SKA 项目的插值模块的 MPI 并行化分析与实现

完成日期： 2016 年 5 月 21 日

指导教师签字：_____

答辩小组成员签字：_____

_____

_____

SKA 项目的插值模块的 MPI 并行化分析与实现

摘　要

为了提升 SKA 工程项目中 gridding 插值模块程序的计算性能，提高数据卷积计算的计算速度，对 gridding 算法进行研究。首先先采用 Intel Vtune 软件对该模块代码进行热点分析，根据分析结果得到热点信息。再对热点部分利用快速排序法通过加快数据存取速度提高性能。然后根据并行计算思想原理，分析热点函数的可并行性，根据分析结果对外部循环进行 MPI 并行化处理，将并行后的程序在多个不同节点上运行，随着节点数的增加，比较其性能提高程度，计算性能提高比率。

关键词：　热点分析；MPI 并行；快速排序；卷积公式

MPI parallel analysis and implementation of the interpolation module of SKA project

Abstract

In order to improve the computational performance of the gridding interpolation module program in the SKA project,and to improve the calculation speed of Convolution computation ,the gridding algorithm is studied. Firstly, the module code is analyzed by Intel Vtune software, and the hot spot information is obtained according to the analysis results. Then it uses the fast sorting method to improve the performance of data access speed. And next, baseing on the parallel computing theory, it analyses Parallel property of hot function. According to the result,processing the outside cycle by MPI parallel , and running parallel program in multiple different nodes. With the increase of the number of nodes, the performance is compared, and the improved ratio is computed.

Keywords: hotspot analysis; MPI parallel; quick sort; convolution formula

目　录

0 引言

人类居住的地球，只是宇宙中的一个小小的星球，目前人类对太空的认识，就像沙滩上的一粒沙。人类对太空的求知欲望，将是太空探索的永恒动力。随着科技的发展，人类对太空的奥秘有着更加浓厚的兴趣，科学家们研究宇宙中各类物体和现象以及它们产生的原因和可能导致的结果。只有通过研究宇宙，才能知道宇宙的起源，地球的诞生，以及许许多多待解决的问题。当地球面临资源枯竭、生态破坏、发展瓶颈，人类需要寻找更好的生存空间。所以研究宇宙环境是全人类的事业。

1 绪论

1.1 背景介绍

在过去的几十年内，人们应用地基望远镜和空间望远镜来观测各类宇宙现象，由于它们采用光学镜面，其尺寸和制造工艺极大的限制了我们的观测要求。根据 NASA／ESA 的哈勃望远镜所探测到的最远距离和最暗光度，人们只能观测到大约 130 多亿光年的距离。即使是其后继者詹姆斯－韦伯空间望远镜的镜面和 TMT 地基望远镜也只提升了十几倍的观测能力。它依然不能满足人类的观测需求。因此改变望远镜的结构和类型是重中之重。射电望远镜很好地解决了镜面限制的问题。而且可以探测到镜面望远镜不能探测到的许多东西。精度是空间望远镜的千百倍。多年来，世界各国都在努力合作建立更大的射电望远镜，这将成为人类探测宇宙的绝佳武器。

射电望远镜 SKA 在建造工程和运行阶段需要用到高速度的计算机或超级计算机，其程序在结构上属于串行程序，串行过程中只能有一个进程或节点来处理数据和计算数值结果，对于 SKA 每天产生的数据量以 PB 计算，若采用此串行程序，效率极低，速度慢，耗时长，无法在短时间内得到我们需要的结果。无法将数据准时的转化成产品给其他领域来使用。因此我考虑更改其程序结构，利用并性技术对串行程序并行化，运用现有的硬件资源，提高程序的运算速度，减少耗时，提早的完成它的工作。

2 SKA 工程项目

2.1 SKA

平方公里阵列射电望远镜（Square Kilometre Array，以下简称 SKA）是国际天文界计划建造的世界最大综合孔径射电望远镜，是由多国政府及国家研究机构联合筹资，全球约 20 个国家 100 家科研机构的天文学家和工程师参与的国际大科学工程。SKA 工程将在南非和澳大利亚建立 3300 面 15 米口径反射面天线、250 个直径约 60 米的致密孔径阵列及 250 个直径 180 米的稀疏孔径阵列，它们分布在 3000 公里范围形成望远镜阵列中。望远镜并不是仅仅成群地集中在核心区域，而是将按多个螺旋臂构造分布。天线将从中央核心起分布到广阔地域，组成众所周知的长基线干涉阵列。高频反射面天线将分布在非洲数千平方公里的土地上，孔径阵列天线将从非洲和澳大利亚核心区域延伸 200 公里。[1]

