神威•太湖之光
并行程序设计与优化

（第一版）

Ver1.2

陈德训 刘鑫 编著

国家并行计算机工程技术研究中心

国家超级计算无锡中心

二〇一七年四月

内容说明

“神威•太湖之光”计算机系统是世界上首台峰值运算速度超过十亿亿次量级的超级计算机，也是中国第一台全部采用自主技术构建的世界第一的超级计算机。为让高性能计算应用开发人员更好的在“神威•太湖之光”计算机系统上开发出高效的并行应用软件，本书在整理“神威•太湖之光”计算机系统相关手册的基础上，增加了国家并行计算机工程技术研究中心多个研发团队多年来在“申威 26010”异构众核系统上的优化经验总结，这些团队包括：科学与工程计算研发团队、基础语言编译研发团队、并行程序设计环境研发团队、MPI 与性能评测研发团队等，在此表示感谢！

“神威•太湖之光”计算机系统部署在国家超级计算无锡中心，面向科学和工程计算的各个领域，使用的处理器是“申威 26010”异构众核处理器，该处理器采用了针对该处理器定制的申威指令系统，与 X86 指令系统不兼容。为了满足部分超算用户使用与 X86 指令系统兼容的商用应用软件系统的需求，在系统研制过程中增加了部分 X86 多核处理器作为辅助计算系统，但在“神威•太湖之光”计算机系统的主要指标中没有体现该部分资源。本书所介绍的大部分内容只涉及“申威 26010”异构众核处理器部分，X86 多核处理器部分因为采用了标准的集群系统架构，因此在本书中没有过多的描述。

本书在第三章和第四章介绍了“神威•太湖之光”加速线程库和 OpenACC 两种主要的并行编程模型；第五章介绍了串行编程的优化方法；第六章介绍高性能扩展数学库的使用方法；第七章介绍了超节点内与超节点间的通信优化措施；第八章介绍了几种常用的性能测试工具；第九章介绍了两个实际应用软件的优化实例，其中“航天飞行器跨流域数值模拟统一算法并行软件”由国家计算流体力学实验室李志辉老师团队开发，国家并行计算机工程技术研究中心应用团队负责优化，“全球大气浅水波方程显式求解软件”由中科院软件所杨超研究员和清华大学薛巍、付昊桓等老师所带领的团队联合研发并优化，是 2016 年度“戈登•贝尔”奖的预先研究课题。

该书主要面向“神威•太湖之光”计算机系统应用软件的开发和优化人员。本书可作为参考书或手册使用。

由于时间仓促以及环境的复杂性，其中有些不足或错误之处敬请各位读者指出并 Email 到 adch@263.net 或 yyylx@263.net,我们将在下一个版本中加以更正，在此先表示感谢！

编者

2017 年 3 月 16 日

目 录

第1章　系统简介

“神威•太湖之光”高效能计算机系统是由中国国家并行计算机工程技术研究中心研制的新一代超级计算机，该系统 2015 年 12 月完成研制，现部署在中国国家超级计算无锡中心。根据 2016 年 6 月 20 日国际 TOP500 排名公布的最新数据，“神威•太湖之光”计算机系统峰值运算速度每秒 12.54 亿亿次，持续运算速度每秒 9.3 亿亿次，性能功耗比每瓦 60.5 亿次。“神威•太湖之光”是世界上首台峰值运算速度超过 10 亿亿次量级的超级计算机，也是中国第一台全部采用自主技术构建的世界第一的超级计算机。

“神威•太湖之光”计算机系统采用面向高性能计算的可扩展多态复合架构，采用高密度组装、高效率直流供电、全机水冷等关键技术，配备精确的资源调度管理、丰富的并行编程语言和开发环境。

“神威•太湖之光”计算机系统采用的“申威 26010”异构众核处理器是由上海高性能集成电路设计中心通过自主技术研制，采用 64 位自主申威指令集，全芯片 260 核心，芯片标准工作频率 1.5GHz，峰值运算速度 3.168TFLOPS。

“神威•太湖之光”计算机高速计算系统峰值运算速度 125.436PFLOPS，内存总容量 1024TB，访存总带宽 4473.16TB/s，高速互连网络对分带宽 70TB/s，I/O 聚合带宽 341GB/s，实测 LINPACK 持续运算速度 93.015PFLOPS，LINPACK 效率 74.153%，系统功耗 15.371MW，性能功耗比 6051.131MFLOPS/W；辅助计算系统峰值运算速度 1.085PFLOPS，内存总容量 154.5TB；磁盘总容量 20PB。

1.1 系统总体架构

“神威•太湖之光”计算机系统是一台十亿亿次量级超大规模并行处理计算机系统，采用基于高密度弹性超节点和高流量复合网络架构、面向多目标优化的高效能体系结构。系统由高速计算系统、辅助计算系统、高速计算互连网络、辅助计算互连网络、高速计算存储系统、辅助计算存储系统和相应的软件系统等组成。计算资源由高速计算系统和辅助计算系统提供，存储资源由高速计算存储

系统和辅助计算存储系统提供。高速计算系统和辅助计算系统通过云管理环境进行统一管理，为用户提供统一的系统视图。系统总体结构如图 1-1 所示。

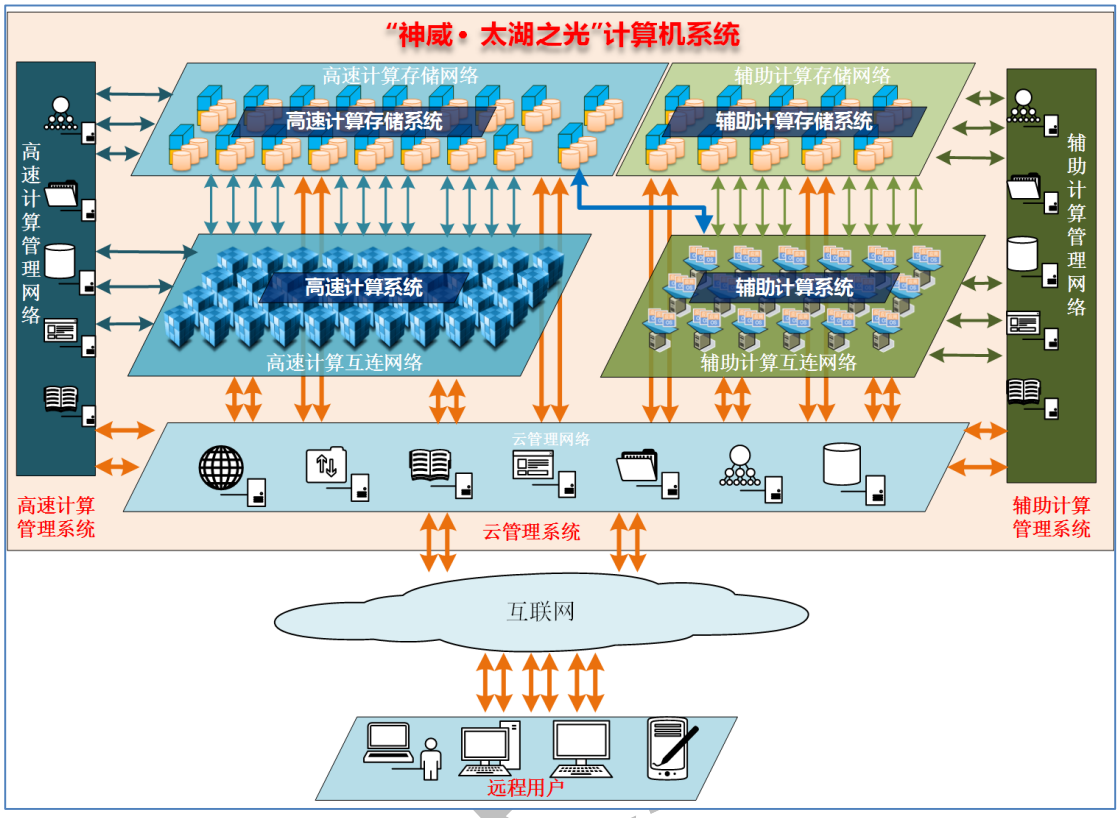


图 1-1 “神威•太湖之光”系统总体架构图

根据应用开发的需要，下面重点介绍与应用软件开发人员比较关心的“申威 26010”异构众核处理器、互连网络、语言环境的功能和性能指标。

1.2 “申威 26010”异构众核处理器

“申威 26010”异构众核处理器是由上海高性能集成电路设计中心通过自主技术研制，采用片上计算阵列集群和分布式共享存储相结合的异构众核体系结构，使用 64 位自主申威指令系统。“申威 26010”异构众核处理器集成了 4 个运算核组共 260 个运算核心，核组间支持 Cache 一致性。每个核组包含 1 个运算控制核心（主核）和 1 个运算核心阵列（从核阵列），运算核心阵列由 64 个运算核心（从核）、阵列控制器、二级指令 Cache 构成，4 个核组的物理空间统一编址，运算控制核心和运算核心均可以访问芯片上的所有主存空间。处理器集成

4 路 128 位 DDR3 存储控制器、8 通道 PCIe3.0、千兆以太网接口和 JTAG 接口。具体参见图 1-2 所示。

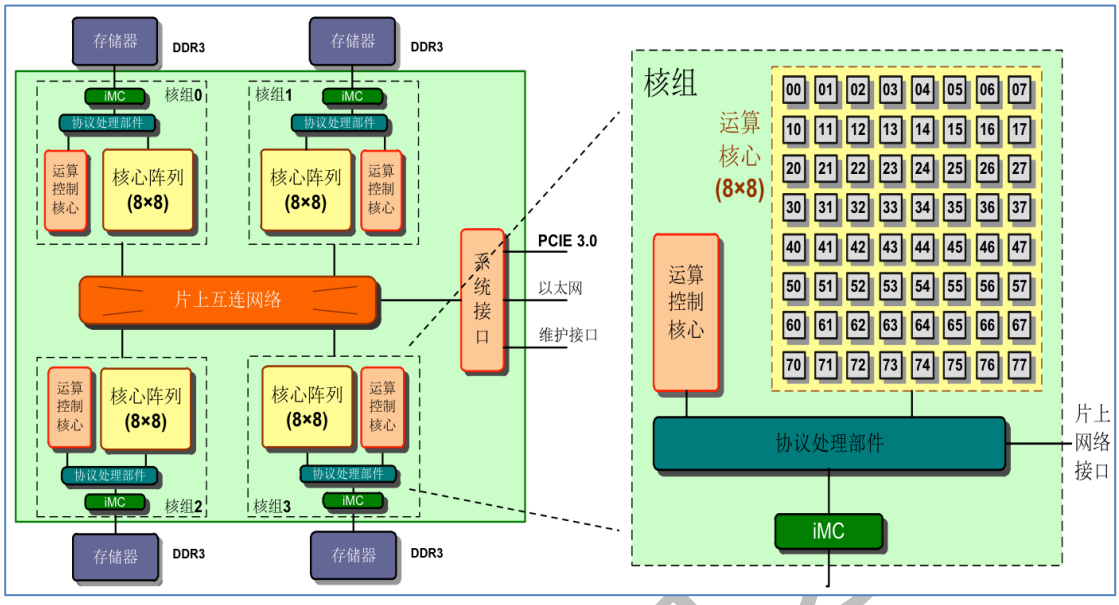


图 1-2“申威 26010”异构众核处理器架构图

1.2.1 主要技术指标

“申威 26010”异构众核处理器主要性能指标如下：

1） 主核和从核工作频率 1.5GHz；

2） 计算能力：

a） 主核的峰值速度为双精度浮点 24GFLOPS、单精度定点 16.5GIPS；

b） 从核的峰值速度为双精度浮点 12GFLOPS、单精度定点 13.5GIPS；

c） 单芯片峰值速度为双精度浮点 3.168TFLOPS、单精度定点 3.522TIPS。

3） 主核存储系统：

a） 一级指令 Cache：32KB；

b） 一级数据 Cache：32KB；

c） 二级 Cache：512KB。

4） 从核存储系统：

a） 一级指令 Cache：16KB；

b） 二级指令 Cache：同一核组内运算核心共享 64KB；

c） 每个核心含 64KB 可重构局部数据存储，支持配置为软硬件协同 Cache 使用。

5） 集成 4 路 128 位 DDR3 存储控制器，访存带宽为 136.51GB/s，支持最大主存容量配置 128GB；

6） 芯片采用 PCIe 3.0 标准接口，8 通道，双向峰值带宽 16GB/s；

7） 以太网标准接口带宽为 1000Mbps。

1.2.2 主核性能参数

1） 主核 Cache 延迟

表格 1-1 主核 Cache 延迟拍数

项目	Cycles
DL1	4
DL2	13
可 Cache 空间延迟	154

2） 主核访问 ldm 延迟

表格 1-2 主核访问 ldm 延迟拍数

项目	Cycles
Ldm 访问延迟	94

3） 主核带宽性能

表格 1-3 主核 stream 性能测试

项目	访存带宽
Stream Triad	9.918GB/s

4） 主核跨核组访存性能

主核跨核组访存性能下降约 20%，实际应用课题因为编译器预取优化，访存性能将下降约 10%。

1.2.3 从核基础性能

1） gld/gst 性能

gld/gst 延迟测试时使用 ldl，gld&gst 使用先写后读的方式；带宽测试时使用 vldd 测试。

表格 1-4 从核延迟

内容	Cycles
gld 单核延迟	177
gld&gst 延迟	278

2） ld/st 性能

表格 1-5 从核 ld/st 性能

内容	测试结果
Ld 延迟 cycle	4
Ld 带宽 GB/s	47.90

3） DMA 性能

表格 1-6 单从核模式 DMA_GET 性能

粒度	1pebw(GB/s)	64pe bw(GB/s)
8B	0.05	0.99
16B	0.10	1.99
32B	0.20	3.92
64B	0.41	7.96
128B	0.80	15.77
256B	1.47	28.88
512B	2.65	28.98
1024B	4.28	27.97
2048B	6.33	30.48
4096B	8.14	30.18
8192B	9.47	30.78
16384B	10.60	30.94

“申威 26010”异构众核处理器的访存带宽理想值为 134.4GB/s，表 1-6 为单核组 DMA_GET 的实测性能数据，测试时主存地址递增、每次都判回答字。

4) 寄存器通信性能

表格 1-7 点对点寄存器通信性能

测试项目	Cycles
点对点延迟	10 拍

表格 1-8load 并广播性能

测试项目	Cycles
行广播延迟	14 拍
列广播延迟	14 拍

1.2.4 从核访问主存方式

“神威•太湖之光”计算机系统每个计算节点包含一个众核处理器，内存 32GB，众核处理器每个核组本地内存 8GB，从核可以通过 gld/gst 直接离散访问主存，也可以通过 DMA 方式批量访问主存，从核阵列之间可以采用寄存器通信方式进行通信。

每个从核局部存储空间大小为 64KB，指令存储空间为 16KB。

1.3 高速计算系统

“神威•太湖之光”高速计算系统采用紧耦合超节点架构，由 40960 块“申威 26010”异构众核处理器、20480 块计算板节点组成，每块计算节点板包含 2 颗“申威 26010”异构众核处理器，并通过计算插件板、计算超节点和计算机仓等模式进行系统扩展，构成 125.436PFLOPS 高速计算系统，高速计算系统体系结构扩展如图 1-3 所示。

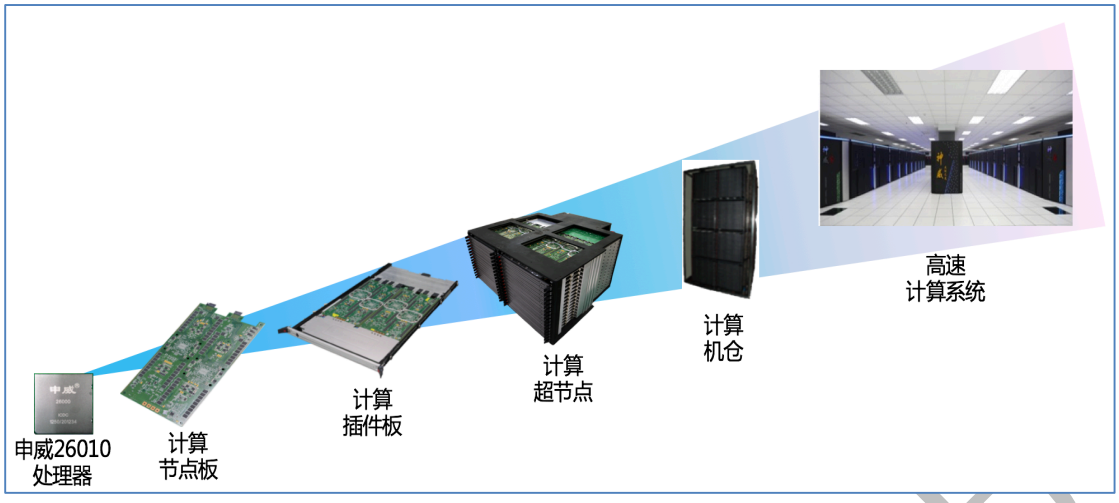


图 1-3 高速计算系统体系结构扩展示意图

计算节点板是高速计算系统的基本组成单元，以“申威 26010”异构众核处理器为核心，访存带宽 136.51GB/s，主存容量 32GB，网络接口双向带宽 16GB/s。

计算超节点由 256 个计算节点经过全互连的交叉开关紧密互连而成，主要技术指标：

1）运算节点数：256 个；

2）峰值性能：811.01TFLOPS；

3）访存带宽：34.95TB/s；

4）主存总量：8TB。

1.4 辅助计算系统

“神威•太湖之光”辅助计算系统采用 X86 集群架构，InfiniBand FDR 网络互连，网络接口双向带宽 16GB/s，主要配置如下：

1） 普通计算节点：

- 980 台，每台计算节点包含 2 路 12 核心英特尔至强 E5-2680 V3 处理器，主频 2.5GHz、DDR4 内存 128GB；
- 处理器总核数：23520 个；
- 内存总容量：122.5TB；
- 通信网络带宽：双向 14GB/s。

2） 大内存计算节点：

- 32 台，每台计算节点包含 8 路 16 核心英特尔至强 E7-8860 V3 处理器，主频 2.2GHz、DDR4 内存 1TB、SAS 本地存储 7TB、SSD 本地存储 500GB；
- 处理器总核数：4096 个；
- 内存总容量：32TB；
- 通信网络带宽：双向 28GB/s。

3） GPU 计算节点：

- 64 台，每台计算节点包含 2 路 8 核心英特尔至强 E5-2630 V3 处理器，主频 2.4GHz、DDR4 内存 128GB、7KRPM 本地存储 1TB、SSD 本地存储 1.6TB、一块 NV Tesla K40 计算加速卡（具有 2880 个流处理器;显存 12GB;双精度 浮点性能可达 1.4TFLOPS）；
- 处理器总核数：1024 个；
- 通信网络带宽：双向 14GB/s。

1.5 互连网络

互连网络包括高速计算互连网络和辅助计算互连网络，分别是负责高速计算系统和辅助计算系统的互连。

1.5.1 高速计算互连网络

高速计算互连网络负责把高速计算系统所有计算节点和存储节点连接为一个有机的整体，互连网络采用多层级胖树交叉的混合拓扑网络结构，实现全机计算节点和存储服务节点的高带宽、低延迟通信，有效支持计算密集、通信密集和 I/O 密集等多种类型课题的运行。高速计算互连网络拓扑连接如图 1-4 所示。

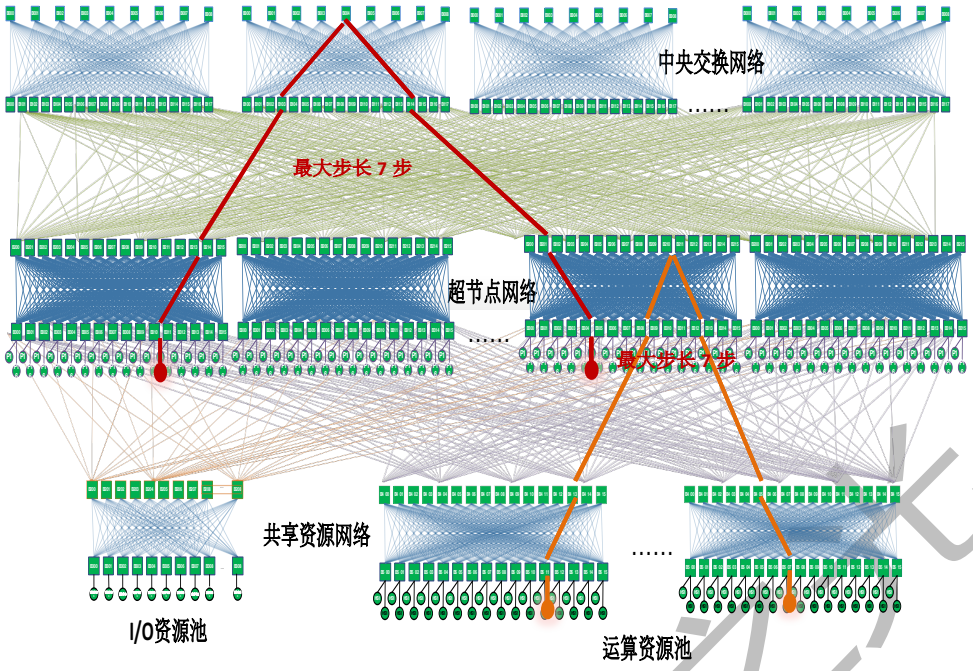


图 1-4 高速计算互连网络拓扑连接示意图

计算超节点网络模块采用两级胖树全交叉的互连结构，实现 256 颗“申威 26010”异构众核处理器之间的全连接通信，支持超过 65536 核心的高效并行计算。超节点网络模块之间通过中央交换网络模块实现互连，并同时直接连接到共享资源池网络模块。

中央交换网络模块完成超节点网络模块之间的上层网络互连。根据对各种规模并行课题的分析，并结合工程可实现性，对上层网络和下层网络互连的带宽进行合理配置。超节点网络按照 4:1 裁剪连接到中央交换网络，按照每 4 颗 CPU 分配 1 个网络端口连接到中央交换网络，即 256 个 CPU 一共分配 64 个网络端口连接到中央交换网络。

1.5.2 辅助计算互连网络

辅助计算互连网络用于辅助计算系统和辅助存储系统内所有节点之间的高速互连，采用标准商用高速 Infiniband 交换系统，双向通信带宽达到 112Gbps。

1.6 存储系统

“神威•太湖之光”包括高速计算存储系统和辅助计算存储系统，总共容量为 20PB。高速计算存储系统包含：online1、online2，辅助计算存储系统包括 GPFS 和 EMC，系统管理员可根据用户和课题使用特点分配不同的存储资源。

- online1：面向高速计算系统和辅助计算系统使用
- online2：面向高速计算系统
- GPFS：面向辅助计算系统
- EMC：面向云计算系统和数据备份

1.7 语言环境

“神威•太湖之光”计算机系统语言环境包括基础编程语言、并行编程语言和接口、用户使用环境、基础编程环境和工具等四部分组成，如图 1-5 所示。

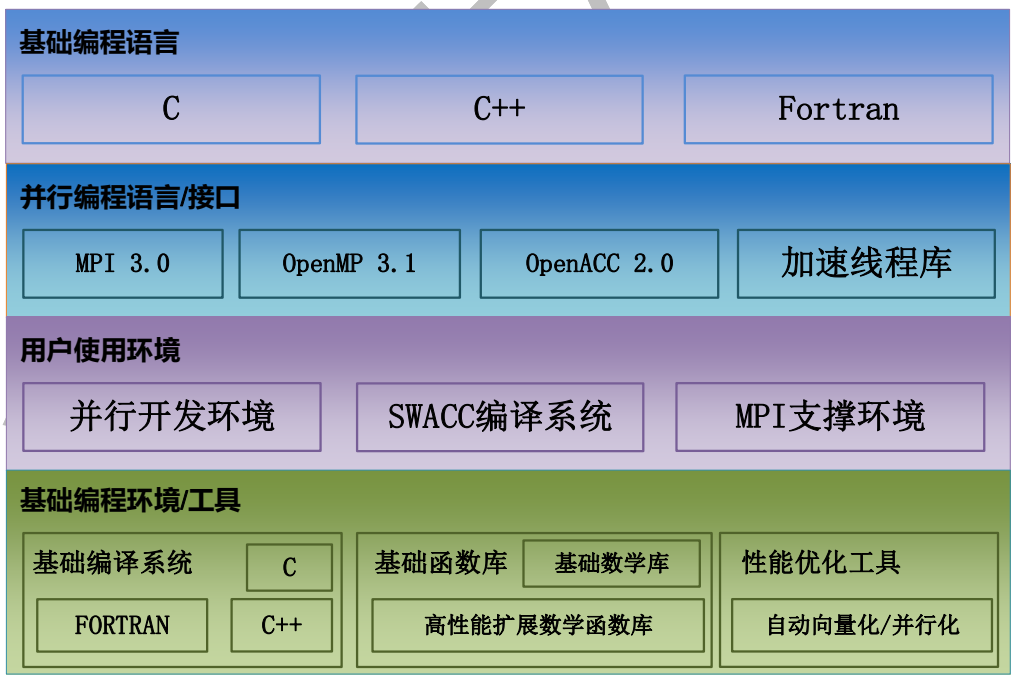


图 1-5 “神威•太湖之光”计算机系统语言环境组成结构图

1.7.1 基础编程语言

“神威•太湖之光”计算机系统支持主流的基础编程语言，满足实际课题编译需求。主要编程语言包括：

- C 语言，支持 C99 标准；
- C++语言，支持 C++03 和 C++11 标准，目前从核不支持 C++语言；
- Fortran 语言，支持 Fortran2003 标准中主要的功能。

1.7.2 并行编程语言/接口

“神威•太湖之光”计算机系统支持与国际接轨的并行编程标准，包括 MPI3.0、OpenMP3.1、Pthreads、OpenACC2.0，支持消息并行编程模型、共享并行编程模型、加速并行编程模型，满足科学与工程计算课题开发和移植的多样性需要。同时提供为“申威 26010”异构众核处理器结构特点定制的加速线程库编程接口，满足部分追求极致性能应用课题的开发需求。

1.7.3 用户使用环境

“神威•太湖之光”计算机系统的并行开发环境，为本地用户提供图形界面的集成开发环境，包括代码的编辑、编译、调试和性能监测。为远程用户提供字符界面的开发环境，用户通过远程终端调用系统提供的命令行工具来使用，兼容 Linux 的绝大部分命令。

1.7.4 基础编程环境

“神威•太湖之光”计算机系统的基础编程环境是所有上层语言及工具的基础，提供基础语言、主从异构编程、基础函数库、自动向量化/并行化等支持，提供丰富高效的编译优化功能。

1.8 运行模式

根据“申威 26010”异构众核处理器结构特点，“神威•太湖之光”计算机系统为课题提供两种内存运行模式：

1） 核组私有模式：该模式是“神威•太湖之光”系统应用课题的主要使用的模式，每个核组共享使用各自的 8GB 私有内存；

2） 全片共享模式：为满足部分课题大内存的需求，每个运行进程都可以访问到全片 32GB 内存。

每种模式中可支持多种具体的并行方式，具体如表格 1-9 所示，其中 L1 表示第一级并行，L2 表示第二级并行，L3 表示第三级并行。

表格 1-9 “神威•太湖之光”计算机系统单节点并行模式

序号	分类	使用方式	应用场景
1	核组私有模式	L1：4 个 MPI 进程	纯主核应用，每个主核运行一个 MPI 进程，可以采用该模式来调试和运行用于已有的纯 MPI 并行程序
		L1：4 个 MPI 进程 L2：4 个从核组(OpenACC*/Athread)	该并行方法是系统最主要使用方式。采用两级并行，第一级并行运行在主核上，第二级并行运行在从核组上，主核与从核组一一对应
2	全片共享模式	L1：1 或 2 个 MPI 进程	纯主核应用，MPI 进程内存需求大于 8GB；可以把全片或 2 个核组的内存供 1 个 MPI 进程使用
		L1：1 个 MPI 进程 L2：4 个线程(OpenMP/Pthreads)	纯主核应用，采用传统的 MPI+OpenMP 或 MPI+Pthreads 两级并行
		L1：2 个 MPI L2：2 个线程(OMP/Pthreads)	纯主核应用，传统两级并行；MPI 进程的内存需求

			大于 8GB、但小于 16GB
		L1：1 或 2 个 MPI 进程 L2：1 或 2 个核组 (OpenACC*/Athread)	MPI 私有空间内存需求大，使用从核； 全片的内存或 2 个核组的内存供 1 个 MPI 进程使用
		L1：1/2 个 MPI L2：OMP/Pthreads L3：Athread	三级并行课题；满足少部分课题大规模扩展效率和大内存需求；该方法很少使用
		master 模式	全片共享模式的上述用法中均包含一种 MPI 的 master 使用模式，即 0 号进程独占一个节点内存(内存需求大)，其他节点内存正常使用。 使用方式是在作业提交命令行处加上“-master”选项就可以，该道作业的 0 号进程就可以独占一个计算节点，使用到 32GB 内存

1.9 异构并行方法

“神威•太湖之光”计算机系统大部分主要采用主从加速并行、主从协同并行、主从异步并行和主从动态并行四种异构并行方法，如图 1-6 所示。

1.9.1 主从加速并行

主从加速并行方法是“神威•太湖之光”计算机系统大部分课题采用的两级并行方法，应用课题的计算核心通过 Athread 或者 OpenACC*被加载到从核上进行加速计算，而主核只完成应用程序的通信、I/O 和部分串行代码的计算，从核在计算核心段过程中，主核处于等待状态，直到从核完成该核心段的计算任务。参见图 1-6-A 所示。

1.9.2 主从协同并行

这种并行方法是主核和从核作为对等的个体进行并行计算，根据各自计算能力进行负载分配，共同完成核心段的计算。如果课题对 LDM 需求不高，在 LDM 范围内从核可以完成整个核心段的计算，则该方式可以减少访存开销，并能够获得较好的并行效果。但该并行方法在任务或数据划分时比较复杂，如果不要求极致的并行和计算效率，一般不考虑该种并行方法。参见图 1-6-B 所示。

1.9.3 主从异步并行

该种并行方法是在从核进行加速计算的同时，主核不等待，而是进行其他如计算、通信或 I/O 等操作，从而提高主从协作的计算效率。该并行方法实现起来比较简单，但优化效果比较好，所以大部分应用课题都采用该种并行方法进行优化。参见图 1-6-C 所示。

1.9.4 主从动态并行

该种并行方法是主核负责任务分配，从核负责任务计算并回写计算结果，该并行方法比较适合从核计算任务的计算时间不固定或者针对一些任务并行的程序，为了提高计算和并行效率，可以考虑采用两级主从并行方式进行大规模并行计算，第一级是 MPI 进程级的主从并行，第二级为异构众核处理器核组内部的主从并行。参见图 1-6-D 所示。

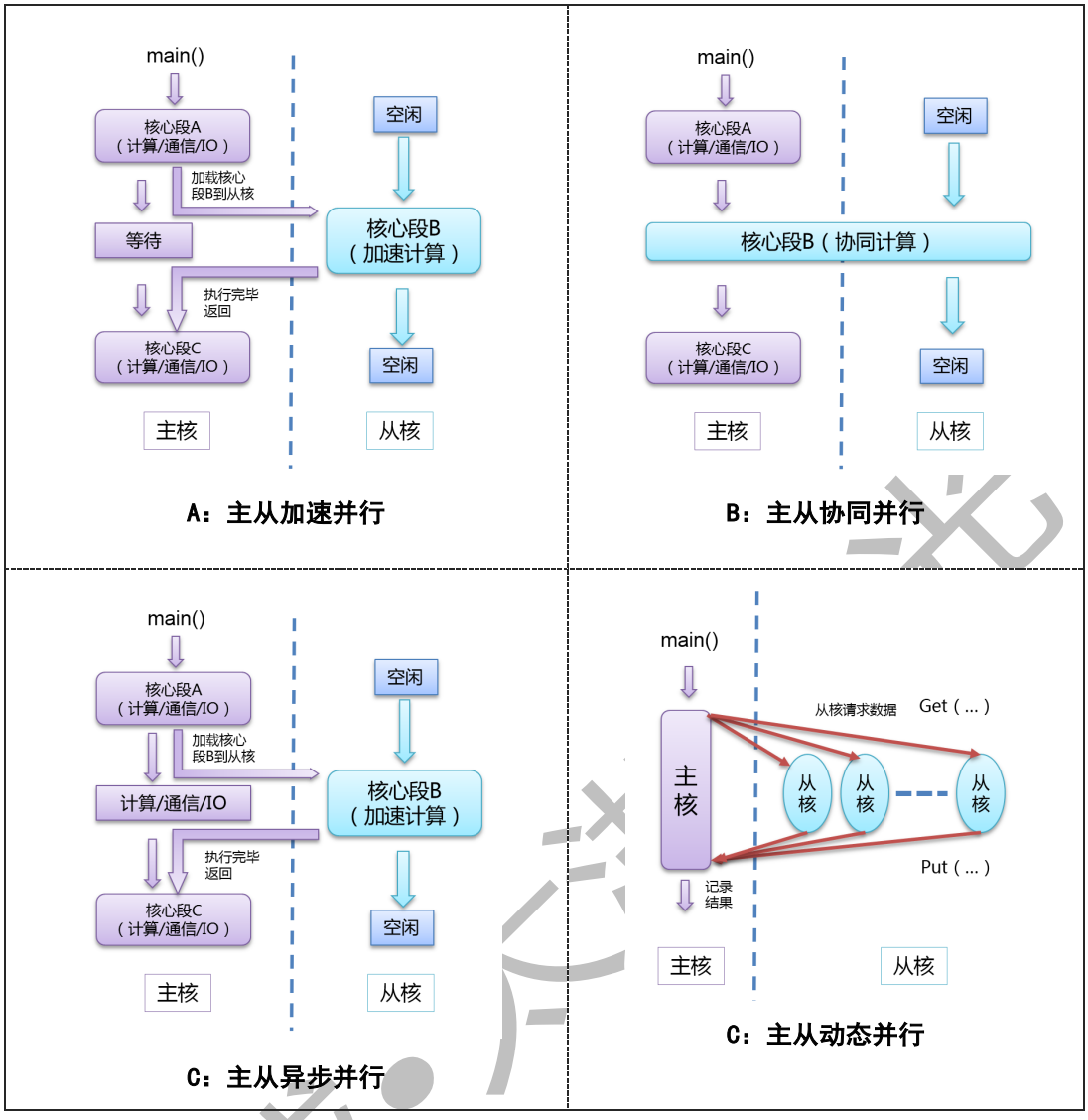


图 1-6 “神威•太湖之光”计算机系统异构并行方法

第2章　用户使用流程

使用部署在国家超级计算无锡中心的“神威•太湖之光”计算机系统，可以到国家超级计算无锡中心或者远程在互联网上通过 VPN 连接来使用，但前提都需要向国家超级计算无锡中心申请系统账户。

2.1 账户申请

使用“神威•太湖之光”计算机系统资源，必须先申请系统账户。通过联系国家超级计算无锡中心市场推广部门人员或登入国家超级计算无锡中心官网（http://www.nsccwx.cn），在“上机指南”页面下载《国家超级计算无锡中心用户上机申请表 2016》并填写，发送到 info@mail.nsccwx.cn，超算中心管理人员会尽快根据你的上机申请与您联系并开户。

系统管理员会根据您申请表的用户信息，为您提供 VPN 连接账户、系统账户、远程使用“神威•太湖之光”计算机系统的步骤等信息。

2.2 用户环境

用户通过 VPN 进行网络连接，采用支持 SSH 协议的客户端就可以在远程登入“神威•太湖之光”计算机系统。在一个账户下，即可以使用“申威 26010”异构众核处理器提供的高速计算系统，也可以使用 X86 多核处理器提供的辅助计算系统，选择何种计算资源，由用户在作业提交的命令行根据程序运行的需求来指定。

在数据存储方面，系统管理员会根据您申请账户时的需求情况，在您账户的 HOME 目录挂上您所需的高速计算存储系统或辅助计算存储系统。

用户登入的环境是标准的 Linux 环境，绝大部分的 Linux 命令都可以使用，用户可以在该环境下进行程序代码的开发、编译和连接，并通过系统提供的作业管理命令，提交可执行程序到相应的计算资源进行程序的调试与计算。

用户远程登入的环境只提供程序开发、调试和小规模数据处理使用，用户切不可在该环境下运行用户的程序任务，系统管理员有权阻止这些程序的运行。

2.3 计算资源

超算中心为每个新用户提供了两个免费的开发调试资源队列：

- 队列 q_sw_expr：免费的高速计算系统开发调试队列
- 队列 q_x86_expr：免费的辅助计算系统开发调试队列

免费开发调试队列的作业任务可运行的最长时间为 60 分钟，最大并行规模为 64，当用户完成程序调试和优化后，需要大规模计算资源时需要再申请收费资源。

2.4 存储资源

“神威•太湖之光”计算机系统提供若干个全局存储资源，系统管理员会根据用户申请账户时的需求情况，选择一个或两个适合用户课题特点的全局文件系统，用户登入系统时，就可以看到所选择的全局文件系统已经挂在 HOME 目录下面。如果需要，后续也可以根据课题运行情况调整使用不同的全局文件系统。

为系统管理需要，“神威•太湖之光”计算机系统只为用户提供少量的 HOME 存储空间（暂定 100MB），用户在实际使用时，建议所有与课题相关的代码或数据都保存在全局文件系统中。

切记：用户的 HOME 目录可以保存环境配置文件和少量的源代码，但不要保存用户数据，特别是不要保存用户课题计算输出的数据。

2.5 编译环境

“神威•太湖之光”计算机系统包含由“申威 26010”异构众核处理器研制的高速计算系统和由 X86 多核处理器组装的辅助计算系统两部分，因此该系统也部署了两个完全不同的编译系统环境。

2.5.1 高速计算系统编译环境

“神威•太湖之光”计算机系统高速计算系统使用神威系列编译环境。

1） 神威基础编译命令：

神威基础编译环境包括 C、C++和 Fortran 编译器，针对“申威 26010”异构众核处理器的主核、从核的程序编译和链接采用了同一编译命令不同编译选项来实现，具体参见表格 2-1 所示，常用编译选项参见表格 2-2。

- C 语言：
  - 主核代码：sw5cc -host［选项］文件名
  - 从核代码：sw5cc -slave［选项］文件名
  - 混合链接：sw5cc -hybrid［选项］主核文件名 从核文件名
- C++语言：
  - 主核代码：sw5CC -host［选项］文件名
  - 从核代码：不支持
  - 混合链接：sw5CC -hybrid［选项］主核文件名 从核文件名
- Fortran 语言：
  - 主核代码：sw5f90 -host［选项］文件名
  - 从核代码：sw5f90 -slave［选项］文件名
  - 混合链接：sw5f90 -hybrid［选项］主核文件名 从核文件名

表格 2-1 神威基础编译命令列表

语言代码	主核代码	从核代码	混合链接
C	sw5cc -host	sw5cc -slave	sw5cc -hybrid
C++	sw5CC -host	不支持	sw5CC -hybrid
Fortran	sw5f90 -host	sw5f90 -slave	sw5f90 -hybrid

2） MPI 编译命令：

- C 语言：mpicc ［选项］文件名
- C++语言：mpiCC ［选项］文件名
- Fortran77 语言：mpif77［选项］文件名
- Fortran90 语言：mpif90［选项］文件名

3） OpenACC*编译命令：

- Fortran 语言：swafort ［选项］文件名

- Fortran 编译指示格式：添加!$acc 编译指示名 [子语[[,]子语]...]

