = a1 * arr[i + 1] + 1.0;
        arr2[i + 1] = b1* 23.33;
    }
```

通过这个展开，可以获得加倍的性能。

在 MASNUM-WAM 计算部分优化的最后，程序中大部分计算循环都进行了指令展开，获得了不错的加速效果。

4.4.4. DMA 压缩

propagat + implsch 计算过程需要格点以及所有临近格点的数据，总计 9 个格点：

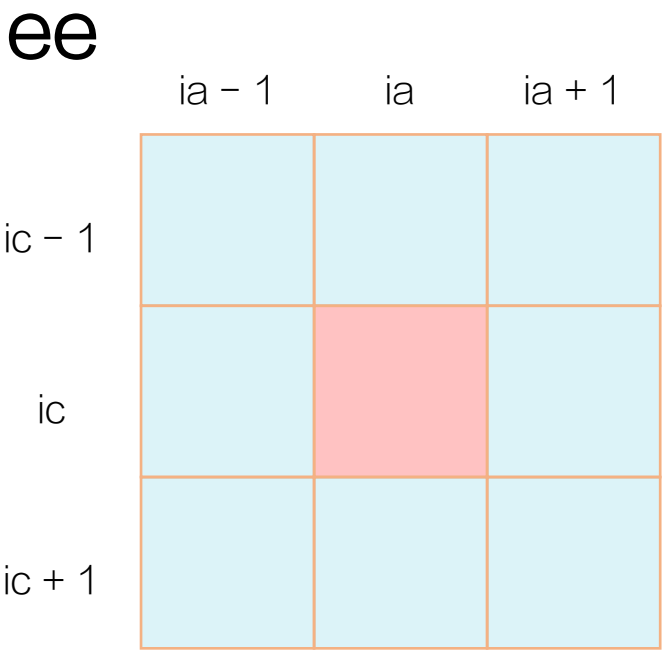

图 6 propagat + implsch 过程所需格点数据示例

连续的两个格点的计算依赖的 9 个格点的数据中，有 6 个格点是重叠的：

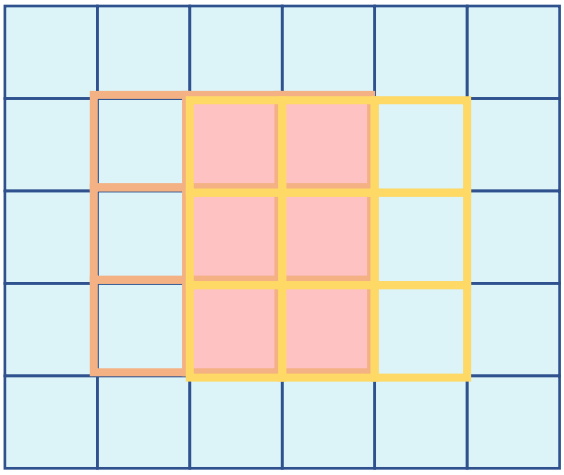

图 7 连续两个格点 propagat + implsch 格点重叠示例

从核在进行计算之前，需要通过 DMA 将格点数据从主存读取到从核 LDM。64 个从核的 DMA 总带宽有限， 访存的开销将会制约所有从核的计算性能。因而，在迭代中，可以利用上一个格点计算使用到的格点数据，而不是从主存去读取。通过这种方式，可以节省大量 DMA 带宽。

4.4.5. DMA 异步

DMA 指令是异步进行的。从核可以调用 DMA 函数来进行主存的异步读写，然

后通过 flag 值来判断 DMA 操作是否完成。在进行 DMA 异步之前，流程如下：

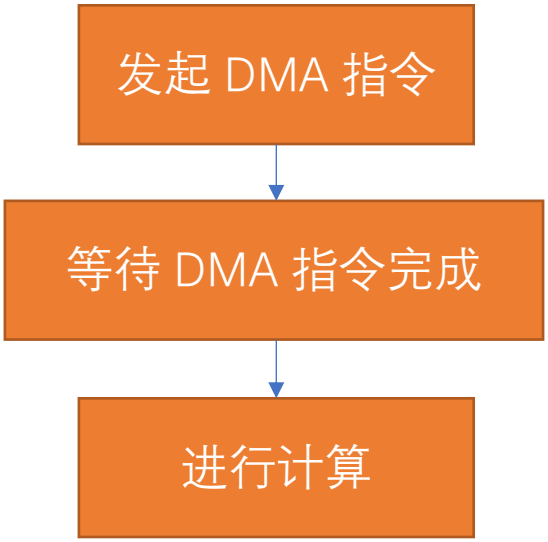

图 8 非异步 DMA 从核代码执行流程

因此，可以在本次迭代计算过程中，预取下一次迭代所需的格点的数据，通过流水线的方式掩盖 DMA 开销。

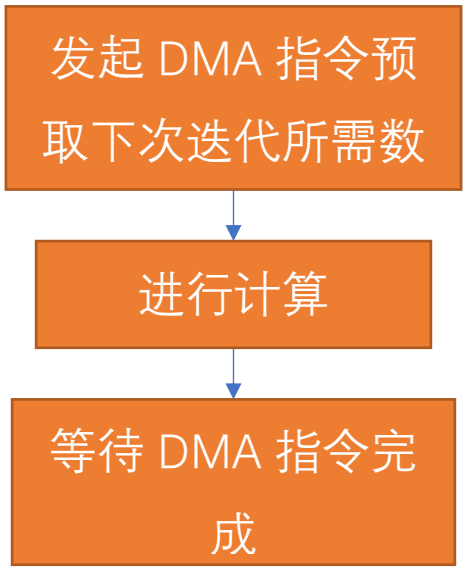

图 9 异步 DMA 从核代码执行流程

4.5. 负载均衡优化

MASNUM-WAM 程序中每个进程的计算量和该进程分配区域的海洋格点数成正比。因此所有进程的执行速度受制于海洋格点最多的进程。在此定义网格划分的效率为：最大网格数÷平均网格数。

原版代码的网格划分方式是，首先逐列遍历网格，当海洋格点数达到期望时，则将已经遍历的若干列分配给这一列的进程。然后，在行方向上也同理。这样会导致一个问题：除了最后一进程列和一列上的最后一个进程，前面的划分总是会大于期望。对于负载均衡算法的改进方法是，每当完成一部分的划分，就更新期望数值，使得网格划分更为平均。表 3 是原版的负载均衡算法和改进后的负载均衡算法的效率：