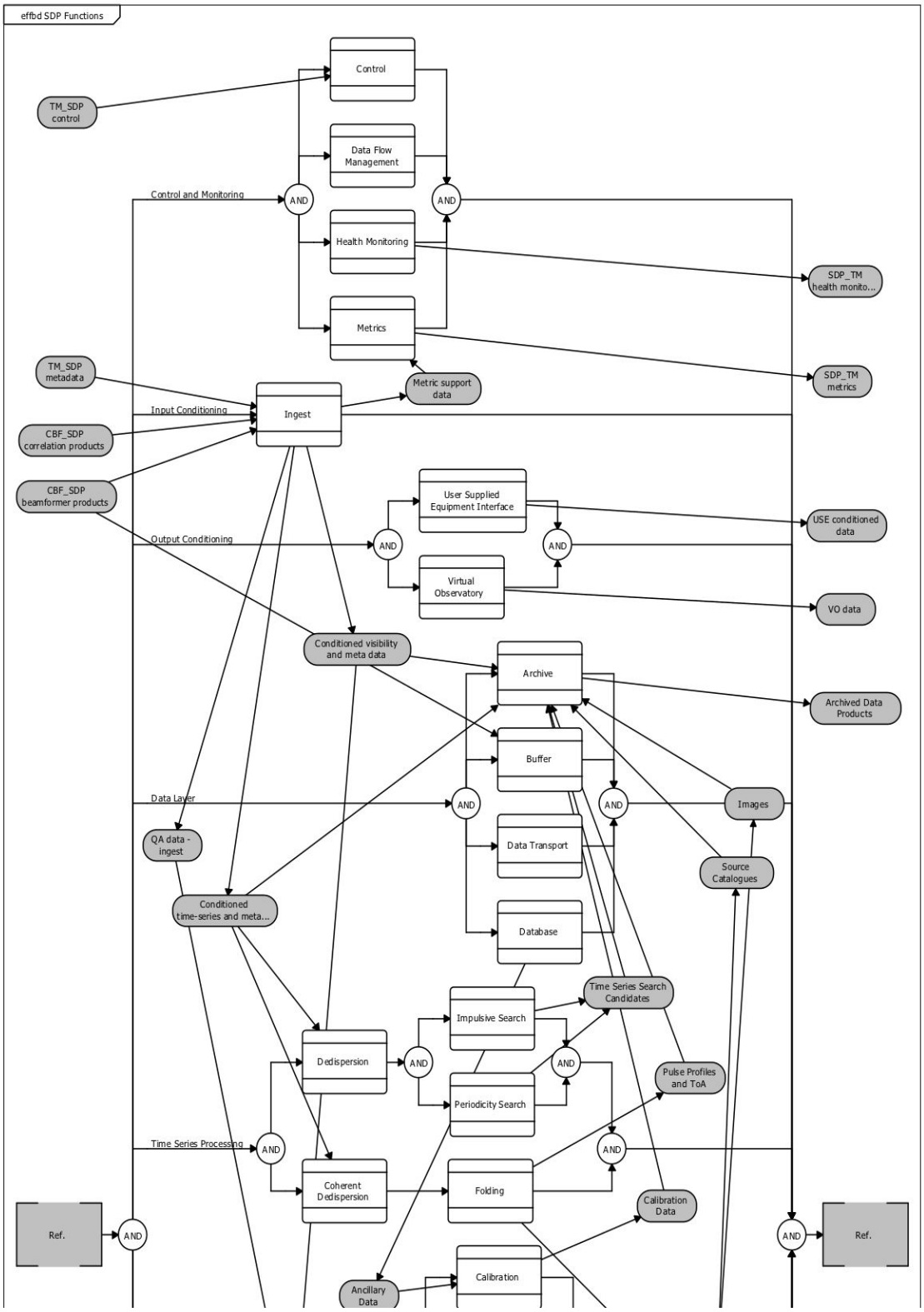
在这样一个阵列中，物理距离分开了这些望远镜，他们之间的距离是通过射电信号到达每个接收机的时差精确计算出来的。计算机就可以计算出如何组合这些信号来合成相当于这两个区域之间宽度那么大的单个天线所接收的信号。如此，干涉测量技术使天文学家可以在阵列中模拟出一个和天线最大分布距离一样尺寸的望远镜，如果需要的话，可以使用几个子阵列来模拟一个小望远镜，甚至可以模拟多个小望远镜。这样胜于建一个巨型天线，这种干涉测量构造的适应性在某些方面超过了单个巨型天线的性能。这个系统可以用作一个巨型望远镜或多个小型望远镜，或在两者之间任意组合。螺旋形配置给出了天线之间不同的长度（基线）和角度使得 SKA 有非常高分辨率的成像能力。每一个望远镜将连接至中央核心，中央核心将通过相关器组合来自每一个望远镜的数据并整合成易于处理的大小的数据包。这些数据将通过高速网络在全球范围传递到科学家的电脑屏幕上，以供他们应用。

SKA 将致力于回答关于宇宙的一些基本问题，如第一代天体如何形成、星系演化、宇宙磁场作用、引力本质、地外理性生命、暗物质和暗能量等。SKA 项目涉及天文、无线电、信息科学、力学、机械、计算数学与系统科学、土木工程等众多领域；集成众多高科技成果，如高性能低造价宽带天线制造、高品质接收机、海量数据传输存储与处理、软件和计算、信号处理、系统工程、新材料和新工艺、绿色能源等，需要通过广泛的国际合作来实现。这将带给世界一个前所未有的科技工具，使各领域出现全新的发展。[2]

2.2 SKA 项目结构

SKA 工程分为 SKA1 和 SKA2 两个阶段建造和实施。现阶段 SKA1 正在建设中。SKA1 将建设 10%左右的望远镜，并且包括两种设备: 非洲的反射面天线(SKA1 中频)和澳大利亚的低频天线(SKA1 低频)。SKA 项目包含了基础建设在内的 11 个工作包。分别为：组装、集成和验证 Assembly, Integration and Verification (AIV)，中央信号处理 Central Signal Processor (CSP)，天线 Dish (DSH)，基础设施建设（澳大利亚） Infrastructure Australia (INFRA AU)，基础设施建设（南非） Infrastructure South Africa (INFRA SA)，低频孔径阵列 Low-Frequency Aperture Array (LFAA)，中频孔径阵列 Mid-Frequency Aperture Array (MFAA)，信号与数据传输 Signal and Data Transport (SaDT)，科学数据处理 Science Data Processor (SDP)，望远镜管理 Telescope Manager (TM)，宽带单像素馈源 Wideband Single Pixel Feeds (WBSPF)。

其中科学数据处理 SDP 包主要工作是将望远镜得到的原始数据转化处理成可用的数据或数据产品，它包括计算硬件平台、软件和数据处理算法。SDP 由许多功能模块和相关的数据流组成。为了使各部分之间相互协作运行，必须有一个综合的、良好的结构系统。下面给出 SDP 的功能模块之间的结构系统。[3][4]


effbd SDP Functions
Control
TM_SDP control
Data Flow Management
Control and Monitoring
AND
AND
Health Monitoring
SDP_TM health monito...
Metrics
TM_SDP metadata
SDP_TM metrics
Metric support data
Input Conditioning
Ingest
CBF_SDP correlation products
CBF_SDP beamformer products
User Supplied Equipment Interface
USE conditioned data
Output Conditioning
AND
AND
Virtual Observatory
VO data
Conditioned visibility and meta data
Archive
Archived Data Products
Buffer
Data Layer
AND
AND
Images
QA data - ingest
Data Transport
Source Catalogues
Conditioned time-series and meta...
Database
Time Series Search Candidates
Impulsive Search
Dedispersion
AND
AND
Pulse Profiles and ToA
Periodicity Search
Time Series Processing
AND
AND
Coherent Dedispersion
Folding
Calibration Data
Ref.
AND
AND
Ref.
Calibration
Ancillary Data