- C 语言：swacc [选项] 文件名
  - C 语言编译指示格式：添加#pragma acc 编译指示名 [子句[[,]子句]...]

4) MPI+OpenACC*编译命令：

- Fortran 语言：swafort [选项] 文件名
  - Fortran 编译指示格式：添加!$acc 编译指示名 [子语[[,]子语]...]
  - MPI 库函数由 swafort 负责链接
- C 语言：swacc [选项] 文件名
  - C 语言编译指示格式：添加#pragma acc 编译指示名 [子句[[,]子句]...]
  - MPI 库函数由 swacc 负责链接

5) MPI+Athread 编译命令：

- C 语言：
  - 主核代码：sw5cc -host [选项] 文件名
  - 从核代码：sw5cc -slave [选项] 文件名
  - 混合链接：mpicc -hybrid [选项] 主核文件名 从核文件名
- C++语言：
  - 主核代码：sw5CC -host [选项] 文件名
  - 从核代码：不支持（使用 C 语言）
  - 混合链接：mpiCC -hybrid [选项] 主核文件名 从核文件名
- Fortran 语言：
  - 主核代码：sw5f90 -host [选项] 文件名
  - 从核代码：sw5f90 -slave [选项] 文件名
  - 混合链接：mpif90 -hybrid [选项] 主核文件名 从核文件名

OpenACC*编译器常用编译选项参见表格 2-3。

表格 2-2 基础编译环境常用编译选项

选项	作用
-c	为每个源文件生成一个中间目标文件，但是不进行链接
-g	为之后的调试生成调试符号信息
-I<dir>	为预处理头文件添加查找<路径>
-l<library>	在链接阶段指定需要链的库文件
-L<dir>	为链接阶段添加查找<路径>
-lm	在链接阶段使用 libm 数学库，在 C 程序中若使用了 exp、log、sin 及 cos 等函数则需要调用该库
-o <filename>	指定生成的可执行(库)文件的名字
-O1 ~ -O3	生成高级优化的可执行代码
-pg	为 gprof 分析程序生成反馈信息
-msimd	打开 simd 功能模块
-OPT：IEEE_arch=1	浮点异常处理（Fortran）
-mieee	浮点异常处理（C++）
-convert big_endian	按照大端读写文件（Fortran）
-freeform -fixedform	Fortran 自由书写格式 Fortran 固定书写格式

表格 2-3 OpenACC*编译器常用编译选项

编译选项	说明
-|--(h|help)	显示帮助
-SCFlags	指定的选项将被传递到编译 device 程序的串行编译器，若同时传递多个选项需使用逗号进行分割，中间不能有空格，例如： swafort -SCFlags -extend_source,-O3 hello.c
-HCFlags	指定的选项将被传递到 host 程序的基础编译器，若同时传递多个选项需使用逗号进行分割，中间不能有空格。
-LFlags	指定的选项将被传递到链接器，若同时传递多个选项需使用逗号进行分割，中间不能有空格。
-priv	辅助选项：对加速区进行私有化变量分析，给出变量的读写属性，遇到无法识别的函数调用则分析终止
-priv:ignore_call	辅助选项：对加速区进行私有化变量分析，给出变量的读写属性，忽略加速区内可能存在的函数调用的影响
-arrayAnalyse	辅助选项：数组访问模式分析，可以给出可 copy 的数组的建议
-ldmAnalyse	辅助选项：设备内存使用情况分析，编译之后在运行时会反馈各加速计算区内对设备内存的使用情况及相关建议
-preinline	辅助选项：对含有#inline 指示的函数调用语句中的函数进行内联，不对其他的加速指示进行处理，主要用于辅助函数内联
-preinline:all	辅助选项：在-preinline 的基础上，对同一个函数内的所有同名的被调用函数进行内联。
-dmaReuse	针对 data copy 指示，自动进行数据重用优化
-autoSwap	自动对适合的数组进行转置处理和优化
-v|-version	显示编译器和运行时库的版本号
-Minfo	显示编译器关于程序的分析信息
-keep	保留编译的中间文件
-dumpcommand [dumpfile]	将对中间文件的编译命令输出到 dumpfile 中

2.5.2 辅助计算系统编译环境

“神威•太湖之光”计算机辅助计算系统采用 X86 服务器构建,部署了 Intel 系列编译环境。

1) 串行程序编译命令:

- Fortran 语言：ifort［选项］文件名
- C 语言： icc［选项］文件名
- C++语言：icpc［选项］文件名

2) MPI 并行程序编译命令：

- Fortran 编译器：mpiifort［选项］文件名
- C 语言编译器：mpiicc［选项］文件名
- C++语言编译器：mpiicpc［选项］文件名

3) Intel 编译器常用编译选项：

表格 2-4 Intel 编译器常用编译选项

编译选项	说明
-01	最大优化速度，但是关闭一些会增大文件大小而对速度提升很小的选项
-02	最大优化速度（默认）
-03	最大优化速度，并打开更多激进的但不是对所有程序都能提高性能的选项
-0	同-02
-0s	打开优化选项，但是关闭一些会增大文件大小而对速度提升很小的选项
-00	关闭优化选项
-fast	等同打开 -xHOST -03 -ipo -no-prec-div -static
-Ofast	等同打开-03 -no-prec-div optimizations
-fno-alias	不采用代码混淆(用于保护代码，防反编译)
-fno-fnalias	不采用内部函数混淆，只混淆外部调用
-nolib-inline	禁用固有函数在内部扩展
-ftz	将非正常数值置为 0
-fp-model precise	保证数值不变优化

-convert big_endian	按照大端读写文件
-assume byterecl	确定文件读写按照字节或长字单位
-mcmodel=large	编译所需内存大于 2GB 时使用该选项

2.6 作业提交

用户使用“神威•太湖之光”计算资源，必须通过系统的作业管理系统进行作业的提交，作业提交命令为 bsub。

- 向高速计算系统队列 q_sw_expr 中提交互式作业 myjob，该作业使用 1 个节点，4 个主核并行：

  bsub -I -q q_sw_expr -N 1 -np 4 ./myjob

- 向高速计算系统队列 q_sw_expr 中提交互式作业 myjob，该作业共使用 64 个主核并行，每个主核带 64 从核：

  bsub -I -q q_sw_expr -n 64 -cgsp 64 ./myjob

- 向辅助计算系统队列 q_sw_expr 中提交互式作业 myjob，该作业共使用 32 个并行规模：

  bsub -I -q q_sw_expr -n 32 ./myjob

作业提交成功后,将显示一行包含 jobid 的提示信息,其中包括作业 id 号，如“Job <102> is submitted to queue < q_sw_expr >”，此时，jobid 就是 102，它是全局唯一的。一旦作业提交成功，用户对作业的各种操作就可以通过这个 jobid 来实现的。

2.6.1 命令格式

```
bsub   [-h] [-v]
bsub   [-f sub_script]
bsub   [-I]
       [-p] [-pr]
       [-q queue_name]
       [-n num_mpes [-master | -allmaster] | -N num_nodes]
       [-np node_mpes]
       [-nodempemap node_mpe_map](0x1-0xf)
```

```
[-mpecg mpe_cgs]
[-cgsp spe_in_cg | -min_cgsp min_spe_in_cg | -rtp spe_rtp |
-asy]
[-mpmd]
[-nobind]
[-noredun]
[-node nodelist]
[-linpack_node node_file]
[-exclu | -shared]
[-js job_proj]
[-lfs_proj lfs_proj]
[-J jobname]
[-jobtype job_type]
[-i infile]
[-o outfile]
[-k ckpt-period-minutes]
[-cross]
[-spebusy]
[-jobpre        plugin_name]
[-jobpost       plugin_name]
[-cnodepre      plugin_name]
[-cnodepost     plugin_name]
[-taskpre       plugin_name]
[-taskpost      plugin_name]
[-switchnode    nodenum_in_switch]
[-midnode       nodenum_in_mid]
[-cabnode       nodenum_in_cab]
[-health        health_level]
[-a ccc [-gs groupsize] | mpi | omp [-pt threads_in_process]]
[-perf | -preff perf_file]
[-sw3runarg "arg1 arg2 ..."]
[-b]
[-parse]
[-PARSE <all | master | slave>]
[-quick]
[-m             value]
[-share_size    size]
[-priv_size     size]
```

```
[-cross_size    size]
[-ro_size       size]
[-pe_stack      size]
[-host_stack    size]
[-sw3run        sw3run_name]
[command        [argument...]]
```

2.6.2 参数说明

- -h 显示帮助信息
- -I 提交交互式作业，使作业输出在作业提交窗口，无该选项时为批式作业
- -q 向指定的队列中提交作业，必选项
- -p/-pr 在作业输出中打印作业分配的节点列表及位图
- -exclu | -shared 指定使用独占/共享模式
- -n 指定需要的所有主核数
- -N 指定需要的节点个数
- -master 主任务模式，0 占用所在计算节点的全部内存资源
- -allmaster 全片主任务模式
- -np 指定每节点内使用的主核数
- -mpecg 指定每个主核使用核组个数
- -cgsp 指定每个核组内需要的从核个数，指定时该参数必须<=64。
- -min_cgsp 指定每个核组内需要的最小从核个数
- -asy 指定使用非对称资源，表示各个核组内使用的从核个数可以不同
- -nodempemap 指定节点内的主核位图
- -noredun 不使用冗余机舱节点
- -js 指定作业对应的课题别名
- -lfs_proj 指定作业使用的局部文件代号
- -node 指定运行作业的节点列表
- -cross 要求全部分配满主核的 CPU（4 主核的 CPU）
- -mpmd 提交 mpmd 模式的作业

- -J 指定作业名
- -job_type 指定作业课题类型
- -health 指定分配资源的健康度级别
- -perf 打开性能监测功能
- -o 将作业的 stdout 和 stderr 的输出定向到指定文件，可选项
- -switchnode 指定每个 switch 中分配的节点数
- -midnode 指定每个中板分配的节点数
- -cabnode 指定每个机舱分配的节点数
- -b 指定从核栈位于局存
- -share_size 指定核组共享空间大小
- -priv_size 指定每个核上私有空间大小
- -cross_size 指定交叉段大小
- -ro_size 指定只读空间大小
- -m value 提供从核自陷模式的控制，指定-m 2 时，将浮点控制状态寄存器 fpcr 的最后两位设为 01，允许除不精确结果之外的所有浮点算术异常自陷，相当于编译器使用-OPT:IEEE_arith=2 选项；指定-m 1 时，将 fcpr 最后两位设为 00，允许所有浮点算术异常自陷，相当于编译器使用-OPT:IEEE_arith=1 选项；其他所有值将不对默认的 fpcr 进行修改。
- -pe_stack 指定从核栈空间大小，默认为 64K
- -host_stack 指定主核栈空间大小，默认为 8M
- -sw3run 指定使用特殊的加载器

2.7 众核并行示例

2.7.1 加速线程库示例

2.7.1.1 Fortran 语言示例

该示例的主要功能是求解两个数组对应元素的平方之差。

1) 串行代码：example1.f90

```
program main
    implicit none
    real,dimension(imin:imax,jmin:jmax):: a, b, c
    integer :: i,j,k
    ! init part
    do j= jmin,jmax
       do i= imin,imax
          a(i,j)=i+j-0.8888
          b(i,j)=i+j+7.7777
       end do
    end do

    ! excute
    do j= jmin,jmax
       do i= imin,imax
          c(i,j)=a(i,j)*a(i,j)- b(i,j)*b(i,j)
       end do
    end do
end
```

该段代码众核并行后将分为两个部分，即主核代码和从核代码。

2) 主核代码：master.f90

```
program main
    implicit none
    integer,parameter::imin=1,imax=1000,jmin=1,jmax=1000
    integer::i,j
    real,dimension(imin:imax,jmin:jmax)::a,b,c
    integer,external :: slave_fun          ! 指定从核函数名
    common /shared_g1/ a,b,c               ! 共享数组

    do j= jmin,jmax
       do i= imin,imax
          a(i,j)=i+j-0.8888
          b(i,j)=i+j+7.7777
       end do
    end do
    call athread_init()                    ! 初始化从核
```

```
    call athread_spawn(slave_fun, 1)     ! 从核并行任务开始
    call athread_join()                  ! 等待从核任务结束
    call athread_halt()                  ! 结束从核
end
```

3) 从核代码：slave.f90

```
subroutine fun
    implicit none
    integer,parameter::imin=1,imax=1000,jmin=1,jmax=1000
    real,dimension(imin:imax,jmin:jmax)::a,b,c
    common /shared_g1/ a,b,c                    ! 共享变量
    ! 从核局存变量申请
    int eger,dimension(imin:imax):: a_slave,b_slave,c_slave
    integer i,j
    integer slavecore_id,reply
    !$omp threadprivate (/local_g1/)          ! 从核编译引导语句

    call athread_get_id (slavecore_id)        ! 获得从核逻辑 id
    do j= jmin,jmax
       ! 从核计算任务绑定
       if(mod(j, 64)+1.eq.slavecore_id) then
          reply=0
          ! 读入数据
          call athread_get(0,a(imin,j),a_slave(imin), &
              (imax-imin+1)*4,reply,0,0,0)
          call athread_get(0,b(imin,j),b_slave(imin), &
              (imax-imin+1)*4,reply,0,0,0)
          do while (reply.ne.2)       ! 等待数据读成功
          end do
          do i= imin,imax
            !  计算
            c_slave(i) = a_slave(i) * a_slave(i) &
                       + b_slave(i) * b_slave(i)
          end do
          reply=0
          ! 写回数据
          call athread_put(0,c_slave(imin),c(imin,j), &
               (imax-imin+1)*4,reply,0,0)
```

```
            do while (reply.ne.1) ! 等待数据写成功
            end do
         end if
      end do
      return
end
```

4) athread_get_id 函数

athread_get_id 函数的 Fortran 接口目前还未封装到缺省函数库中，还需要手工增加一个接口函数，如文件 athread_get_id.c 所示。

```
#include <slave.h>
#include <math.h>
#include <stdio.h>

void  athread_get_id_(int*t)
{
        *t = athread_get_id(-1);
}
```

5) 主从核分开编译：

- sw5cc -slave -c athread_get_id.c
- sw5f90 -slave -c slave.f90
- sw5f90 -host -c master.f90
- sw5f90 -O3 -hybrid athread_get_id.o slave.o master.o -o test

从核代码用 sw5f90 -slave 编译，主核代码用使用 sw5f90 -host 编译，使用 sw5f90 -hybrid 进行链接生成最终的可执行文件。

6) 提交作业命令：

```
bsub -I -b -q q_sw_expr -n 1 -cgsp 64 -share_size 4096
     -host_stack 128 ./test
```

其中：

- -I：选项表示提交交互式作业，使作业输出在作业提交窗口；
- -b：表示从核函数栈变量放在从核局部存储上，该选项是获取加速性能必须的提交选项；

- -q：向指定的队列中提交作业；
- -n：指定作业需要的所有主核数；
- -cgsp：指定每个核组内需要的从核个数，指定时该参数必须<=64；
- -share_size：指定核组共享空间大小，一般最大可以用到 7600MB；
- -host_stack：指定主核栈空间大小，默认为 8M，一般设置为 128MB 以上。

从上面的典型众核代码可以看出，在 Fortran 语言中，众核并行时需要将核心计算相关的数据通过 common 进行共享。主核程序中首先初始化加速线程库，然后调用需要执行的从核程序的接口，等待从核计算结束，最后结束加速线程库。从核程序主要实现共享数据的读入、计算和回写操作。

对主从核而言，共享存储空间都是可见的，这也意味着主从核都可以直接访问和修改共享存储空间的数据。采用 Fortran 编写程序时，数据共享通常采用 common 方式实现，因此一定要保证相应 commnon 名字和数组大小的一致性。通过 common 的方式实现数据的共享数组时，对数组的大小是有要求的——即数组的大小必须是固定的。对于动态大小的数组，可直接 common 共享数组指针，或者 common 指向该数组的 Cray 指针。表格 2-5 给出三种 common 使用的例子。

表格 2-5 三种 common 方式

… n=10; m=10 real v(n,m) common/fun_v / v …	… real,pointer v(:,:) common/fun_v / v …	… real v_host(n,m) pointer(v_p,v_host) common/fun_v / v_p v_p=loc(v(1,1)) …

三种 common 方式的 DMA 通信性能存在差异，实验结果表明，数组指针的 DMA 性能最差，Cray 指针次之，静态数组最优。

2.7.1.2 C 语言示例

该示例的主要功能是实现数组的除法。

1) 主核代码：master.c

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <athread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define J 64
#define I 1000
double a[J][I],b[J][I],c[J][I],cc[J][I];
double check[J];
unsigned long counter[J];

extern SLAVE_FUN(func)();

// 节拍计数器
static inline unsigned long rpcc()
{
    unsigned long time;
    asm("rtc %0": "=r" (time) : );
    return time;
}

int main(void)
{
    int i,j;
    double checksum;
    double checksum2;
    unsigned long st,ed;

    printf("!!!!!!!!!! BEGIN INIT !!!!!!!!!!\n");
    fflush(NULL);
    for(j=0;j<J;j++) {
        for(i=0;i<I;i++){
            a[j][i]=(i+j+0.5);
            b[j][i]=(i+j+1.0);
        }
    }
    st=rpcc();
```

```
    for(j=0;j<J;j++) {
        for(i=0;i<I;i++){
            cc[j][i]=(a[j][i])/(b[j][i]);
        }
    }
    ed=rpcc();
    printf("the host counter=%ld\n",ed-st);

    checksum=0.0;
    checksum2=0.0;
    athread_init();
    st=rpcc();
    athread_spawn(func,0);
    athread_join();
    ed=rpcc();
    printf("the manycore counter=%ld\n",ed-st);
    printf("!!!!!!!!!! END JOIN !!!!!!!!\n");
    fflush(NULL);

    for(j=0;j<J;j++) {
        for(i=0;i<I;i++){
            checksum=checksum+c[j][i];
            checksum2=checksum2+cc[j][i];
        }
    }
    printf("the master value is %f!\n",checksum2);
    printf("the manycore value is %f!\n",checksum);
    athread_halt();
    printf("!!!!!!!!!! END HALT !!!!!!!!!\n");
    fflush(NULL);
}
```

1) 从核代码：slave.c

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "slave.h"
```

```
#define J 64
#define I 1000

__thread_local volatile unsigned long get_reply,put_reply;
__thread_local volatile unsigned long start,end;
__thread_local int my_id;
__thread_local double a_slave[I],b_slave[I],c_slave[I];

extern double a[J][I],b[J][I],c[J][I];
extern unsigned long counter[64];

void func()
{
    int i,j;

    my_id = athread_get_id(-1);
    get_reply = 0;
    // 获取数据
    athread_get(PE_MODE,&a[my_id][0],&a_slave[0],I*8,&get_reply,
                0,0,0);
    athread_get(PE_MODE,&b[my_id][0],&b_slave[0],I*8,&get_reply,
                0,0,0);
    while(get_reply!=2); // 等待数据加载完成
    for(i=0;i<I;i++){
        c_slave[i]=a_slave[i]/b_slave[i];
    }
    put_reply=0;
    // 回送数据
    athread_put(PE_MODE,&c_slave[0],&c[my_id][0],I*8,
                &put_reply,0,0);
    while(put_reply!=1); // 等待数据回送完成
}
```

2) 编译链接过程：

- sw5cc -host -c master.c
- sw5cc -slave -c slave.c
- sw5cc -hybrid master.o slave.o -o test

从核代码用 sw5cc -slave 编译，主核代码用使用 sw5cc -host 编译，使用 sw5cc -hybrid 进行链接生成最终的可执行文件。

3) 提交作业命令：

作业提交方式请参考 Fortran 语言示例。

2.7.2 OpenACC*示例

“神威•太湖之光”计算机系统第二种众核并行的方法就是采用 OpenACC*并行，OpenACC*并行也是“神威•太湖之光”计算机系统最主要的并行方法，如果课题不追求极致的计算效率，一般都采用这种并行实现。

表格 2-6 介绍的Fortran 语言程序代码就是采用OpenACC*并行方法实现的。

表格 2-6 采用 OpenACC*并行的程序代码

```
program main
    implicit none
    real ,dimension(1:512,1:64):: a, b, c
    integer :: i,j

    !init part
    do j= 1,64
       do i= 1,512
          a(i,j)=i+j-0.8888
          b(i,j)=i+j+7.7777
       end do
    end do

    !execute
    !$ACC PARALLEL LOOP COPYIN(a, b) COPYOUT(c) LOCAL(i)
    do j= 1,64
       do i= 1,512
          c(i,j)=a(i,j)*a(i,j)- b(i,j)*b(i,j)
       end do
    end do
    !$ACC END PARALLEL LOOP
end program
```

将需要使用众核加速执行的程序核心段(execute)部分的循环前后使用OpenACC*加速编译指示进行标注，如代码中加粗部分所示，其中 PARALLEL 表明该部分代码是需要加载到从核进行加速执行的并行代码；LOOP 表示下面紧跟着的 j 循环需要在加速线程间进行并行划分；COPYIN(a,b)表明 a、b 数组需要拷贝到从核的 LDM 空间，因为 a、b 数组是只读，不需要更新回主存；COPYOUT(c)表明 c 数组在计算结束后需要拷贝回主存；LOCAL(i)表明变量 i 存放在 LDM 中，是从核加速线程私有的。

使用 swafort 进行编译。该程序的执行效果是，如果以 64 个加速线程运行，则每个线程会执行 64 个 j 循环中的一个，相应的会将 a、b、c 数组对应的数据在 LDM 中申请空间，并执行对应的数据拷贝和计算操作。

初始化部分的代码也可以参照该方法进行移植。OpenACC*的具有用法详见后续章节介绍。

2.7.3 全片共享模式示例

“神威•太湖之光”计算机系统的一个计算节点由一颗“申威 26010”众核处理器组成，每个计算节点 32GB 内存，也就是一颗“申威 26010”众核处理器可以访问 32GB 内存。全片共享模式是指一颗众核处理器 4 个核组共享 32GB 节点内存。全片共享方式支持主从混合编译，在混合链接时需要使用-allshare 选项。根据课题的需求，全片共享模式分为 master 模式和非 master 模式，应用程序运行时，作业管理会根据命令行参数给应用程序分配所需的内存空间。

2.7.3.1 全片共享 master 模式

应用程序在实际运行时，满足如下内存特点的课题可以采用全片共享 master 模式：主进程（一般规定为 0 进程）的内存需求大于 7500MB，其他进程的内存需求小于 7500MB。这种情况下，建议选择使用 master 模式提交作业，下面通过实例进行说明。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include "mpi.h"
#define DIM_I 1024
#define DIM_J 1024
#define DIM_K 256

int main(int argc, char *argv[])
{
  unsigned long *a,*b,*c,*d;
  unsigned long sum_total,size1;
  int myid, numprocs,i;
  extern void slave_func();

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  sum_total=0;
  if(myid==0){
    size1=DIM_I*DIM_J*DIM_K;
    a=(unsigned long*)malloc(size1*sizeof(unsigned long));
    b=(unsigned long*)malloc(size1*sizeof(unsigned long));
    c=(unsigned long*)malloc(size1*sizeof(unsigned long));
    d=(unsigned long*)malloc(size1*sizeof(unsigned long));
    for(i=0;i<size1;i++){
        a[i] = 1; b[i] = 2; c[i] = 3; d[i] = 4;
        sum_total = sum_total + a[i] + b[i]/2 + c[i]/3 + d[i]/4;
    }
    sum_total = sum_total/4;
    if(sum_total==size1){
        size1=4*8*size1/(1024*1024);
        printf("Process %d Memery size : %dMB and sum = %d \n",
                myid,size1,sum_total);
    }
  } else {
    size1=DIM_I*DIM_J;
    a=(unsigned long*)malloc(size1*sizeof(unsigned long));
    b=(unsigned long*)malloc(size1*sizeof(unsigned long));
    c=(unsigned long*)malloc(size1*sizeof(unsigned long));
    d=(unsigned long*)malloc(size1*sizeof(unsigned long));
```

```
    for(i=0;i<size1;i++){
       a[i] = 1; b[i] = 2; c[i] = 3; d[i] = 4;
       sum_total = sum_total + a[i] + b[i]/2 + c[i]/3 + d[i]/4;
    }
    sum_total = sum_total/4;
    if(sum_total==size1){
        size1=4*8*size1/(1024*1024);
        printf("Process %d Memery size : %dMB and sum = %d \n",
                myid,size1,sum_total);
    }
  }
  MPI_Finalize();
}
```

在程序中，数组数据类型为 long，在实际运行时，0 进程需要的总内存需求是 8G，其他进程内存需求较小。用户在提交程序时，为了满足内存需求，0 进程需要使用全片共享模式，而其他进程的私有空间足够存储数据，建议使用 master 模式提交作业：

```
bsub -b -I -pr -q q_sw_expr  -n  5  -cgsp  64  -allmaster
     -host_stack0 256 -cross_size0 10000 -host_stack 256
     -cross_size 2500 ./a.out
```

上述提交命令行，-allmaster 说明程序需要使用 master 全片共享模式，-cross_size0 指定 0 进程所需内存大小， -cross_size 指定其他进程所需内存大小。

2.7.3.2 全片共享非 master 模式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"

#define DIM_I 1024
#define DIM_J 1024
#define DIM_K 256

int main(int argc, char *argv[])
```

```
{
  unsigned long *a,*b,*c,*d;
  unsigned long sum_total,size1;
  int myid, numprocs,i;
  extern void slave_func();

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  sum_total=0;
  size1=DIM_I*DIM_J*DIM_K;
  a=(unsigned long*)malloc(size1*sizeof(unsigned long));
  b=(unsigned long*)malloc(size1*sizeof(unsigned long));
  c=(unsigned long*)malloc(size1*sizeof(unsigned long));
  d=(unsigned long*)malloc(size1*sizeof(unsigned long));
  for(i=0;i<size1;i++){
      a[i] = 1; b[i] = 2; c[i] = 3; d[i] = 4;
      sum_total = sum_total + a[i] + b[i]/2 + c[i]/3 + d[i]/4;
  }
  sum_total = sum_total/4;
  if(sum_total==size1){
      size1=4*8*size1/(1024*1024);
      printf("Process %d Memery size : %dMB and sum = %d \n",
              myid,size1,sum_total);
  }
  MPI_Finalize();
}
```

程序中数据数组的数据类型为 long，每个进程的内存需求是 8G。用户在提交程序时，为了满足内存需求，每个进程都需要使用全片共享模式，建议使用非 master 模式提交作业：

```
bsub -b -I -pr -q q_sw_expr -n 5 -np 1 -cgsp 64 -sw3runarg
     "-a 1" -host_stack 256 -cross_size 10000 ./a.out
```

上述提交命令行，通过-sw3runarg "-a 1" -np 1 实现全片共享，每个进程所需的内存空间大小由-cross_size 参数来指定。

第3章　加速线程库程序设计

“神威•太湖之光”计算机系统加速线程库(Athread 库)是针对两级并行编程模型(主从加速编程模型)所设计的程序加速库,其目的是为了用户能够方便、快捷地对核组内的线程进行灵活的控制和调度,从而更好地发挥核组内多从核并发执行的加速性能。本书仅介绍一些用户最常用功能，更详细的加速线程库接口请参见《“神威•太湖之光”编译系统用户手册》。

3.1 主核加速线程库

主核加速线程库主要是提供主核程序使用的 Athread 接口，主要用于控制线程的初始化、启动、结束等一系列操作。用户可以通过调用以下接口实现相关功能。

3.1.1 初始化线程库

函数名：

athread_init

函数说明：

完成加速线程库的初始化，在使用任何线程库接口前必须使用该接口。

参数说明：

无

返回值：

- 成功：返回 0，表示加速线程初始化成功；
- 失败:其它任何返回值都表示出现了错,如果监测到以下任一情况，athread_init()将失败，并返回相应的值：
  - -EINVAL：初始化失败。

附加说明：

无

3.1.2 创建线程组

函数名：

athread_spawn

函数说明：

在当前进程中添加新的受控线程组。最先执行的任务由函数 fpc 指定；函数 fpc 的参数由 arg 提供。

参数说明：

- start_routine fpc：函数指针；
- void * arg：函数 f 的参数起始地址。

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_spawn()将失败，并返回相应的值：
  - -EINVAL：线程组创建失败。

附加说明：

调用时 athread_spawn 接口时，启动核组中的所有可用从核资源。如果需要知道线程组在核组中所占用的从核资源信息，可以调用相关接口得到该类信息。

3.1.3 等待线程组终止

函数名：

athread_join

函数说明：

显式阻塞调用该线程组，直到指定的线程组终止。指定的线程组必须位于当前的进程中。

参数说明：

无

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错：
  - -EINVAL：核组内无线程组运行。

附加说明：

无

3.1.4 关闭线程组流水线

函数名：

athread_halt

函数说明：

在确定线程组所有从核无相关作业后，停滞从核组流水线，关闭从核组，该从核组在本进程无法再次使用。

参数说明：

无

返回值：

- 成功：返回 0，线程所占用的从核资源被成功关闭；
- 失败：其它任何返回值都表示出现错误，如果监测到以下任一情况，athread_halt()将失败，并返回相应的值：
  - -EINVAL：无法正常关闭从核资源。

附加说明：

必须保证从核组无任何用户作业才能使用该函数。

3.1.5 进入满核组快速工作模式

函数名：

athread_enter64

函数说明：

athread_spawn 和 athread_join 机制提供了对任意数量和形状的线程进行控制的机制。如果用户希望在包含 64 个从核的满配核组上快速启动线程并追求更高的效率，athread_enter64 提供了进入这种快速模式的机制。

参数说明：

无

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_enter64()将失败，并返回相应的值：
  - -EINVAL：无法进入满核组工作模式。