表 3 原版及优化后负载均衡效率对比

进程数	156456 格点分割效率		40208 格点分割效率	
	原版	优化后	原版	优化后
2	0.9964	0.9997	0.9948	0.9975
4	0.9908	0.9981	0.9893	0.9945
8	0.9949	0.9976	0.9830	0.9933
16	0.9815	0.9900	0.9508	0.9847
32	0.9701	0.9847	0.9541	0.9718
64	0.9632	0.9732	0.9280	0.9606

4. 6. 通信异步解耦

每个进程的 ee 数组的后两个维度的区域划分如图所示：

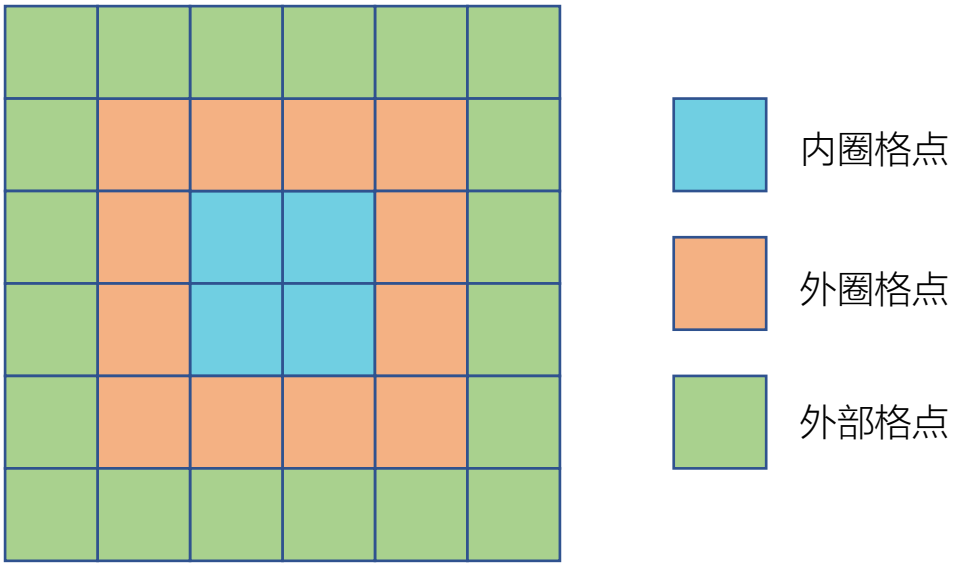

图 10 波谱数组格点空间分布

内圈格点和外圈格点组成了该进程划分得到的网格区域，外部格点的信息则

需要通过边界交换从其余进程获取[3]。

原版代码的通信过程是，首先进行垂直方向上的通信，然后再进行水平方向上的通信。但是，原版代码的水平通信依赖于垂直通信的完成。垂直通信如下图所示：

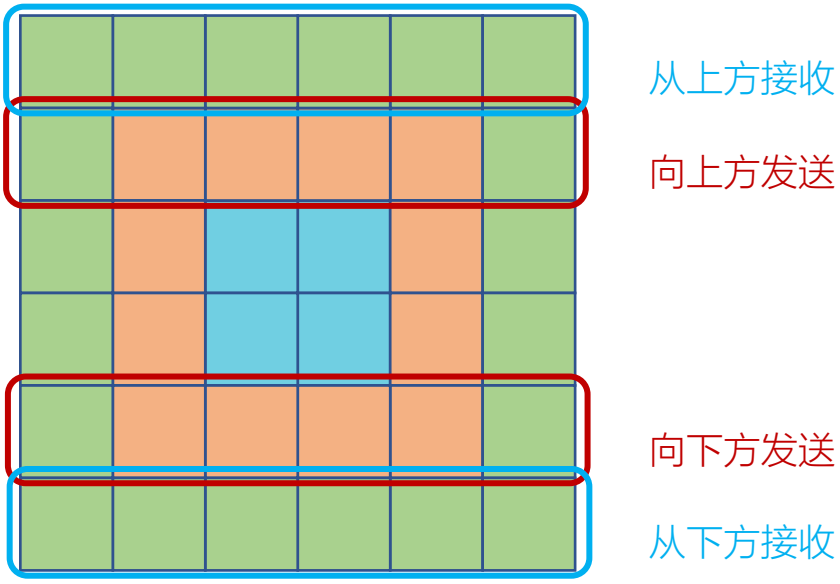

图 11 垂直通信阶段

而水平通信阶段发送的一部分与垂直通信阶段接收的一部分重合，如图：

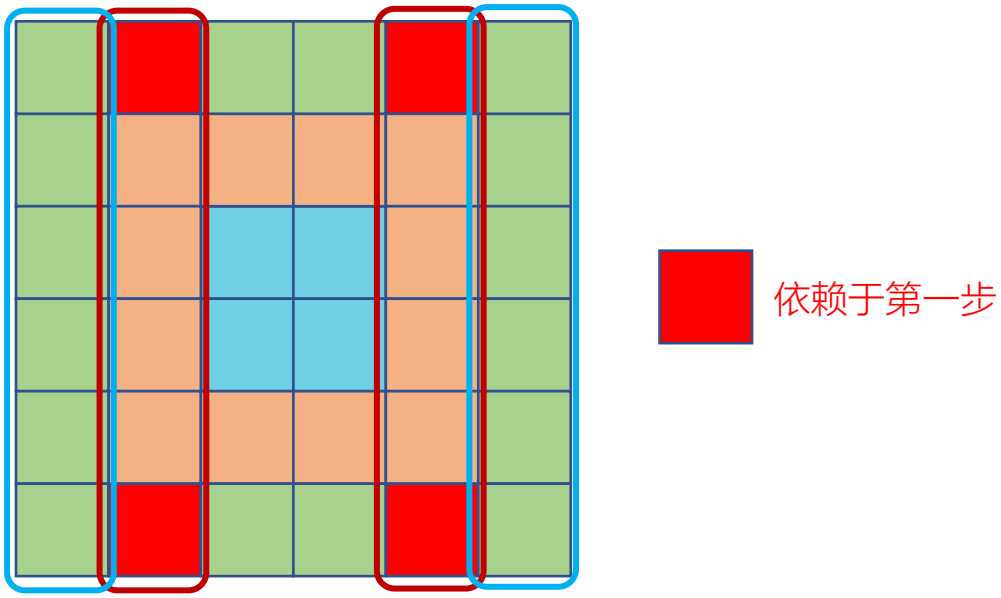

图 12 水平通信阶段

因此原版代码的水平通信必须在垂直通信执行完之后才能执行。所以，要实

[3] 外部格点的信息来自于其余进程的外圈格点

现通信和计算的异步，则必须修改通信算法，使得水平通信和垂直通信能够同时进行。代码的实现中，将两个通信阶段的发送和接收的区域进行了缩减[4]，如图：

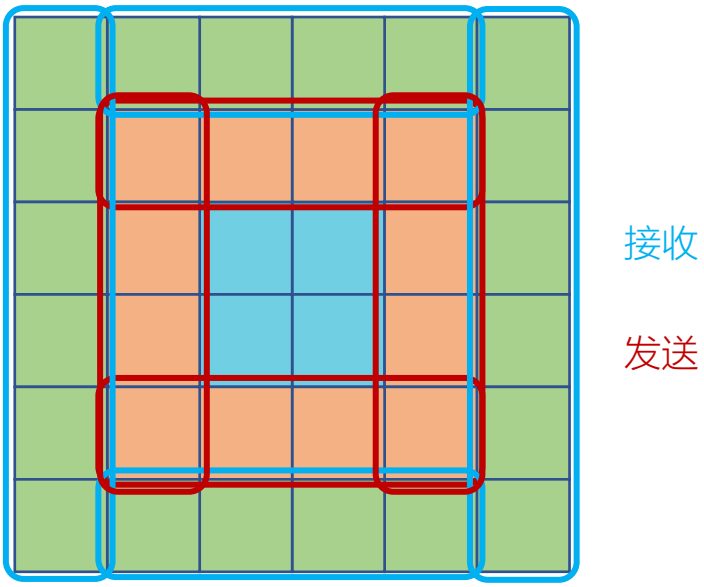

图 13 解耦后的通信区域

此外，MPI 的非阻塞通信指令并非是完全并行的。MPI 非阻塞通信指令只是将通信的请求提交给了 MPI 守护进程，具体的通信实现是由 MPI 守护进程实现的。MPI 在运行的时候提供了许多参数，可以用于指导 MPI 守护进程规划通信规则。但可以确定的是，不管在哪一种参数下，MPI 守护进程都会受制于系统本身的网络通信能力。