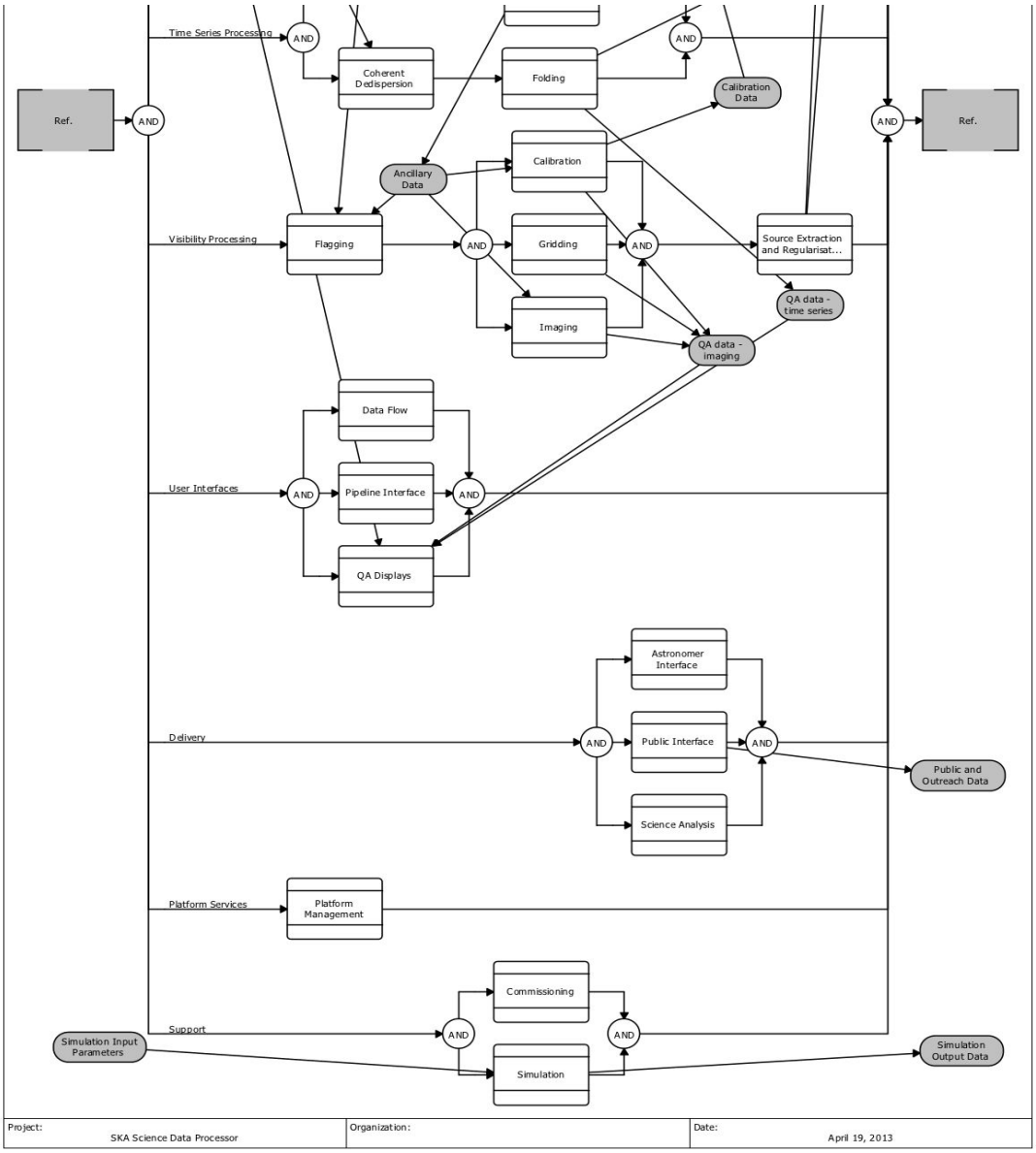


图 1 ： SDP 结构示意图[5]

上图中的 Gridding 模块是整个 SKA 项目中耗时最长，速度最慢，数据计算量最大的模块，是我们需要优化的重点部分。

通过上述结构我们可以看到，数据通过 Flagging 之后被送入 Gridding 模块，然后在该模块中进行数据处理，处理之后的数据将会做快速傅里叶变换（FFT），然后进入 Imagining 模块，此模块会产生我们需要的数据产品。所以重点的部分还是在 Gridding 的操作，它在整个环节中占据很大的比重。在之后的模块将详细叙述 Gridding 模块的结构和原理。

3 Gridding 模块

3.1 gridding 算法结构

gridding 模块的功能是利用 3200000 离散样本点数值通过卷积计算出 4096*4096 的

数据点数值。其所在程序的运行流程如下图所示：

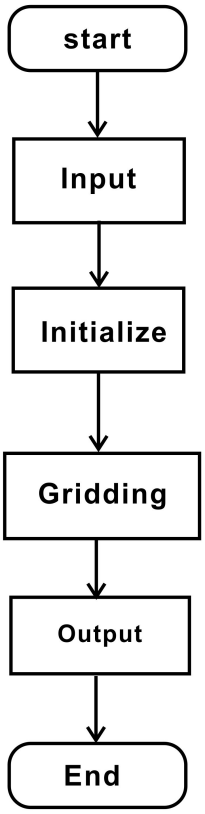


图 2 ： 程序流程图

上图在 I/O 中间的初始化和 gridding 是最主要的部分。此模块主要由以下几部分函数组成：

表 1 ： 函数功能表

名称	函数
随机数产生函数	randomInt()
初始化函数	init()
运行函数	runGrid()
插值运算模块函数	gridKernel()
初始化 C 函数	initC()
初始化 C 偏移函数	initCOffset()
输出函数	printGrid()
获得尺寸函数	getSupport()

其功能结构通过下面的 UML 图展示出来：

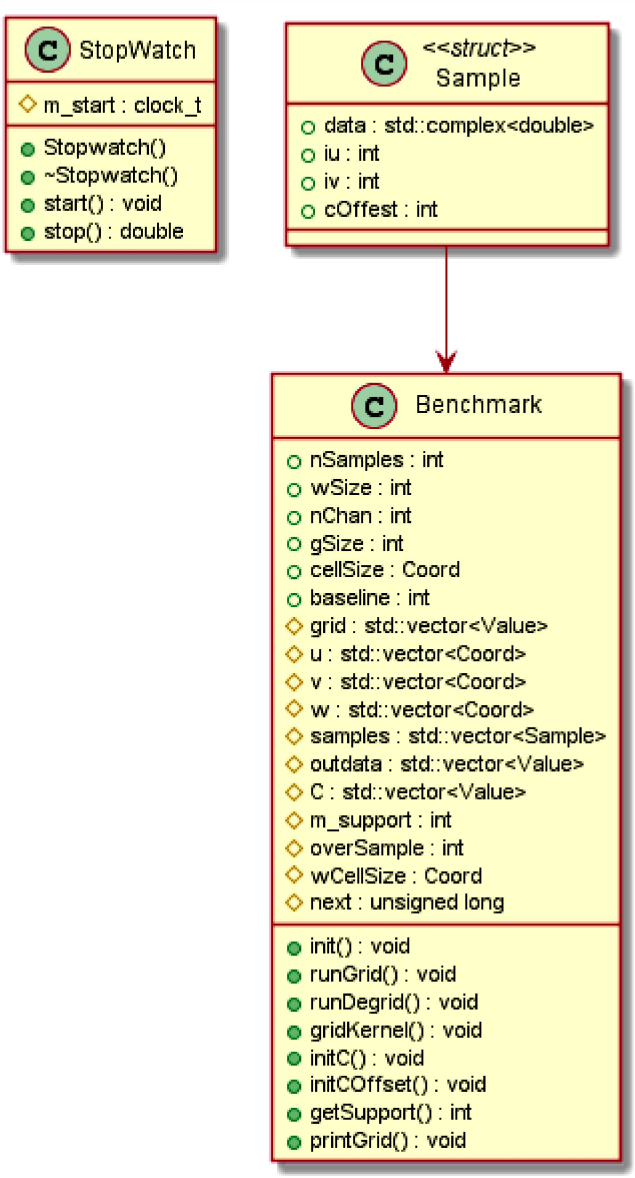


图 3 ：程序 UML 图

上述函数及原始数据所涉及的相关文件见下表：

表 2 ：相关文件表

文件名	概要
input.dat	输入文件，包括程序的一些重要参数
randnum.dat	输入文件，包括 3200000 样本数据
grid.dat	输出文件，包括计算结果
log.dat	输出文件，包括 Gridding time and the Gridding rate
Benchmark.h Benchmark.cc	Gridding 核心文件，包括初始化、gridding、输出函数
Stopwatch.h Stopwatch.cc	计时功能文件
tConvolveMPI.h tConvolveMPI.cc	程序入口文件