附加说明：

athread_enter64 是进入满核组快速线程模式的入口，如果用户希望控制的线程数量或形状并非满配的模式，则使用该快速模式将导致不可预知的错误。

3.1.6 快速模式下创建线程组

函数名：

athread_spawn64

函数说明：

在满核组快速工作模式下添加新的受控线程组。最先执行的任务由函数 fpc 指定；

参数说明：

- start_routine fpc：函数指针；
- void * arg：目前无意义。

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_spawn64()将失败，并返回相应的值：

- -EINVAL：线程组创建失败。

附加说明：

调用时 athread_spawn64 接口时，将快速启动满核组的 64 个从核资源进入指定任务。

3.1.7 快速模式下等待线程组终止

函数名：

athread_join64

函数说明：

快速模式下显式阻塞调用该线程组，直到指定的线程组终止。指定的线程组必须位于当前的进程中。

参数说明：

无

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错
  - -EINVAL：核组内无线程组运行。

附加说明：

无

3.1.8 退出满核组快速工作模式

函数名：

athread_leave64

函数说明：

用户调用该接口将离开快速工作模式，后继对线程的控制回到可以对任意形状和数量进行控制的普通 spawn 和 join 模式

参数说明：

无

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错。

附加说明：

无

3.1.9 满核组快速工作模式示例

3.1.9.1 从核程序

```
int fun64( ){
    ……
}
int fun(arg){
   ……
}
```

3.1.9.2 主核程序

```
int main()
{
    int tid1,tid2;

    athread_init();   // 线程库初始化
    athread_enter64();// 进入快速工作模式
    for (i=0; i<n; i++) {
        athread_spawn64(fun64, 0); // 快速模式下创建和等待线程组
        athread_join64();
    }
    athread_leave64( ); // 退出快速模式
    // 进入普通 spawn 和 join 模式
```

```
    // 关闭线程所占从核流水线，该从核无法在本进程中继续使用
    athread_spawn(fun, arg);
    athread_join();
    athread_halt();
}
```

3.1.10 获取核组最大线程总数

函数名：

athread_get_max_threads

函数说明：

使用该接口得到当前进程的可用的所有核组线程资源总数。

参数说明：

无

返回值：

- 成功：返回核组内可用最大线程个数(1<=ret<=64)；
- 失败:其它任何返回值都表示出现了错,如果监测到以下任一情况，athread_get_max_threads 将失败，并返回相应的值：
  - -EINVAL：返回失败，出现异常。

附加说明：

无

3.1.11 设置并行区线程总数

函数名：

athread_set_num_threads

函数说明：

使用该接口设置当前进程下一个并行区启动的所有核组线程总数。

参数说明：

int num：线程总数

返回值：

- 成功：0；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_set_num_threads 将失败，并返回相应的值：
  - -EAGAIN：设置线程数目超过了可用线程资源，将返回该值；
  - -EINVAL：并行区线程数设置失败，比如上个并行区任务未完成。

附加说明：

无

3.1.12 获取并行区线程总数

函数名：

athread_get_num_threads

函数说明：

使用该接口得到当前进程启动的所有核组线程总数。

参数说明：

无

返回值：

- 成功：返回当前核组内线程个数(1<=ret<=64)；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get_num_threads 将失败，并返回相应的值：
  - -EINVAL：返回失败，出现异常。

附加说明：

无

3.1.13 强制线程结束

函数名：

athread_cancel

函数说明：

该函数用来终止指定线程，线程 ID 以及资源可以立即收回。

参数说明：

int id：指定退出线程号

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_cancel 将失败，并返回相应的值：
  - -ESRCH：没有找到指定线程 ID 对应的线程；
  - -EINVAL：ID 号非法；
  - -EFAULT：ID 线程对应从核故障。

附加说明：

线程结束可以通过以下方法来终止执行：

- 从线程的第一个过程返回，即线程的启动例程。参考 athread_create。
- 调用 athread_cancel，提前退出。

3.1.14 中断管理

函数名：

athread_signal

函数说明：

接收中断，并指定中断处理函数。

参数说明：

- int signo：中断信号
- start_routine fpc：中断处理函数

返回值：

无

附加说明：

参考 athread_sigqueue()接口。

3.1.15 异常管理

函数名：

athread_expt_signal

函数说明：

挂载特定异常的处理函数。

参数说明：

- int signo：异常信号
- start_routine fpc：异常处理函数

返回值：

无

附加说明：

1） 异常类型：根据从核异常类型，用户可以挂载以下 18 种异常处理信号：

```
enum Exception_Kind_1 {
     UNALIGN = 0,     // 不对界异常
     ILLEGAL,         // 非法指令
     FPE,             // 浮点异常
     IOV,             // 整数溢出
     PCOV,            // PC 溢出
     CLASSE,          // 分类异常
     SELLDWE,         // 向量查表异常
     DFLOWE,          // 数据流异常
     IFLOWE,          // 指令流异常
     ATOMOV,          // 原子操作溢出
     SBMDE,           // SBMD 异常
     SYNE,            // 同步异常
     RCE,             // 寄存器通信异常
     CHANNAL,         // 通道异常
     SRSETE,          // 从核软复位异常
     IOE,             // IO 访问异常
```

```
    OUTRANK,          // 存控 rank 越界异常
    MTAG,             // 多个源同事写 CTAG 错
};
```

2) 挂载方式

athread_expt_signal

3) 异常处理函数

用户自定义异常处理函数原型为：

handler_t * handler(int signum, siginfo_t *sinfo, struct sigcontext *sigcontext)

- 第一个参数：signum 为异常信号
- 第二个参数：sinfo 的信息如下：
  - sinfo->si_signo = signum; // 异常信号
  - sinfo->si_errno = ? ; // 异常向量 2
  - sinfo->si_pid = peid; // 出现异常的从核号
  - sinfo->si_uid = cgid; // 出现异常的核组号
  - sinfo->si_ptr=eaddr; // 造成从核异常的可能访问的地址
- 第三个参数：sigcontext 的前三个参数用来保留特定信息，如下：
  - sigcontext-> sc_onstack // 从核异常 PC 是否精确标志
  - sigcontext->sc_pc // 从核异常 PC
  - sigcontext-> sc_mask // 从核数据流异常访存信息

sc_pc 如果为 0 表示没有异常 PC，不为 0 的情况下根据 sc_onstack 判断是否为精确 PC。

sc_onstack 为 0 表示没有精确异常 PC，此时 sc_pc 记录的是发生异常时的非精确 PC；sc_onstack 为 1 表示有精确异常 PC，此时 sc_pc 记录的是发生异常时的精确 PC。

sc_mask 项只对数据流异常 DFLOWE 有效，为 4b`0001 表示 LD 主存，为 4b`0010 表示 ST 主存，为 4b`0100 表示 DMA_GET，为 4b`1000 表示 DMA_PUT。

sigcontext 其它各个域的内容由 OS 确定并原样传给用户处理程序。

对上述 Exception_Kind_1 中特定的异常，异常向量 2 表示更细分的异常类型，用户可以根据异常向量 2 的值进行进一步处理。它们的定义如下：

4) 浮点异常

```
enum Ekind_FPE_2 {
      OVI = 0,      // 整数溢出
      INE,          // 非精确结果
      OVF,          // 下溢
      UNF,          // 上溢
      DZE,          // 除数为 0
      INV,          // 无效操作
      DNO,          // 非规格化数
};
```

5) 数据流异常：

```
enum Ekind_DFLOW_2 {
      RCSRINV = 0, // rcsr 索引号无效
      LDME,         // LDM 相关异常
      OTHERS,       // 其他需要判定的数据流异常
      MEMANS = 5,   // 从核收到访存异常错误响应
      CODC,         // 从核流水线 1 CODC 指令转换 LDM 局存地址时
溢出
      CLASS_ST,     // CLASS_ST 指令的访问地址越权或越界
};
```

6) SBMD 异常：

```
enum Ekind_SBMD_2 {
      MATCHE = 0,   // SBMD 匹配异常和 SBMD 查询异常
      CMBIO,        // SBMD 集中管理部件 IO 访问异常
      CMBCK,        // SBMD 集中管理部件检查的 SBMD 异常
      OTHER,        // 其他 SBMD 异常
};
```

7) 同步异常：

```
enum Ekind_SYN_2 {
      NOSELF = 0,   // 从核发出的同步向量中不包含本从核
      SWITHNO,      // 可降级同步不使能的情况下，与不在位的从
核进行硬件同步操作
```

```
};
```

8） 通道异常：

```
enum Ekind_CHANNAL_2 {
      WRCHE = 0,    // WRCH 使用了非法的通道号
      DMAE,         // DMA 产生 DMA 描述符静态检查异常
      DMAW,         // DMA 产生 DMA 描述符静态检查警告
      DMAMEM,       // DMA 访问了不在位的核组主存
      DMALDM,       // DMA 除广播模式和广播行模式外，访问不在位
的从核 LDM
      ANSWD,        // 回答字地址越界或不对界
      CHNO,         // 非法通道号
      SYNVEC,       // 检查到本从核不在同步向量中
};
```

发生异常时，线程库会报出第一个捕获的异常，这个异常可以是核组异常也可以是从核异常。如果用户没有挂载相应该异常的处理函数，那么线程库会打印异常相关的所有信息，halt 发生异常的从核，并退出程序；如果用户挂载了该异常的处理函数，则进入用户定义的异常处理函数。

3.1.16 主核取从核局存地址

宏名：

IO_addr(elment, penum, cgnum)

宏说明：

主核可以通过 IO 访问局存，该宏返回从核 LDM 变量 element 的 IO 地址。

参数说明：

- element：从核程序内的局存私有变量__thread_local elment，主核使用 IO_addr 接口前必须加 extern __thread 声明成外部变量
- penum：核号
- cgnum：核组号

返回值：

返回 IO 地址

附加说明：

IO_addr 宏不会判断 penum 和 cgnum 的合法性，由用户保证。

举例说明：

1） 从核程序 slave.c

```
……
__thread_local char ch='b';
__thread_local long para[10]
__attribute__((__aligned__(128)))= {0x123, 0x234, 0x345,
0x456};
……
```

2） 主核程序 main.c

```
……
extern long __thread para[];
extern char __thread ch;
unsigned long  LDM_addr;
LDM_addr = IO_addr(para[0], 23, 0);
……
```

这样，LDM_addr 等于第 0 号核组第 23 号从核上 para[0]的 IO 地址。

3.1.17 主核访问从核局存

宏名：

h2ldm(elment, penum, cgnum)

宏说明：

主核可以通过 IO 访问局存，该宏直接在主核对从核 LDM 变量 element 进行 IO 存取操作

参数说明：

- element：从核程序内的局存私有变量__thread_local elment，主核使用 IO_addr 接口前必须加 extern __thread 声明成外部变量
- penum：核号
- cgnum：核组号

返回值：

typeof(element)

附加说明：

h2ldm 宏不会判断 penum 和 cgnum 的合法性，由用户保证。

举例说明：

1） 从核程序 slave.c

```
……
__thread_local char ch='b';
__thread_local long para[10]
__attribute__((__aligned__(128)))= {0x123, 0x234, 0x345,
0x456};
……
```

2） 主核程序 main.c

```
……
extern long __thread para[];
extern char __thread ch;
unsigned long  LDM_addr;
h2ldm(para[0], 23, 0) = 0xaaa;
h2ldm(ch, 5, 1) = 'z';
……
```

主核对 0 号核组第 23 号从核上局存变量 para[0]赋值；主核对 1 号核组第 5 号从核上的局存变量 ch 赋值。

备注：

线程 create 和 spawn 接口只带一个参数接口。而主核访问从核局存接口可以用作主核对核组加速区进行批量的参数赋值。比如：从核程序专门申请一块局存空间用作存放参数，主核在启动不同的加速区之前可以对这块参数区进行赋值；如果每个从核上分配相同的参数，通常最优的做法是只对某一个从核进行参数赋值，然后从核通过寄存器通信将参数广播到核组上所有的从核。

3.1.18 设置线程存活掩码

函数名：

__set_still_living_mask

函数说明：

在核组程序运行的过程中，线程组创建时的从核工作位图可能因为某些从核异常或其他原因而无法正常完成分配的任务，并导致主核无法等待线程组正常终止。__set_still_living_mask 提供一种机制，用来设置线程依然存活的掩码，并进而控制线程组终止时等待到达一致点的从核位图。

该函数主要用于并行 C 库实现从核降级功能。

参数说明：

unsigned long mask。

mask 的 64 位值均有效，当 mask 的某一位置为 1 时，表示当前核组对应此位的从核依然存活；当 mask 的某一位置为 0 时，表示当前核组对应此位的从核已经发生了某种异常，在等待线程组结束时，不需要等待该从核结束。

返回值：

无

附加说明：

无

3.1.19 线程空闲状态查询

函数名：

unsigned long athread_idle ()

函数说明：

使用该函数可获得当前从核线程组是否处于空闲状态。

参数说明：

无

返回值：

如果从核组处于空闲状态则返回当前可用的从核位图；否则返回 0。

附加说明：

这里的空闲状态是指等待任务的状态，也即：线程组未执行用户 spawn 的从核加速线程或线程组已经完成用户 spawn 的从核加速线程。

举例说明：

```
……
unsigned long idleslave;
if (idleslave=athread_idle()){
    用从核优化(spawn)；
    ……
}
……
```

3.2 从核加速线程库

从核加速线程库主要是提供从核程序的 athread 接口，主要用于从核线程的线程识别、中断发送等一系列操作。用户可以通过调用以下接口实现相关功能。

3.2.1 获得线程逻辑标识号

函数名：

athread_get_id

函数说明：

使用该函数获得本地单线程的逻辑标识号(即 ID 号)。该接口主从核通用。

参数说明：

int core：指定查询物理核号，-1 默认本地从核(从核接口有效)。

返回值：

- 成功：线程逻辑 ID 号(0<=ret<=63)；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_get_id 将失败，并返回相应的值：
  - -EINVAL：返回失败，出现异常。

附加说明：

只要当线程是被用 athread_create 创建单线程接口启动的线程才会有绑定的标识符。

举例说明：

```
……
int myid;
myid = athread_get_id(-1);
……
```

3.2.2 获得线程物理从核号

函数名：

athread_get_core

函数说明：

使用该接口获得对应线程的物理从核号。该接口主从核通用。

参数说明：

int id：指定查询线程号，-1 默认本地线程(从核接口有效)。

返回值：

- 成功：线程所占从核物理号(0<=ret<=63)；
- 失败:其它任何返回值都表示出现了错,如果监测到以下任一情况，athread_get_core 将失败，并返回相应的值：
  - -EINVAL：返回失败，出现异常。

附加说明：

返回值低 6 位有效，低 3 位为列号，高 3 位为行号，分别表示核组 8*8 阵列。

举例说明：

```
……
int mycore;
mycore = athread_get_core(-1);
……
```

3.2.3 数据接收 GET

函数名：

athread_get

函数说明：

从核局存 LDM 接收主存 MEM 数据，进行主存 MEM 到从核 LDM 的数据 get 操作，将 MEM 的数据 get 到 LDM 指定位置。传输模式由 mode 指定，如果是在广播模式和广播行模式下，要进行屏蔽寄存器配置；在其它模式下，mask 值无效。

参数说明：

extern int athread_get(dma_mode mode, void *src, void *dest, int len, void *reply, char mask,int stride,int bsize);

- dma_mode mode：DMA 传输命令模式；
- void *src：DMA 传输主存源地址；
- void *dest：DMA 传输本地局存目标地址；
- int len：DMA 传输数据量，以字节为单位；
- void *reply：DMA 传输回答字地址，必须为局存地址，地址 4B 对界；
- char mask：DMA 传输广播有效向量，有效粒度为核组中一行，某位为 1 表示对应的行传输有效，作用于广播模式和广播行模式；
- int stride：主存跨步，以字节为单位；
- int bsize：行集合模式下，必须配置，用于指示在每个从核上的数据粒度大小；其它模式下，在 DMA 跨步传输时有效，表示 DMA 传输的跨步向量块大小，以字节为单位。

返回值：

- 成功：返回 0；
- 失败:其它任何返回值都表示出现了错,如果监测到以下任一情况，athread_get 将失败，并返回相应的值：
  - -EINVAL：返回失败，出现异常。

附加说明：

```
typedef enum {
        PE_MODE,
        BCAST_MODE,
        ROW_MODE,
        BROW_MODE,
```

```
        RANK_MODE
} dma_mode;
```

举例说明：

```
……
__thread_local long  ldm_a[64];
__thread long  mem_b[64];
__thread_local  int reply=0;
athread_get(PE_MODE,mem_a,ldm_a,64*8,&reply,0,0,0);
//以单从核模式，从核私有连续段以 mem_a 为基地址取 64*8B 数据放入从核局存以 ldm_a 为基地址的连续 64*8B 的空间中，其中广播向量、主存跨步、向量块大小均为 0。
……
```

3.2.4 数据发送 PUT

函数名：

athread_put

函数说明：

从核局存 LDM 往主存 MEM 发送数据，进行从核 LDM 到主存 MEM 的数据 put 操作，将 LDM 的数据 put 到 MEM 指定的位置。传输模式由 mode 指定，不支持广播模式和广播行模式。

参数说明：

int athread_put(dma_mode mode, void *src, void *dest, int len, void *reply,int stride,int bsize)

- dma_mode mode：DMA 传输命令模式；
- void *src：DMA 传输局存源地址；
- void *dest：DMA 传输主存目的地址；
- int len：DMA 传输数据量，以字节为单位；
- void *reply：DMA 传输回答字地址，必须为局存地址，地址 4B 对界；
- int stride：主存跨步，以字节为单位；

- int bsize：行集合模式下，必须配置，用于指示在每个从核上的数据粒度大小；其它模式下，在 DMA 跨步传输时有效，表示 DMA 传输的跨步向量块大小，以字节为单位。

返回值：

- 成功：返回 0;
- 失败:其它任何返回值都表示出现了错,如果监测到以下任一情况，athread_put 将失败，并返回相应的值:
  - -EINVAL：返回失败，出现异常。

附加说明：

DMA 传输命令模式参考 athread_get 接口附加说明。

3.2.5 物理地址数据接收

函数名：

athread_get_p

函数说明：

物理地址数据接收，除源主存地址为物理地址外，其它使用方式与 athread_get 一致。

3.2.6 物理地址数据发送

函数名：

athread_put_p

函数说明：

物理地址数据发送，除目的主存地址为物理地址外，其它使用方式与 athread_put 一致。

3.2.7 DMA 栏栅

函数名：

athread_dma_barrier

函数说明：

发起 dma 栏栅，对应 dma 命令操作码为 barrier_group

参数说明：

无参数

3.2.8 核组内同步

函数名：

athread_syn

函数说明：

核组内从核同步控制。

参数说明：

- scope scp：同步范围控制
- int mask：同步屏蔽码

返回值：

- 成功：返回 0；
- 失败：其它任何返回值都表示出现了错，如果监测到以下任一情况，athread_syn 将失败，并返回相应的值：
  - -EINVAL：返回失败，出现异常。

附加说明：

```
typedef enum {
        ROW_SCOPE,    // 行同步，低 8 位有效
        COL_SCOPE,    // 列同步，低 8 位有效
        ARRAY_SCOPE, // 全核组同步，低 16 位有效，其中 16 位中
                      // 低 8 位为列同步屏蔽码、高 8 位为行同步
                      // 屏蔽码
}scope;
```

3.3 使用 DMA 接口注意事项

3.3.1 athread_get 函数

athread_get 是 DMA 读入接口程序，有 8 个实参。样例如下：

athread_get(0, master(1), slave(1), lenth, reply, 0, 0, 0)

其中第 1 个实参和读取模式相关，一般都设置为 0。第 2 个实参是待读入数据的起始地址。第 3 个实参是存放读入数据的从核数组起始地址。第 4 个实参是读入数据总长度。第 5 个实参是回答字。第 6 个是 mark，一般设置为 0。第 7 个和第 8 个是主存跨步加载时的跨步长度和每次读入数据的长度，非跨步时都设置为 0。

3.3.2 athread_put 函数

athread_put 是 DMA 写回接口程序，有 7 个实参。样例如下：

athread_put(0,slave(1),master(1),lenth,reply,0,0)

和读入接口程序相比，写回接口程序少了第 6 个实参，并且第 2 个实参是待写回的从核变量地址，而第 3 个则是写回目标地址。

在实际“申威 26010”异构众核编程时使用 DMA 接口程序时需要注意以下几个方面：

1) 虽然接口程序访问的是地址，但在写接口实参时却直接写数组元素 master(1)，而不是写成 loc(master(1))。这是由 Fortran 语言本身的特点所决定的，第 5 个实参回答字也是类似；
2) 对于操作数据总长度、跨步长度和单次操作长度，均是字节为单位，因此实际编程过程中这些长度都是操作数据元素个数乘以该元素类型的字节数；
3) 回答字 reply 是一个整数型变量，也可以是整数型数组的一个元素，不允许仅仅写数组名而不具体到元素。

3.3.3 主存跨步读写

跨步读写方面，目前仅支持主存跨步，即可以对离散在共享主存区上的若干数据块进行跨步读写操作。实际编程时要注意跨步长度的确定，和 MPI 不同，这里的跨步长度是从前一个数据块的结束到下一个数据块的开始。如从共享浮点变量 v(im,jm,km)读入 km 个长度为 im 的数据块，操作数据的总长度为 im*km*4，跨步长度为 im*jm*4-im*4，每次操作的长度为 im*4。

实际编程过程中，接口程序参数错误会导致从核计算错误或读写操作失败。如错误的主从核存储地址错误，会导致计算错误、段错或 LDM 访问越界。操作数据总长度书写错误则会导致错误的写覆盖。当运行报出“DMA discriptor”错误时，一般需要仔细检查 DMA 接口程序所有实参是否正确。

上面介绍的是 DMA 读写数据的接口，但实际运行时 DMA 的相关硬件操作是独立的，因此必须通过软件判断相应回答字值的改变，来进一步判断读写操作是否完成。因此一个完整的 DMA 读写过程往往需要三个步骤，即回答字的初始化、调用 DMA 读写接口和判断回答字的值。参见图 3-1 所示：

```
……
reply=0                          ! 初始化回答字为 0
call athread_put(0,c_slave(1),c(1,my_id),100*4,reply,0,0)
! 如果 DMA 完成操作，回答字 + 1
do while (reply.ne.1)      ! 判断回答字的值
end do
……
```

图 3-1 完整的 DMA 操作软件过程

实际操作过程中，由于 reply 回答字判断过程在 -O2 或更高级的优化时被编译器优化掉，而导致程序结果不正确。因此回答字判断过程需要单独写一个文件，并单独 -O0 编译，在众核编程过程中通常以 reply_wait(reply,1)来代替上述过程。

3.3.4 从核同步操作

从核的同步一般有三种：从核阵列的行同步、列同步和全阵列的同步。从核阵列的行同步和列同步操作，在个别情况下采用寄存器 DMA 通信时才会涉及到。而在应用软件中一般只需要全阵列同步功能，该同步接口函数为 sync_array()。即当从核中出现直接访问共享存储区，并且后续从核计算需要保证共享存储区内存影像一致时，需要调用该同步接口。但需要特别注意，该同步操作是所有从核同步，所以当该程序接口调用不当时，程序容易挂死。

第4章　OpenACC*程序设计

OpenACC*语言是在OpenACC2.0文本的基础上，针对“申威26010”异构众核处理器结构特点进行适当的精简和扩充而来的。OpenACC 是由 OpenACC 组织(www.openacc-standard.org)于2011年推出的众核加速编程语言，2013年发布2.0文本。OpenACC是以编译指示的方式提供众核编程所需的语言功能，其主要目的是降低众核编程的难度。本文简要介绍“神威•太湖之光”计算机系统中SWACC编译系统的使用以及其所支持的OpenACC*语言的用法和说明。

4.1 程序设计步骤

在“神威•太湖之光”计算机系统上使用 OpenACC*进行应用程序设计、移植与优化通常遵循的步骤：

1) 完成串行或MPI程序在多核环境下的正确性调试；
2) 搜寻程序中可并行的核心循环代码段，通常可通过以下几个条件或方法来确定：
   a) 核心循环代码段必须为串行代码，在该代码段中不能包含有消息通信功能的代码；
   b) 可以使用类似 gprof 性能工具来确定程序代码中的热点函数，并确定其中的核心代码段；
   c) 支持核心代码段中的函数调用，但通常情况下被调用函数最好是纯函数，或者使用 routine 指示标注被调用的函数。
3) 在核心循环代码段前加上“#pragma acc parallel loop”编译指示；
4) 确定核心代码段中需要私有化的变量，并添加 private 子句：
   a) 在私有化变量分析方面，可以使用 -priv 选项编译，调用 SWACC 编译系统私有化分析功能，SWACC 编译系统会进行私有化变量分析，并将显示分析结果（由于编译器分析存在局限性，需要对分析结果进行确认，以免出现遗漏或错误的情况）；

b) 在数组访问分析方面，可以额外添加 -arrayAnalyse 选项使用 SWACC 编译系统的分析功能进行分析。

5) 编译并运行，确定私有化变量是否正确。

通过 OpenACC*进行应用程序的设计、移植主要包含三个步骤：

1) 找到核心循环；

2) 添加 parallel loop 指示，并确定 private 变量；

3) 编译运行，确定运行结果是否正确，如果结果不正确继续第 2)步，直到程序在从核上正确运行。

使用 OpenACC*对应用程序进行优化过程中，需要考虑如下因素：

1) 充分利用从核的 LDM 空间，关键数据应尽量都存放在 LDM 中；

2) 尽量缩小数据传输和计算的开销比例。并行程序的计算和消息通信开销比例与应用程序的并行算法和实现有关；

3) 合理设置循环分块大小。可以灵活使用 tile 属性，将关键数据尽可能多的放在 LDM，循环分块会影响放入 LDM 中的数据量；

4) 提高数据传输效率。可以使用打包子句将分散的数据整合成大块的数据、使用转置子句将不连续的数据访问变成连续的数据访问；

在程序移植优化方面，需要分析核心循环代码段的数据访问情况，适当的添加 copy、local、pack 等指示，将核心数据导入从核 LDM 中进行访问和计算，使用 OpenACC* 进行程序移植、优化过程中常见的方法和编译指示用法可参见表格 4-1 所示。

表格 4-1 OpenACC*常见用法与编译指示/子句对照表

序号	编译指示子句用法	功能	说明
1	local(变量列表)	在 LDM 中为临时变量(标量、小数组)申请空间	在众核并行代码中需要被改写后再使用，并且其数值不需要更新回主存中的变量，包括数组和标量，可以使用 local 子句在 LDM 中创建相应的存储空间，local 中的变量没有初始值。

2	copyin(变量列表)	只读的标量拷贝到 LDM 中	copyin 的标量将在 LDM 中申请空间，并将原始变量的值赋给 LDM 中对应的变量。
3	copyin(变量列表) annotate(entire(变量列表))	只读小数组整个拷贝到 LDM	copyin 的数组默认会进行分块拷入，当有小数组需要整个拷入 LDM 时，采用 entire 暗示的方式通知编译器。此时对应的数组将在每个加速线程的 LDM 中申请空间，并将原始数组的数值拷贝到 LDM 中。
4	copy copyin copyout	数组拷入拷出 LDM	按照循环的划分方式将数组分割，以数据片段的方式在 LDM 中申请空间，并在主存和 LDM 间进行数据传输。如果数组与被划分的循环非直接线性仿射则无法对数据进行分割。
5	collapse 子句	最外层循环量较小，并行度不够	collapse 的作用是将紧嵌套的若干循环合并进行并行划分，适用于最外层循环量较小，而且紧嵌套若干层可并行的循环的情况。
6	增加 tile 子句，或者增大 tile 的分块大小	LDM 空间没用满	循环的 tile 大小会直接影响到与之关联的数组的分块情况，增大 tile 的块大小可以将每次拷入拷出 LDM 的数据增大。
7	对数组访问相关的内层循环前添加 loop 指示，并根据需要使用 tile 子句	parallel copy 的多维数组太大，LDM 存放不下	当仅对最外层循环划分时，某些多维数组的数据量过大，无法一次性放入 LDM，则可以在与数组访问相关的内层循环前添加 loop 指示，用于将对应的循环分块处理，相应的会对相关的数据也分块，则可以将大数据划分成小的数据块。
8	loop 指示 tilemask 暗示子句 tile 子句	数组访问与并行循环无关，与内层循环相关，使用	当数组访问与最外层并行循环无关、仅与内层循环相关时，使用 parallel copy 编译器无

		parallel copy 无法拷入 LDM	法确定数组如何分块存储，整个数组通常无法整个放到 LDM 中。则可以在与数组访问相关的内层循环前添加 loop 指示，用于将对应的循环分块，并根据需要设置 tile 子句。另外，在某些情况下，又希望对某些数组屏蔽新增加的 loop tile 信息，可以使用 tilemask 子句，提升某些数组的数据传输效率。
9	data copy	数组访问的数据区间是在运行时确定的	这种情况下需要动态的数据传输，可以使用 data copy 在需要数据的代码前描述所需的数据信息，并可以根据需要使用 if 子句控制数据的重用。
10	cache 子句	离散访问的数组无法放入 LDM	对于离散访问的数组，编译器无法知道拷贝哪些数据到 LDM 中，因此 copy 子句不适用这种情况，可以使用 cache 子句解决这个问题，另外根据程序特点可以使用 readonly 暗示子句对 cache 进行优化，只读的 cache 数据不用刷新回主存。
11	data present	加速函数内需要用到 parallel copy 的数据	当 parallel copy 的数据在 parallel 内所调用的函数中需要使用时，需要实现在 parallel 中拷贝到 LDM 中的数据在加速函数内的重用和映射，可以用 data present 描述重用的数据信息。
12	pack/packin/packout 子句	具有多个访问模式相同的标量或数组，想进一步提升数据传输效率	数据打包子句的作用是将具有相同数据访问模式的变量打包成新的数据结构，可以实现一次数据传输拷贝多个数据对象的目的。另外，当加速段本身在循环内、且打包的数据在该

			循环内不改变时，可以使用 data index 设置数据点，将打包操作外提到循环之外。
13	swap/swapin/swapout 子句	优化跨步数组访问的性能	当二维数组是按列访问时，其访问属于跨步访问，高维数组以此类推。这种情况下访存及数据传输效率较低，可以使用转置子句对原始数据重新布局改善数据传输效率。另外，当加速段本身在循环内、且被转置的数据在该循环内不改变时，可以使用 data index 设置数据点，将转置操作外提到循环之外。
14	routine、data copy、data present 指示	加速代码中有函数调用	加速代码中有函数调用的情况有三种处理方式：1)在被调用函数的定义处使用 routine 指示，表明该函数的定义将生成加速版本，同时可以根据情况使用 data present 和 data copy 来控制函数内的数据重用和传输；2)将函数内联；3)将函数改造成纯函数，即函数所需的数据均通过参数传递。
15	组合子句	临时数组太大，无法放入 LDM	当程序中需要私有化的临时数组很大，使用 local 无法将该数组存放到 LDM 时，可以使用组合子句，将该数组先 private，保证其私有属性，之后根据情况使用 copy 或者 cache 将其数据分段导入 LDM，保证程序性能。

上述具体用法及示例参见《“神威•太湖之光”计算机系统 OpenACC*用户手册》。

4.2 程序优化示例

本节使用矩阵乘程序来说明使用 OpenACC*进行程序移植和优化的过程。

4.2.1 串行代码

矩阵乘串行程序代码 matrixMul.c，如图 4-1 所示。

```
 1 /*
 2  * FILE: matrixMul.c
 3  */
 4
 5 #include <stdio.h>
 6 #include <stdlib.h>
 7 #include <sys/time.h>
 8
 9 #define M 1024
10 #define N 1024
11 #define K 1024
12
13 int A[M][N];
14 int B[N][K];
15 int C[M][K];
16
17 // 计时函数
18 double
19 timer()
20 {
21     double t;
22     struct timeval tv;
23     gettimeofday(&tv, NULL);
24     t = (double)tv.tv_sec*1.0+(double)tv.tv_usec*1e-6;
25     return t;
26 }
27
28 int main()
29 {
```

```
30      int i,j,k;
31      double t1, t2;
32
33      // init A
34      for(i = 0; i < M; i ++)
35          for(j = 0; j < N; j ++)
36              A[i][j] = i;
37
38      // init B
39      for(i = 0; i < N; i ++)
40          for(j = 0; j < K; j ++)
41              B[i][j] = j;
42
43      t1 = timer();
44      // matrix multiply, C=A*B
45
46      for(i = 0; i < M; i ++)
47      {
48
49          for(k = 0; k < K; k ++)
50          {
51
52              for(j = 0; j < N; j ++)
53              {
54                  C[i][k] +=  A[i][j] * B[j][k];
55              }
56          }
57      }
58      t2 = timer();
59          printf("Matrix-Multiply  A[%d][%d]  *  B[%d][%d],  use
time:%.2f\n", M,N,N,K,t2-t1);
60      return 0;
61  }
```

图 4-1 矩阵乘串行程序代码

图 4-1 中例子的核心段是 46~57 行的循环段，即进行矩阵乘运算的代码段。另外，为了比较移植和优化前后的程序性能，在核心代码段前后增加了计时代码。

4.2.2 程序修改

1) 首先使用 swacc 分析该串行程序，并记录串行版本的运行时间。
   - **编译：**swacc matrixMul.c －O3
   - **运行：**提交到高速计算系统资源以单进程方式执行，运行时间：T_Serial = 4.10 秒（注意：运行时间与课题数据规模、运行环境、编译环境等多个因素有关，仅供参考）。
2) 找到 matrixMul.c 代码的核心循环，并在核心循环前添加#pragma acc parallel loop 指示。
   - 在 matrixMul.c 的第 45 行添加**#pragma acc parallel loop** 指示。
3) 使用”swacc －priv”选项编译程序，根据私有变量自动分析结果或通过手工分析，添加 private 子句。所谓需要私有化的变量就是每个从核线程都可能会去改写，并且改写的内存区域有重叠的变量。具体到 matrixMul.c，需要私有化的变量有三个：i、j、k。
   - 在 matrixMul.c 第 45 行的编译指示中添加私有化子句：**#pragma acc parallel loop private(i, j, k)**
4) 使用 swacc 编译，生成可执行程序后，可以使用 1 个主核加 64 个从核的规模运行，这说明通过 OpenACC*改造过的代码可以正确运行于“神威•太湖之光”计算机系统上。

4.2.3 程序优化

OpenACC*程序优化的基本思想是将程序中的关键数据尽可能多的放到 LDM 中，并设法提高程序中数据的传输效率。

1) **将标量、小数组放入 LDM：**对于这个程序只需要将 private 子句换成 local 子句，即将 i、j、k 三个标量放在每个加速线程的 LDM 中，其性质仍然是加速线程私有的。
   - 第 45 行编译指示变为：**#pragma acc parallel loop local(i, j, k)**
   - 编译运行，验证正确性和效果。

2) **将关键数组拷贝到 LDM**：程序中访问的数组有三个，A、B、C，其中 A 和 B 是只读的，C 是先读后写的，对于只读的数据只需要拷入(copyin)，只写的数据只需要拷出(copyout)，否则既需要拷入也需要拷出(copy)。在程序中第 45 行添加数据拷贝子句。

- 第 45 行编译指示变为：**#pragma acc parallel loop local(i, j, k) copyin(A, B) copy(C)**
- 编译运行，验证正确性和效果，使用同样规模运行时间约 T_Copy = 9.02 秒。