非阻塞发送操作 isend 在提交请求后，发送请求就进入了 MPI 守护进程的队列。在匹配到目标进程相对应的接收请求后，则开始进行通信传输。因此，isend 和 irecv 在实现上并非完全是异步的，异步的程度取决于系统的能力，系统能够同时执行的请求数量是有限的。

原有的算法中，不管是哪一个进程，都会先提交与在其上方的进程的通信请求，然后再提交与在其下方的进程的通信请求，待垂直通信完成后，然后依次提交左方和右方的通信请求，再通过 waitall 等待所有水平通信请求完成。假如按照这个顺序提交请求，那么发送和接收的匹配将会是非常随机的。

由于发送和接收的匹配非常随机，因此大大提高了少量通信请求被长期搁置的可能性。因此，在本文的代码中，对于不同进程非阻塞通信请求的提交顺序，

[4] 值得注意的是，在水平边界交换的过程中，所有进程发送的数据量依然等于接收的数据量，因为在进行了通信修改之后，部分外圈格点的数据需要发送给不止一个进程

有不同的要求。在这里定义进程的二维坐标如图[5]：

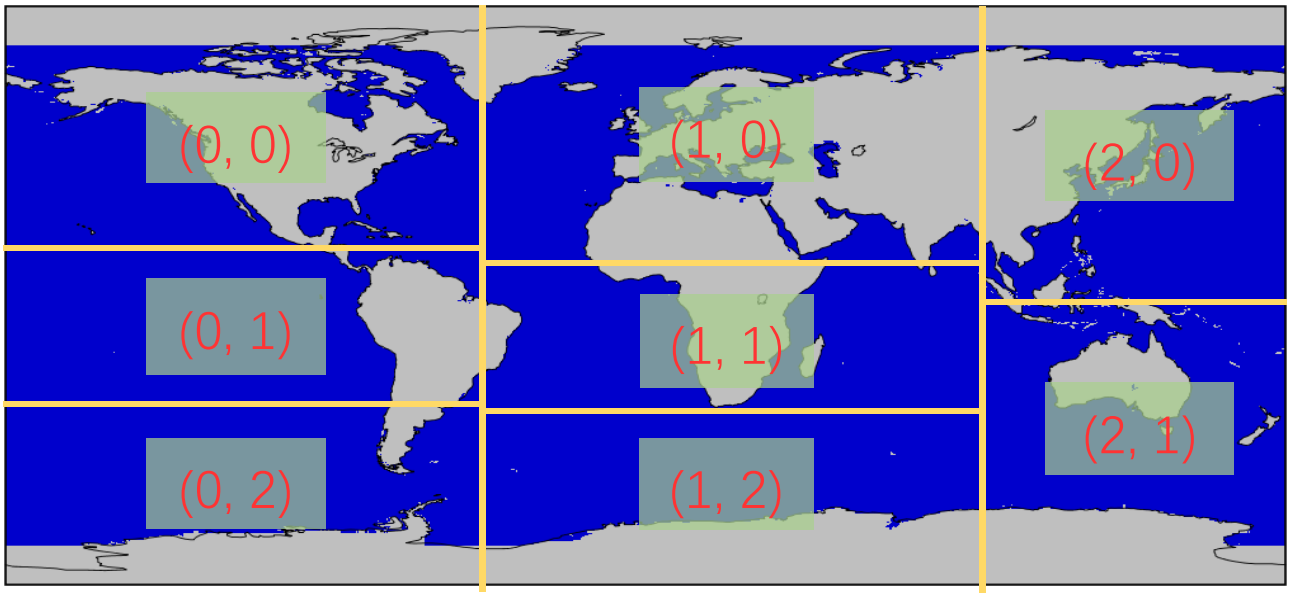

图 14 网格划分和进程坐标

在垂直方向通信请求提交顺序如图：

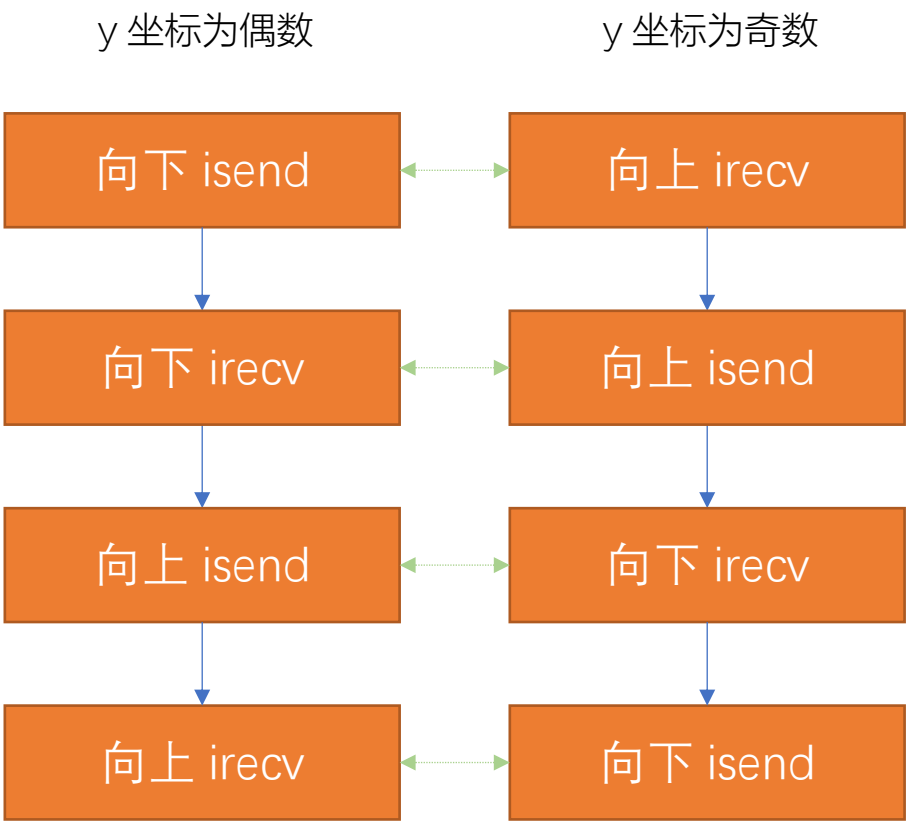

图 15 垂直方向通信请求提交顺序

[5] “上”方向指 y 坐标较高的方向

上图中请求的提交顺序，确保了 MPI 守护进程异步通信队列请求的有序匹配，减少了通信总过程的周期。

水平方向的通信请求也同理。在进行了这一小节的优化后，实现了边界交换的异步进行，同时也提高了边界交换的效率。

4.7. 计算过程解耦

要实现代码的可拓展性，很重要的一方面在于计算和通信的 overlap。由于通信针对的只是外圈和其余进程接触的部分，因而可以先完成外圈的 ee 的计算，然后在通信的过程中，再进行内圈的 ee 的计算。

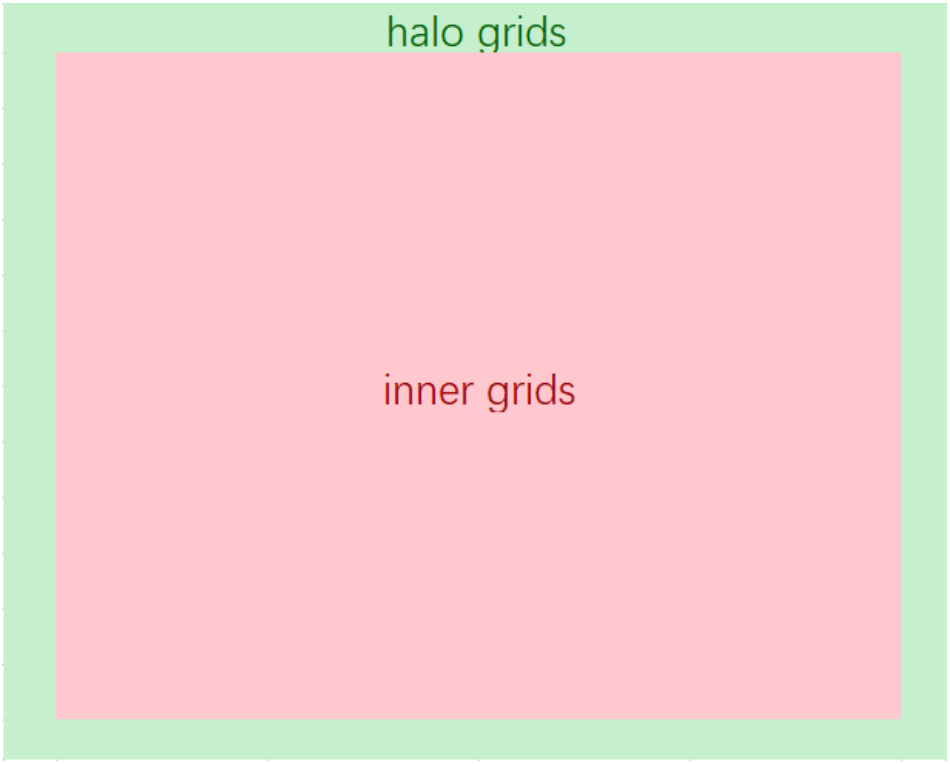

图 16 单通信异步格点分区

原版代码中的边缘交换函数 updatev 并不是异步的，因此首先 updatev 改写成两个函数。update_communication 函数调用 isend 和 irecv 操作，update_wait 函数则通过 waitall 操作等待异步通信的完成。