3.2 插值模块原理

Gridding 算法中的插值部分运用了数学科学中的卷积公式。因为 Gridding 是平面的图像处理，所以采用的是二维卷积公式。下面介绍一下二维卷积公式。[6]

$$grid(u,v)=\int_{-support}^{support}\int_{-support}^{support} Samples(u-x,v-y)\times C(-x,-y)dxdy \qquad \text{公式 1}$$

$$grid(u,v)=\sum_{x=-support}^{support}\sum_{y=-support}^{support} Samples(u-x,v-y)\times C(-x,-y) \qquad \text{公式 2}$$

考虑到 Samples 的大小都是 N，则整个 Samples 的运算量应该为:

$$Nsamples\times(2\times support+1)\times(2\times support+1)$$

3.3 程序的性能

根据上述结构和原理，gridding 算法可以对原数据进行处理得到我们需要的结果，然后在传递到下一步操作。由于原数据量十分巨大，紧紧依靠此程序的单一计算性能过低，速度太慢，以致于无法满足我们的需求。因此需要对算法进行调整以缩短运行的时间。我们需要先对源程序进行分析，找出时间消耗过大的部分，在这里我们用到了 Vtune 软件来分析哪些是我们需要关注的热点部分。

3.3.1 Vtune 软件

英特尔 VTune™性能分析器一个比较强大的性能分析软件。主要包括三个小工具：
（1）Performance Analyzer：性能分析，找到软件性能比较热的部分，一般也就是性能瓶颈的关键点。
（2）Intel Threading Checker：用于查找线程错误, 能够检测资源竞争、线程死锁等问题。
（3）Intel Threading Profiler：线程性能检测工具,多线程可能存在负载比平衡, 同步开销过大等线程相关的性能问题，该工具可以帮你发现每一个线程每一时刻的状态。
这三个工具在使用的过程中可以根据个人需要组合使用。[7]

首先介绍 Vtune 软件的安装方法。所需要的工具和平台为：Linux 系统、Vtune 安装文件（含 Vtune Driver Kit 安装文件）、License 文件。
第一步：在 Intel 官网上下载 vtune_amplifier_xe_2015_update4.tar.gz （可选择合适的版本）文件
第二步：在 Linux 系统上使用命令解压 vtune_amplifier_xe_2015_update4.tar.gz
命令为：#tar -zxvf vtune_amplifier_xe_2015_update4.tar.gz
第三步：进入解压得到的 vtune_amplifier_xe_2015_update4 文件夹
命令为：#cd vtune_amplifier_xe_2015_update4
第四步：运行 install.sh 文件。在安装的过程中需要 license 的地方，选择通过 license 文件激活。

命令为：#./install.sh
第五步：配置环境变量
命令为：#vi ~/.bashrc
添加内容为：export PATH=$PATH:$NETCDF/bin:/opt/intel/vtune_amplifier_xe_2015/bin64
第六步：测试：在命令行中输入# amplxe-gui，如果正确出现软件界面表明安装完毕。

3.3.2 运行结果与分析

(1) 收集数据
先使用 Intel Performance Analyzer 软件。收集热点信息时使用 amplxe 运行软件并用－collect hotspots 参数。
命令为：amplxe-cl -collect hotspots -r ./hotspots ./tConvolve
(2) 分析数据
热点收集完毕后，如果没有使用-r 指定文件夹，vtune 会在当前目录下生成 result 文件夹，指定路径后，结果就在指定文件夹中。如图 4 所示。

图 4 ： 命令展示界面图

首先在命令行中打开带界面的 Performance Analyzer。
命令为：＃amplxe-gui
打开后在界面中寻找 Open Result ,再加载之前生成的文件＊.amplxe,就会出现程序的各种信息，然后根据图示和信息标签就可以分析程序热点和瓶颈了。以下为本次运行的结果。
根据运行操作，打开 GUI 界面加载统计数据，显示如下。

————————————————————Summary————————————————————

>>Elapsed Time: 185.20s

Total Thread Count: 1
Paused Time: 0.00s
>>CPU Time: 184.27s
Spin Time: 0.00s
Overhead Time: 0.00s
>> Effective Time: 184.27s
Idle: 0.04s
Poor: 184.23s
Ok: 0.00s
Ideal: 0.00s
Over: 0.00s

>>Top Hotspots

This section lists the most active functions in your application.Optimizing these hotspot functions typically results in improving overall application performance.

表 3 ： 函数运行时间表

Function	CPU Time/s
Benchmark::runGrid	181.69s
Benchmark::init	0.95s
cos	0.61s
_IO_fwrite	0.34s
fread	0.29s
[Others]	0.37s

>>CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously.Spin and Overhead time adds to the idle CPU value.

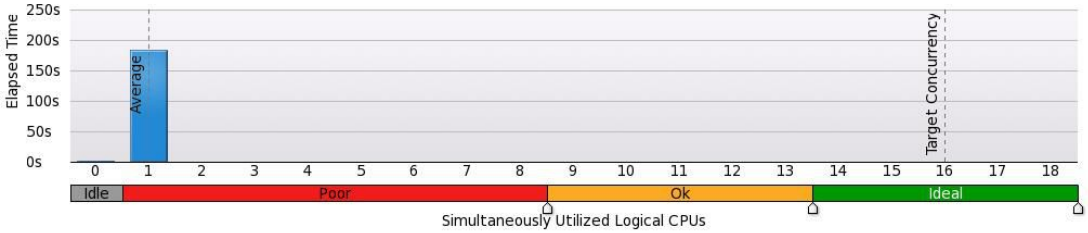


图 5 ： CPU 使用率统计图

>>Collection and Platform Info

Application Command Line: ./tConvolve

Operating System: 2.6.32-431.el6.x86_64 Red Hat Enterprise Linux Server release 6.5 (Santiago)