可以看到，程序的运行时间不仅没有变快，反而变慢了。注意一下编译时的屏幕输出或者使用-keep 选项查看变换后的_slave 中间文件，可以发现在加速代码中对 B 数组的访问仍然是直接访问主存数据，并没有使用 LDM 空间。

分析程序可以发现，B 数组的访问与并行循环 i 是无关的，也就是说加速线程每执行一次并行循环迭代 i 都需要访问 B 数组所有的数据，由于 LDM 容量有限，B 数组无法整个拷入加速线程的 LDM，因此对 B 数组的访问被处理成直接访问主存数据。

而 A、C 数组的访问是跟并行循环 i 线性相关的，也就是说跟随着并行循环 i 的划分，每个加速线程所需要读取的 A 数组的数据和需要改写的 C 数组的数据也是不同的，因此，对于 A、C 数组，可以仅将所需的数据放入对应加速线程的 LDM。

3) **解决 B 数组拷贝到 LDM 的问题**：数组 B 之所以没有放入 LDM 中，是因为数组 B 相关的循环没有划分，导致数据太大没法放入 LDM，解决数据过大无法放入 LDM 的办法是对没有进行划分的循环使用 tile 进行分块，使得循环分块后对应的数据能够拷入 LDM，对于这个程序具体实现上有两种方式：

a) **方式 1：对 k 循环 tile，并对数组 B 转置拷入(swapin)**

B 数组访问相关的循环变量是 k 和 j，首先可以选择对 k 循环 tile，其用意是将循环分块执行，同时可以使相关的数据按照同样的分块方式拷贝到 LDM 中。具体做法是：

- 在第 48 行，k 循环前添加 **#pragma acc loop tile(2) annotate(tilemask(C))**

- 最外层 i 循环的 loop 指示表明 i 循环将在多个加速线程间并行划分
- 而内存的 k 循环上的 loop 指示表明 k 循环将在同一个加速线程内分块执行，此处 tile 指定分块大小为 2。
- 由于数组 C 的访问也与 k 相关，对 k 循环分块之后，会影响到 C 数组的数据传输，添加 annotate(tilemask(C))的作用是，通知编译器 C 数组忽略 k 循环的分块信息，即 C 数组与 k 相关的维度（最低维）不分块，整个拷入 LDM，这样可以获得高效的数据传输。

- 修改第 45 行的编译指示，将 copyin(B)改成 **swapin(B(dimension order:1, 2))**
  - 第 45 行编译指示变成**#pragma acc loop local(i,j,k) copyin(A) copy(C) swapin(B(dimension order:1, 2))**
  - 之所以使用 swapin 代替 copyin 是因为与数组 B 的访问相关的下标时 j 和 k，分别对应的是 B 数组的高维和低维，而 j 循环在 k 循环之内，也就是说 B 数组的访问是不连续的。不连续的数据访问会导致数据传输效率较低，而 swapin 的作用是对 B 数组的原始数据按照指定的维度顺序进行转置，使用转置后的数据代替原始 B 数组的访问，这样可以使不连续的数据访问变成连续的数据访问。
- 上述两个修改完成后，编译运行，此时的矩阵乘时间为 T_Tile_K = 0.45 秒。

b) 方式 2：对 j 循环 tile

方式 1 是对 k 循环 tile，这个程序还可以对 B 数组的高维分块，即对应的将 j 循环分块执行。

- 在第 51 行，j 循环前添加 **#pragma acc loop tile(2) annotate(tilemask(A))**
  - 与方式 1 的含义相同，该编译指示表明将对 j 循环以 2 为块大小进行分块，相应的会影响到与 j 相关的数据拷贝，包括数组 A

和 B，同时通过 tilemask(A)的暗示子句通知编译器，A 数组忽略该分块信息，即 A 数组与 j 相关的维度不分块。

- 编译运行，该版本矩阵乘的时间为 T_Tile_J = 0.53 秒。
- 方式 2 的运行时间之所以比方式 1 略长，是因为在方式 2 中数组 B 的访问方式仍然是先高维后低维，这对程序的性能有一定影响。
- 因此，可以尝试修改原始程序，调整 j、k 循环的顺序，之后再按照本节介绍的方法进行移植和优化。

至此，该程序基本上修改和优化完毕，优化后的核心段可以获得 9 倍左右的性能加速，后续还可以基于 SWACC 编译器生成的中间代码，进行二次开发，对程序代码核心段中的循环采用 SIMD 进行优化，二次开发的方法见后续章节，SIMD 的使用方式请参考相关用户手册。

4.3 二次开发

在使用 SWACC 调试通过加速程序之后，如果需要在编译器生成的中间代码的基础上进一步分析和优化改造，则可以按下列步骤操作。

1) 中间文件的建立

以 test.c 程序为例，在该文件目录下输入指令：

```
swacc test.c -dumpcommand mk
```

其中-dumpcommand 的用法可以参见《“神威•太湖之光”计算机系统 OpenACC 用户手册》介绍的 SWACC 编译选项，其作用是保留中间文件，并将对中间文件的编译命令写到 mk 文件中。

命令行执行后，将在当前目录下生成 host 程序文件和 device 程序文件，同时编译 host 和 device 程序文件的命令将被写入指定的用户自定义文件(例子中是 mk)中。host 和 device 程序文件的命名规则如下（以 c 程序为例，Fortran 类似）：

a) host 程序文件：源文件名_host.c，如 test_host.c；

b） device 程序文件：源文件名_slave_行号.c，其中行号为加速区指示所在的行号，如有多个 parallel loop 指示，则会分别生成不同的 device 程序文件，如 test_ slave_7.c。

2） 中间文件的分析改造

用户可根据需要对中间文件（host 程序文件、device 程序文件和编译命令文件）进行修改。

3） 中间文件的重新编译

执行之前保留的中间文件编译过程，执行 sh 编译命令文件名（例子中为 sh mk）即可。

第5章　串行优化

5.1 常用编译选项优化

1) -O0 表示没有优化，Debug 中的-g 选项与该选项完全兼容。
2) -O1 最小程度的优化，在编译时间上与-O0 没有很大区别。该优化只局限于线性代码（基本块），例如窥孔优化和指令调度。-O1 级别的优化编译时间最短。
3) -O2 选项是默认编译优化选项，需要较长的编译时间，但也带来了更加突出的优化效果：
   - 针对内层循环：
     - 循环展开
     - 简单 if 转化
     - 循环相关优化
   - 二遍指令调度
   - 基于第一遍指令调度的全局寄存器分配
   - 函数级的全局优化
     - 局部冗余代码删除
     - 无效 store 删除
     - 控制流优化
     - 强度减弱优化，循环结束判断替换优化。
     - 跨基本块的指令调度
   - -O2 选项意味着-OPT:goto=on,该选项将 goto 结构转化为更高级的结构，例如 for 结构。
   - -O2 选项设定-OPT:Olimit=6000.
4) -O3 选项打开更多的优化开关，使用户的程序得到最快的运行速度（注意：有可能优化过头导致程序运行出错）。该选项除了包括所有-O1,-O2 的优化之外，还包含了如下选项：

- -LNO:opt=1,打开循环嵌套优化开关.
- -OPT 选项和下列选项的组合
  - OPT:roundoff=1
  - OPT:IEEE_arith=2
  - OPT:Olimit=9000
  - OPT:reorg_common=1

为避免程序在运行过程中出现浮点异常情况，建议使用 swcc/swf90/swCC 缺省-O2 编译时，加上“-OPT:IEEE_arithmetic=2”编译选项；使用 swgcc/swg++ 缺省-O2 编译时，加上“-mieee”选项。

5.2 编译选项优化

由于主从核需要分别进行编译，在编译选项优化上也有一些差异，下面分开介绍主核和从核的编译优化选项。

5.2.1 主核编译选项优化

5.2.1.1 数据预取

1) -CG:pf_L1_ld:pf_L1_st:pf_L2_ld=0:pf_L2_st=0

此选项可以改变预取指令的类型，决定生成的预取指令是否是优先淘汰的。

2) -CG:pref_opt=0

CG 阶段的一个有关循环中的数据预取优化的选项，对部分课题有效。

3) -LNO:prefetch=0

关闭数据预取的选项。

4) -LNO:pf_ahead=3

调整预取提前量的选项，表示提前 3 个 Cache 行预取。

5) -LNO:prefetch_iter_min=TRUE

两次数组访问间相差超过 10 次迭代的情况下也实施预取。

5.2.1.2 循环优化

1) -LNO:full_unroll=5

循环展开，循环迭代次数为常量，且小于等于 5 的循环进行完全循环展开。

2) -LNO:vintr=0

关闭将循环中的 sin 转换成 vsin 的选项，关闭该优化有可能让编译器进行更多其它优化，对部分课题有效，缺省为-LNO:vintr=1。

3) -LNO:fis_scc=1

循环优化中进行较激进的循环分裂的优化，对一个循环中没有相关性的语句集都分裂成一个单独的循环。对循环中有 Cache 冲突严重的课题该选项可能有效，缺省为-LNO:fis_scc=0。

4) -LNO:fission=0

关闭所有循环分裂的选项，缺省为-LNO:fission=1。

5) -LNO:blocking=0

关闭循环分块的选项，缺省为-LNO:blocking=1。

5.2.1.3 其它优化

1) -OPT:alias=strongly_typed

别名分析相关的选项，加上该选项，编译器认为不同类型之间的访存不存在别名，程序中有大量通过指针访存的，优化效果会比较明显。但如果不能保证程序中不同类型之间不会访问相同内存位置，使用该选项有可能引起正确性问题。

2) -OPT:div_split=1

将除法优化成乘倒数的优化，精度上有可能有一定损失。

3) -WOPT:combine=0

关闭全局优化中某些操作组合的优化，对部分课题有效。

4) -OPT:unroll_times=8

循环展开选项，循环展开 8 次。缺省为 4 次。对于循环中语句较多的情况，该选项经常与-OPT:unroll_size=512 一起使用（可以写成这样：

-OPT:unroll=8:unroll_size=512，也可以分开写），表示展开后循环中的指令数只要不超过 512 条，就展开 8 次，否则按照编译器缺省展开次数展开。

5) -OPT:Olimit=0

对较大程序进行优化时，不考虑所建数据结构是否太大的因素。

6) -OPT:roundoff=3

一般针对 fortran 课题，对运算顺序进行调整以寻求更多的优化机会。对部分课题优化效果明显，但可能造成精度损失。

7) -Ofast

该选项将多项较激进的优化打开，包括上述 2、5、6、7。

8) -IPA

过程间分析，可以进行常量传播，inline 等多项优化。

9) -OPT:IEEE_arithmetic=2

兼容 IEEE754 浮点标准的选项,相当于多种商用编译器中的-ieee 选项(gcc 的-mieee 选项）。

5.2.1.4 反馈式编译优化

为获得更优的程序优化效果，“神威•太湖之光”计算机系统的基础编译器支持反馈式编译优化，要编译两遍。

1) 第一遍编译增加如下选项：

```
-O2/-O3  -fb_create Mid  -fb_phase=4  -fb_type=2/10
-fb_edge_profile  *.c  -lcginstr  -lstdc++
```

2) 第二遍使用如下编译：

```
-O2/-O3  -OPT:feed=Mid*  *.c
```

5.2.2 从核编译选项优化

与主核略有不同，因从核无 cache，只有 LDM，因此不需要预取类优化和循环类优化方法。其他优化方法主要有：

1) -OPT:alias=strongly_typed/-OPT:alias=disjoint

别名分析相关的选项，加上该选项，编译器认为不同类型之间的访存不存在别名，程序中有大量通过指针访存的，优化效果会比较明显。但如果不能保证程序中不同类型之间不会访问相同内存位置，使用该选项有可能引起正确性问题。

2) -OPT:div_split=1

将除法优化成乘倒数的优化，精度上有可能有一定损失。

3) -WOPT:combine=0

关闭全局优化中某些操作组合的优化，对部分课题有效。

4) -OPT:Olimit=0

对较大程序进行优化时，不考虑所建数据结构是否太大的因素。

5) -OPT:roundoff=3

一般针对 fortran 课题，对运算顺序进行调整以寻求更多的优化机会。对部分课题优化效果明显，但可能造成精度损失。

6) -Ofast

该选项将多项较激进的优化打开。

7) -IPA

过程间分析，可以进行常量传播，inline 等多项优化。

8) -inline 函数名

将频繁调用的函数内联，提高程序运行效率。

5.2.3 快速数学库的使用

“神威•太湖之光”高效能计算机系统提供两种基础数学库，一个是严格遵守 ieee 754 标准的数学库，一个是损失部分精度（最后两位有效数字）的快速数学库，建议用户根据课题精度需要选取合适的数学库进行链接。

1) 标准数学库，编译器自动链接：-lm

2) 快速数学库，用户需要手动链接：

- -lmfast_slave 损失部分精度；
- -lmldm_1KB_slave 与主核数学库相同精度，但需要额外占用 1KB 的 LDM 空间；

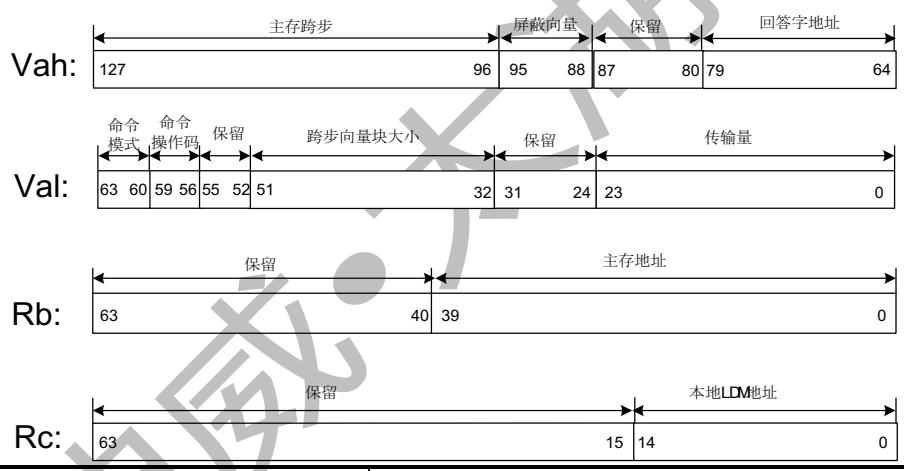
- -lmldm_4KB_slave　　与主核数学库相同精度但需要额外占用 4KB 的 LDM 空间；
- -lmldm_6KB_slave　　与主核数学库相同精度但需要额外占用 6KB 的 LDM 空间。

5.3 从核访存优化

5.3.1 DMA intrinsic

DMA intrinsic 的主要功能是在从核的 LDM 和主存之间进行数据的传输，DMA 操作只能由从核进程发起。DMA 传输的峰值性能在主存地址为 128B 对界，且传输量为 128B 倍数的时候。

图 5-1 DMA 参数各字段功能



dma　　va, rb, rc	发起 DMA 请求和 DMA 栏栅操作命令

DMA 为异步操作，流水线 0 发送 DMA 命令到从核中的通道缓冲后即结束，可继续运行，通道缓冲满的情况下流水线 0 暂时停止运行。

数据 DMA 中使用 DMA 的命令模式包括：

- 单从核模式
- 广播模式
- 行模式
- 广播行模式
- 行集合模式

用户在程序中使用编译器提供的 DMA 接口时，必须包括头文件：

#include “dma.h”

1) 扩充的关键字：

- DMA 描述符（dma_desc）：dma_desc 使用 128 位的向量数据类型；
- DMA 模式（DMA_MODE）：

```
enum DMA_MODE {
    PE_MODE,
    BCAST_MODE,
    ROW_MODE,
    BROW_MODE,
    RANK_MODE,
};
```

- DMA 操作码（DMA_OP）：

```
enum DMA_OP {
    DMA_PUT,
    DMA_GET,
    DMA_PUT_P,
    DMA_GET_P,
    DMA_BARRIER=5,
};
```

2) 扩充的 intrinsic:

- dma_set_size
- dma_get_size
- dma_set_reply
- dma_get_reply
- dma_set_op
- dma_get_op
- dma_set_mode
- dma_get_mode
- dma_set_mask
- dma_get_mask
- dma_set_bsize

- dma_get_bsize
- dma_set_sleng
- dma_get_sleng
- dma_desc_init
- dma
- dma_wait
- dma_barrier

5.3.1.1 dma_set_size

用法：

```
void dma_set_size (dma_desc *dma_d, int size);
```

功能：

设置 DMA 描述符的数据传输量属性。例如：

```
dma_set_size(dma_d, 256);
```

5.3.1.2 dma_get_size

用法：

```
void dma_get_size(dma_desc *dma_d, int *size);
```

功能：

得到当前 DMA 描述符的数据传输量属性。

5.3.1.3 dma_set_reply

用法：

```
void dma_set_reply(dma_desc *dma_d, int *reply);
```

功能：

设置 DMA 描述符的回答字属性。例如：

```
__thread_local reply;
```

```
dma_set_reply(dma_d, &reply);
```

5.3.1.4 dma_get_reply

用法：

```
void dma_get_reply(dma_desc *dma_d, int *reply);
```

功能：

得到当前 DMA 描述符的回答字属性。例如：

```
__thread_local reply;
dma_get_reply(dma_d, reply);
```

5.3.1.5 dma_set_op

用法：

```
void dma_set_op(dma_desc *dma_d, dma_op op);
```

功能：

设置 DMA 描述符的 DMA 操作属性。例如：

```
dma_set_op(dma_d, dma_put);
```

5.3.1.6 dma_get_op

用法：

```
void dma_get_op(dma_desc *dma_d, int *dma_op);
```

功能：

得到当前 DMA 描述符的 DMA 操作属性。例如：

```
dma_get_op(dma_d, dma_op);
```

5.3.1.7 dma_set_mode

用法：

```
void dma_set_mode(dma_desc *dma_d, dma_mode mode);
```

功能：

设置 DMA 描述符的 DMA 模式属性。例如：

```
dma_set_op(dma_d, pe_mode);
```

5.3.1.8 dma_get_mode

用法：

```
void dma_get_mode(dma_desc *dma_d, int *dma_mode);
```

功能：

得到当前 DMA 描述符的 DMA 模式属性。例如：

```
dma_get_mode(dma_d, dma_op);
```

5.3.1.9 dma_set_mask

用法：

```
void dma_set_mask(dma_desc *dma_d, int mask) ;
```

功能：

设置 DMA 描述符的屏蔽码属性。例如：

```
dma_set_mask(dma_d, 1);
```

备注：

只用于广播和广播行模式。

5.3.1.10 dma_get_mask

用法：

```
void dma_get_mask(dma_desc *dma_d, int  *mask);
```

功能：

得到当前 DMA 描述符的屏蔽码属性。例如：

```
dma_get_mask(dma_d, mask);
```

5.3.1.11 dma_set_bsize

用法：

```
void dma_set_bsize(dma_desc *dma_d, int bsize);
```

功能：

设置 DMA 描述符的跨步向量块大小属性。例如：

```
dma_set_bsize(dma_d, 0x32);
```

备注：

- 只用于跨步模式以及行集合模式；
- 行集合模式下 bsize 是传输到每个从核上的块大小。

5.3.1.12 dma_get_bsize

用法：

```
void dma_get_bsize(dma_desc *dma_d, int  *bsize);
```

功能：

得到当前 DMA 描述符的跨步向量块大小属性。例如：

```
dma_get_bsize(dma_d, bsize);
```

5.3.1.13 dma_set_stepsize

用法：

```
void dma_set_stepsize(dma_desc *dma_d,  int length) ;
```

功能：

设置 DMA 描述符的主存跨步长度属性。例如：

```
dma_set_stepsize(dma_d, 0x128);
```

备注：

只用于跨步模式以及行集合模式。

5.3.1.14 dma_get_stepsize

用法：

```
void dma_get_stepsize(dma_desc*dma_d, int *length);
```

功能：

得到当前 DMA 描述符的主存跨步长度属性。例如：

```
dma_get_stepsize(dma_d, length);
```

备注：

只用于跨步模式以及行集合模式；

5.3.1.15 dma_desc_init

用法：

```
void dma_desc_init(dma_desc *dma_d);
```

功能：

对当前的 DMA 描述符 dma_d 进行初始化。初始化的缺省设置为：

- 主存跨步长度：0
- 主存跨步向量块大小：0
- dma 模式：pe_mode
- dma 操作：dma_get
- dma 回答字地址：
- 屏蔽码：0
- 数据传输量：128B

5.3.1.16 dma

用法：

```
void dma(dma_desc dma_d, long mem, long ldm);
```

功能：

根据参数指定的 DMA 描述符、主存起始地址和局存起始地址，发起 DMA 操作。

5.3.1.17 dma_wait

用法：

```
void dma_wait( int *reply, int count);
```

功能：

根据参数指定的回答字地址等待，直到回答字的值等于 count。

5.3.1.18 示例

使用 DMA intrinsic 可以实现 DMA 描述符的设置、DMA 传输和 DMA 等待的异步，可以提高从核 DMA 效率，图 5-2 是 DMA intrinsic 的宏定义。

```
#ifndef  _ADMA_H_INCLUDE
#define  _ADMA_H_INCLUDE

#define A_DMA_IGET(mode,src,dest,len,re_addr)    \
({                                               \
    dma_desc __da__=0;                           \
    dma_set_op(&__da__, DMA_GET);                \
    dma_set_mode(&__da__, mode);                 \
    dma_set_size(&__da__, len);                  \
    dma_set_reply(&__da__, re_addr);             \
    dma(__da__, src, dest);                      \
 })

/*backup athread_get*/
#define A_DMA_GET(mode,src,dest,len,re_addr)     \
({                                               \
    dma_desc __da__=0;                           \
    dma_set_op(&__da__, DMA_GET);                \
    dma_set_mode(&__da__, mode);                 \
    dma_set_size(&__da__, len);                  \
```

```
    dma_set_reply(&__da__, re_addr);              \
    dma(__da__, src, dest);                       \
    dma_wait(re_addr,1);                          \
 })

#define A_DMA_GET_SET(da,mode,len,re_addr)        \
({                                                \
    dma_set_op(&da, DMA_GET);                     \
    dma_set_mode(&da, mode);                      \
    dma_set_size(&da, len);                       \
    dma_set_reply(&da, re_addr);                  \
})

#define A_DMA_GET_RUN(da,src,dest)                \
({                                                \
    dma(da, src, dest);                           \
})

#define A_DMA_GET_WAIT(re_addr,n)                 \
({                                                \
    dma_wait((re_addr), n);                       \
})

#define A_DMA_PUT_SET(da,mode,len,re_addr)        \
({                                                \
    dma_set_op(&da, DMA_PUT);                     \
    dma_set_mode(&da, mode);                      \
    dma_set_size(&da, len);                       \
    dma_set_reply(&da, re_addr);                  \
})

#define A_DMA_PUT_RUN(da,src,dest)                \
({                                                \
    dma(da, src, dest);                           \
})

#define A_DMA_PUT_WAIT(re_addr,n)                 \
```

```
({                                                  \
    dma_wait((re_addr), n);                         \
})

#endif  //_ADMA_H_INCLUDE
```

图 5-2 DMA intrinsic 宏定义

在实际程序设计中，按顺序调用 A_DMA_GET_SET、A_DMA_GET_RUN 和 A_DMA_GET_WAIT 三个函数就可以实现 DMA_GET 的功能。尤其是在反复调用 DMA 传输的循环体内，可以省去每次设置 DMA 描述符的开销，如图 5-3 所示。

```
……
dma_desc gva = 0;
volatile unsigned long hs_reply = 0;
A_DMA_GET_SET(gva, PE_MODE, sizeA, &hs_reply);
For(){
   hs_reply = 0;
   A_DMA_GET_RUN(gva, &(global_hs[beginIhs][beginJhs]),
                 compute_data.hs[5]);
   ……
   A_DMA_GET_WAIT(&hs_reply, 1);
}
……
```

图 5-3 DMA intrinsic 使用示例

5.3.2 双缓冲模式

提高“申威 26010”异构众核处理器加速性能的关键是如何降低从核或隐藏通信开销。采用双缓冲模式，就是当需要多轮次的 DMA 读写操作时，在从核的局部存储空间上申请 2 倍于通信数据大小的存储空间，以便存放两份同样大小且互为对方缓冲的数据。双缓冲通信通过编程来控制和实现，具体方法：除了第一轮次（最后一轮次）读入（写回）数据的通信过程之外，当从核进行本轮次数据计算的同时，进行下一轮次（上一轮次）读入（写回）数据的通信。

在采用双缓冲模式时，通常需要定义双缓冲标识：index 表示当前轮次的变量，next 表示下一轮次的变量，而 last 表示上一轮次的变量。由于采用双缓

冲模式，从核的双缓冲变量 a_slave(:)为 a_slave(:,1:2)所代替，其中 a_slave(:,1) 和 a_slave(:,2)互为缓冲区。注意此时回答字变量也用 get_reply(1:2)和 put_reply(1:2)代替原来的 reply 变量。双缓冲模式示例如图 5-4 所示。

```
.....
do k = kmin, kmax
   ! 从核计算任务绑定
   if (mod((k-kmin),corenum)+1.eq.slavecore_id) then
      j=jmin
      index=mod((j-jmin),2)+1
       put_reply(mod( (j-jmin)+1,2)+1)=1
      !读入首批数据
      get_reply(index)=0
      call athread_get(0,a(imin,j,k),a_slave(imin,index),
           (imax-imin+1)*4,get_reply(index),0,0,0)
      call athread_get(0,b(imin,j,k),b_slave(imin,index),
           (imax-imin+1)*4,get_reply(index),0,0,0)
       do j=jmin,jmax
          index= mod((j-jmin),2)+1
          next = mod((j-jmin)+1,2)+1
          last = next
          !读入下一轮次计算所需要的数据
         if (j.lt.jmax)then
            get_reply(next)=0
            call athread_get(0,a(imin,j+1,k),a_slave(imin,next),
                 (imax-imin+1)*4,get_reply(next),0,0,0)
            call athread_get(0,b(imin,j+1,k),b_slave(imin,next),
                 (imax-imin+1)*4,get_reply(next),0,0,0)
         end if
          do while (get_reply(index).ne.2)
          enddo        ! 等待本轮次计算所需数据读入完毕
          do i=imin,imax
             c_slave(i,index)=a_slave(i,index)*a_slave(i,index)+
                              b_slave(I,index)*b_slave(i,index)
          enddo
          put_reply(index)=0
```

```
            call athread_put(0,c_slave(imin,index),c(imin,j,k),
                 (imax-imin+1)*4,put_reply(index),0,0)
            do while (ptu_reply(last).ne.1)
            enddo ！等待上一轮次数据写回，第一轮次不必等待直接通过。
         enddo
         do while (ptu_reply(index).ne.1)
         enddo    ！等待最后一批数据写回
      endif
   enddo
   ……
```

图 5-4 双缓冲模式示例

在双缓冲机制具体实现过程中，尤其注意各个双缓冲标识、回答字判断位置等。通过研究该双缓冲过程可以发现，当通信开销小于计算开销时，从核加速才会达到理想的并行加速效果。

5.3.3 原子操作

在多进程共享访问操作时，为了确保数据的完整性，引进了原子操作功能，原子操作是不可分割的，在执行完毕之前不会被任何其它任务或事件中断，图 5-5 为将 n 值原子加到地址为_addr_的数据上的示例。

```
#define updt_addw(_n_, _addr_)                          \
{                                                       \
    unsigned long __tmp__;                              \
    asm volatile( "vshuffle $31, %1, 0x01, %1\n\t"  \
                  "ldi    %0, 4(%1)\n\t"            \
                  "updt   %0, 0(%2)\n\t"            \
                  :"=r"(__tmp__):"r"(_n_),"r"(_addr_):"memory"); \
}
```

图 5-5 将 n 值原子加到地址为_addr_的数据上

5.3.4 寄存器通信接口

寄存器通信可以实现核组内同行、同列从核间细粒度通信。图 5-6 为寄存器通信接口示例。

```
// 向同行目标从核送数
#define LONG_PUTR(var,dest) \
asm volatile ("putr %0,%1\n"::"r"(var),"r"(dest):"memory")

// 读行通信缓冲
#define LONG_GETR(var) \
asm volatile ("getr %0\n":"=r"(var)::"memory")

// 向同列目标从核送数
#define LONG_PUTC(var,dest) \
asm volatile ("putc %0,%1\n"::"r"(var),"r"(dest):"memory")

// 读列通信缓冲
#define LONG_GETC(var)  \
asm volatile ("getc %0\n":"=r"(var)::"memory")
```

用法示例：

```
void CommLeftToRight(int rowid,int colid)
{
        int i,j,k,l,p,dest,length;
        length=(NIS+3)*DIMNV;
        if( colid < 7 ) {
                p=0;
                for(i=0;i<=NIS+2;i++)
                for(j=NJS;j<=NJS;j++)
                for(l=0;l<DIMNV;l++){
                        CommBuff[p]=qy0_s[i][j][l];
                        p=p+1;
                }
                dest=colid+1;
                for(i=0;i<length;i++)
                        LONG_PUTR(CommBuff[i],dest);
        }
        if( colid > 0 ) {
                for(i=0;i<length;i++)  LONG_GETR(CommBuff[i]);
                p=0;
                for(i=0;i<=NIS+2;i++)
```

```
                for(j=0;j<=0;j++)
                for(l=0;l<DIMNV;l++) {
                        qy0_s[i][j][l]=CommBuff[p];
                        p=p+1;
                }
        }
}
```

图 5-6 寄存器通信接口示例

说明：1）读行（或列）通信缓冲，缓冲宽度为向量寄存器宽度。通信缓冲是一个 FIFO 的缓冲，并且具有读后清的属性。如果通信缓冲为空则停顿。2）向同行（或同列）上目标从核发送数据，数据宽度为向量寄存器宽度。PUT 可以点对点操作或广播操作，当广播时目标方至少一个满时停顿。dest 低 4 位有效，其中 dest[3]为通信类型位，为 0 时表示点对点操作，则 dest[2:0]是一个 3 位目标位向量，指示目标从核号；当 dest[3]为 1 时表示广播操作，忽略 dest[2:0]位。

5.3.5 离散存储调整为连续存储

在很多科学计算类课题算法中，对主要变量位置坐标的每个方向都要进行相应的差分求解，因此不可避免的存在离散访存，并且导致计算开销急剧增加。因此在针对存在大量离散访存的程序段的众核并行设计过程中，首先在计算核心上将原来离散的数组调整成方便通信的读入和写回的存储顺序，然后计算核心进行通信读入数据、计算和通信写回数据，最后计算核心将写回的数据再次调整回原来的存储顺序，见图 5-7 和图 5-8。尽管相较于原来的算法，增加了前后两个数组存储顺序调整的过程，但由于上述过程都是在计算核心上来完成的，两个存储顺序调整所导致的计算核心开销增加的不大。另外，计算方面则由于计算核心对专属局部存储空间访问延迟小，使得计算核心计算开销大大减小。综合考量后，众核并行后的加速效果还是比较理想的。

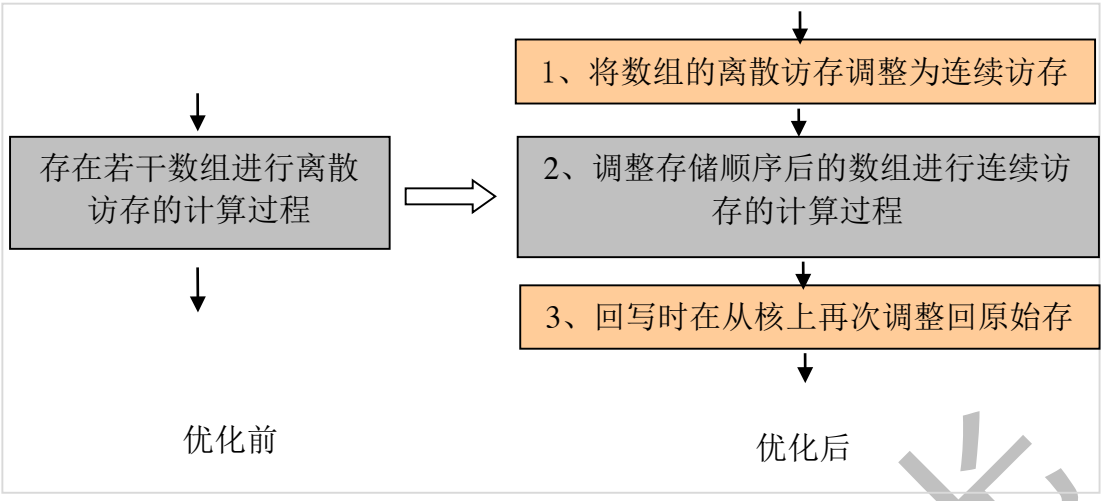


图 5-7 离散访存问题的优化示意图

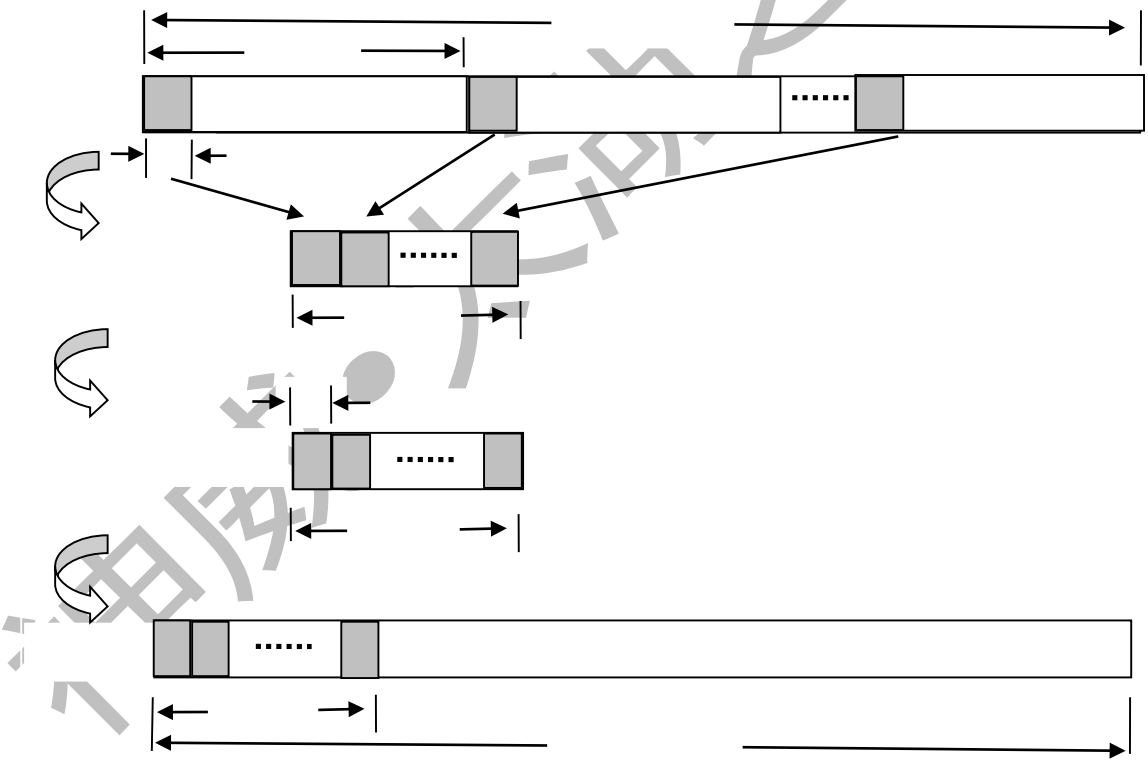


图 5-8 单个计算核心上数组存储顺序调整过程示意图

5.4 短向量优化

5.4.1 合理使用数组

为了有效的进行向量化，程序中对数组的操作需要尽量的连续访问，这意味着对于多维数组，尽量选择在最右边的维（C 语言）进行向量化。另外，尽量让数组作为结构的一个域，而不是使用结构类型的数组。例如表格 5-1 的例子中，右边数组更好向量化。