计算中耗时的部分主要在于 progagat + implsch。由于 smooth + mean 计算时间非常短，用其进行通信掩盖无法取得显著效果。最终计算流程如下：

表 4 单通信异步计算流程

过程	对 ee，e 的操作
外圈 progagat + implsch	得到外圈的 e
updatev_communication	启动交换边缘的 e
内圈 progaget + implsch	释放 flag，从核开始进行内圈 e 计算
updatev_wait	等待 e 边缘交换完成
等待内圈 progaget + implsch	等待从核的内圈 e 计算完成的 flag
smooth + mean1	通过 e 计算 ee
updatev	交换边缘的 ee

通过上面的通信与计算的异步实现，掩盖了每次迭代中的一次通信的时间。但是，每次迭代需要进行两次边缘交换通信，这意味着上述实现无法完全掩盖通信时间，可拓展性依然有限。

由于内圈 propagat + implsch 只依赖于上一次迭代 smooth + mean1 和 ee 的边缘交换的结果，且迭代中 e 的交换边缘和内圈 propagat + implsch 互不影响，因此，可以将内圈进行两等分，这样的话两份内圈的 propagat + implsch 过程分别可以用来掩盖两次通信的时间。划分方式如下：

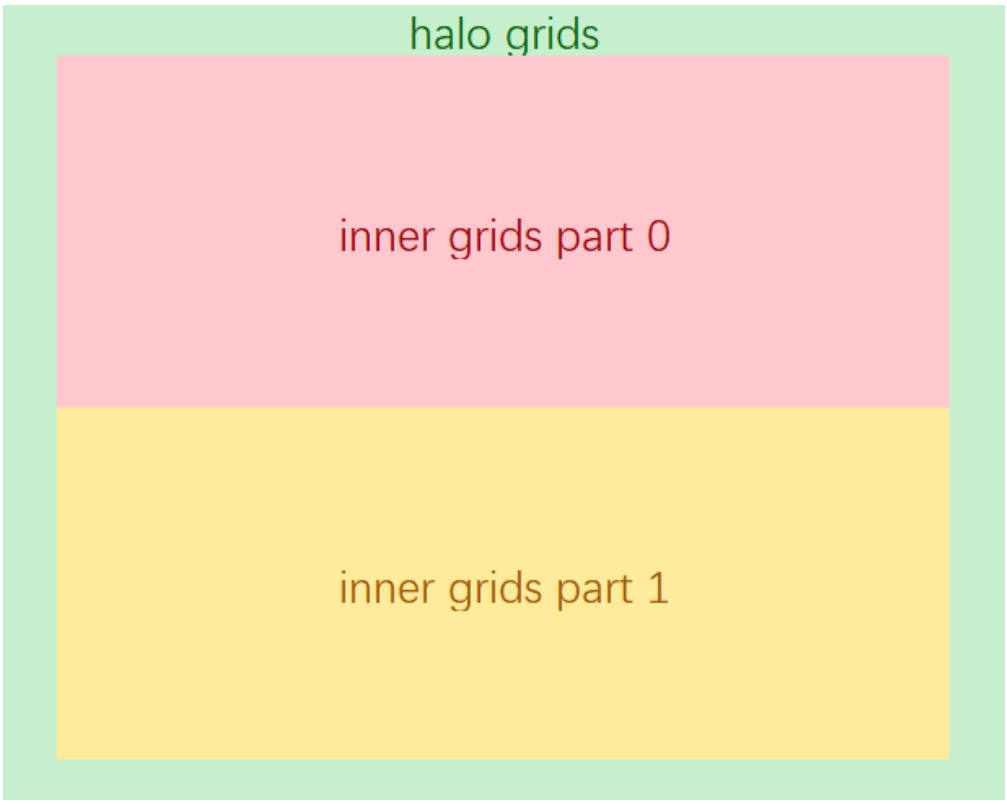

图 17 双通信异步格点分区

将内圈划分为 part 0 和 part 1。下一次迭代 part 0 的 propagat + implsch 计算被提前到本次迭代 smooth + mean1 刚结束的时候，用以掩盖 ee 边缘交换的时间，过程如下所示：

表 5 双通信异步计算流程

过程	对 ee，e 的操作
外圈 progagat + implsch	得到外圈的 e
updatev_communication	启动交换边缘的 e
part 1 内圈 progaget + implsch	释放 flag，从核开始进行内圈 part 0 的 e 计算
updatev_wait	等待 e 边缘交换完成