Computer Name: cu01

Result Size: 3 MB

Collection start time: 13:10:24 29/04/2016 UTC

Collection stop time: 13:13:29 29/04/2016 UTC

>>CPU

Name: Intel(R) microarchitecture code named Haswell Server

Frequency: 2.4 GHz

Logical CPU Count: 16

————————————————Bottom-up————————————————

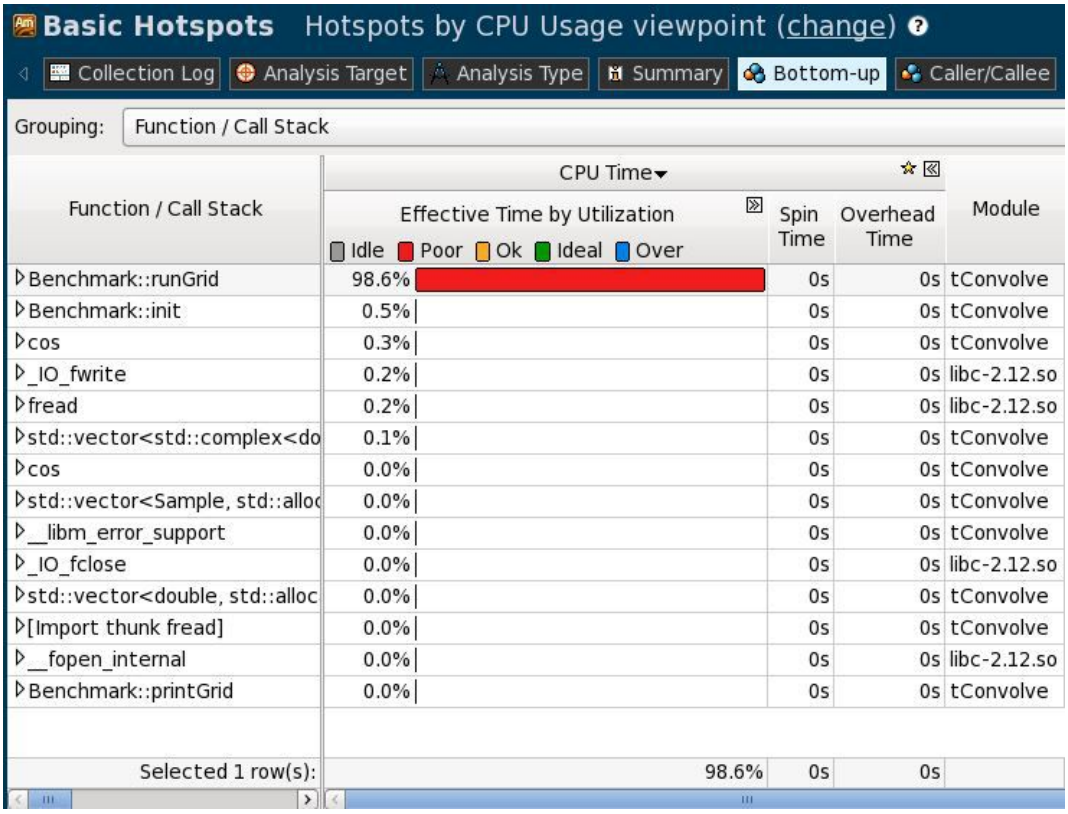


图 6 ： 热点耗时图

根据结果我们可以看出在 Benchmark 中 runGrid 函数运行时间占据 98.6%，而 runGrid 函数的内容便是运行 gridkernel 函数，因此 gridkernel 函数占据了大部分的时间，拖延运算速度。所以我们需要把 gridkernel 函数进行处理。下面先给出 gridKernel 函数代码。

3.4 插值模块关键代码

插值模块的 gridKernel()函数如下所示：

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The next function is the kernel of the gridding.
// The data are presented as a vector. Offsets for the convolution function
// and for the grid location are precalculated so that the kernel does
// not need to know anything about world coordinates or the shape of
// the convolution function. The ordering of cOffset and iu, iv is
// random.
//
// Perform gridding
//
// data - values to be gridded in a 1D vector
// support - Total width of convolution function=2*support+1
// C - convolution function shape: (2*support+1, 2*support+1, *)
// cOffset - offset into convolution function per data point
```

```
// iu, iv - integer locations of grid points
// grid - Output grid: shape (gSize, *)
// gSize - size of one axis of grid
void Benchmark::gridKernel(const int support,
                           const std::vector<Value>& C,
                           std::vector<Value>& grid, const int gSize)
{
    const int sSize = 2 * support + 1;

    for (int dind = 0; dind < int(samples.size()); ++dind) {
        // The actual grid point from which we offset
        int gind = samples[dind].iu + gSize * samples[dind].iv - support;

        // The Convoluton function point from which we offset
        int cind = samples[dind].cOffset;

        for (int suppv = 0; suppv < sSize; suppv++) {
            Value* gptr = &grid[gind];
            const Value* cptr = &C[cind];
            const Value d = samples[dind].data;
            for (int suppu = 0; suppu < sSize; suppu++) {
                *(gptr++) += d * (*(cptr++));
            }

            gind += gSize;
            cind += sSize;
        }
    }
}
```

gridKernel 函数被 runGrid()函数调用运行，runGrid()函数如下所示：

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//runGrid()函数
void Benchmark::runGrid()
{
    gridKernel(m_support, C, grid, gSize);
}
```

由于此函数为三个嵌套循环处理数据，所以我们考虑应用并行手段来处理它。常用的并行手段是 MPI 和 openMP 并行，本文将主要介绍 MPI 并行方法，下面将详细的介

绍如何用 MPI 进行并行化操作。

4 MPI 并行

4.1 MPI

Massage Passing Interface(MPI)是消息传递函数库的标准规范，支持 Fortran 和 C 语言，消息传递指的是并行执行的各个进程具有自己独立的堆栈和代码段，作为互不相关的多个程序独立执行，进程之间的信息交互完全通过显示地调用通信函数来完成。MPI 是一个新的库描述，且具有上百个可移植性的函数调用接口，在 Fortran 和 C 语言中可以直接对这些函数进行调用。各个厂商或组织遵循这些标准实现自己的 MPI 软件包，典型的实现包括开放源代码的 MPICH、LAM MPI 以及不开放源代码的 Intel MPI。由于 MPI 提供了统一的编程接口，我们只需要设计好并行算法，使用相应的 MPI 库就可以实现基于消息传递的并行计算。同时，MPI 支持多种操作系统，包括大多数的类 UNIX、Linux 和 Windows 系统。[8][9][10]

MPI 程序常用的编译器有 mpiCC /mpicc /mpif77 /mpif90 /C++ /C/F77 /F90。下面介绍常用的 MPI 函数：

表 4 ： 函数功能表

功能	含义
MPI_Init()	启动 MPI（初始化）
MPI_Finalize()	结束 MPI 计算
MPI_Comm_size()	确定进程数
MPI_Comm_rank()	返回进程 ID
MPI_Send()	发送消息
MPI_Recv()	接收消息
MPI_Wtime()	从过去某时刻到当前消耗的时间（秒）
MPI_Wtick()	MPI_Wtime()返回结果的精度
MPI_Initialized()	检查是否 MPI_Init 被调用过
MPI_Aboet()	中止 MPI 程序，用于出错处理

4.2 可并行性分析

由于原程序在 gridKernel 部分消耗过多时间，因此考虑用并行的思想来处理这个函数。根据并行的思想可以得到以下流程图。将任务同时分配给多个进程同是进行计算，再将计算结果合并。[11]