表格 5-1 数组的向量化比较

```
struct {
   float a, b, c, d, e, f;
}s[99];

int i, n;
for ( i=0; i < n; i++ )
     x += s[i].a * s[i].e;
```

```
struct {
   float a[99], b[99], c[99],
         d[99], e[99], f[99];
}s;
int i, n;
for ( i=0; i < n; i++ )
     x += s.a[i] * s.e[i];
```

5.4.2 对函数调用的处理

如果要向量化的循环中包含一个函数调用，占用很大开销，可以通过将该函数 inline 进循环体或者将该函数体向量化的方法来解决。表格 5-2、表格 5-3、表格 5-4 为函数体的向量化前后程序。

表格 5-2 常规函数调用程序

```
unsigned fc[FCL],q,k;

for (k=0;k<FCL;k++) {
  if (pjoyl(fc[k]&q))
     s--;
  else
     s++;
}
int pjoyl(unsigned s)
```

```
{
  int i;
  for(i=16;i>=1;i>>=1)
    s ^= s>>i;
  return s%2;
}
```

表格 5-3 函数体 inline 调用

```
int fc[FCL],q,k,tmp[8],i;
intv8 v,vq,vi;

vq = simd_set_intv8(q,q,q,q,q,q,q,q);
for (k=0;k<FCL;k+=8) {
    simd_load(v,&fc[k]);
    v=v&vq;
    for (i=16;i>=1;i>>=1) {
        vi=simd_set_intv8(i,i,i,i,i,i,i,i);
        v ^= v>>vi;
        simd_store(v,tmp);
        for (i=0;i<8;i++)
            if (tmp[i]%2) s--;
            else  s++;
    }
}
```

表格 5-4 函数体向量化调用

```
unsigned fc[FCL],q,t[8],k;
intv8 vq,v;

vq = simd_set_intv8(q,q,q,q);
for (k=0;k<FCL;k+=8) {
    simd_load(v,&fc[k]);
    v=v&vq;
    vpjoyl(v,t);
    for (i=0;i<8;i++)
       if (t[i]%2) s--;
       else  s++;
}
```

```
void vpjoyl(intv8 v,int*r)
{
  int i; intv8 vi;
  for (i=16;i>=1;i>>=1) {
      vi = simd_set_intv8(i,i,i,i);
      v ^= v>>vi
  }
  simd_store(v,r);
}
```

在上面的例子中，扩展类型变量 vs（程序中是 v）作为函数的参数传递进 vpjoyl。在 SIMD 扩展中，扩展类型在函数调用过程中的使用约定需要与浮点类型一致。

5.4.3 更有效的使用主核 Cache

高效地使用各级 Cache 对于性能的提高是极为重要的，到主存储器中访问数据要比在一级 Cache 中访问数据慢数十倍。为了更好地使用 Cache，程序需要尽量使用同一个 Cache 行的所有数据而不是各不同 Cache 行的部分数据，而且程序最好能在数据被替换出 Cache 以前尽量多的重用这些数据。当然，为了从 SIMD 部件中获得性能的提升，也要求程序最好访问连续的内存区域，这一点来讲，Cache 与 SIMD 部件对程序的要求是一样的。

例如在矩阵乘的例子中，第一个矩阵通常按照行优先的顺序访问，第二个矩阵则要按照列优先的顺序访问，显然对第一个矩阵的访问具有空间局部性，因为访问的是连续地址，而对第二个矩阵的访问每一次访问的经常是一个新的 Cache 行。为了实施向量化，通常需要将第二个矩阵的某一列数据逐一装入，然后重新组织到向量寄存器中，这样会增加很大的开销。下面以图 5-9 和图 5-10 的程序示例来说明优化方法。

```
for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
        for (k=0;k<n;k++) c[i][j] = c[i][j] + a[i][k]*b[k][j];
        // 如果直接进行向量化，则程序变换为：
        for (k=0; k<n; k+=4) {
```

```
                simd_load(va, &a[i][k]);
                vb <- (b[k][j], b[k+1][j], b[k+2][j], b[k+3][j])
                vt += va * vb;
            }
            simd_store(vt, temp);
            c[i][j] = temp[0]+temp[1]+temp[2]+temp[3];
        }
    }
```

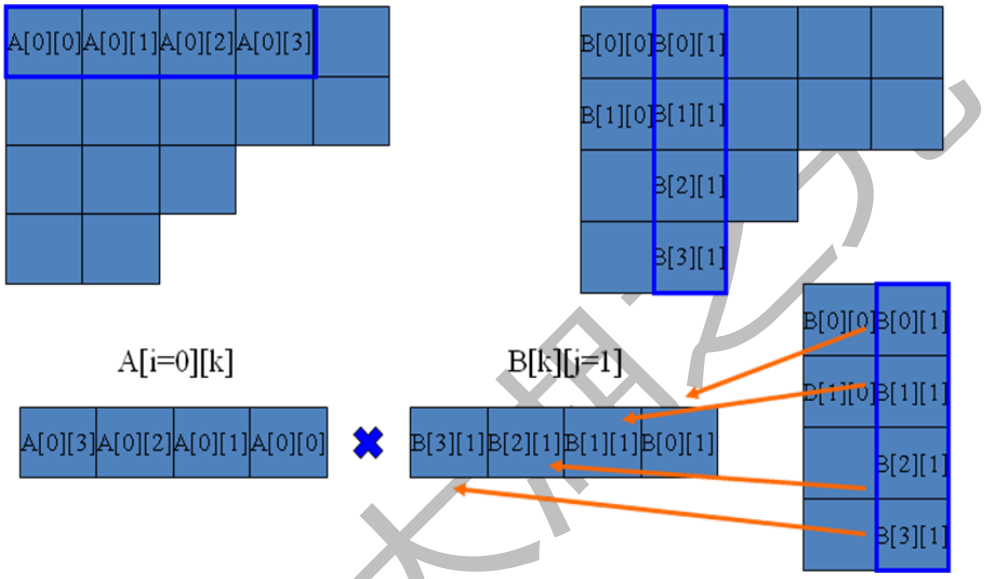


图 5-9 矩阵乘向量化未优化示例

向量化的时候，数组 A 连续的 4 个元素可以直接装载到向量中，而数组 B 对应作乘法的元素在内存中并不连续，相当于是对多个 cache 行的访问，这样就需要花费较大的开销去拼接。但是，如果把程序做一下变换，就可以很好的避开访问不同 cache 行的开销，把 k 级循环和 j 级循环互换，然后再做向量化，如图 5-10 代码所示。

由于 j 是最内层循环，因此对 B[k][j]的访问是连续的，而 A[i][k]在 j 级循环里面仅相当于一个常数，这样就可以用数组 A 的一个元素同时去乘数组 B 的连续四个元素，完全屏蔽了不连续的 cache 访问。

```
for (i=0;i<n;i++) {
     for (j=0;j<n;j++) {
          for (k=0;k<n;k++) {
               c[i][j] = c[i][j] + a[i][k]*b[k][j];
          }
    }
}
// 向量化后程序变换为：
for (i=0;i<n;i++) {
     for (k=0;k<n;k++) {
          simd_loade(&a[i][k]);
          for (j=0;j<n;j += 4) {
               simd_load(vb,&b[k][j]);
               vt = va*vb;
               simd_load(vs, &c[i][j]);
               vs += vt;
               simd_store (vs, &c[i][j])
          }
     }
}
```

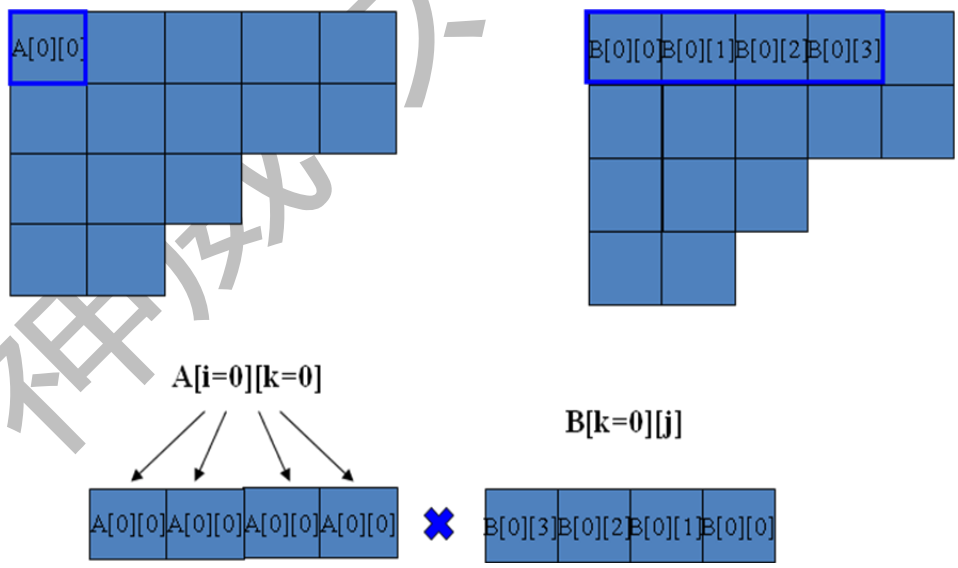


图 5-10　矩阵乘向量化优化示例

5.4.4 对界问题的处理

使用 simd_load 或 simd_store 操作进行变量映射的时候，需要保证标准类型变量为 32 字节对界（对于 floatv4，只需 16 字节对界）。如果出现了不对界的访存，执行的时候操作系统将最终处理该访存引起的异常，这将付出更昂贵的代价。编译器可以保证数组变量的起始地址为 32 字节对界，其余要求用户保证。

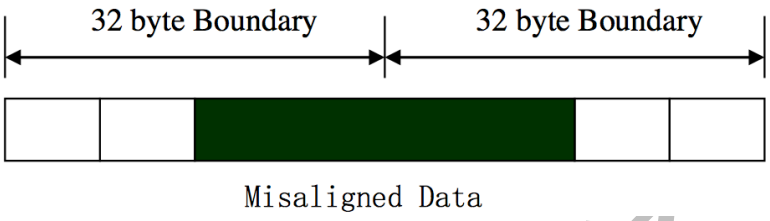


图 5-11 数据 Cache 中跨 32 字节的不对界访存

对于有些不对界的情况，程序可以进行调整，如图 5-12 所示：

```
double a[n],b[n+2];
for ( i=0; i<n; i++) {
    a[i]=a[i]+b[i+2];
}
```

图 5-12 处理扩展类型的对界问题

在向量化图 5-12 循环时，程序若保证数组 a 的对界要求，必然就不能保证数组 b 的对界要求，这时可以使用不对界访存接口直接处理。可以使用手册提供的两个不对界访存接口，分别是向量不对界取 simd_loadu 和向量不对界存 simd_storeu。使用不对界访存指令可以不考虑向量类型 32 字节对界（floatv4 为 16 字节对界)的要求，只需满足存取基本单元的对界要求即可，如 int 和 float 类型需要 4 字节对界，double 类型需要 8 字节对界。程序转换如图 5-13 所示。

```
#include “simd.h”
double  a[n],b[n];
doublev4  aa, bb,bb1,vtmp;
for ( i=0; i<(n/4)*4; i += 4) {
    simd_load(aa, &(a[i]));
    simd_loadu(bb, &(b[i+2]));
    aa=aa+bb;
    simd_store(aa,&a[i]);
```

```
}
for ( i=(n/4)*4; i<n; i++) {
       a[i]=a[i]+b[i+2];
}
```

图 5-13 数组不对界访存接口示例

调用不对界访存接口直接装入 b[i+2]，然后可直接做向量加法操作。

5.4.5 循环下标的处理

通过数组循环下标的处理，把多维改成一维数组，然后再使用向量化进行优化。

```
void stencil13(Real *in, Real *out, int width, int height, Real
coe1, Real coe2, Real coe3, int count)
{
   Real  *fin = in;
   Real  *fout = out;
   int s,i,j;
   for(s = 0; s < count; s ++) {
      for(j = 2; j < height-2; j ++)  {
          int c = j*WIDTHP+1;
          int n = c-WIDTHP;
          int s = c+WIDTH;
          int w = c-1;
          int e = c+1;
          int ne = n+1;
          int nw = n-1;
          int se = s+1;
          int sw = s-1;
          int nn = n-WIDTHP;
          int ee = e+1;
          int ww = w-1;
          int ss = s+WIDTHP;
          for(i = 2; i < width-2; i ++) {
              Real vw = coe2*coe1*fin[w]
                      + (0.05+coe3)*0.2*coe2*fin[ww]
```

```
                        + 0.1*coe2*fin[c]
                        + (0.13+coe2)*1.1*coe2*fin[sw]
                        + 0.9*coe2*fin[nw];
                Real ve = coe2*coe1*fin[e]
                        + (0.05+coe3)*0.2*coe2*fin[ee]
                        + 0.1*coe2*fin[c]
                        + (0.13+coe2)*1.1*coe2*fin[se]
                        + 0.9*coe2*fin[ne];
                Real vn = coe2*coe1*fin[n]
                        + (0.05+coe3)*0.2*coe2*fin[nn]
                        + 0.1*coe2*fin[c]
                        + (0.13+coe2)*1.1*coe2*fin[nw]
                        + 0.9*coe2*fin[ne];
                Real vs = coe2*coe1*fin[s]
                        + (0.05+coe3)*0.2*coe2*fin[ss]
                        + 0.1*coe2*fin[c]
                        + (0.13+coe2)*1.1*coe2*fin[sw]
                        + 0.9*coe2*fin[se];
                Real vc = coe2*coe1*fin[c]
                        + (0.05+coe3)*0.2*coe2*fin[w]
                        + 0.1*coe2*fin[e]
                        + (0.13+coe2)*1.1*coe2*fin[n]
                        + 0.9*coe2*fin[s];
                fout[c] = coe3*vw + coe3*ve + coe1*vn
                        + coe1*vs + coe2*vc;
                c++; n++; s++; w++; e++; ne++; nw++;
                se++; sw++; nn++; ee++; ww++; ss++;
            }
        }
        Real *tmp = fin;
        fin = fout;
        fout = tmp;
    }
    return;
}
```

图 5-14 数组降维示例

图 5-14 stencil13 函数将二维数组 fin,fout 改为一维数组，所有原二维数组的 13 点 stencil 计算改为一维数组的计算，方便程序的向量化。

具体主从核 SIMD 接口详见《神威基础编译器用户手册》。

5.5 除法平方根优化

“申威 26010”异构众核处理器的除法和平方根操作，无法进行流水运算，可以采用乘加的软件模拟方式来更有效的利用浮点流水线，会得到比较大的性能提升。图 5-15 和图 5-16 分布为采用软件模拟的方式实现除法和平方根的运算。

```
long magic=0x7fde5f73aabb2400;
double ONE=1.0;
union{
    double d;
    long i;
} p;
inline double soft_divide( double a,double b)
{
    double y,q;
    p.d=b;
    p.i=magic-p.i;
    y=p.d*b-ONE;
    q=-p.d*y+p.d;
    y=y*y;
    q=q*y+q;
    y=y*y;
    q=q*y+q;
    y=y*y;
    q=q*y+q;
    q=q*a;
    return q;
}
void main( void )
{
    double a=123.0;
    double b=0.000000000001234567;
```

```
    double q;
    double volatile r;

    r=1/b;
    q=soft_divide(a,b);
    printf("%lf %lf %lf \n",q,a/b,a*r);
}
```

图 5-15 除法操作软件模拟

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
long magic=0x5fe6eb50c7b537a9;
inline double soft_sqrt(double y) {
    long i;
    double x2, q, r, z;

    x2 = y * 0.5;
    z = y;
    i = *(long *) &z;
    i = magic - (i >> 1);
    z = *(double *) &i;
    z = z * (1.5 - (x2 * z * z));
    z = z * (1.5 - (x2 * z * z));
    z = z * (1.5 - (x2 * z * z));
    q = y*z;
    r = y - q*q;
    z = q + r*z;
    return z;
}

double y=123.32352;
main()
{
    printf("res=%lf %lf %lf\n", y, sqrt(y), soft_sqrt(y));
}
```

图 5-16 平方根操作软件模拟

5.6 自动向量化

“申威 26010”异构众核处理器主、从核有 256 位的 SIMD 扩展部件，自动向量化系统就是针对该 SIMD 扩展部件特征，对应用程序进行依赖关系分析和并行性分析，发掘应用程序中蕴含的数据并行，生成面向从核心和主核心的可执行向量程序。

“神威•太湖之光”自动向量化系统基于神威编译器开发，并作为一个独立版本进行发布，通过指定版本号 5.421-sw-gy 即可以进行使用。

5.6.1 自动向量化基本用法

自动向量化通过发掘应用程序源代码中蕴含的数据级并行，在可执行代码生时成时，用 SIMD（Single Instruction Multiple Data，单指令多数据）指令替换标量指令，提高程序的运行效率。

5.6.1.1 确定编译器版本

程序自动向量化系统基于神威编译器开发，作为神威编译器的一个版本，在使用时需要指定版本号为“5.421-sw-gy”。通过选项-ver NO.可以指定编译器的版本号。使用命令 sw5cc -ver list 可以查看编译器支持的版本，支持自动向量化的版本为 5.421-sw-gy，如图 5-17 所示。

```
5.421-sw-429:   SWCC Compilers: Version 5.421-sw-429 by gzr at devcomp.yichu.jn on 2016-01-21 09:18:47 +0800
5.421-sw-430:   SWCC Compilers: Version 5.421-sw-430 by gzr at devcomp.yichu.jn on 2016-01-25 09:17:54 +0800
5.421-sw-431:   SWCC Compilers: Version 5.421-sw-431 by gzr at devcomp.yichu.jn on 2016-01-26 08:51:03 +0800
5.421-sw-433:   SWCC Compilers: Version 5.421-sw-433 by gzr at devcomp.yichu.jn on 2016-02-22 11:25:40 +0800
5.421-sw-434:   SWCC Compilers: Version 5.421-sw-434 by gzr at devcomp.yichu.jn on 2016-03-24 09:08:50 +0800
5.421-sw-435:   SWCC Compilers: Version 5.421-sw-435 by gzr at devcomp.yichu.jn on 2016-03-28 16:42:11 +0800
5.421-sw-436:   SWCC Compilers: Version 5.421-sw-436 by gzr at devcomp.yichu.jn on 2016-03-29 15:27:19 +0800
5.421-sw-437:   SWCC Compilers: Version 5.421-sw-437 by gzr at devcomp.yichu.jn on 2016-04-06 15:25:05 +0800
5.421-sw-gy:    SWCC Compilers: Version 5.421-sw-gy by swgy2 at sn007 on 2016-03-14 11:30:15 +0800
```

图 5-17 编译器版本列表

5.6.1.2 编译选项

自动向量化需要在编译选项为-O3 的基础上进行，与自动向量化有关的选项有：

1) -LNO:simd=n

控制自动 SIMD 向量化，n：0 为关闭，1 为打开。该选项默认为 0。

示例：

```
1 int main()
2 {
3     int i,A[128], B[128];
4
5     for(i=0; i<128; i++)
6         A[i] = B[i];
7
8     return A[0];
9 }
```

图 5-18 SIMD 示例 exam1.c

使用命令：

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 exam1.c
```

提示信息：

```
(exam1.c:5) LOOP WAS VECTORIZED
```

2) -LNO:simd_report=n

控制向量化信息的打印，n：0 时只打印向量化失败循环的信息，1 时只打印向量化成功循环的信息，2 时同时打印向量化成功和失败循环的信息。该选项默认值为 1。

示例：

```
1 int main()
2 {
3     int i,A[128], B[128];
4
5     for(i=0; i<128; i++)
6         A[i] = B[i];
```

```
 7
 8      for(i=0; i<128; i++)
 9          A[i+1] = A[i];
10
11      return A[0];
12 }
```

图 5-19 SIMD 示例 exam2.c

使用命令：

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1:simd_report=2 exam2.c
```

提示信息：

```
(exam2.c:5) LOOP WAS VECTORIZED.
(exam2.c:8)Loop has dependencies.Loop was not vectorized.
```

5.6.1.3 OpenACC*程序自动向量化

通过 SWACC 编译器提供的选项-HCFlags 和-SCFlags，可以分别向编译运算控制核心代码和运算核心代码的编译器传递参数，即通过选项-HCFlags -ver,5.421-sw-gy 和-SCFlags -ver,5.421-sw-gy 分别实现主核心代码和从核心代码的向量化。

示例：

```
 1       PROGRAM main
 2
 3       integer a(128),b(128),c(128)
 4       integer i,j
 5
 6       do i = 1,128
 7           a(i) = 10
 8           b(i) =6
 9       enddo
10 !$ACC region do local(i,j) copyin(a,b) copyout(c)
11       do j = 1,128
12           do i = 1,128
13               c(i) = a(i) + b(i)
```

```
14              enddo
15          enddo
16
17          write(*,*) c(64)
18          END
```

图 5-20　SIMD 示例 test.f

使用命令：

swacc -HCFlags -ver,5.421-sw-gy -SCFlags -ver,5.421-sw-gy -O3 -LNO:simd_report=2 test.f

提示信息：

(test_host.f:7) LOOP WAS VECTORIZED.

(test_slave_10.f90:31) LOOP WAS VECTORIZED.

5.6.1.4 MPI 程序的自动向量化

使用 MPI 编译器时通过选项-ver 5.421-sw-gy，可以指定基础编译器的版本为 5.421-sw-gy，实现自动向量化。

示例：

```
1 #include "mpi.h"
2
3 int main( int argc, char *argv[])
4 {
5     int myid, numprocs, i, j, n;
6     int mysum=0, sum=0, a[128];
7
8     MPI_Init(&argc,&argv);
9     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
10    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
11
12    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
13
14    for (i = myid + 1; i <= n; i += numprocs)
15    {
```

```
16        for(j=0; j<128; j++)
17          {
18            mysum += a[j];
19          }
20      }
21
22      MPI_Reduce(&mysum,&sum,1,MPI_DOUBLE,MPI_SUM,0,
                    MPI_COMM_WORLD);
23      MPI_Finalize();
24      printf("sum is %d",sum);
25
26      return 0;
27  }
```

图 5-21 SIMD 示例 mpi_test.c

使用命令：

```
mpicc -ver 5.421-sw-gy  -O3 -LNO:simd=1 mpi_test.c
```

提示信息：

```
(mpi_test.c:16) LOOP WAS VECTORIZED.
```

5.6.2 自动向量化支持的功能

5.6.2.1 循环级向量化

自动向量化实现了循环级的向量发掘和代码生成，能够发掘循环迭代间的向量并行性。

示例：

```
1 #include<stdio.h>
2 #define N 128
3 int main(){
4   double a[N][N],b[N][N],c[N][N];
5   int i,j;
6
7   for(i=0;i<N;i++)
8     for(j=0;j<N;j++)
```

```
 9     {
10       a[i][j]=10;
11       b[i][j]=6;
12     }
13
14   for(i=0;i<N/2;i++)
15     for(j=0;j<N;j++)
16       c[i][j]=a[i][j]+b[i][j];
17
18   return 0;
19 }
```

图 5-22 循环级向量化示例

使用命令:

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 1-loop.c
```

输出提示信息:

```
(1-loop.c:8) LOOP WAS VECTORIZED.
(1-loop.c:15) LOOP WAS VECTORIZED.
```

5.6.2.2 控制流向量化

自动向量化针对程序中存在的控制流，利用“申威 26010”提供的向量条件选择指令和向量比较指令，实现控制流的向量化。

示例:

```
 1 #include<stdio.h>
 2 #define N 128
 3 int main(){
 4   float a[N]={0};
 5   float b[N]={1.1,2.2,3.3};
 6   float c[N]={1,2,3};
 7   int i;
 8
 9   for(i=0;i<N;i++){
10     if(b[i]<c[i])
11       a[i]=b[i]+a[i];
```

```
12      else
13        a[i]=c[i]-a[i];
14    }
15
16    return 0;
17 }
```

图 5-23 控制流向量化示例

使用命令：

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 2-ctrl_flow-f4.c
```

输出提示信息：

```
(2-ctrl_flow-f4.c:9) LOOP WAS VECTORIZED.
```

5.6.2.3 归约操作向量化

针对程序中存在的加法、乘法、Count、Max/Min等归约操作进行识别，实现归约向量代码的生成。

示例：

```
 1 #include<stdio.h>
 2 #define N 128
 3 float sum=1;
 4
 5 int main()
 6 {
 7     float a[N]={1,2,3};
 8     int i;
 9
10     for(i=0;i<N;i++)
11         sum+=a[i];
12
13     return 0;
14 }
```

图 5-24 归约操作向量化示例

使用命令：

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 3-reduction-d4.c
```

输出提示信息：

```
(3-reduction-d4.c:10) LOOP WAS VECTORIZED.
```

5.6.2.4 不对齐向量化

针对程序中存在的不对齐访存情况，当不能通过循环剥离等优化生成对齐指令时，可以通过不对齐访存指令实现向量化。

示例：

```
 1 #include<stdio.h>
 2 #define N 128
 3 int main()
 4 {
 5   double a[N],b[N],c[N];
 6
 7   int i;
 8
 9   for(i=0;i<N;i++)
10     a[i]=3;
11   for(i=0;i<N;i++)
12     b[i]=4;
13
14   for(i=0;i<N/4;i++)
15     c[i]=a[i+1]+b[i+2];
16
17   return 0;
18 }
```

图 5-25 不对齐向量化示例

使用命令:

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 4-unalign-d4.c
```

输出提示信息：

```
(4-unalign-d4.c:9) LOOP WAS VECTORIZED.
(4-unalign-d4.c:11) LOOP WAS VECTORIZED.
```

(4-unalign-d4.c:14) LOOP WAS VECTORIZED.

5.6.2.5 类型转换向量化

针对程序中存在的单精度到双精度等类型转换操作，支持实现类型转换的向量化。

示例：

```
 1 #include<stdio.h>
 2 #define N 128
 3 int main(){
 4   float a[N]={0};
 5   double b[N]={1,2,3};
 6   int i;
 7
 8   for(i=0;i<N;i++)
 9     a[i]=b[i];
10
11   return 0;
12 }
```

图 5-26 类型转换向量化示例

使用命令:

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 5-cvt-df.c
```

输出提示信息：

```
(5-cvt-df.c:8) LOOP WAS VECTORIZED.
```

5.6.2.6 数学函数向量化

针对程序中存在的数学函数调用，利用基础向量数学库中的向量函数接口实现向量化，目前实现了 sin、cos、tan 等常用数学函数的向量化版本。

示例：

```
 1 #include<stdio.h>
 2 #include<math.h>
```

```
 3 #define N 128
 4 int main(){
 5   double a[N]={0};
 6   double b[N]={1,2,3};
 7
 8   int i;
 9
10   for(i=0;i<N;i++)
11     a[i]=sin(b[i]);
12
13   return 0;
14 }
```

图 5-27 数学函数向量化示例

使用命令:

```
sw5cc -ver 5.421-sw-gy -O3 -LNO:simd=1 6-math_func-d4.c
```

输出提示信息：

```
(6-math_func-d4.c:10) LOOP WAS VECTORIZED.
```

注意：在数学函数向量化时需要连接 SIMD 数学库，使用选项 -lm_sw2_simd_noieee -lm_sw2_noieee。

5.6.3 更好的使用自动向量化功能

通过自动 SIMD 向量化功能获得更好的程序性能提升，建议程序员在编写程序时注意以下几点。

5.6.3.1 减少指针的使用

依赖关系分析是进行程序变换和向量化发掘的基础，而指针引起的别名问题会严重影响自动 SIMD 向量化功能的依赖分析；另一方面，用指针访问数据时对界分析也会变得更加复杂。建议在编写程序时使用尽量少用指针，尤其是核心循环内的数组操作用数组形式访问，不要使用指针。

示例：

```
int main()
{
    int  A[128], B[128]
    ……
        func(A, B);
        ……
}

int func(int * a, int * b)
{
        ……
        for( i=0, i<128, i++)
            *(a+i) = *(b+i)
        ……
}
```

图 5-28 减少指针使用示例一

图 5-28 代码段中的函数 func 使用了指针 a，b 作为参数，func 内的循环通过指针 a，b 对 main 函数中定义的数组 A，B 进行操作。然而在自动向量化系统进行依赖关系分析时，因为指针 a，b 可能的别名问题，无法对 func 内的循环的依赖关系作出准确分析，只能保守的认为可能存在依赖关系，放弃向量化的机会。指针别名问题是编译器进行程序分析和优化面临的重要挑战，程序员编写程序时应当减少指针的使用，或通过 restrict 关键字告诉编译器更多的信息。将示例代码段中的函数 func 改为如图 5-29 所示。或者使用 restrict 关键字告诉编译器指针 a，b 指向不重叠的内存空间，即改为如图 5-30 所示。

```
int func(int a[], int b[])
{
……
for( i=0, i<128, i++)
    a[i] = b[i]
……
}
```

图 5-29 减少指针使用示例二

```
int func(restrict int * a, restrict int * b)
```

```
{
      ……
      for( i=0, i<128, i++)
          *(a+i) = *(b+i)
      ……
}
```

图 5-30 减少指针使用示例三

5.6.3.2 减少结构体使用

向量访存只能访问连续的地址空间，结构体的使用会使得访存不连续的情况增多。核心循环中如果有结构体数组操作，程序本身可能有很高的向量并行性，然而由于访存的不连续可能无法获得好的性能，建议这种情况下把结构体进行拆分，把对结构体数组的操作变成对多个数组的操作。

示例：

```
struct
{
    int a;
    float b;
} x[128], y[128];

for(i=0; i< 128; i++)
{
    x[i].a = y[i].a;
    x[i].b = y[i].b;
}
```

图 5-31 减少结构体使用示例一

图 5-31 示例代码段中的循环是对结构体数组成员进行操作，在内存访问访问时并不连续，这会导致向量化程序访存不可连续，需要拼接完成数据存取，严重影响程序性能，应将代码改为如图 5-32 示例所示。

```
int x_a[128], y_a[128];
float x_b[128], y_b[128];
```

```
for(i=0; i<128; i++)
{
    x_a[i] = y_a[i];
    x_b[i] = y_b[i];
}
```

图 5-32 减少结构体使用示例二

5.6.3.3 对循环不变量的外提

在循环内对循环不变量进行赋值，会影响程序的执行效率，可以把赋值语句提到循环外。目前自动向量化不能对复杂的循环不变量进行外提，为了获得高效的向量程序，建议编写串行程序时把对循环不变量的赋值语句提到循环外执行。

示例：

```
do k=1,nf
  do j=1,ny
    do i=1,10
      j1=j_offset(npy)+j-1
      ua(i,j,k)=uat(i,j1,k)
    enddo
  enddo
enddo
```

图 5-33 循环不变量外提示例一

图 5-33 代码段中，i 循环内的语句 j1=j_offset(npy)+j-1 对 j1 变量进行赋值，而 j1 对 i 循环是一个循环不变量，所以可以把这条赋值语句提到 i 循环外，这样可以提高程序运行的效率和方便自动向量化分析。代码可优化为图 5-34 所示。

```
do k=1,nf
  do j=1,ny
    j1=j_offset(npy)+j-1
    do i=1,10
      a(i,j,k)=uat(i,j1,k)
    enddo
```

```
    enddo
enddo
```

图 5-34 循环不变量外提示例二

5.6.3.4 利用向量化提示信息修改程序

通过选项-LNO:simd_report=2 可以输出向量化成功和失败时的提示信息，可利用向量化失败时的提示信息对程序进行相应修改。

向量化失败时的提示信息主要有：

表格 5-5 向量化失败提示信息列表

提示信息	说明
Loop has calls or Gotos.	循环内有函数调用或 goto。
Loop is not innermost.	不是最内层循环（目前只支持最内层循环）。
Non-contiguous array x reference exists.	对数组 x 的引用不连续。
Index variale lives at the exit of OpenMP loop.	循环索引变量在 OpenMP 的 lastprivate。
Loop upper bound can not be std.	循环上界太复杂，不能标准化。
Loop has inline assembly.	循环内有内嵌汇编。
Loop upper bound too complicated.	循环上界太复杂。
Vectorization is not likely to be beneficial.	循环向量化可能无收益。
Loop has dependencies.	循环有依赖。
Too few iterations.	跌代数太少。

Loop has 0 vectorizable ops.	循环内没有向量操作。
Loop has to be split.	循环需要进行分布。
Loop has to be scalar-expanded.	循环需要做标量扩展。

用户可依据表格 5-5 信息确认导致向量化失败的原因报告，通过修改源代码消除影响向量化的语句，并再次提交自动向量化系统进行编译。

第6章　高性能扩展数学库

“神威•太湖之光”计算机系统高性能扩展数学库 xMath（众核版和片上多核版）是一套基于“申威 26010”异构众核处理器，具有单核组众核并行化和片上多核并行化特征的众核扩展数学库，该数学库由**中国科学院软件研究所并行软件与计算科学实验室**针对“申威 26010”异构众核处理器研究开发。

6.1 模块说明

“神威•太湖之光”计算机系统 xMath 扩展数学库包括以下 5 个子模块：

- BLAS
- LAPACK3.5.0
- FFT 信号处理子程序;
- 稀疏线性系统求解子程序包
- ScaLAPACK2.0.2

其中 ScaLAPACK 模块本软件未做改动，请参考其用户手册，其余 4 个模块的函数的详细功能和使用方法参见《“神威•太湖之光”计算机系统 xMath 用户手册》。

6.2 使用方法

6.2.1 编译链接

xMath 扩展数学库需要使用 sw5f90 进行链接，由于内部使用了从核函数，所以需要加上-hybrid 参数。

1) 编译：sw5cc -host -o test.o test.c
2) 路径：/usr/sw-mpp/swcc/sw5gcc-binary/lib
3) 链接：sw5f90 -hybrid -o test test.o -lxMath_multicore -lxMath_manycore