等待 part 1 内圈 progaget + implsch	等待从核的 part 1 内圈 e 计算完成的 flag
smooth + mean1	通过 e 计算 ee
updatev_communication	启动交换边缘的 ee

part 0 内圈 progaget + implsch	释放 flag，从核开始进行内圈 part 0 的 e 计算
updatev_wait	等待 ee 边缘交换完成
等待 part 0 内圈 progaget + implsch	等待从核的 part 0 内圈 e 计算完成的 flag

通过这个实现，当计算和通信的耗时比例大于一定程度时，通信的时间将几乎被完全掩盖，进而取得接近线性的加速比。

第五章 性能与评估

5.1. 结果分析

对于本文最终的代码，使用了三个算例进行性能测试，算例信息如下：

表 6 算例信息

算例名称	分辨率	时间步长（min）	模拟天数
exp1	1°	10	3
exp2	0.5°	5	1
exp3	0.1°	1.5	1

Efficiency 的计算依据来自于 2016 年戈登贝尔奖提名的 MASNUM-WAM 论文中的数据[2]：

表 7 Efficiency 计算依据

分辨率	核组数	迭代步数	运行时间(s)	Efficiency
(1/100)°	163840	28800	810.7	36.22%

三个算例的评测结果如下：

表 8 算例测试结果

算例	核组数	原版运行时间（s）	运行时间（s）	Speed-up	Efficiency（%）
exp1	2	12990.79	52.2292	248.7266	69.0838
exp2	4	23937.32	68.2505	350.7274	69.4713
exp3	256	44732.36	约 90	约 500	约 70%

表现出了良好的弱可拓展性。

通过与原版代码的输出进行比较，验证了代码的数学等价性。各个算例的结果可视化结果如下：

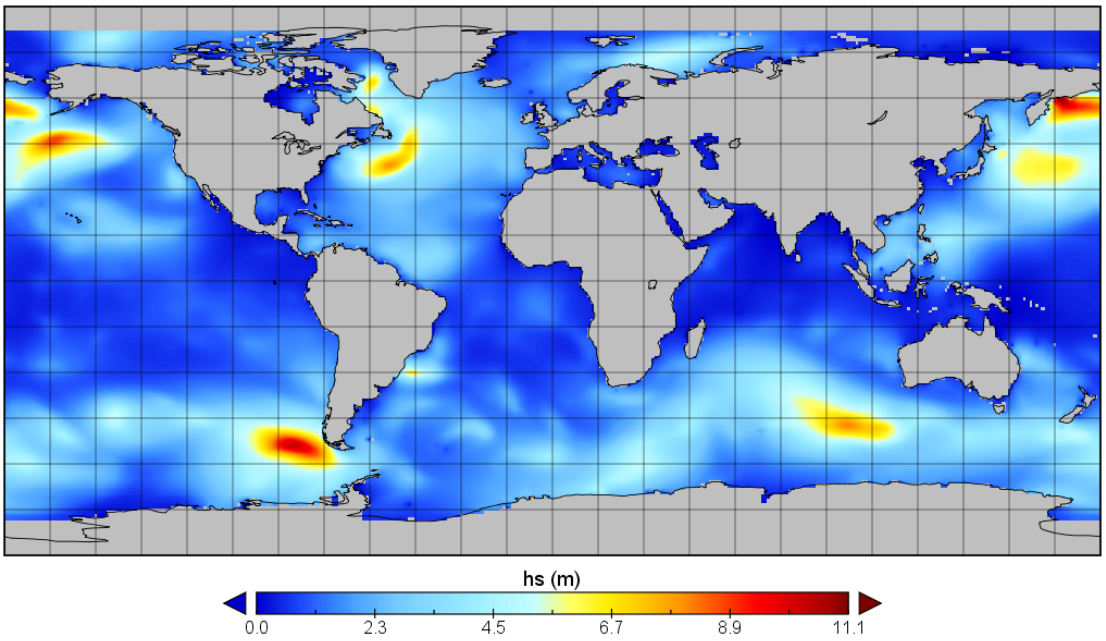

图 18 exp1 的运行结果可视化

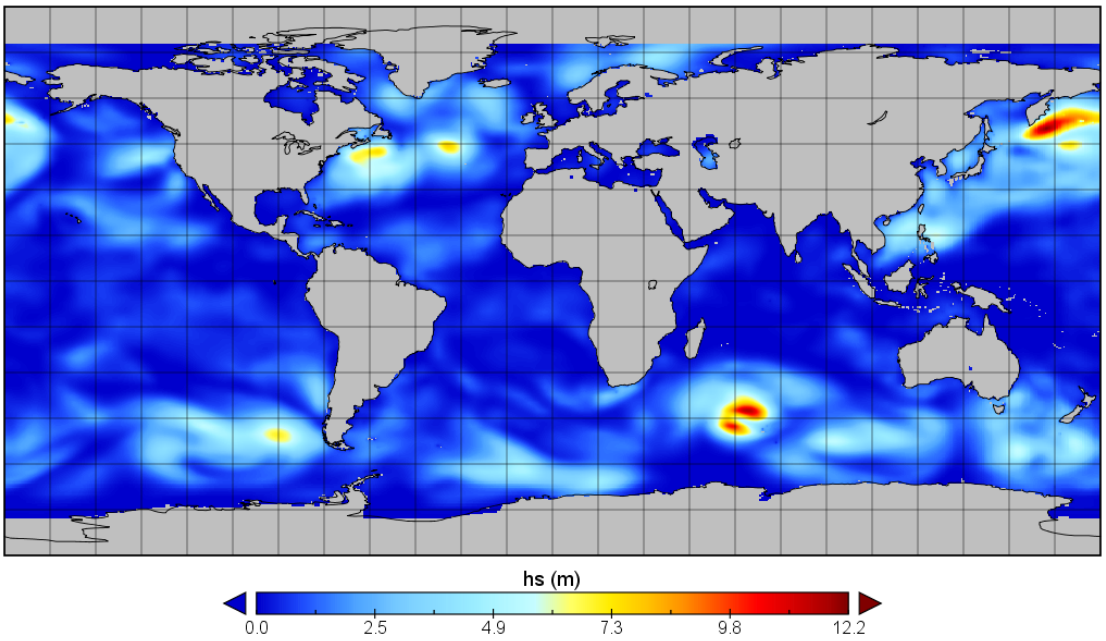

图 19 exp2 运行结果可视化

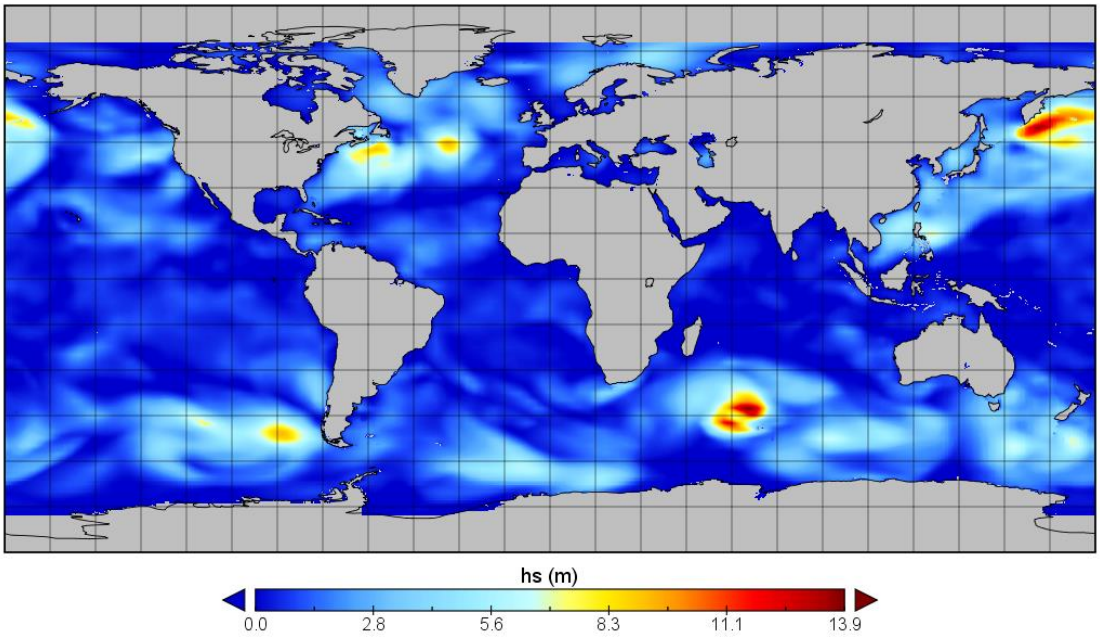

图 20 exp3 运行结果可视化