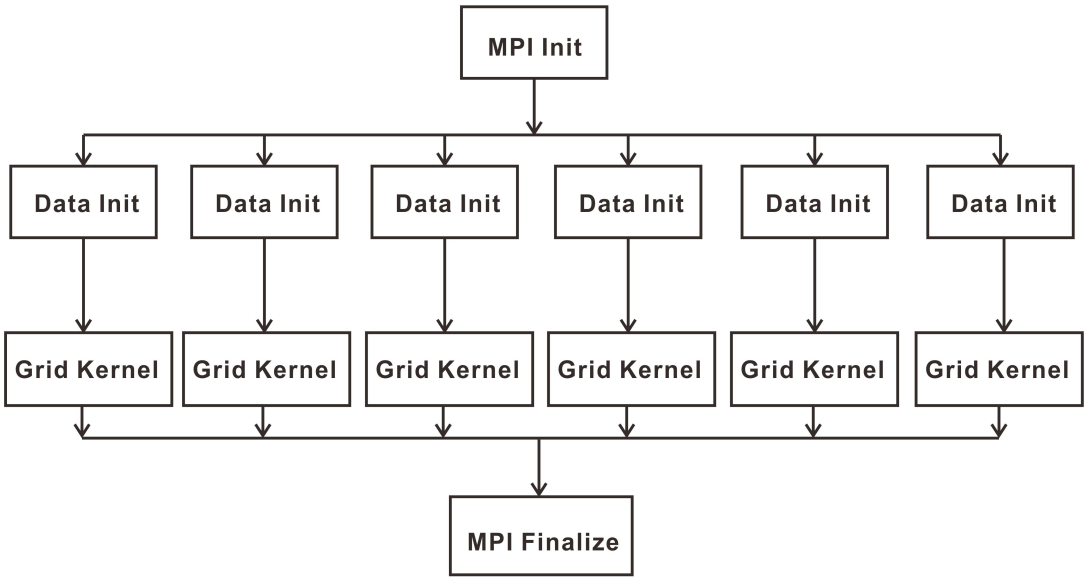


图 7 ： 并行结构图

并行思路一：并行“dind”循环（代码中红色标注）

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// data - values to be gridded in a 1D vector
// support - Total width of convolution function=2*support+1
// C - convolution function shape: (2*support+1, 2*support+1, *)
// cOffset - offset into convolution function per data point
// iu, iv - integer locations of grid points
// grid - Output grid: shape (gSize, *)
// gSize - size of one axis of grid
void Benchmark::gridKernel(const int support,
                           const std::vector<Value>& C,
                           std::vector<Value>& grid, const int gSize)
{
    const int sSize = 2 * support + 1;

    for (int dind = 0; dind < int(samples.size()); ++dind) {
        // The actual grid point from which we offset
        int gind = samples[dind].iu + gSize * samples[dind].iv - support;

        // The Convoluton function point from which we offset
        int cind = samples[dind].cOffset;

        for (int suppv = 0; suppv < sSize; suppv++) {
            Value* gptr = &grid[gind];
            const Value* cptr = &C[cind];
```

```
                const Value d = samples[dind].data;
                for (int suppu = 0; suppu < sSize; suppu++) {
                    *(gptr++) += d * (*(cptr++));
                }

                gind += gSize;
                cind += sSize;
            }
        }
}
```

并行思路二：并行“suppv”和“suppu”循环（代码中红色标注）

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// data - values to be gridded in a 1D vector
// support - Total width of convolution function=2*support+1
// C - convolution function shape: (2*support+1, 2*support+1, *)
// cOffset - offset into convolution function per data point
// iu, iv - integer locations of grid points
// grid - Output grid: shape (gSize, *)
// gSize - size of one axis of grid
void Benchmark::gridKernel(const int support,
                           const std::vector<Value>& C,
                           std::vector<Value>& grid, const int gSize)
{
    const int sSize = 2 * support + 1;

    for (int dind = 0; dind < int(samples.size()); ++dind) {
        // The actual grid point from which we offset
        int gind = samples[dind].iu + gSize * samples[dind].iv - support;

        // The Convoluton function point from which we offset
        int cind = samples[dind].cOffset;

        for (int suppv = 0; suppv < sSize; suppv++) {
            Value* gptr = &grid[gind];
            const Value* cptr = &C[cind];
            const Value d = samples[dind].data;
            for (int suppu = 0; suppu < sSize; suppu++) {
                *(gptr++) += d * (*(cptr++));
            }
```

```
                gind += gSize;
                cind += sSize;
            }
        }
    }
```

并行思路二的分析：根据上述原理进行编程与测试，我们会发现内层循环在计算量上非常小，时间占比也很小。如果进行了并行化处理，那么并行及通信的开销与外层循环结合时产生的时间开销反而更大，并行效果反转。因此我们不对内层循环做并行化处理。

综上所述，我们对最外层循环做并行化处理。

当我们进行并行操作，给不同的进程分配数据时，因为 gind 是随机的，在读取数据时，CPU 将数据从内存中取出来进行计算，CPU 会首先在内存中的缓存中寻找是否有需要的数据，如果有，那么 CPU 直接从缓存中取数据进行计算。相反的，如果缓存中没有，那么 CPU 便会从内存中取数据进行计算，并同时在缓存中建立新的数据块。当计算完成时，缓存中的数据就会被清除。如此，在下一次计算时，如果应用到了相同的数据或部分数据就无法直接从缓存中取得。需要花时间再从内存中寻找。当 CPU 在缓存中找到数据，则被定义为“命中”，反之被定义为“未命中”，每一次未命中会造成数据延时或者等待，当数据量很大很复杂时，就需要花费大量的时间，造成资源的浪费。如果我们在做之前进行了排序操作，CPU 需要数据时直接在缓存中可以找到，就直接在缓存中取数据。排序之后每一步计算都会用到部分重复的数据。在此之前，内存将数据读取到缓存中，缓存使用完成之后，数据还可以被继续使用，如此便可以省略从内存中读取所耗得时间。由于缓存的读取速度远远大于内存的读取速度，因此，在做排序之后对数据的读取是有利的，减少了内存的读取次数。提高了性能。[12]

具体观察此段代码：

```
int gind = samples[dind].iu + gSize * samples[dind].iv - support;
```

gind在做卷积运算时是在样本中进行移动的，那么移动的过程中有些数据就会被多次利用，因此缓存中的这些数据如果是在排序以后的，可以继续被使用。如果没有排序，那么就要在样本中重新搜索。每一次运算都要重新搜索，耗费了大量时间。因此对数据进行排序操作，排序中常用的速度快，稳定性高的排序方法是快速排序法。下面用快速排序来进行操作。

4.3 排序操作

快速排序法的基本思想是通过一次排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此使整个数据变成有序序列。在平均状况下，排序 n 个数要进行 O(n log n)次比较。快速排序通常明显比其他排序算法更快更稳定，因为它的内部循环可以在大部分的架构上很有效率地被实现出来。