6.2.2 运行方法

xMath 扩展数学库需使用全部从核资源,提交任务时需要指定满从核数(64);另外，本数学库内部通过从核函数的局部变量使用从核 LDM，所以需要添加-b 参数，否则性能会极差。作业提交命令行示例:

```
bsub -I -b -N 1 -q q_sw_expr -cgsp 64  ./test
```

由于本数学库使用了大量从核 LDM，所以有可能会覆盖用户编写的从核函数中使用的 LDM 静态变量(__thread_local 声明的变量)。如果出现此种情况，需要设置环境变量 XMATH_PRESERVE_LDM=1，此时该数学库会对 LDM 使用前和使用后分别进行保存和恢复操作。示例如下:

```
export XMATH_PRESERVE_LDM=1
bsub -I -b -N 1 -q q_sw_expr -cgsp 64 ./test
```

6.3 BLAS 模块

BLAS(Basic Linear Algebra Subprograms)是一个线性代数核心子程序的集合，主要包括向量和矩阵的基本操作。它是最基本和最重要的数学库之一，广泛应用于计算科学，特别是在科学建模中。

BLAS 分成三个等级：

- 1 级：进行向量-向量操作；
- 2 级：进行向量-矩阵操作；
- 3 级：进行矩阵-矩阵操作。

6.3.1 BLAS Level 1 函数列表

表格 6-1 BLAS Level 1 函数列表

子程序组	数据类型	描述
?asum	s、d、sc、dz	向量元素求和
?axpy	s、d、c、z	标量-向量乘
?copy	s、d、c、z	向量复制

?dot	s、d	点积
?sdot	sd、d	扩展精度点积
?dotc	c、z	共轭点积
?dotu	c、z	非共轭点积
?nrm2	s、d、sc、dz	欧几里得范数
?rot	s、d、cs、zd	Plane 变换
?rotg	s、d、c、z	Givens 变换
?rotm	s、d	修改的 plane 变换
?rotmg	s、d	修改的 Givens 变换
?scal	s、d、c、z、cs、zd	向量-标量乘
?swap	s、d、c、z	向量-向量交换
i?amax	s、d、c、z	向量最大绝对值的下标
i?amin	s、d、c、z	向量最小绝对值的下标

6.3.2 BLAS Level 2 函数列表

表格 6-2 BLAS Level 2 函数列表

子程序组	数据类型	描述
?gbmv	s、d、c、z	带状矩阵-向量积
?gemv	s、d、c、z	矩阵-向量积
?ger	s、d	秩 1 更新
?gerc	c、z	共轭矩阵秩 1 更新
?geru	c、z	非共轭矩阵秩 1 更新
?hbmv	c、z	Hermitian 带状矩阵-向量积
?hemv	c、z	Hermitian 矩阵-向量积
?her	c、z	Hermitian 矩阵秩 1 更新
?her2	c、z	Hermitian 矩阵秩 2 更新
?hpmv	c、z	Hermitian packed 矩阵-向量积
?hpr	c、z	Hermitian packed 矩阵秩 1 更新
?hpr2	c、z	Hermitian packed 矩阵秩 2 更新
?sbmv	s、d	对称带状矩阵-向量积
?spmv	s、d	对称 packed 带状矩阵-向量积
?spr	s、d	对称 packed 矩阵秩 1 更新

?spr2	s、d	对称 packed 矩阵秩 2 更新
?symv	s、d	对称矩阵-向量积
?syr	s、d	对称矩阵秩 1 更新
?syr2	s、d	对称矩阵秩 2 更新
?tbmv	s、d、c、z	三角带状矩阵-向量积
?tbsv	s、d、c、z	系数矩阵为三角带状矩阵的线性方程组求解
?tpmv	s、d、c、z	三角 packed 矩阵-向量积
?tpsv	s、d、c、z	系数矩阵为三角 packed 矩阵的线性方程组求解
?trmv	s、d、c、z	三角矩阵-向量积
?trsv	s、d、c、z	系数矩阵为三角矩阵的线性方程组求解

6.3.3 BLAS Level 3 函数列表

表格 6-3 BLAS Level 3 函数列表

子程序组	数据类型	描述
?gemm	s、d、c、z	矩阵-矩阵乘
?hemm	c、z	Hermitian 矩阵-矩阵乘
?herk	c、z	Hermitian 矩阵秩 k 更新
?her2k	c、z	Hermitian 矩阵秩 2k 更新
?symm	s、d、c、z	对称矩阵-矩阵乘
?syrk	s、d、c、z	三角矩阵秩 k 更新
?syr2k	s、d、c、z	三角矩阵秩 2k 更新
?trmm	s、d、c、z	三角矩阵-矩阵乘
?trsm	s、d、c、z	三角矩阵的线性矩阵-矩阵方程组求解

6.4 LAPACK 模块

LAPACK 是 Linear Algebra PACKagre 字母的缩写。LAPACK 能够解决，如线性方程组求解，线性最小二乘问题求解，特征值和奇异值等问题。LAPACK 还可以实现矩阵分解和条件数的估计等相关计算。

LAPACK 包含驱动程序(driver)、计算程序(computational)、辅助程序(auxiliary)。驱动程序用于解决一个完整问题。计算程序用于执行一个单独的计算任务。辅助程序用于执行分块算法的子任务、完成低水平计算。

LAPACK 提供了计算稠密矩阵和带状矩阵的功能，但是没有提供计算稀疏矩阵的功能。对实数型和复数型数据，LAPACK 提供相同的计算功能。

LAPACK 的具体使用方法详见《xMath 扩展数学库用户手册》。

6.5 FFT 模块

xMath 库的 FFT 模块，主要提供给用户一组快速进行离散傅里叶变换(DFT)的接口，用户可以调用他们进行所需要的信号处理的变换。

表格 6-4 xMath FFT 支持的功能

变换维度	1D、2D、3D
变换类型	复数到复数(C2C)
变换规模	任意大于 1 的正整数(2 的幂，非 2 的幂)
带跨步的输入、输出	1D，2D，3D 计算都支持
批量计算	仅 1D 计算支持

xMath FFT 的具体使用方法详见《xMath 扩展数学库用户手册》。

6.6 性能特点

对使用了__thread_local 关键字在 LDM 中保存数据的程序，xMath 提供 LDM 现场保护功能(默认关闭)。用户能够通过设置环境变量 xMath_PRESERVE_LDM=1 或 WEXML_PRESERVE_LDM=1 来开启该功能。但因为这项功能的实现需要在内存和

LDM 之间来回传输多余数据，因此会对 xMath 库的性能产生不利影响，因此建议若客户程序中没有使用__thread_local 关键字，则可确保该功能一直处于关闭状态。xMath 库中函数的实际性能受输入的影响较大，在接下来的各个小节中，将说明 xMath 中各个模块在何种输入条件下能够达到相对较优的性能。

6.6.1 BLAS 模块

1） 从核版

BLAS 模块三级函数性能均显著受到计算规模的影响，一般情况下，计算规模越大，性能越优。其中，Level 1 和 Level 2 级函数具有访存受限的特点，本库在此两级函数中进行了线程数自动调优的机制充分利用访存带宽，使其在一般情况下也有较优性能，因此不需要用户进行其它设置。另一方面， Level 3 级函数具有计算密集型的特点，通过计算与访存重叠的方式进行了优化，用户在调用时矩阵首地址为 32Bytes 对齐，且矩阵维度 M 维为 128 的倍数，N 维为 256 的倍数，K 维为 512 的倍数时性能较优。

2） 主核版

主核版仅包含主核函数，最大线程数为 4，用户调用时需通过标准的 OpenMP 环境变量(OMP_NUM_THREADS)进行设置。主核版 BLAS 显著受到线程数以及计算规模的影响，一般情况下，线程数越大、矩阵规模越大，性能越高；另外由于此版本函数仅包含主核函数，矩阵地址为 32Bytes 字节带来的性能提升可能不大。

6.6.2 LAPACK 模块

1） 从核版

LAPACK 模块的性能显著地受到计算规模的影响，因此在矩阵规模为 2048（经验值)之下时，并不建议使用主核版 LAPACK。另外，矩阵首地址是否为 32 Byte 对齐也对性能的高低有一定影响，因此建议用户在产生矩阵时，使用类似 memalign 函数，产生首地址为 32 Byte 整数倍的存储空间。在从核 LAPACK 库中，用户不需要自行调整矩阵的分块大小和配置每个 LAPACK 函数使用的从核数，LAPACK 库有线程数自动适应功能，确保在一般情况下能产生较优的性能。

2） 主核版

与从核情况类似，用户在矩阵规模为 1024（经验值）之下时，不应将线程数量设置过多。主核 LAPACK 的线程使用数量可以使用标准的 OpenMP 环境变量(OMP_NUM_THREADS)进行设置。另外矩阵首地址是否为 32 Byte 对主核 LAPACK 的 性能并无显著影响。用户也不需要设置分块大小，各个函数的最优分块性能业已配置并固化，确保在一般情况下的较优性能。

6.6.3 FFT 模块

1） 从核版

xMath 从核版 FFT 使用访存计算重叠的双缓存技术，性能与输入数据规模相关。对于一维 FFT，数据规模小于 512 时，计算量较少，由主核运行，大于 512 时，调用 8 或 64 个从核运行，数据规模越大，性能越高，例如数据规模为 131072 时，性能较优。对于多维 FFT 计算，行数较大时，性能较优。同时，性能与数据首地址 32Byte 对齐息息相关，建议用户使用类似 memalign 函数，以产生首地址为 32Byte 整数倍的存储空间。

2） 主核版

xMath 主核版 FFT 性能同样与输入数据规模相关，数据规模较大时，性能较高，例如，数据规模为 131072 时，性能较优，当数据规模较小时（1024），不宜将线程数量设置过多。

第7章　通信优化

在并行程序设计过程中，通信的效率直接影响并行软件的计算效率和加速比，直接影响并行程序的可扩展，特别是在万核以上大规模并行环境下，通信的优化就显得更为重要。根据“神威•太湖之光”计算机系统的结构特点，本章主要介绍超节点内和超节点间两种类型的通信优化。

7.1 超节点内通信优化

7.1.1 超节点内冲突的原因

“神威•太湖之光”计算机系统超节点内理论上说是全连接，任意点到任意点都有满带宽。但是也存在两种类型的冲突：

1) 同一节点内不同核组之间的通信竞争；
2) 超节点内的 256 个节点被分为 16 组，组内的 16 个节点在同一路由器上，任意通信都无冲突，但组间的通信需要经过路由器中转，如果通信模式不合适，仍然存在冲突可能。

7.1.2 节点内通信冲突解决

系统实际网络带宽是有限的，理论上如果通信量非常巨大，在程序运行过程中通信将占据大部分的时间，这样节点内的通信冲突将无法完全解决。假如应用的模式是有序的先做一段计算，再进行一段通信，而且通信模式不是阻塞的全局通信（隐含着同步），那么就可对并行程序的进程映射进行适当调整来减轻通信冲突的影响。

以 MPI 并行程序为例，默认配置下进程的分布以节点为单位，同一节点的四个主核分配到的 MPI 进程号是连续的。而一般应用程序的邻近进程在同一时刻任务也是相似的，就会出现同时计算，同时通信的情况。这样在程序通信时，网络带宽将被 4 个核组平分，而计算的时候，网络空闲完全浪费。

假如并行算法中相邻位置的进程通信存在时间差（如 HPL，通信以流水方式向后扩散），就可以把同一节点的不同核组映射到 MPI 节点模板中距离较远的位置，尽可能使每个进程通信时其他进程都在计算。

7.1.3 不同路由器间的通信冲突避免

超节点内不同路由器间的通信，选择按目标节点号模 16 的路由器进行。因此当多个进程向模 16 相同的目标节点通信时就会有冲突发生。例如，CPU0~15 向 CPU(0~15)*16 发消息，则 16 个消息都经过路由器 0，带宽会降到 1/16。

最理想的情况是任意 CPU 与相等距离的 CPU 通信，此时带宽为满带宽。建议用户设计通信算法与节点映射算法时考虑通信进程的相互位置。

7.2 超节点间通信优化

“神威•太湖之光”计算机系统超节点间存在 1/4 网络带宽裁剪，超节点间的路由规则是按照目标处理器编号模 64,选择 64 条通信链路中的一条进行通信。

基于超节点间的路由规则，一种较好的物理映射方法是将一个超节点的连续 64 个核组作为一列，依次排列，且将同一处理器的不同核组间隔 3 列，得到最优的排列方式，如图 7-1 为最优物理映射的排列方式。

该排列方式保证了超节点内和超节点间网路带宽的充分利用：

1) 同一处理器的 4 个核组映射于间隔的不同列上，由于处在节点模板的不同位置，同时通信的可能性比较低，保证了在每个处理器不同核组上运行的进程出处理器的网络带宽不被裁剪；

2) 对于行方向通信，超节点间只会有 64 个核组同时进行通信并且目标处理器号模 64 不等。根据超节点间的路由规则，通信无冲突地使用了 64 条链路，也相当于网络带宽不裁剪；超节点内连续 16 个源处理器的目标处理器也是连续的，且模 16 不等，自然满足超节点内的路由规则。当用户课题以边界通信为主时，每个超节点与其他超节点同时通信的数量为 64，避免了超节点间通信网络带宽裁剪带来的损失。

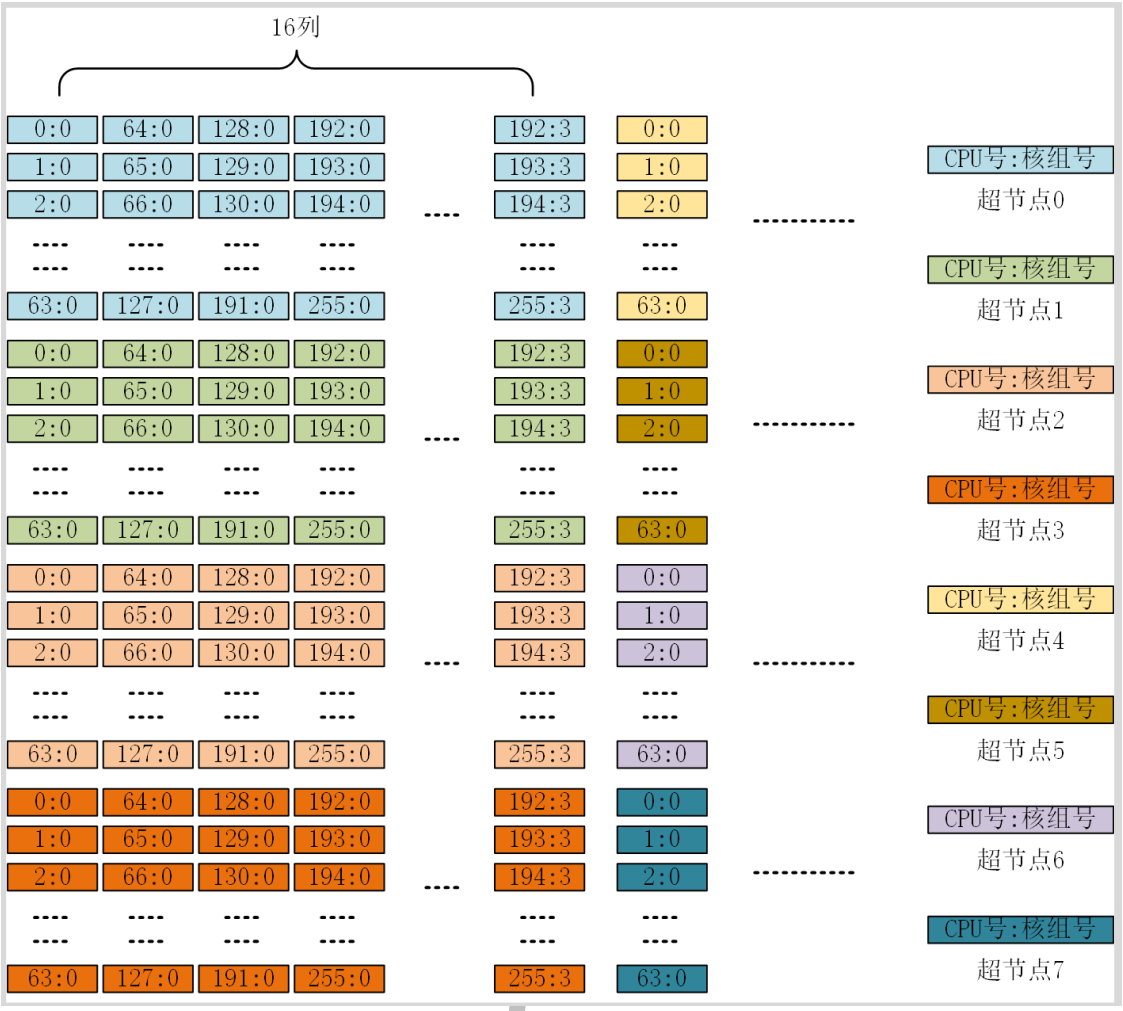


图 7-1 最优物理映射的排列方式

7.3 通信隐藏优化

科学与工程计算一般应用课题的运行模式为：计算——通信——计算。在传统多核结构下，计算任务分配给每个核，因此在计算时难以完全隐藏通信开销。在“申威 26010”异构众核架构下，处理器间消息的收发由主核完成，主要的计算部分由从核完成。当消息传递需要的时间比计算短时，通信时间可以完全被计算时间所隐藏，显著地提高了并行课题的运行效率。

对于计算与通信之间存在相关性的并行课题，我们提出了基于流水的通信隐藏算法，如图 7-2 所示。

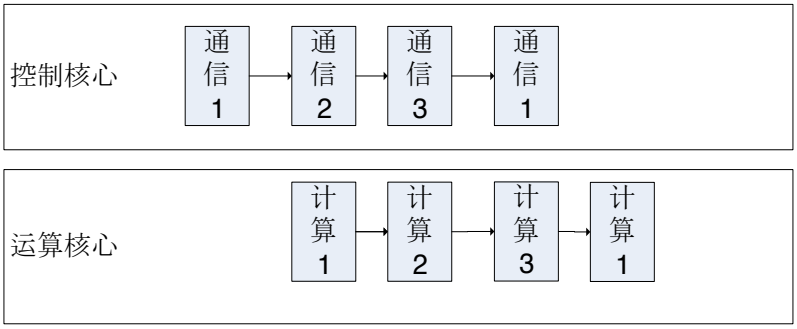


图 7-2 通信隐藏算法

主要方法是：把完整的通信分为几部分，除了最开始的一段通信必须串行执行外，其余的所有通信都可以与计算并行（在某些多核系统的 MPI 非阻塞通信的实现也采用了该种方式）。

本算法应用于整机 HPL 测试，取得很好效果，算法中行交换与矩阵乘有先后关系，必须完成行交换后才能计算。通信隐藏典型代码如图 7-3 所示。

```
……
HPL_pdlaswp (PBCST, &test, PANEL, nn ); // 执行 nn 宽度的行交换
    ……                                  // 计算函数
athread_spawn64(slave_control_curr,0);  // 计算，当完成 nn 宽度的计
                                           算后等待余下的行交换完成
HPL_pdlaswp(PBCST, &test, PANEL, nextnn); // 执行余下的行交换
athread_join64();                       // 主核等待从核计算完成
……
```

图 7-3 通信隐藏典型代码

7.4 资源池的用法

在上节对应用软件通信的分析可以看出，在大规模系统下，由于通信冲突的存在，为了提高通信有效带宽往往需要完整的超节点结构。

“资源池”的概念和实现方法是“神威”系列计算机系统的重要创新之一，当系统计算能力达到一定规模时，研制者们假设超节点内会出现个别失效节点而不能及时更换的情况，为了系统的可用性和可维性考虑，在“神威”系列计算机系统的方案设计和实现上增加了资源池概念。资源池本身是一种特殊的超节点，它的作用就是当某些超节点内的某些节点异常时，作业管理会从资源池中分配相

应的节点替换那些异常节点，并保证在该超节点内部新组成的 256 个处理器在网络连接上还是全交换的。

为保证资源池的使用效率，资源池的使用有以下要求：

7.4.1 数量限制

为保证网络通信带宽，每个超节点从每个资源池超节点至多获取两个节点；并建议选取资源池中满配置的节点。

如果超节点缺少的节点超过 2 个（按满配节点计算，即缺少的核组超过 8 个），建议从每个资源池超节点各取 2 个节点，直到超节点达到满配。

7.4.2 位置限制

从资源池取到的节点与超节点内的节点之间的通信可以视为超节点内的通信，路由算法与超节点内相同。因此建议从资源池内取得的节点应与原超节点空缺的节点位置相同或模 16 相等。

从资源池取得的节点也应放在空缺的节点的相同位置。用户程序运行时，可以自行指定节点列表，列表文件每一行指定一个全局核组编号号，当超节点内部分核组或节点不可用时，在节点列表中直接以资源池的核组号代替。

资源池的数量与节点列表指定方式与系统实现有关，可参考《神威操作系统环境手册》。

第8章　性能分析

8.1 核心段分析

当进行程序代码优化时，首先要找到程序代码运行过程中占主要运行时间的核心代码段，进而对核心代码段进行分析与优化，最后达到代码优化的目的。可以通过如下步骤获得程序的核心代码段。

1） 提交作业时加：--sw3runarg="-p -f" 命令行选项；

2） 作业运行结束后生成： gmon.out 文件；

3） 在 gmon.out 目录下执行：gprof 可执行程序 gmon.out

通过上述三个步骤，在 gmon.out 目录下，可以看到主从核的性能分析数据，从核函数带 slave 前缀。具体如图 8-1 所示：

```
  %   cumulative  self          self     total
time   seconds  seconds calls Ts/call Ts/call  name
98.15  12.75    12.75                         slave_Array_Waiting_For_Task
0.77   12.85    0.10                          _I/O_vfscanf_internal
0.38   12.90    0.05                          athread_halt
0.31   12.94    0.04                          slave_BARFIT
0.23   12.97    0.03                          ____strtod_l_internal
0.08   12.98    0.01                          __mpn_impn_sqr_n_basecase
0.08   12.99    0.01                          slave_bipol
……
```

图 8-1 主从核代码段分析数据

8.2 拍数统计方法

为了更准确分析程序的运行时间，可以先统计出某一程序段运行时的拍数，然后把拍数乘以处理器主频，就可以准确得到该段程序运行时的墙钟时间。

图 8-2、图 8-3、图 8-4 和图 8-5 分布为主核拍数统计代码、从核拍数统代码、从核号获取代码和从核阵列同步代码，在分析优化代码时，可以直接使用这些代码对程序进行分析。

```
void rpcc_(unsigned long *counter)
{
        unsigned long rpcc;
        asm("rtc %0": "=r" (rpcc) : );
        *counter=rpcc;
}
```

图 8-2 主核拍数统计代码

```
void rtc_(unsigned long *counter)
{
        unsigned long rpcc;
        asm volatile("rcsr  %0, 4":"=r"(rpcc));
        *counter=rpcc;
}
```

图 8-3 从核拍数统计代码

```
void get_myid_(int *core_id)
{
        int row,col;
        asm volatile("rcsr   %0, 1" : "=r"(row));
        asm volatile("rcsr   %0, 2" : "=r"(col));
        *core_id=row*8+col+1;
}
```

图 8-4 从核号的获取代码

```
void sync_array_()
{
        int128 sync_tmp;
        asm volatile(                \
            "ldi    %0, 0xff\n"      \
            "sync   %0\n"            \
            "synr   %0\n"            \
            :                        \
            :"r"(sync_tmp):"memory");
}
```

图 8-5 从核阵列同步代码

8.3 主从核性能计数器

本节提供两种性能评测方法用于程序各种运行性能的统计，一种是使用作业提交方式直接得到程序整体运行情况，另一种是利用主从核性能计数器接口，通过代码插桩方式进行程序各部分的精细统计。

8.3.1 通过作业提交选项统计程序整体性能

bsub 提交作业时指定-perf 选项，即可使用默认的配置文件启动性能计数功能；性能测试结果存放于“/home/export/online1/.perfpmu/作业号”目录下，其中作业号即为作业运行时的作业号，目录下对每一个任务产生一个结果文件。运行 Jperf 作业号即可得到程序性能信息。

例如：运行程序 bsub -I -b -perf -q q_sw_expr -n 1 -cgsp 64 -share_size 4096 -host_stack 512 ./sitest 1，如果返回的作业号是 4534672，则运行 Jperf 4534672，性能分析结果如图 8-6 所示：

```
np_gc_count_0(0x1 DMA_GET):640384
np_gc_count_1(0x2 MEM_ACCESS):235932515
np_l2cache_count_0(0x3 L2ICACHE_ACCESS):380471696
np_l2cache_count_1(0x4 L2ICACHE_MISS):30096
Performance counter stats for job( ./sitest ):
 @@--Job----Run Cycles is                                : 14122274773
  --Host---average cycles occur per     ICACHE_ACCESS : 3124.75
  --Host---average cycles occur per   DTB_Single_Miss : 12826770.91
  --Slave--average cycles occur per         Float_Ops : 6.86
  --Slave--average cycles occur per             CYCLE : 1.04
  --CG-----average cycles occur per           DMA_GET : 22052.82
  --CG-----average cycles occur per        MEM_ACCESS : 59.86
  --CG-----average cycles occur per   L2ICACHE_ACCESS : 37.12
  --CG-----average cycles occur per     L2ICACHE_MISS : 469240.92
```

图 8-6 通过作业提交选项统计的程序性能分析结果

8.3.2 主从核性能计数器接口

使用主从核性能计数器函数接口之前必须先调用其初始化函数。注意：

1) 所有初始化函数接口只能在主核程序中调用，从核性能分析的统计函数接口在从核调用，其余均在主核调用；
2) C 语言和 FORTRAN 语言编译时链接“-lswperf” 即可；
3) C++语言在程序的调用处用 extern “C” 添加需要调用的接口，编译时链接 “-lswperf”。

下面具体介绍常用的性能计数统计函数接口。

8.3.2.1 从核 DMA 次数统计

初始化函数：void penv_slave2_dma_init(); 该函数设置从核发起的 DMA 请求计数事件，并初始化；

统计函数：void penv_slave2_dma_count(long *ic)；其中 ic 是保存当前发起 DMA 请求次数，使用的参数是地址形式，该函数只能在从核调用。

如果需要统计一个核心段或者一个核心函数发起 DMA 请求总次数，在核心段或核心函数前后调用该接口，分别得到 ic0 和 ic1,程序通过（ic1-ic0）即可得到该核心段或核心函数发起的 DMA 请求次数。

8.3.2.2 从核 gld 次数统计

初始化函数：void penv_slave2_gld_init()；该函数设置从核 gld 请求计数事件，并初始化。

统计函数：void penv_slave2_gld_count(long *ic)；其中 ic 是保存当前 gld 请求次数，使用的参数是地址形式，该函数只能从核函数调用。

如果需要统计一个核心段或者一个核心函数 gld 请求次数，在核心段或核心函数前后调用该接口,分别得到 ic0 和 ic1,程序通过(ic1-ic0)即可得到该核心段或核心函数发起的 gld 请求次数。

8.3.2.3 从核 IPC 统计

初始化函数：void penv_slave_ipc_init()；该函数设置一些从核计数器的计数事件，并初始化。

统计函数：void penv_slave_ipc (long *ic,long *tc)；其中 ic 是保存当前已运行指令的条数，tc 是保存当前时钟周期数，参数采用地址方式，该函数只能从核函数调用。

如果需要统计一个核心段或者一个核心函数的 IPC，在核心段或核心函数前后调用该接口，分别得到(ic0,tc0)和(ic1,tc1)，程序通过((double)(ic1-ic0))/((double)(tc1-tc0))即可得到该核心段或核心函数的 ipc 信息。

8.3.2.4 从核浮点操作数统计

初始化函数：void penv_slave0_float_ops_init()；该函数设置从核浮点加减乘、乘加类指令等效操作的计数事件，并初始化。

统计函数：void penv_slave_float_ipc_count (long *ic)；其中 ic 保存当前浮点加减乘、乘加类指令等效操作的次数，参数采用地址方式，该函数只能从核函数调用。

如果需要统计一个核心段或者核心函数浮点加减乘、乘加类指令等效操作的次数，在核心段或核心函数前后调用该接口，分别得到 ic0 和 ic1，程序通过(ic1-ic0)即可得到该核心段或核心函数的浮点加减乘、乘加类指令等效操作的次数。

8.3.2.5 从核浮点 IPC 统计

初始化函数：void penv_slave_float_ipc_init ()；该函数设置一些从核计数器的计数事件，并初始化。

统计函数：void penv_slave_float_ipc_count (long *float_ic,long *tc)；其中 float_ic 是保存当前已运行浮点指令如浮点加减乘、乘加类指令等效操作

指令计数（每个标量加减乘+1，每个标量乘加+2，每个向量加减乘+4，每个向量乘加+8），tc 是保存当前时钟周期数，参数采用地址方式，该函数只能从核函数调用。

如果需要统计一个核心段或者一个核心函数的浮点 IPC，在核心段或核心函数前后调用该接口，分别得到(ic0,tc0)和(ic1,tc1)，程序通过((double)(ic1-ic0))/((double)(tc1-tc0))即可得到该核心段或核心函数每拍多少个浮点操作数，该值理想值为8。

8.3.2.6 从核指令脱靶次数统计

初始化函数：void penv_slave1_l1icache_miss_init()；该函数设置从核 L1 ICACHE 脱靶次数的计数事件，并初始化。

统计函数：void penv_slave1_l1icache_miss_count(long *ic)；其中 ic 是保存当前 L1 ICACHE 脱靶次数，参数采用地址方式，该函数只能从核函数调用。

如果需要统计一个核心段或者一个核心函数 L1 ICACHE 脱靶次数，在核心段或核心函数前后调用该接口，分别得到 ic0 和 ic1，程序通过(ic1-ic0)即可得到该核心段或核心函数的 L1 ICACHE 脱靶次数。

8.4 GDB 用法

“神威•太湖之光”计算机系统程序调试采用 swgdb 工具，该工具支持在用户的登入服务器上直接调试正在运行的作业。

8.4.1 环境条件

1) 在编译可执行程序时，必须加-g 编译选项；
2) 为让程序出现段违例时不退出，保留调试现场，需要加如下环境变量：
   export MV2_HANG_WHEN_ERROR=1

3） 若程序出错现场太快，可以在程序出错前的代码中设置死循环，使用 gdb 改变循环条件，继续调试；采用死循环调试代码及步骤如图 8-7 所示，该例子是 FORTRAN 程序。

实现死循环的代码：

```
j=1              !  申请全局变量 j
do while(j==1)
   i=1
enddo
```

生成的可执行程序为 a.out，调试过程：

```
$ gdb a.out
(gdb) run
(gdb) ^C              // 中断程序执行 Control + C
(gdb) p  j            // 打印变量 j 的值
    $1 = 1
(gdb) p j = 0         // 修改变量 j 的值
(gdb) p j
     $2 = 0
(gdb) c               // 继续执行后续代码
```

图 8-7 死循环调试代码及步骤

8.4.2 调试步骤

1） 在编译可执行程序时，必须加-g 编译选项；

2） 设置环境变量：export MV2_HANG_WHEN_ERROR=1

3） 进入可执行程序目录；

4） 生成符号表文件：xobjdump -d ./a.out 2>&- 1>&-

5） 提交作业时加 -debug 选项：

bsub -I -debug -p -q q_sw_expr -n 16 -cgsp 64 ./a.out

通过 bjos 查询作业信息，记住作业号和运行节点编号。如作业号为：6527895，运行节点为：40152-40155

6） 启动作业调试服务（每次提交作业都需要运行一次，因为作业退出后自动停止），格式：bdebug 作业号

bdebug 6527895

7） 执行调试命令：swgdb 可执行程序 节点号

swgdb a.out 40152

8） 在 swgdb 提示符下输入相应命令，这些命令与 GDB 命令相同：

a） attach 进程号：选择被调试的进程

b） detach 进程号：结束当前进程的调试

c） where： 列出函数调用关系

注：由于从核 gdb 调试比较复杂，建议采用原始的打印方式进行调试，该调试工具不支持。

8.5 局存使用情况统计工具

程序名称：ldmreport

功能说明：检查“神威•太湖之光”计算机系统可执行程序中所有使用__thread_local ，__thread_local_fix，__thread_kernel(“kernel_name”)关键字申明的局存私有变量的名称、地址和大小。

运行格式：ldmreport 可执行程序

输出信息：程序中所有局存私有变量、可重用局存私有变量的名称、地址和大小；LDM 变量总的对界属性和总大小。

其他说明：

1） ldmreport 依赖程序中的符号信息,可执行程序正常编译链接生成后，不要进行 strip 操作，否则无法获取局存使用信息；

2） 如果程序总的静态 LDM 变量空间超过 64K，会打印报警信息。这里静态 LDM 变量空间包括以下几个部分：

a） 用户使用 preserve 选项保留的局存栈空间；

b） __thread_local + __thread_local_fix 的静态空间；

c） __thread_kernel 空间中最大的一个。

示例：

```
[pfsxxcl@sn22 blast]$ ldmreport ./blastp.swgccok
====  tdata_local max align: 16B, data num: 10, total size: 10144B
indx:0          addr:0x10       size:8          name:_PC
indx:1          addr:0x18       size:1          name:_PEN
indx:2          addr:0x19       size:1          name:_ROW
indx:3          addr:0x1a       size:1          name:_COL
indx:4          addr:0x1b       size:1          name:_CGN
indx:5          addr:0x1c       size:4          name:_MYID
indx:6          addr:0x20       size:8048       name:s_data_ldm
indx:7          addr:0x1f90     size:2052       name:s_subject_ldm
indx:8          addr:0x2794     size:4          name:rply1
indx:9          addr:0x2798     size:4          name:rply2
```

图 8-8 ldmreport 应用示例

8.6 从核函数栈空间深度统计工具

程序名称：checksp

功能说明：检查“神威•太湖之光”计算机系统可执行程序的从核函数占用栈空间的大小，若不指定特定函数名，则会输出所有 kernel 函数的所需栈空间大小。

运行格式：checksp 可执行程序［函数名］

输出信息：

```
……
BeCalledFunc(selfstack_size,childstack_size) [total_size]
  Funcname(selfstack_size,childstack_size)  [total_size]
```

函数的调用层次关系由缩进表示，被调用函数比原函数向前缩进两个字符。

- selfstack_size：函数自身所需栈空间大小。
- childstack_size：子函数所需栈空间大小。
- total_size：函数实际所需栈空间大小，为前两者的和。
- UNKNOWN：无法判断该函数栈空间大小。

其他说明：

以下几种函数无法判断栈空间大小（输出 UNKNOWN）：

1） 函数指针；

2） 递归函数。

对于一些常用的库函数，由于以上两种原因无法计算栈空间大小，我们给出了初步的预测估计值，如 printf 函数，给出一个估计的栈空间大小 1088，这个值可以根据实际情况调整。