5.2. 强可拓展性测试

强可拓展性指的是在算例规模不变的情况下，在不同的并行规模体现出的可拓展性。相对于弱可拓展性，强可拓展性更加能够反馈程序可拓展性的优化结果。

为了测试代码的强可拓展性，不同核组数将用于各个算例的测试。为了测得更加准确的结果，三个算例其余参数不变，模拟天数改为原本算例的四倍。测试结果如下：

表 9 exp1 测试结果

核组数	运行时间（s）	Efficiency（%）	加速比
2	208.28269	71.3495	1
4	106.82069	67.5561	1.9498
8	59.21401	60.9348	3.5175
16	35.28174	51.1340	5.9034

表 10 exp2 测试结果

核组数	运行时间（s）	Efficiency（%）	加速比

2	530.87422	72.4982	1
4	271.76466	70.8102	1.9534
8	138.81233	69.3155	3.8244
16	72.09038	66.7346	7.3640
32	39.06489	61.5761	13.5895

表 11 exp3 测试结果

核组数	运行时间（s）	Efficiency（%）	加速比
64	1436.2218	69.7856	1
128			
256			
512			
1024			

上面的测试结果中，代码的强可拓展性表现可以接受。

第六章 总结与展望

6.1. 总结

本文主要工作总结如下：

1. **分析了任务需求**。通过对于 MASNUM-WAM 原版程序热点和特性的分析，以及对于神威太湖之光平台的分析，确定了移植于优化的任务需求。
2. **完成了代码移植**。将通用的 FORTRAN 代码进行改写，得到适用于神威太湖之光平台的 C 和 FORTRAN 混编代码。
3. **实现了多方面的优化。**针对算法本身和 SW26010 特性，对代码进行优化。

6.2. 展望

6.2.1. 通过寄存器通信共享格点数据

虽然本论文尝试减少 DMA 的开销，但是 DMA 的开销还是相当大。

SW26010 一个核组内的从核阵列上的 64 个从核支持寄存器通信。因此，如果一个从核在下一次迭代中所需要的格点数据在另一个从核的本次迭代已经获取，那么在下一次迭代之前就可以通过寄存器通信获取该格点的数据，而不是通过 DMA 从主存中读取。