算法的步骤为：

(1)、从数列中选出一个元素，把它当成一个基准。

(2)、重新排序数列，要求所有元素比基准值小的摆放在基准前面，所有元素比基准

值大的摆在基准的后面（如果有相同的数可以放到任意一边）。在这个操作之后，该基准就处于数列的中间位置。

(3)、用递归的方法按上面操作把小于基准值元素的子数列和大于基准值元素的子数列排序。最后就得到排序后的数列了。

排序后我们就可以进行并行操作了。排序的结果将在后面叙述。

4.4 并行操作

我们并行操作使用的集群硬件信息如下表所示:

表 5 ： 硬件平台信息表

名称	组件	型号
CPU 节点*15	CPU	Intel Xeon E5-2630V3*2
	内存	Samsung 8G ECC Registered DDR4 2133Mhz *4
	硬盘	160G MLC SSD
GPU 节点*2	CPU	Intel Xeon E5-2630V3 *2
	内存	Samsung 8G ECC Registered DDR4 2133Mhz *8
	硬盘	160G MLC SSD
MIC 节点*2	CPU	Intel Xeon E5-2630V3 *2
	内存	Samsung 8G ECC Registered DDR4 2133Mhz *8
	硬盘	160G MLC SSD

其中，CPU 节点有 15 个，GPU 节点 2 个，MIC 节点 2 个，每个节点有 16 个核心，每个核心对应一个进程，一共可以开 304 个进程。

在硬件搭建完成安装软件平台，软件平台具体参数如下所示：

表 6 ： 软件平台信息表

名称	版本
系统	Redhat Enterprise Linux Server release 6.5
函数库	i_mkl_2015.1.133
MPI	IMPI-5.0.2.044
编译器	ifort-15.0.1 icc-15.0.1
性能分析软件	Vtune_amplifier_xe_2015.1.1.38310

环境搭建完成后，就可以进行并行运算了。

根据上述可并行性分析和并行思路，作如下实现。

```
#include "Benchmark.h"

// System includes
#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
#include <limits>
#include <mpi.h>
Benchmark::Benchmark()
        : next(1)
{
}

int Benchmark::randomInt()
{
    const unsigned int maxint = std::numeric_limits<int>::max();
    next = next * 1103515245 + 12345;
    return ((unsigned int)(next / 65536) % maxint);
}

void Benchmark::init()
{
    // Initialize the data to be gridded
    MPI_Comm_size(MPI_COMM_WORLD, &np_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    u.resize(nSamples);
    v.resize(nSamples);
    w.resize(nSamples);
    samples.resize(nSamples*nChan);
    outdata.resize(nSamples*nChan);


    Coord rd;
    FILE * fp;
    if( (fp=fopen("randnum.dat","rb"))==NULL )
    {
      printf("cannot open file\n");
      return;
    }
```

```
    for (int i = 0; i < nSamples; i++) {
        if(fread(&rd,sizeof(Coord),1,fp)!=1){printf("Rand number read error!\n");}
       u[i] = baseline * rd - baseline / 2;
        if(fread(&rd,sizeof(Coord),1,fp)!=1){printf("Rand number read error!\n");}
        v[i] = baseline * rd - baseline / 2;
        if(fread(&rd,sizeof(Coord),1,fp)!=1){printf("Rand number read error!\n");}
        w[i] = baseline * rd - baseline / 2;

        for (int chan = 0; chan < nChan; chan++) {
            if(fread(&rd,sizeof(Coord),1,fp)!=1){printf("Rand number read
error!\n");}
            samples[i*nChan+chan].data = rd;
            outdata[i*nChan+chan] = 0.0;
        }
    }
    fclose(fp);

    grid.resize(gSize*gSize);
    grid.assign(grid.size(), Value(0.0));

    // Measure frequency in inverse wavelengths
    std::vector<Coord> freq(nChan);

    for (int i = 0; i < nChan; i++) {
        freq[i] = (1.4e9 - 2.0e5 * Coord(i) / Coord(nChan)) / 2.998e8;
    }

    // Initialize convolution function and offsets
    initC(freq, cellSize, wSize, m_support, overSample, wCellSize, C);
    initCOffset(u, v, w, freq, cellSize, wCellSize, wSize, gSize,
                m_support, overSample);
}

void Benchmark::runGrid()
{
    gridKernel(m_support, C, grid, gSize);
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
bool cmp(Sample a,Sample b){
        return (a.iu+4096*a.iv)<(b.iu+4096*b.iv);
```

```
}

void Benchmark::gridKernel(const int support,
                           const std::vector<Value>& C,
                           std::vector<Value>& grid, const int gSize)
{
    long *start,*end,*size;
    int i,j,k;
    if(my_rank==0){
        gridrecv.resize(gSize*gSize);
    }
    start=(long *)malloc(sizeof(long)*np_size);
    end=(long *)malloc(sizeof(long)*np_size);
    size=(long *)malloc(sizeof(long)*np_size);

    i=samples.size()/np_size;
    j=samples.size() % np_size;
    if(j==0){
        for(k=0;k<np_size;k++){
            start[k]=i*k;
            end[k]=i*(k+1)-1;
            size[k]=end[k]-start[k]+1;
        }
    }else{
        for(k=0;k<j;k++){
            start[k]=(i+1)*k;
            end[k]=start[k]+(i+1)-1;
            size[k]=end[k]-start[k]+1;
        }
        for(k=j;k<np_size;k++){
            start[k]=(i+1)*j+(k-j)*i;
            end[k]=start[k]+i-1;
            size[k]=end[k]-start[k]+1;
        }
    }

    const int sSize = 2 * support + 1;
    std::sort(&samples[start[my_rank]],&samples[end[my_rank]],cmp);
    for (int dind = start[my_rank]; dind <= end[my_rank]; ++dind) {

        // The actual grid point from which we offset
```

```
            int gind = samples[dind].iu + gSize * samples[dind].iv - support;
            // The Convoluton function point from which we offset
            int cind = samples[dind].cOffset;
            const Value* cptr = &C[cind];
            const Value d = samples[dind].data;

            for (int suppv = 0; suppv < sSize; suppv++) {
                Value* gptr = &grid[gind];
                for (int suppu = 0; suppu < sSize; suppu++) {
                    *(gptr++) += d * (*(cptr++));
                }
                gind += gSize;
                  }
        }

MPI_Reduce(&grid[0],&gridrecv[0],gSize*gSize,MPI_DOUBLE_COMPLEX,MPI_SUM
,0,MPI_COMM_WORLD);
}
```

4.5 排序及并行结果

程序运行结束后可以通过查看 SKA 目录下 log.dat 文件。此文件中输出了运行总时间和 gridding rate。按照上述快速排序的操作对原程序进行排序。分别测试排序前后在单节点运行的时间。得到下表。

表 7 排序时间对比表（单位：秒）

名称/次数/项目		Time
排序前	第 1 次	181.98
	第 2 次	181.65
	第 3 次	181.23
	Average	181.62
排序后	第 1 次	107.70
	第 2 次	107.83
	第 3 次	107.72
	Average	107.75
提升比率（倍）		1.68