示例：

对于图 8-9 所示的函数调用关系。

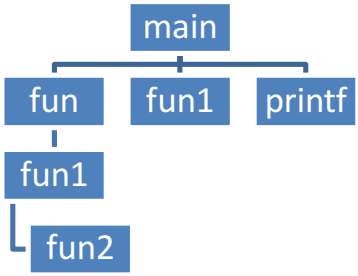


图 8-9 checksp 应用示例

使用 checksp 的输出格式如图 8-10 所示。

```
   slave_fun2(16,0) [16]
   slave_fun1(16,16) [32]
   slave_fun(16,32) [48]
   slave_fun1(32) computed    // fun1 已被计算过，不再重复计算
   slave_printf(1088) estimated // 估计函数的栈空间大小
slave_main(32,1088) [1120] // 最高层函数的栈空间情况，高亮黄色
```

图 8-10 checksp 输出格式示例

对于函数指针，将输出 func pointer(UNKNOWN)。

8.7 函数长度统计工具

程序名称： checklen

功能说明： 检查“神威•太湖之光”计算机系统可执行程序中某指定函数的起始 PC 和代码长度。

运行格式： checklen 可执行程序 函数名称

输出信息： 程序中函数 func_name 的起始 PC 值和代码长度。

第9章　工程实例

9.1 航天飞行器跨流域数值模拟统一算法并行软件

9.1.1 概述

航天飞行器从外层空间再入大气层（距地面 300 公里到 20 公里），先后经历高稀薄自由分子流区、过渡流区、滑移流区和连续流区，所遇气动环境十分复杂。如何准确可靠地求解飞行器跨越飞行各流域复杂空气动力学问题，一直是对流体力学研究工作者的挑战，迄今未有成熟可靠的研究方法。

国家计算流体力学实验室李志辉老师团队通过研究确立描述各流域微观分子输运现象的 Boltzmann 简化速度分布函数方程，发展了基于微观分子概率统计原理的气体运动论离散速度坐标技术，并将其应用于气体分子速度空间，利用一套在物理空间和时间上连续而速度空间离散的分布函数来代替原分布函数对速度空间的连续依赖性，将各流域统一的气体分子速度分布函数方程化为在各个离散速度坐标点处基于时间和位置空间具有非线性源项的双曲型守恒方程。应用拓展了计算流体力学有限差分方法，借助非定常时间分裂技术和 NND 格式，发展直接求解气体分子速度分布函数的气体运动论耦合迭代数值格式。研究发展了可有效模拟高低不同马赫数绕流问题的气体运动论离散速度数值积分方法，用于对速度分布函数进行宏观取矩，确定物理空间各点处的宏观流动参数，由此建立从稀薄流到连续流各流域飞行器绕流问题气体运动论算法统一算法。

由于开展求解 Boltzmann 速度分布函数方程的差分统一算法研究需要在位置空间与速度空间组成的六维相空间进行，必须依靠现代超大规模并行计算机系统。如果开展真实飞行器再入大气层各流域高超音速绕流问题较为成熟的统一算法研究预计需要使用的并行计算内存资源状况将会达到数百 TB 量级、浮点运行速度每秒亿亿次量级的并行计算机资源，随着飞行器来流马赫数的增大，不同高度的飞行器绕流所需要的计算机资源还会大大增加。为了充分发挥“神威•太湖之光”计算机系统的超强计算能力，必须对现有软件系统进行基于 MPI 和众核的

并行化改造与移植，在并行处理技术方面，主要解决了基于速度空间的区域分解策略、并行向量归约、众核并行求解和并行 I/O 四个问题。

9.1.2 基于速度空间的区域分解策略

三维绕流气体运动论数值模拟的计算空间是由离散速度坐标空间和位置空间组成的六维空间，可形成二相空间。算法分成两个主要部分，一部分是在各个离散速度点处基于时间和位置空间求解关于气体分子速度分布函数 f 的具有非线性源项的双曲守恒方程；另一部分是基于速度分布函数 f，使用 Gauss-Hermite 无穷积分法，求解任一时刻 t 位置空间（x,y,z）中的气体密度 ρ、温度 T、流动速度（Ux,Uy,Uz）、应力张量（τxx, τxy, τxz, τyy, τyz, τzz）、热流矢量(qx,qy,qz)等宏观流动参数。虽然问题的整个求解空间是由$(i,j,k,\sigma,\delta,\theta)$组成的六维空间，然而，涉及的变量有些定义在整个六维空间中，有些则定义在子空间（i,j,k）或（σ,δ,θ）中，所以，在用数据并行方法研究这一课题的并行方案时，区域分解方法是研究的重点。

设Ω是三维绕流气体运动论数值模拟的求解空间，Ω_r是位置空间(x,y,z)，Ω_v是离散速度坐标空间（σ,δ,θ），则有：

$$\Omega = \Omega_r \times \Omega_v$$

设处理器数为N_p，将Ω分解为N_p个子空间Ω_i，满足：

$$\Omega = \sum_{i=1}^{N_p} \Omega_i \quad \text{且} \quad \Omega_i \cap \Omega_j = \Phi \qquad (i \neq j)$$

并将子空间Ω_i的数据映射到相应的处理器PE_i中。显然，这样的分解策略有三种。

第一种是位置空间Ω_r的分解策略。按某种方式将Ω_r分解成N_p个子空间Ω_{ri}，使$\Omega = \sum_{i=1}^{N_p} \Omega_{ri}$ 且 $\Omega_{ri} \cap \Omega_{rj} = \Phi \qquad (i \neq j)$，取$\Omega_i = \Omega_{ri} \times \Omega_v$。

第二种是离散速度坐标空间Ω_v分解策略。类似于第一种策略，将Ω_v分解成N_p个子空间Ω_{vi}，取$\Omega_i = \Omega_r \times \Omega_{vi}$。

第三种是混合分解策略，将Ω_r、Ω_v分别按类似于上述方式分解成N_p个子空间Ω_{ri}, Ω_{vi}，取$\Omega_i = \Omega_{ri} \times \Omega_{vi}$。

通过变量依赖关系、数据通信、并行可扩展性三个方面分析，可以得出，当Ω_r空间网格点数并非很大（例如，在百万点以下）采用Ω_v分解策略，当Ω_r空间网格点数很大（例如在千万点以上），应采用Ω_r分解策略。在当前数值模拟的需求和并行机能力限制下，一般情况下Ω_r空间网格点数至多在百万量级，因此采用了Ω_r分解策略。由于在速度空间的三个方向没有数据相关性，不存在数据交换的问题，因此采取三维并行划分，减小空间需求；且在计算过程中对划分后的数据空间采取动态申请与释放，这样不仅解决了计算空间需求问题，还最大限度地达到了计算负载平衡，最大限度的局部计算和尽量少的通信。

9.1.3 主从加速并行

由前面分析可知，该算法基于离散速度空间和位置空间二相空间，离散速度空间与位置空间基本无数据相关性，因此该课题具有天然的多级并行性，该课题在离散速度相空间实现 MPI 并行，因此可以尝试在位置空间挖掘更细粒度的并行性，实现“基于消息传递编程模型的 MPI 进程级并行+基于共享变量编程模型的从核线程级并行”的两级并行方式。考虑到“申威 26010”异构众核处理器局部存储有限，难以存放整个离散位置坐标空间计算过程中所需要的变量，所以可选择针对每个计算核心函数进行循环级的从核并行，如图 9-1 所示。

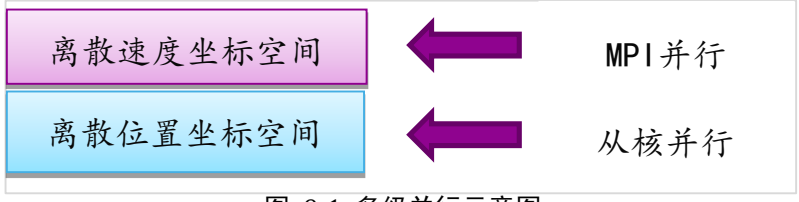


图 9-1 多级并行示意图

对位置空间循环结构重新进行从核编程，采用的并行方案如下：以最外层循环变量来进行从核并行。计算核心执行部分可分为两部分，一部分是计算核心的计算部分，另外一部分则是计算核心发起通信请求并读入或写回数据的部分，即从核数据通信部分（以下简称数据通信部分）。从核并行后整个 3 层循环构成一个单位，称为一个从核并行域。一个从核并行域的简单结构如图 9-2 所示。其中 My_id 是对应的计算核心的逻辑编号(这里逻辑编号从 1 开始),CoreNumber 是用于计算的从核心数。可以看出，该从核并行算法设计方案也适用于其它领域的某些具有多重循环的高性能计算课题。

原程序段	从核并行程序段

do k=1,km
  do j=1,jm
  do j=1,jm
  ... ...
  enddo
  enddo
enddo
	do k=My_id,km, CoreNumber
	  do j=1,jm
	  DMA_Get !从主存读入计算所需数据
	  do i=1,im
	  ... ...
	  enddo
	  DMA_Put !将计算结果写回主存
	  enddo
	enddo

图 9-2 从核并行编程方案示意图

根据上面介绍的从核并行编程方案，如果某程序在单核处理器上计算开销为 T，在具有 CoreNumber 个计算核心的众核处理器上实现并行后计算核心计算部分开销为 T/CoreNumber。如果忽略计算核心初始化开销，计算核心上单轮次数据通信部分开销为P，则计算核心上 N 轮次数据通信开销总计为 P×N，则对于该程序相应的众核加速比计算方式如图 9-3 所示。

$$\text{众核加速比} = \frac{\text{单核处理器计算开销}}{\text{众核并行后总开销}} = \frac{T}{(T/\text{CoreNumber}) + (P \times N)}$$

图 9-3 众核加速比计算公式

由上式可推知，从核心数据通信部分开销越小，得到的加速比越大，加速效果就越理想。因此从核编程优化策略主要是如何减少从核心的数据通信部分开销，并降低其在从核心总开销所占比重。

在本章节也会介绍一些在众核架构下其它的特殊编程优化策略。

9.1.3.1 数据布局优化

数据布局优化主要有两部分内容，首先要将计算任务合理均衡的分配到每个从核心上，即实现从核心并行层次上的细粒度负载平衡，其次是采用一些具体的优化手段减少从核心数据通信部分的开销。

从核心并行层次上的细粒度负载平衡采用基于离散位置空间的区域分解来完成，可采用一维分解、二维分解和三维分解的方式，因二维分解和三维分解均涉及到离散访存，为增加通信数据的长度、减少数据通信次数，可以采用在 K 方向一维分解的方式进行从核心区域分解。

在针对程序的具体优化手段方面，通过对核心段程序变量依赖关系的分析，发现若干变量具有相同的数组形式，且在迭代过程中，这些变量的值不变。因此在优化过程中，将这些数组进行拼接或合并。如在变换坐标 η 方向差分算子的求解程序中，从核心需要读入如下变量：jcb(i,j,k)、ritab(i,j,k)、rix(i,j,k)、riy(i,j,k)、riz(i,j,k)，优化时可以先在迭代过程开始前将这些变量合并为 rixyzbjcb(5,i,j,k)变量，在此后的迭代过程中，计算核心进行数据通信时只需读入 rixyzbjcb 数组即可。这样一来，从核心数据通信次数大大减少，不仅提高带宽利用效率，也缓解了多从核心多次发起通信请求所导致的竞争。实际优化实验结果表明，通过拼接或合并数组能够使数据通信开销大大减小。

9.1.3.2 计算与访存的相互隐藏

对于“申威 26010”异构众核处理器，如何实现最大限度的隐藏从核心对其它核外存储空间的访问延迟是获得良好性能加速的关键。在三维绕流气体运动论数值模拟众核并行算法设计和编程过程中，采用了双缓冲机制，实现计算和通信最大限度的相互隐藏并取得良好加速效果。所谓双缓冲机制，就是在从核心的局部存储空间上申请 2 倍于通信数据大小的存储空间，以便存放两份同样大小且互为对方缓冲的数据。

通信双缓冲通过编程来控制和实现，具体过程如下：除了第一轮次（最后一轮次）读入（写回）数据的通信过程之外，当从核心进行本轮次数据计算的同时，进行下一轮次（上一轮次）读入（写回）数据的通信。此时从核心数据通信部分开销分为两部分，一部分是不可隐藏部分，另外则是可以与计算开销相互隐藏部分。其中不可隐藏部分开销为第一轮次读入与最后一轮次写回的数据通信开销之和，即单轮次通信部分的开销 P。相应的可以与计算开销相互隐藏的数据通信开销为 P ×（N-1），此时计算核心上计算开销仍为 T/CoreNumber。那么众核加速比的计算表达式具有如图 9-4 形式：

$$\text{众核加速比} = \frac{T}{\text{Max}[T/\text{CoreNumber}, P\times(N-1)] + P}$$

图 9-4 通信双缓冲机制众核加速比计算公式

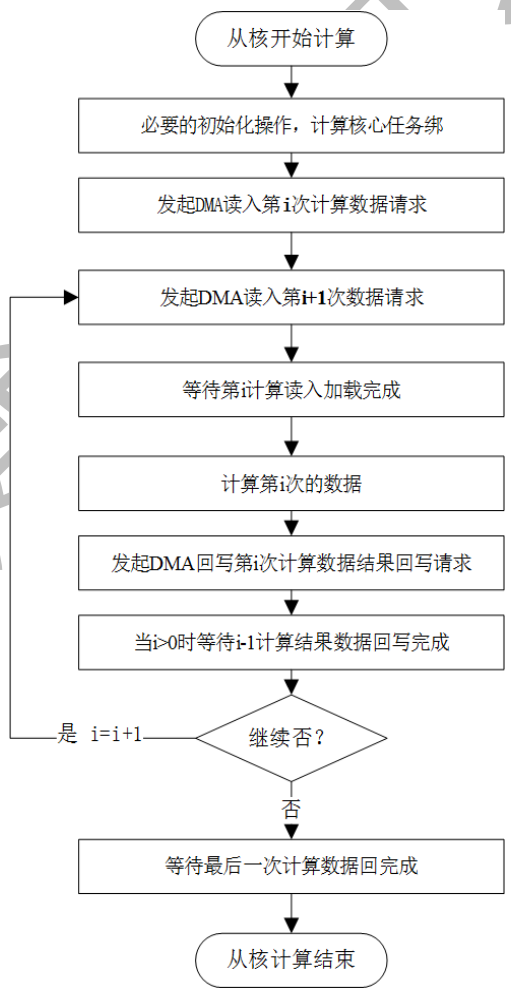


图 9-5 从核心双缓冲机制示意图

上面众核加速比计算表达式表明，从核心计算开销和部分的通信开销实现了隐藏。考虑计算密集型程序的情形，从核心计算开销大于通信开销，而单轮次的通信开销又特别小时，根据上面计算表达式得到众核加速比将会接近CoreNumber。如果将众核加速比与众核具有的从核心个数之比定义为众核并行效率，那么此时众核并行效率也将接近于1，即计算密集型课题更能够充分地发挥众核的优势。

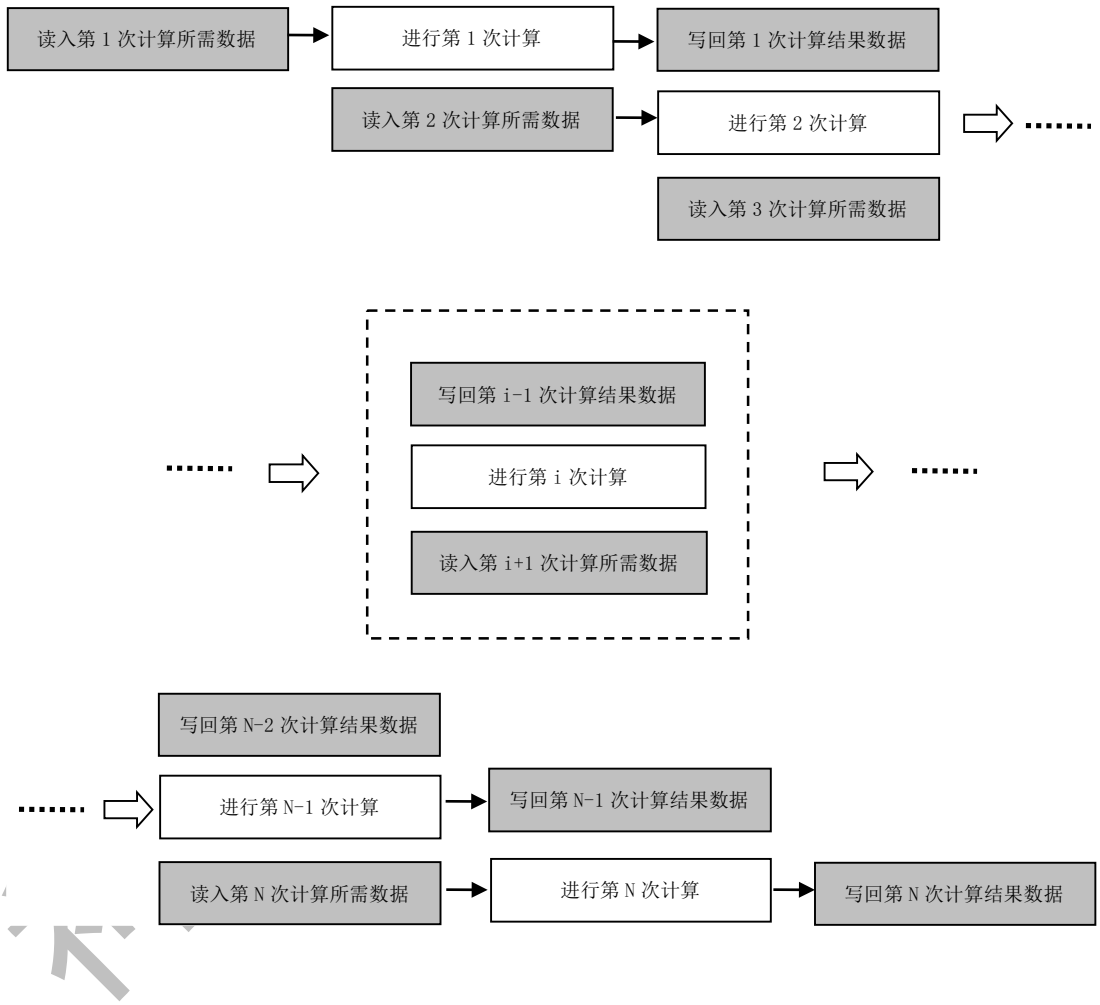


图 9-6　双缓冲机制下通信和计算互相隐藏原理示意图

9.1.3.3 其它优化策略

新的众核架构体系促使产生新的编程策略。从核心专属的局存空间不大，但访问延迟小。因此在针对存在不可避免的大量离散访存的程序进行众核并行算法设计时，可以利用从核的特点来避免离散访存并获得理想加速比。

在该课题原始算法中，对主要变量位置坐标的每个方向都要进行相应的差分求解，因此不可避免的存在离散访存，并且导致计算开销急剧增加。因此在针对存在大量离散访存的程序段的众核并行设计过程中，首先在计算核心上将原来离散的数组调整成方便通信的读入和写回的存储顺序，然后从核心进行通信读入数据、计算和通信写回数据，最后从核心将写回的数据再次调整回原来的存储顺序。尽管相较于原来的算法，增加了前后两个数组存储顺序调整的过程，但由于上述过程都是在从核心上来完成的，两个存储顺序调整所导致的从核心开销增加的不大。另外，计算方面则由于从核心对专属局部存储空间访问延迟小使得从核心计算开销大大减小。在众核并行优化过程中，有些从核并行程序段实现超线性的加速。

9.1.4 优化效果

9.1.4.1 从核并行加速情况

鉴于众核并行算法加速效果仅在从核并行域部分体现，因此为更直观的体现性能加速效果，在优化验证时采用的位置空间网格规模为 75×36×51，离散速度空间网格规模为 16×16×14。优化验证的对象包括速度分布函数双曲守恒方程的各个求解过程以及整体过程的从核程序的加速效果，单个“申威 26010”异构众核处理器上相应的优化验证结果如表格 9-1 所示。

表格 9-1 核心程序模块从核并行效率

各功能模块	从核并行效率
二阶龙格库塔法求解部分 1	0.37
ξ 方向差分方程求解程序	1.32
η 方向差分方程求解程序	0.80
ζ 方向差分方程求解程序	0.23
二阶龙格库塔法求解部分 2	0.35
求解双曲守恒方程过程整体	0.68

从表格 9-1 结果可以看出，该课题的众核加速效果比较理想。对于 ξ 方向差分方程求解则由于原算法中存在大量离散访存，完成从核并行后获得超线性

加速，从核并行效率达到 1.32； η 方向差分方程求解也获得理想的加速效果；值得注意的是，ζ 方向差分方程求解过程中由于存在从核直接离散访问主存的情况导致加速效果不理想，需要进一步的优化。整体上能够较好的发挥众核处理器的性能优势，获得良好的加速效果。

9.1.4.2 大规模并行加速比

为了验证超大规模并行可扩展性与加速性能，在同一网格计算量设置 $61 \times 25 \times 31 \times 300 \times 250 \times 250$，分别在 243750 至 7312500 从核心数上进行了加速比测试，实测所用进程与计算时间及所得加速比、并行效率如表格 9-2 所示。

表格 9-2 并行计算实测加速比

从核数	243750	406250	609375	812500	1015625	1625000	2031250
时间(秒)	46.69	28.37	18.24	13.86	11.10	6.99	5.74
理论加速比	1	1.67	2.5	3.33	4.17	6.67	8.33
实测加速比	1	1.65	2.56	3.37	4.21	6.68	8.13
并行效率	1	0.99	1.02	1.01	1.01	1.00	0.98

从核数	2437500	3250000	4062500	4875000	6500000	7312500
时间(秒)	4.77	3.67	2.90	2.45	1.92	1.82
理论加速比	10	13.33	16.67	20	26.67	30
实测加速比	9.79	12.72	16.1	19.06	24.32	25.65
并行效率	0.98	0.95	0.97	0.95	0.91	0.85

9.2 全球大气浅水波方程显式求解软件

9.2.1 概述

浅水波方程是大气模拟类问题中最基础，也是最重要的一种，其反映了核心的大气模拟计算特征，是试验和开发新一代大气模式动力框架不可或缺的步骤。为此，中科院软件所杨超研究员和清华大学薛巍、付昊桓老师的联合研发团队，

设计和开发了新一代、面向混合架构的可扩展大气浅水波方程求解器。该系统一方面可以用于大气动力框架计算方法的研究和试验；另一方面希望能为未来 E 级计算系统的设计和开发提供一个重要的测试样例，为百万核可扩展应用的研制提供一个优化范本。

9.2.2 算法简介

浅水波方程求解的是如下的二维偏微分方程：

$$\begin{cases} \dfrac{\partial h}{\partial t} + \nabla \cdot (h\mathbf{v}) = 0, \\ \dfrac{\partial (h\mathbf{v})}{\partial t} + \nabla \cdot (h\mathbf{v} \otimes \mathbf{v}) = \Psi_C + \Psi_G, \end{cases}$$

采用显式方案，通过离散化，该方程可以转换为如下形式。此形式为典型的模板计算(stencil)。

$$\overline{X}(t^{(m)}) = X(t^{(m-1)}) - \Delta t \mathcal{L}(X(t^{(m-1)})),$$
$$X(t^{(m)}) = \frac{1}{2}\left\{X(t^{(m-1)}) + \overline{X}(t^{(m)})\right\} - \frac{1}{2}\Delta t \mathcal{L}(\overline{X}(t^{(m)})),$$

采用球立方网格（cubed sphere），它将地球看成一个有六块皮球片拼成的正方体，其形状如图 9-7 图所示：

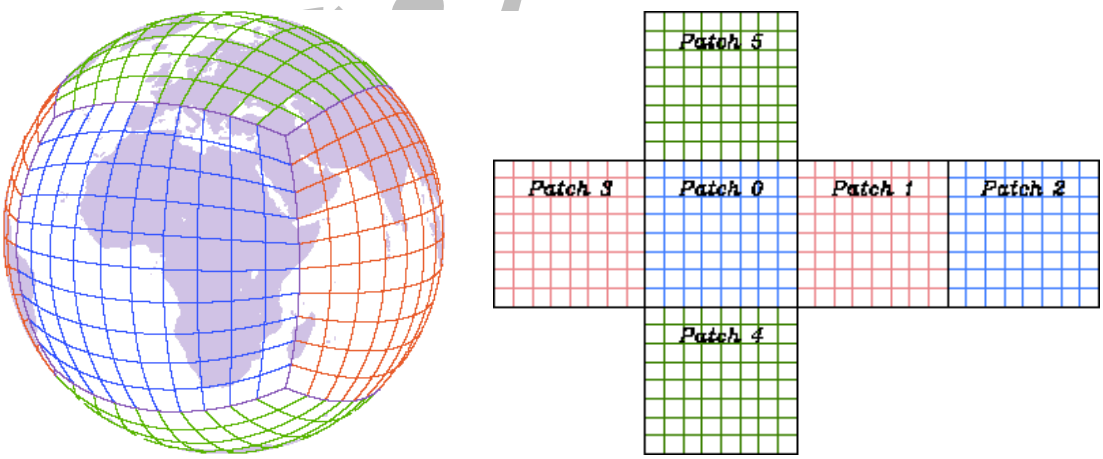


图 9-7 球立方网格系统

在计算时，先按图 9-7 将整个数据空间分成六块(patch)，对应六个 MPI Group。再在每一个 MPI Group 中将数据划分成子块(sub-block)，并分派到不同进程中。Stencil 的计算格式为 13 点，如图 9-8 所示：

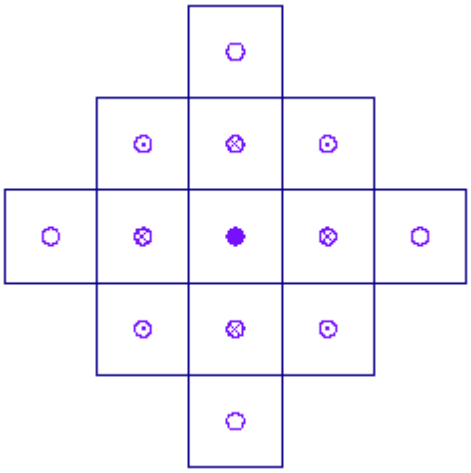


图 9-8 Stencil 格式

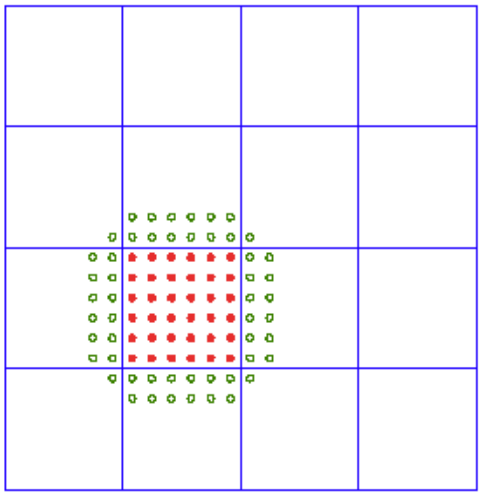


图 9-9 patch 中一个 sub-block 的 halo 区

可以看到，计算每个网格点需要上、下、左、右方向各两个相邻点，以及四个斜方向的相邻点，这样就产生了进程间的数据通信的需求。其中每一个进程的通信模式如图 9-9 所示，这时 cubed-sphere 中一个 patch 被划分为多个 sub-block 的情况。红色点是该进程所需计算的网格点，绿色点是为了计算对应网格点而需要从相邻进程获取的数据。一次 stencil 算法计算的流程如图 9-10 所示。

算法 1： stencil 算法

```
1:  for all six patches do
2:     for all sub-blocks in each patch do
3:        Update halo information
4:        Interpolation for ghost cells when necessary
5:        for all mesh cells in each sub-block do
6:           Compute stencil for the h component
7:           Compute stencil for the hu[1] component
8:           Compute stencil for the hu[2] component
9:        end for
10:    end for
11: end for
```

图 9-10 stencil 算法计算流程

上述 stencil 的计算流程包括四个主要步骤：

1) 邻居通信（Update halo information）。基于 PETSc 的框架，使用其中提供的非阻塞邻居通信接口(VecScatterBegin/End)进行通信；

2) 数据拷贝。通过数据拷贝完成对 stencil 计算的准备；

3) 边界插值（Interpolation for ghost cells when necessary），在使用从来自不同 patch 的邻居中的 halo 区数据之前，需要对其进行线性插值以达到滤波的效果；

4) stencil 计算（Compute stencil for h/ hu1/ hu2），这部分涉及大量计算，包括除法和三角函数，占绝大部分计算时间。

9.2.3 主从异步并行

针对“神威•太湖之光”计算机系统的系统结构特点，基于上节 4 步骤各自的计算特征，设计每个主从结构负责一个 sub-block 的计算，运算主核心进程完成 sub-block 间的邻居通信，数据拷贝和 patch 间的边界插值，将核心计算部分（也就是 stencil 计算）全部放在从核心上，采用多线程方式完成。这种方案一方面充分利用了主从计算资源，又减少了不必要的运算核心线程管理开销。而为了进一步提高计算效率，又引入了通信计算重叠的策略。该策略先将每个进程负责的 sub-block 进行进一步划分。

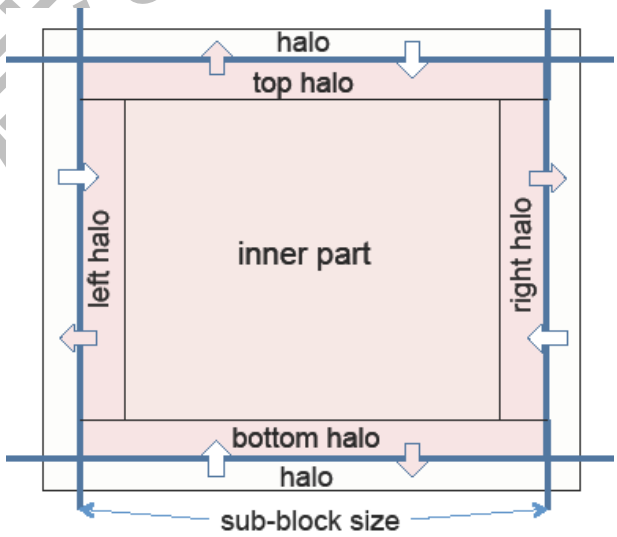


图 9-11 面向计算通信重叠的 sub-block 划分

在该划分中，Inner part 指 stencil 的中心部分，这部分的计算只需要本进程的数据即可完成，不需要等待通信。上、下、左、右四个 halo 区是边界，这部分的计算需要等待相邻的八个进程通信的数据，见图 9-11 所示。通过合理的计算次序安排，可以用 Inner part 的 stencil 计算有效重叠 halo 区通信，其计算流程具体如图 9-12 所示。

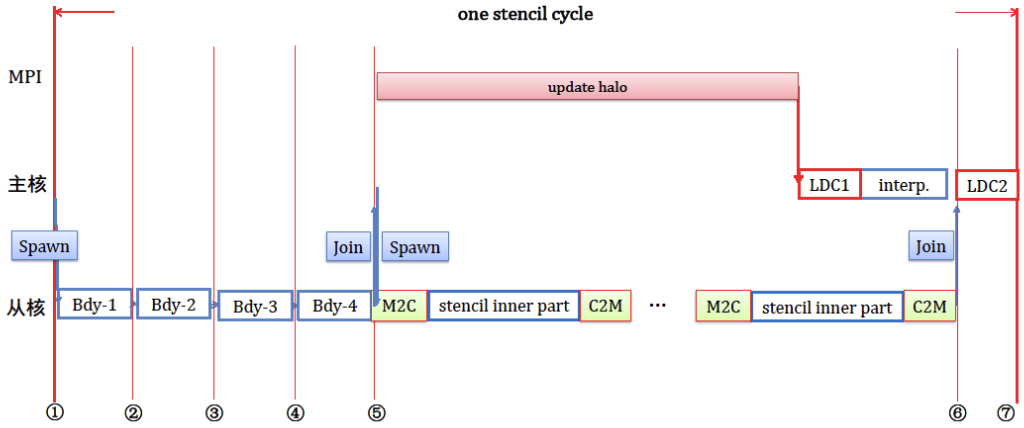


图 9-12 主从结构整体计算方案

为适应全机运行需求，还设计实现了自适应的划分策略。全球大气浅水波方程的新版本可以同时支持 $6N^2$ 和 $6N^M$ 并行度，程序尽可能完成均衡的 sub-block 划分。需要指出的是，$6N^2$ 并行度时有最佳的面积边长比，会有最好的计算效率。

9.2.3.1 从核心上的 stencil 计算方法

如图 9-11 所示，sub-block 的 Inner Part 是计算的核心区，如何在从核心上对 Inner Part 计算提速是对整体计算提速的关键，图 9-13 是计算 Inner Part 的算法方案，即每个从核心计算一个纵条。计算 Inner Part 的算法方案还有其他两种选择：

1) 与图 9-13 类似，每个从核心计算一个横条；

2) 每个从核心计算一个 X*X 的块状结构。

选择图 9-13 对应方案是因为：

1） 条状方案比块状方案更容易写成流水的形式，从而所需要的内存更小；

2） 纵条方案比横条方案在访存连续性上更具有优势。

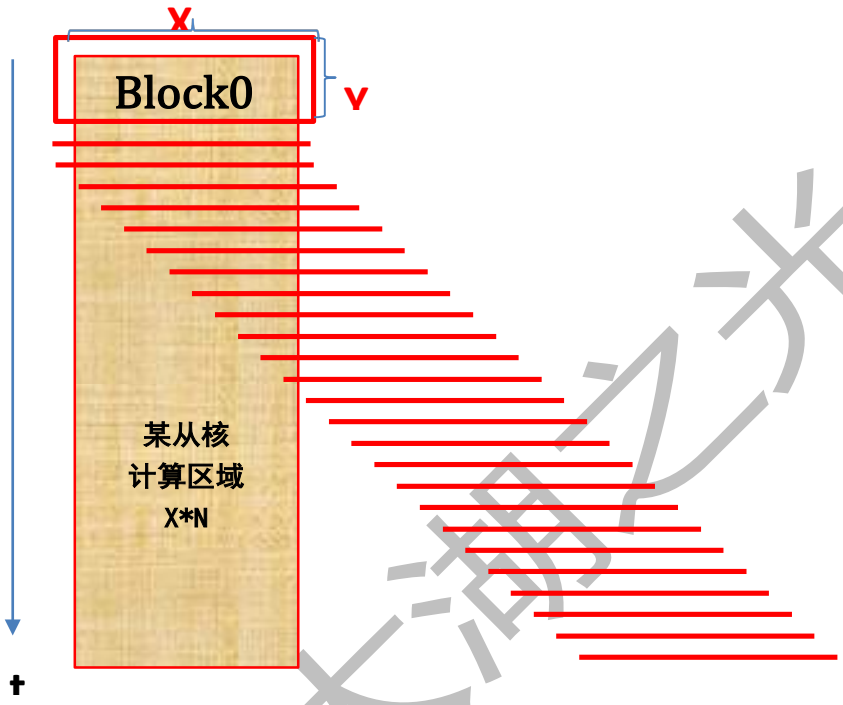


图 9-13 Inner Part 运算核心计算方案

由于从核心自身数据空间大小受限，所以对其分配空间的优化至关重要。对于 Inner Part 来说需要 47*(X+4)*sizeof(double)，由于真正可以用于分配的空间大概是 17KB，则 X<=42，最终选择 X 为 32。在 Inner Part 计算中，主要用到了 x_h、x_hu、x_hv、x1_h、x1_hu、x1_hv、y_h、y_hu、y_hv、hs、co，且都是二维指针，但是宽度不同，对于计算宽度为 X 来说，一部分为 X+4 个 double，一部分为 X 个 double。经过测试，如果空间分配时，按照 X+4 个 double 进行对齐，虽然空间有少量浪费，但性能上有 10%的提升。