6.2.2. 通信压缩

由于只有海洋格点的边界才需要进行交换，因此去除非海洋格点的边界交换，以缩减不必要的通信带来的开销。

6.2.3. 从核内存装入优化

由于 SW26010 的从核不支持跨越两行 cache line 的向量装入，因此代码实现中的许多向量赋值都使用了 simd_set_floatv4 函数。该函数的装入效率远差于直接的 load 指令。

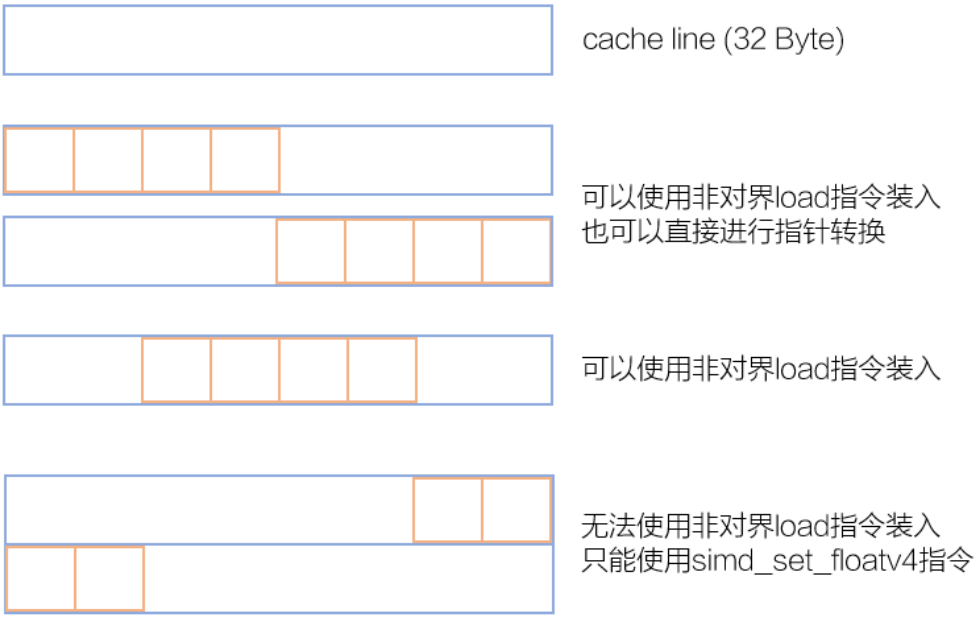

图 21 floatv4 向量类型装入遇到的问题

因此，要提高向量装入的效率，需要进行数据的对齐和重新排列。

参考文献：

[1] 杨永增, 乔方利, 赵伟,等. 球坐标系下 MASNUM 海浪数值模式的建立及其应用[J]. 海洋学报, 2005, 27(2):1-7.

[2] Fangli Qiao , Wei Zhao , Xunqiang Yin , Xiaomeng Huang , Xin Liu , Qi Shu , Guansuo Wang , Zhenya Song , Xinfang Li , Haixing Liu , Guangwen Yang , Yeli Yuan, A highly effective global surface wave numerical simulation with ultra-high resolution, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 13-18, 2016, Salt Lake City, Utah.

[3] Haohuan Fu , Junfeng Liao , Nan Ding , Xiaohui Duan , Lin Gan , Yishuang Liang , Xinliang Wang , Jinzhe Yang , Yan Zheng , Weiguo Liu , Lanning Wang , Guangwen Yang, Redesigning CAM-SE for peta-scale climate modeling performance and ultra-high resolution on Sunway TaihuLight, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 12-17, 2017, Denver, Colorado.