由上表可以看出，排序后的程序在时间上大大缩短了。速度提升比率是之前的 1.68 倍。排序之后再进行并行操作。我们分配不同数量的进程数来运行这个程序，可以得到相应的结果。具体结果如下。

表 8 ：不同进程并行对比表（单位：秒）

进程数	2	4	8	16	32	64	128	240	304
第 1 次	53.72	28.97	20.15	24.65	14.78	9.92	6.00	4.00	2.95
第 2 次	53.56	28.88	19.55	25.25	14.77	9.84	5.99	3.55	2.96
第 3 次	53.54	28.91	19.72	24.36	14.61	9.91	5.98	3.55	2.94
Average	53.60	28.92	19.80	24.75	14.72	9.89	5.99	3.70	2.95

5 MPI 并行效果

根据 MPI 并行结果我们看出并行效果越来越好，下面我们计算并行之后的性能提升比率。

表 9 ： MPI 并行效果

项目/进程数	1	2	4	8	16	32	64	128	240	304
并行前后/秒	181.62	53.60	28.92	19.80	24.75	14.72	9.89	5.99	3.70	2.95
提升比率(倍)	1	3.39	6.28	9.17	7.34	12.34	18.36	30.32	49.09	61.57

我们用图表来描述性能来观察并行效果增长。见下图。

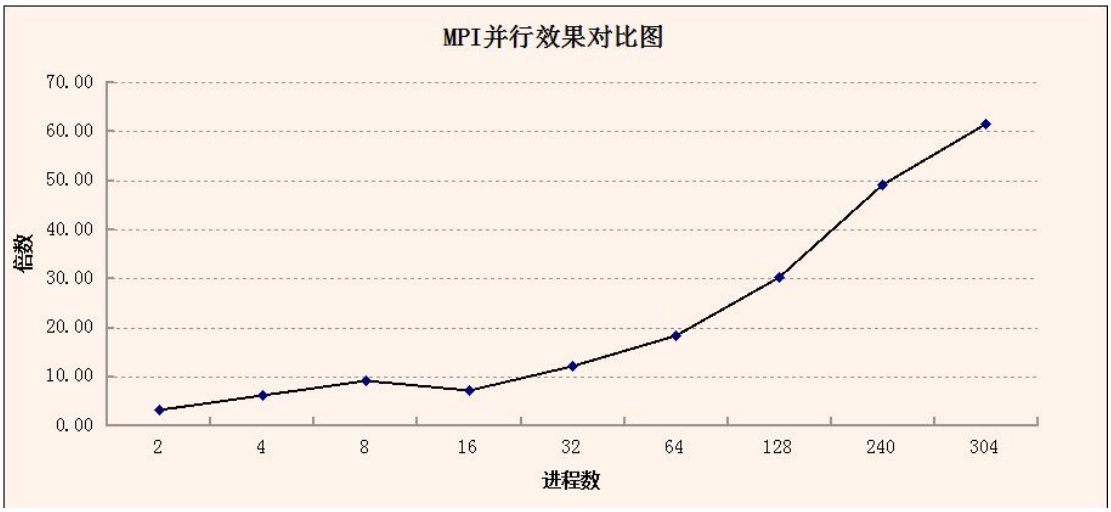


图 8 ： MPI 并行效果对比图

从上图的变化趋势来看，并行计算的思想对程序性能的提升非常大，我们在运行大型软件或大量数据计算时，计算过程往往会花费太多的时间，而导致我们无法在规定的时间范围内获得结果，并行化的出现大大缩短了时间开销，在一定程度内运用所有的硬件资源使程序的运算速度提升一定的幅度。达到我们的需求标准。并行计算将在未来一段时间内为我们做出巨大的贡献。

6 改进及推广

本文以 MPI 并行手段对热点函数进行并行并对程序进行排序操作。除了进程的并行处理技术之外还可以进行线程的并行。利用 openMP 并行技术可以对线程级别进行并行处理，进一步加快速度提高性能。openMP 可以单独使用或结合 MPI 一起使用。在应用 openMP 并行时会遇到以下问题：数据之间相互依赖不具有独立性，需要对程序进行修改，多线程调用“gind”可能会导致内存冲突的可能性增加，以致于计算失败。数据依赖也需要解决向量化的问题。除此之外，由于 gind 的随机性使得缓存的命中率下降，降低计算性能。在解决以上问题之后，并行的性能就会被进一步提升。算法的效率会被大大提高。除此之外，还可以应用 CUDA 技术等来提高性能。[13][14][15]

参考文献

【1】http://www.chinanews.com/gn/2015/11-20/7634126.shtml.　中国新闻网. 2016.05

【2】http://www.nrscc.gov.cn/nrscc/gjhz/ska/skaj/skajj/skawyj/.　国家遥感中心. 2016.05

【3】https://www.skatelescope.org/.　SKA Telescope-Square Kilometre Array. 2016.05

【4】http://china.skatelescope.org/welcome/.　SKA 中国-Square Kilometre Array. 2016.05

【5】SDP Element Concept.　Science Data Processor-SKA Work Packages（官网技术文档）

【6】张辉,胡广书. 基于二维卷积的图像插值实时硬件实现.生物医学工程　（2007 年 06 期）

【7】高奕,毕鹏. 英特尔软件开发工具介绍——Intel~VTune 可视化性能分析器.2004.10

【8】都志辉　编著.　高性能计算之并行编程技术—MPI 并行程序设计

【9】安竹林.　MPICH 并行程序设计环境简介.　2003

【10】陈国良.　并行计算－结构.算法.编程（修订版）. 北京：高等教育出版社，2003

【11】刘明生. 多核并行编程技术在加速数字图像处理中的应用.西安建筑科技大学.2010

【12】杨再奇，浅析缓存的意义，计算机技术应用　（2015 年 10 期）

【13】TaisukeBoku1,MitsuhisaSato,Masazumi Matsubara,Daisuke Takahashi.OpenMPI-OpenMP like tool for easy programming in MPI. . 2007

【14】Dana A.Jacobsen,Julien C.Thibault,Inanc Senocak.An MPI-CUDA Implementation for Messively Parallel Incompressi-ble Flow Computations on Multi-GPU Clusters. 48th AIAA Aerospace sciences Meeting . 2010

【15】Tsutomu Yoshinaga,Ta Quoc Viet.Optimization for Hybrid MPI-OpenMP Programs on a Cluster of SMP PCs

致谢

首先我要感谢我的母校，是他给了我学习的环境和知识的宝库，没有学校的支持，我不会取得任何优异的成绩。

我感谢我的导师张临杰副教授，是她的谆谆教导和细心的指导带领我走向科学研究的道路，她教会了我丰富的知识和本领，让我一生受用。

我感谢我的父母，在我上学期间给我的一切支持。

我感谢我的同学，在我的学习过程中给予我各种帮助和学习指导。

谨向所有关心和帮助我的老师、同学、朋友，以及我的那些亲戚们致以最衷心的感谢！

王秦豫