[4] Chao Yang , Wei Xue , Haohuan Fu , Hongtao You , Xinliang Wang , Yulong Ao , Fangfang Liu , Lin Gan , Ping Xu , Lanning Wang , Guangwen Yang , Weimin Zheng, 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 13-18, 2016, Salt Lake City, Utah.

[5] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The Sunway TaihuLight supercomputer: system and applications. Science China Information Sciences, 59(7):072001, 2016.

[6] 孙卓毅, 李洪平. MASNUM 海浪数值模式并行化实现[J]. 地理空间信息, 2012, 10(5):117-119.Cachin C. Architecture of the Hyperledger blockchain fabric[C]//Workshop on Distributed Cryptocurrencies and Consensus Ledgers. 2016.

[7] 张志远, 周宇峰, 刘利,等. MASNUM 海浪模式的性能特点分析与并行优化[J]. 计算机研究与发展, 2015, 52(4):851-860.

[8] 杨晓丹, 宋振亚, 周姗,等. MASNUM 海浪模式的代码现代化优化[J]. 海洋科学进展, 2017, 35(4):473-482.

[9] 孙玉娟, 乔方利, 王关锁,等. MASNUM 海浪数值模式业务化预报与检验[J]. 海洋科学进展, 2009,

27(3):281-294..

[10] 吴欢, 汪一航, 滕涌,等. 浙江海域 MASNUM 海浪模式在台风“达维”“海葵”及“布拉万”过程的波浪数值模拟[J]. 应用海洋学学报, 2017, 36(2):249-259.