表格 9-3 Inner Part 和边界计算所需数据

计算中心的输入值	PETScScalar **x_h, **x_hu, **x_hv; PETScScalar **x1_h, **x1_hu, **x1_hv; PETScScalar **hs, **co;
计算边界的输	PETScScalar **x_h, **x_hu, **x_hv; PETScScalar **x1_h, **x1_hu, **x1_hv;

入值	PETScScalar **hs, **co; PETScScalar *L_i_h, *L_i_hu, *L_i_hv, *L_i_hs, *L_o_h, *L_o_hu, *L_o_hv, *L_o_hs; PETScScalar *R_i_h, *R_i_hu, *R_i_hv, *R_i_hs, *R_o_h, *R_o_hu, *R_o_hv, *R_o_hs; PETScScalar *B_i_h, *B_i_hu, *B_i_hv, *B_i_hs, *B_o_h, *B_o_hu, *B_o_hv, *B_o_hs; PETScScalar *T_i_h, *T_i_hu, *T_i_hv, *T_i_hs, *T_o_h, *T_o_hu, *T_o_hv, *T_o_hs;

对于 sub-block 的边界计算来说，在数据上，需要比计算中心更多的数据，如表格 9-3，在计算上，也要比计算中心更复杂，这里就需要考虑 2 个问题：

1） 额外数据如何获取？进而，获取额外数据后空间是否足够？

2） 调用边界计算的边的宽度是多少？

对于额外数据，由于对于一个从核心来说，计算上下边界不可能共存，计算左右边界不可能共存，所以在分配空间时判断该从核心是计算哪一个部分，然后获取相应的额外数据。并且，经过计算，对于最为复杂的计算角点的从核心来说，计算宽度为 32，即与计算中心同宽，恰好空间足够；

对于边界计算，一方面访存量明显下降，另一方面，计算函数的判断过程更加复杂，即算的更慢，那么如何选取边界的宽度成为一个问题：如果选择比较大，那么可以通过计算掩藏访存，但是代价是有更多的非边界点用边界函数去算，浪费时间；如果选择的比较小，那么没有浪费计算资源，但是代价是通信将几乎完全暴露出来。经过测试，最终选择了宽度为 2，即为最小值，这样最小程度的降低计算资源的浪费，由于只有 2 条边，则通信暴露的代价还是可以接受的，对于 meshsize 为 1024 的样例来说，边界计算只占用了 2%-3%的时间，还是比较理想的。

9.2.3.2 从核心上的 stencil 计算优化

上节主要介绍了从核心上的 stencil 计算方法，本节则介绍从核心上的 stencil 计算可以采取的三种优化方式。

1） 数据结构从 Aarry Of Structure（AOS）向 Structure Of Array（SOA）转换

表格 9-4 AOS 向 SOA 转换优化

原始版本	优化版本
Typedef struct { PETScScalar H, HU, HV; } ActiveField; ActiveField **x, **x1, **y;	PETScScalar **x_h, **x_hu, **x_hv; PETScScalar **x1_h, **x1_hu, **x1_hv; PETScScalar **y_h, **y_hu, **y_hv

全球大气浅水波方程原有的实现如表格 9-4 左侧所示，这样导致计算过程中跳跃访存（例如 temp=x[i][j-1].H+x[i][j].H+x[i][j+1].H，中间有 2 个 double 的偏移），导致计算效率不佳，更不利于向量化。改进版本表格 9-4 右侧所示，对于任何一个二维数组，访存皆是连续（temp=x_h[i][j-1]+x_h[i][j]+x_h[i][j+1]），这样便为向量化做好了基础准备。不过带来的代价是每次需要在从核心中进行相应的转化。

2） 从核心访存和计算重叠

表格 9-5 访存计算重叠

原始版本	改进版本

loop:
  get 数据;
  计算;
  put 数据
end
	prepare data for step 0;
	i = 0
	loop:
	  等待 step i get 数据结束;
	  get step i+1 数据;
	  计算 step i 结果;
	  等待 step i-1 put 数据结束;
	  put step i 数据;
	  i ++;
	end

由于从核心的访存为异步访存，所以可以通过流水模式将 put 和 get 的时间掩藏在计算之中。采用的方法如表格 9-5 所示，就是将 step i+1 的 get，step i 的计算，以及 step i-1 的 put 同时进行，最大限度的将 put 和 get 掩藏起来。

3） 三角函数调用优化

表格 9-6 三角函数调用优化

以 meshsize=1024 为例 W=32，H=1024
原始版本
for 1:H for 1:W 6 * tan compute end end
优化版本

// 6*tan
compute tanbi_2, tanbj_2, tan05x, tan05y, tan1x, tan1y;
for 1 : H
for 1 : W
use tanbi_2, tanbj_2, tan05x, tan05y, tan1x, tan1y and tan(A+B)=(tanA+tanB)/(1-tanA*tanB) to compute 6 values;
compute;
tanbi_2 = (tanbi_2+tan1x)/(1-tanbi_2*tan1x);
end
tanbj_2 = (tanbj_2+tan1y)/(1-tanbj_2*tan1y);
end

当前系统的三角函数 tan 计算延迟大，且随输入的不同性能有较大波动，所以希望能在速度和稳定性上均得到优化。经过分析，发现原始版本的 tan 计算数量如表格 9-6 原始版本所示，即计算了 6*H*W 次 tan，虽然每次 tan 的参数均不同，但是是有规律的，即 tan 参数的偏移量恒定，这便产生了如表格 9-6 优化版本所示的优化方法，先计算出原始的 tan 值（tanbi_2 和 tanbj_2）以及偏移量（tan05x，tan05y，tan1x，tan1y），然后通过公式 tan(A+B)=(tanA+tanB)/(1-tanA*tanB)计算以后所有的值，这样就将原始的 6*H*W 次 tan 压缩为 6 次 tan 和 6*H*W 次加减乘除组合，因为加减乘除无论在速度上和在稳定性上都远胜于 tan，所以用上述方法实现了 tan 在速度和稳定性上的优化，经测试，以 meshsize=1024 为例，速度提升 40%-50%。

9.2.3.3 从核心访存优化

在调用从核心计算中心时，发现流水带来的收益并不明显。由于 put 和 get 的动作发起花费了大量的时间，无法有效进行计算访存重叠，另外由于访问数据量不大，所以动作发出到完成，花费的时间并不多。针对这一点，可以采用如下 2 步进行优化，具体如下：

1） 将函数改成汇编指令，并提取其中的不变量；

2） 进行 put/get 时采用相对地址减少主存访问次数。

表格 9-7 从核心访存优化

以 x 的 get 为例演示优化过程	

原始版本	loop:
	
	athread_get(PE_MODE,&(global_x[beginI][beginJ]),
	compute_data.x_buffer_B, sizeB*3,
	&x_reply, 0, 0, 0);
	// 进行 2 次主存访问
	beginI ++;
	end

改进 1	#define A_DMA_GET_SET(da,mode,len,re_addr) \ ({ \ dma_set_op(&da, DMA_GET); \ dma_set_mode(&da, mode); \ dma_set_size(&da, len); \ dma_set_reply(&da, re_addr); \ }) #define A_DMA_GET_RUN(da,src,dest) \ ({ \ dma(da, src, dest); \ }) A_DMA_GET_SET(gvb3, PE_MODE, sizeB*3, x_reply); loop: A_DMA_GET_RUN(gvb3,&(global_x[beginI][beginJ]), compute_data.x_buffer_B);

	//进行 2 次主存访问
	beginI ++;
	end

改进 2	ActiveField *my_local_x;
	my_local_x = &(global_x[beginI][beginJ]); //进行 2 次主存访问
	A_DMA_GET_SET(gvb3, PE_MODE, sizeB*3, x_reply);
	loop:
	A_DMA_GET_RUN(gvb3,my_local_x,
	compute_data.x_buffer_B);
	my_local_x += x_n;
	end

9.2.3.4 从核心向量化

由于有之前 AOS 转 SOA 的基础，所以从数据结构上已经完全有可以向量化的基础。结合编译器，对从核心看进行向量化的优化。

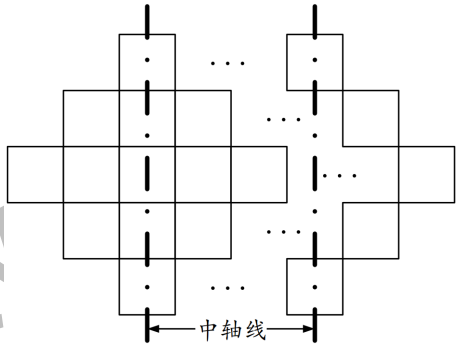


图 9-14 Stencil 对齐情况示意

Stencil 有天生的不对齐的访存特性，以 13 点 stencil 为例，如图 9-14 右图所示，假设按照中轴线对齐，那么访问中轴线两侧的数据都一定是不对齐的访问，即 13 个点的访问中，有 8 个点的访问是不对齐的，需要额外的指令支持。

结合程序本身的一些特性，设计了局部 AOS 转 SOA 的方法，从而在向量化过程中消除所有的不对齐访问。而由于 AOS 转 SOA 的过程是嵌入在循环内部的，编译器几乎不可能识别优化意图，所以这个过程只能手动完成，如图 9-15 所示。

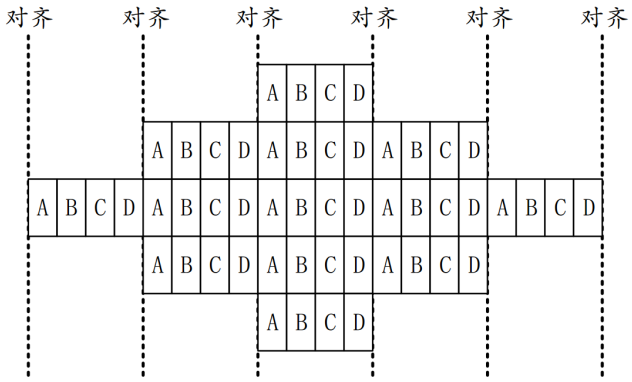


图 9-15 修改后 Stencil 对齐情况示意

程序中有 4 组数据是执行相同的计算模式的，即有 4 个完全相同的 13 点 stencil。我们将这 4 组数据交错排布，如图 9-15 所示，由于系统是按照 4 个 double 对齐，所以每次向量化的访问都将是对齐的访问。当 stencil 部分计算结束后，我们再用 shuffle 指令将 AOS 的结果重新排布为 SOA 的结果，以用于之后的计算。具体如下：

表格 9-8 从核心向量化优化

以 A，B，C，D 为例演示局部 AOS 转 SOA 的优化	

原始版本	//A,B,C,D 为输入，数据类型为 double
	//RA,RB,RC,RD 为局部输出，用于之后的计算，数据类型为 double
	RA=Function(A[j-2][i], A[j-1][i-1], A[j-1][i],
	A[j-1][i+1], A[j][i-2], A[j][i-1], A[j][i],
	A[j][i+1], A[j][i+2], A[j+1][i-1], A[j+1][i],
	A[j+1][i+1], A[j+2][i]);
	RB=Function(B[j-2][i], B[j-1][i-1], B[j-1][i],
	B[j-1][i+1], B[j][i-2], B[j][i-1], B[j][i],
	B[j][i+1], B[j][i+2], B[j+1][i-1], B[j+1][i],
	B[j+1][i+1], B[j+2][i]);
	RC=Function(C[j-2][i], C[j-1][i-1], C[j-1][i],
	C[j-1][i+1], C[j][i-2], C[j][i-1], C[j][i],
	C[j][i+1], C[j][i+2], C[j+1][i-1], C[j+1][i],
	C[j+1][i+1], C[j+2][i]);
	RD=Function(D[j-2][i], D[j-1][i-1], D[j-1][i],
	D[j-1][i+1], D[j][i-2], D[j][i-1], D[j][i],
	D[j][i+1], D[j][i+2], D[j+1][i-1], D[j+1][i],

	D[j+1][i+1], D[j+2][i]); //标红部分为不对齐访问

手动向量化	//X 为交错排布 A，B，C，D 数组后的输入，数据类型为 doublev4
	//R 为局部输出，数据类型为 doublev4
	R=Function(X[j-2][i], X[j-1][i-1], X[j-1][i],
	X[j-1][i+1], X[j][i-2], X[j][i-1], X[j][i],
	X[j][i+1], X[j][i+2], X[j+1][i-1], X[j+1][i],
	X[j+1][i+1], X[j+2][i]);

局部 AOS 转 SOA	//in0，in1，in2，in3 为输入输出，数据类型为 doublev4
	doublev4 o0 = simd_vshff(in1,in0,68);
	doublev4 o1 = simd_vshff(in1,in0,238);
	doublev4 o2 = simd_vshff(in3,in2,68);
	doublev4 o3 = simd_vshff(in3,in2,238);
	in0 = simd_vshff(o2,o0,136);
	in1 = simd_vshff(o2,o0,221);
	in2 = simd_vshff(o3,o1,136);
	in3 = simd_vshff(o3,o1,221);

9.2.3.5 其他优化方法

1） 从核心线程 spawn 调用位置调整

异构众核结构浅水波方程的实现当中，主核心邻居通信启动和从核心线程 spawn 这两个函数调用需要一一完成，其调用顺序不相互依赖。为此，通过测试发现先启动从核心线程 spawn，后启动主核心上的邻居通信，在大规模运行环境下可以取得 10%的性能提升。

2） 采用 MPI 库中提供的复合式连接方法

在大规模运行中，课题启动时间大幅攀升，且 MPI 占用的主存容量增加导致无法运行更大规模的例题。为此，可使用 MPI 库中提供的复合式连接方法，提高了用户程序可使用的内存容量（为应用程序留存了更多可用空间），缩短了超大规模课题启动开销，在半机规模情况下，每个 MPI 进程未优化时可用 7.2G 内存空间，优化后可用 7.5G 内存空间，初始化时间约为 30 秒。

9.2.4 优化效果

本节给出了两方面的性能测试结果和分析：一方面，使用“申威 26010”异构众核结构与单独使用主核心进行相同计算的性能加速情况，验证面向异构众核结构的算法和优化策略的有效性，该项测试采用全球 10km 分辨率的算例，该分辨率是正在研发下一代大气模式目标分辨率；另一方面，测试大规模系统上的可扩展性情况，验证计算通信重叠策略的有效性，并评估并行开销。这部分测试目标是程序计算扩展性，需要采用更高分辨率的测试算例。

测试过程中，针对每个算例，程序进行 100 个时步的模拟，每 10 个时步进行一次结果输出，用以正确性校验。性能测试指标中用的是平均每时步的计算时间。由于输出步需要额外计算与输出，而第一时步需要数据加载，其计算时间大大增加，因此只统计中间 89 个时步的计算时间。值得注意的是，该程序采用的是显式计算方法，因此计算网格规模增加导致步长降低，每个时步计算具有可比性，而没有采用模拟相同时长的平均计算时间作为性能指标。

9.2.4.1 主从核心加速性能测试

表格 9-9 主从核心计算的性能对比

核心类型	单纯主核心程序	从核心计算程序
算例网格总数	6.29M	6.29M
步长(s)	1.6E-4	1.6E-4
算例未知数总数	18.87M	18.87M
进程数	6	6
使用的计算核心数	6	390
平均每时步计算时间(s)	10.051	0.027

从表格 9-9 的结果可见，对于同一算例，主从核心计算程序较纯主核心性能提升超过 256（64*4）倍。仔细分析可知，两个程序的计算复杂度相同，编译也都选取了各自最优的编译参数，但由于结构不同实现不同。性能提升如此之大可能的原因有：

1) 访存对 Stencil 计算性能有明显影响，从核心的寄存器总量要大于主核心的 LLC 容量，使得计算访存性能得到进一步提升；
2) 上述提及的异构众核优化实现采用了 blocking、访存计算重叠等技术进一步提升了计算性能，这部分优化没有在主核心中相应实现；
3) 三角函数调用优化大幅降低了 tan 的使用，从核心和主核心性能差异的问题得到解决；
4) 用局部 AOS 转 SOA 的方法手动向量化，可以最大化的提高向量化的效率，减少不对齐的访存数量，从而减少指令数量。

9.2.4.2 并行可扩展性测试

表格 9-10 弱可扩展性测试

序号	1	2	3	4	5	6	7	8	9
进程总数	1536	6144	13824	24576	38400	55296	75264	98304	124416
计算核心总数	99840	399360	898560	1597440	2496000	3594240	4892160	6389760	8087040
算例网格总数（G）	13.5	54.0	121.5	216.0	337.5	486.0	661.5	864.0	1093.5
平均每时步计算时间(s)	0.217	0.217	0.217	0.217	0.218	0.218	0.212	0.218	0.218
算例网格总数（G）	6.0	24.0	54.0	96.0	150.0	216.0	294.0	384.0	486.0
平均每时步计算时间(s)	0.099	0.099	0.099	0.099	0.099	0.099	0.099	0.099	0.099

算例网格总数（G）	1.5	6.0	13.5	24.0	37.5	54.0	73.5	96.0	121.5
平均每时步计算时间(s)	0.027	0.027	0.027	0.027	0.028	0.028	0.028	/	/

表格 9-11 强可扩展性测试

序号	1	2	3	4	5	6	7	8	9
算例网格总数（G）	13.5								
进程总数	1536	6144	13824	24576	38400	55296	75264	98304	124416
计算核心总数	99840	399360	898560	1597440	2496000	3594240	4892160	6389760	8087040
平均每时步计算时间(s)	0.2173	0.0576	0.0275	0.0173	0.0126	0.0094	0.0072	0.0064	0.0058
并行效率	100.00%	94.38%	87.88%	78.38%	69.21%	64.52%	61.31%	53.30%	46.30%

从表格 9-10 和表格 9-11 的测试结果可以看出，弱可扩展性测试取得了理想的性能结果，在每个核组计算 3072x3072 网格可以有效扩展到 12 万核组规模（合计 809 万核），这说明算法中所提出的计算通信重叠策略的有效性。强可扩展性没有取得接近线性的加速比结果，从 1536 到 12 万核组并行效率为 46%。分析可知，强可扩展性测试中单核组处理的网格规模在不断降低，其计算时间与处理网格数不成线性关系，单核组网格数降低到 1/4，计算时间变为 1/3.77；且随着单核组网格数的降低，该数字还可能进一步下降。究其原因，边界区的网格点计算复杂度要大大高于 Inner Part 的计算时间，而网格数降低到 1/4，这部分的计算时间仅仅降低到 1/2;同时，随着单核组网格规模降低，从核心线程 spawn，MPI 消息拷贝等时间成为不可忽略的部分。

附录 A　作业管理

A.1 基本概念

A.1.1 作业与作业 ID

- **作业**：是用户编写的经过编译后并在主机上运行的用户可执行程序或脚本。用户作业程序的编写可以使用 MPI、OpenMP 等各种编程语言和环境，最终以可执行程序的形式存在，并以作业的形式被提交到主机中运行。任何作业都必须提交到某个指定的队列中调度运行；
- **作业 ID**：每道作业有唯一的作业 ID（整数，JOBID），这也是作业的唯一性标识。
- **计算节点号（nodeid）**：是计算节点的编号，就是每个操作系统核心（一个 IP 地址）所在 CPU 的编号。

A.1.2 作业队列

- **作业队列**：分配给是指定用户的计算资源，用户在作业提交时要指定作业队列名。作业队列由管理员创建和管理、由用户使用。
- **作业流程**：所有用户运行课题以作业的形式提交到指定的队列中，系统为每道作业分配唯一的作业 ID，作业在队列中排队，接受作业管理系统的自动调度运行。运行时，根据作业服务环境的队列属性允许普通用户对作业进行人机交互控制，或者由系统自动批处理。

A.1.3 作业类型

- **串行作业**：采用 C、Fortran 等语言编写的单进程串行应用程序或者 shell 程序；
- **并行作业**：采用 MPI、OpenACC 等编程环境编写的多任务并行应用程序。

A.1.4 作业运行模式

作业服务提供两种作业运行模式：交互作业、批式作业，由用户提交作业时决定。

- **交互作业：**在作业提交命令 bsub 命令行中指定“-I”参数。作业提交后，作业运行信息在终端窗口中实时输出，允许用户进行各种人机交互控制。这种运行模式适合在课题调试时使用。一旦提交窗口被意外关闭，作业将自动转为脱机状态，不会终止，用户可以通过作业联机命令（bonline）重新在终端窗口接收作业的实时输出。交互运行模式下，如果系统发现当前队列中的空闲资源不能满足该作业的资源需求，作业提交时将自动报错退出。
- **批式作业：**在作业提交命令 bsub 命令中不指定“-I”参数。批作业提交方式后，系统根据计算资源和负载情况自动运行，并将结果输出文件保存在文件系统中。这种作业的特点是作业提交运行后，提交窗口可以关闭，作业运行不受任何影响。作业一旦开始运行，作业的标准输出和错误输出都将被自动重定向，如果提交时指定了输出定向的文件，则将输出到对应的文件中，否则系统将进行缓存。作业运行过程中，可以通过相应的命令查询作业输出。批作业运行模式下，即使当前队列中的空闲资源不能满足本作业的需求，作业将排队等待。由于交互作业在实时终端窗口的输入或操作都会对作业产生影响，所以相比之下批式作业更适合业务课题的运行，建议所有业务运行的课题都使用批作业模式提交。

A.1.5 作业运行状态

- PEND：作业等待分配计算资源；
- STARTING：作业正在启动运行。是作业已经分配资源后到作业正式启动完成开始运行之间的过渡状态；
- RUN：作业正在运行。作业已经完成调度并分派，作业占用的系统资源，正在运行中；

- DONE：作业正常完成并退出；
- EXIT：作业异常完成并退出；
- CKPT：作业正在系统级全透明保留，一旦保留完成，作业回复到运行状态。

A.2 队列命名规则

“神威•太湖之光”计算机系统提供两类计算资源：高速计算系统和辅助计算系统。高速计算系统由“申威 26010”异构众核处理器组成，辅助计算系统由 Intel 处理器组成，因此计算资源分成两个资源队列类型：

- **高速计算系统：**q_sw_xxxx，高速计算系统缺省队列：q_sw_expr
- **辅助计算系统：**q_x86_xxxx，辅助计算系统缺省队列：q_x86_expr

其中，q_sw_expr 和 q_x86_expr，主要用于用户程序的开发、移植、调试与优化，只要用户有系统账户，就可以免费使用这两个队列资源，限制条件：

- 每个任务的并行规模不能超过 64
- 每个任务的计算墙钟时间不能超过 1 小时

当用户完成程序的移植和优化后，根据对计算资源的估算，再申请收费的计算资源进行课题的大规模计算。

A.3 作业管理常用命令

A.3.1 作业提交

功能	向指定的计算资源提交用户作业
命令格式	● bsub [-h][-v] ● bsub [-f sub_script] ● bsub [-I] [-p] //list job's nodename and spe_map [-q queue_name] [-n num_procs [-master] | -N num_nodes] [-np node_mpes] [-mpecg mpe_cgs]

	[-cgsp spe_in_cg | -min_cgsp min_spe_in_cg | -rtp spe_rtp | -asy] [-exclu | -shared | -cpuexclu] [-js job_proj] [-lfs_proj lfs_proj] [-J jobname] [-jobtype job_type] // available type: COMM / I/O / COMP / MEM / MIX [-mpmd] [-node nodelist] [-o outfile] [-k ckpt-period-minutes] [-cross] [-switchnode nodenum_in_switch] [-midnode nodenum_in_mid] [-cabnode nodenum_in_cab] [-health health_level] [-b] [-parse] [-PARSE <all | master | slave>] [-quick] [-m value] [-share_size size] [-priv_size size] [-cross_size size] [-ro_size size] [-pe_stack size] [-host_stack size] [command [argument...]]
参数说明	-h 显示帮助信息 -I 提交交互式作业，使作业输出在作业提交窗口，无该选项时为批式作业 -q 向指定的队列中提交作业，必选项 -p 在作业输出中打印作业分配的节点列表及位图 -exclu | -shared | -cpuexclu 指定使用 CG 独占/CG 共享/CPU 独占模式 -n 指定需要的所有主核数 -N 指定需要的节点个数 -np 指定每节点内使用的主核数 -cgsp 指定每个 CG 内需要的从核个数，指定时该参数必须<=64。

	-asy 指定使用非对称资源，表示各个 CG 内使用的从核的个数可以不同 -js 指定作业对应的课题代号 -lfs_proj 指定作业使用的局部文件代号 -node 指定运行作业的节点（CG 列表） -cross 要求分配全片 CPU（4CG 的 CPU） -health 指定分配资源的健康度级别 -o 将作业的 stdout 和 stderr 的输出定向到指定文件，可选项 -switchnode 指定每个 switch 中分配的节点数 -midnode 指定每个中板中分配的节点数 -cabnode 指定每个机舱中分配的节点数 -b 指定从核栈位于局存 -share_size 指定核组共享空间大小 -priv_size 指定每个核上私有空间大小 -cross_size 指定交叉段大小 -ro_size 指定只读空间大小 -m value 提供从核自陷模式的控制，指定-m 2时，将浮点控制状态寄存器fpcr的最后两位设为01，允许除不精确结果之外的所有浮点算术异常自陷，相当于编译器使用-OPT:IEEE_arith=2选项；指定-m 1时，将fcpr最后两位设为00，允许所有浮点算术异常自陷，相当于编译器使用-OPT:IEEE_arith=1选项；其他所有值将不对默认的fpcr进行修改。 -pe_stack 指定从核栈空间大小，默认为64K -host_stack 指定主核栈空间大小，默认为 8M
使用范例	向队列 queue 中提交互式作业 myjob，该作业共使用 1 个节点，四个主核： bsub -I -q queue -N 1 -np 4 ./myjob 作业提交成功后，将显示一行包含 jobid 的提示信息，其中包括作业 id 号，如”Job <102> is submitted to queue <queue>”，此时，jobid 就是 102，它是全局唯一的。一旦作业提交成功，用户对作业的各种操作就可以通过这个 jobid 来实现的。 使用缺省队列 q_sw_expr 可以不使用-q 参数，如： bsub -I -N 1 -np 4 ./myjob
注意事项	1）-I 参数与-o 参数通常不建议放在一起使用。因为使用-o 参数就无法在屏幕输出上看见程序的打印 2）每道作业提交成功后，都会有一个 jobid，这是本作业可以区别于其他作业的唯一特征。系统将保证 jobid 的唯一性。作业

	生命周期中，对作业的任何操作都需要以 jobid 为参数 3）本命令的各种参数都需要在用户程序之前 4）用户必须在 SN 节点上使用

A.3.2 作业终止

功能	终止作业
命令格式	bkill [-h] [-J jobName] [-u user] [-q queue] [-f] [jobId]
参数说明	-h 帮助信息 -u 操作指定用户提交的作业 -q 操作指定队列内的作业 -J 操作指定作业名称的作业 -f 当终止批量作业时不需要确认 jobId 表示作业 id 标识符 说明：当 jobId 指定时，忽略-u -q 选项
使用范例	bkill 1234 终止队列中作业 id 号为 1234 的作业 bkill -q q1 终止队列 q1 内的本用户所有的作业
注意事项	1）使用-u 参数时，管理员可以指定所有用户，普通用户只能指定本用户 2）对于普通用户来说，即使使用-q 参数，也只能操作本用户在指定队列中的作业 3）必须在 SN 节点上使用

A.3.3 作业状态查询

功能	查询队列中作业的状态信息

命令格式	bjobs [-h] [-l] [-w] [-a | -d | -e | -p | -r] [-q queue_name] [-u user_name | -u all] [jobId]
参数说明	-q 指定要查询的队列 -l 长格式显示作业详细信息 -w 全长度显示，当列值的长度超过列宽时，不按列宽进行截取 -a 显示一段时间内的所有作业 -u 显示指定用户的作业，all 是特殊的关键字，可显示所有用户的作业信息 -d 显示最近正常完成的作业 -e 显示最近异常退出的作业 -p 显示处于 pend 状态的作业 -r 显示正在运行的作业 jobid 作业 id 号
使用范例	bjobs 1234 查询 id 号为 1234 的作业的作业运行状态信息。
注意事项	1）-u 参数只面向管理员开放 2）普通用户只能查看属于本用户的作业的信息 3）对于处于调度状态（pend）的作业，如果作业长时间处理 pend 状态，可以用 bjobs -l jobid 来查看作业不能调度运行的原因。 4）用户必须在 SN 上使用

A.3.4 作业输出查询

功能	查询作业的 stdout 和 stderr 输出信息
命令格式	bpeek [-h] [-f] jobId
参数说明	-h 显示帮助信息 -f 持续显示作业的输出信息，类似于 tail 的-f 选项 jobId 指定作业的 id 号
使用范例	bpeek 1234 显示 id 号为 1234 的作业的输出信息
注意事项	1）本命令只能用于查询属于本用户的已经运行的作业的输出信息 2）用户必须在 SN 节点上使用

A.3.5 作业联机

功能	当提交的交互作业窗口关闭后，可在新窗口中通过该命令实时显示该作业输出信息，进行联机操作
命令格式	bonline [-h] jobId
参数说明	-h 显示帮助信息 jobid 指定作业的 id 号
使用范例	bonline 1234 对作业 id 号为 1234 的正在运行中的交互式作业进行联机操作
注意事项	1）只能普通用户操作本用户的作业， 2）操作的对象只能是本用户的正在运行状态的交互作业。否则都是无效的，会报错退出。如果作业处于联机状态，则作业输出不再发送至原实时输出终端，只发送到本输出终端 3）对作业联机后，可以在当前联机的会话窗口看到作业的实时输出信息 4）用户必须在 SN 节点上使用

附录 B　VPN 登入步骤

国家超级计算无锡中心的 VPN 入口有电信、联通和移动三条运营商线路，用户可以选择与本地网络运营商相对应的 VPN 入口。客户端支持 Windows、Linux 和 MacOS 操作系统。

B.1 登入步骤

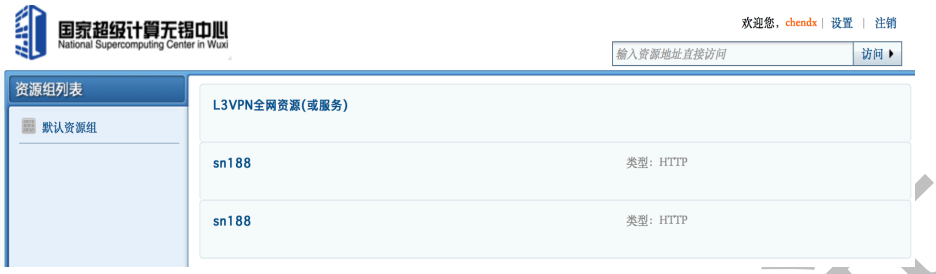
1）使用 WEB 浏览器访问国家超级计算无锡中心官网，网址为：http://www.nsccwx.cn/。

2）在官网首页的右上角选择一条与本地网络运营商相对应线路；

3）输入申请的 VPN 账号和密码：

4) 首次登入会要求安装 SSL VPN 的插件和修改初始密码的页面，用户新密码请自行保存；

5) 登入界面如下图所示，用户就可以通过远程 SSH 直接登入“神威•太湖之光”计算机系统。

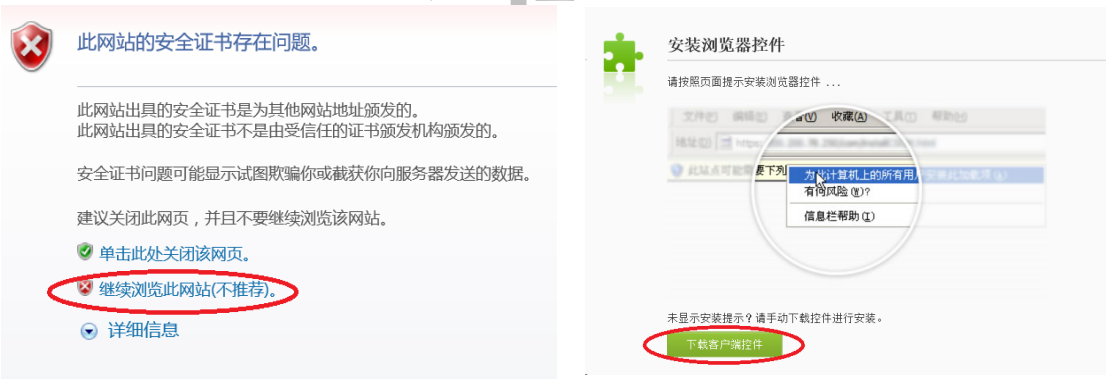


6) 支持 ssh 协议的远程终端有：XSell、SecureCRT、putty 等。

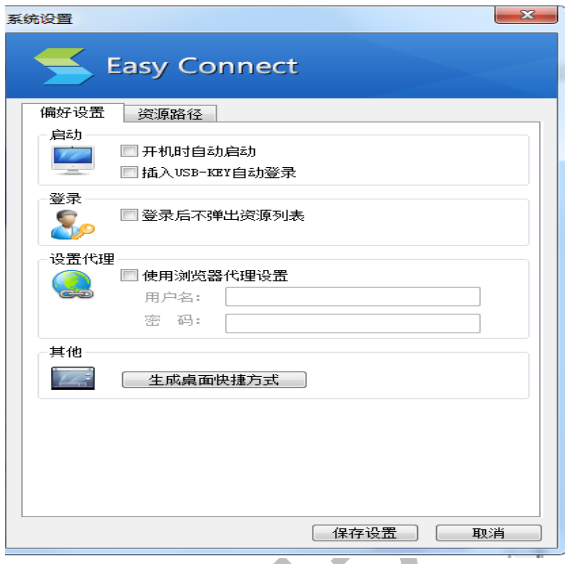
B.2 Windows 连接注意事项

1) 使用 WEB 浏览器访问国家超级计算无锡中心官网，网址为：http://www.nsccwx.cn/。

2) 在官网首页的右上角选择一条与本地网络运营商相对应线路，并选择安全策略和下载客户端控件，如下图所示：



3) VPN 登录成功后，右击 Easy Connect 小图标选择连接状态可以获取一系列 VPN 连接信息，包括虚拟 IP 地址，如下图所示。

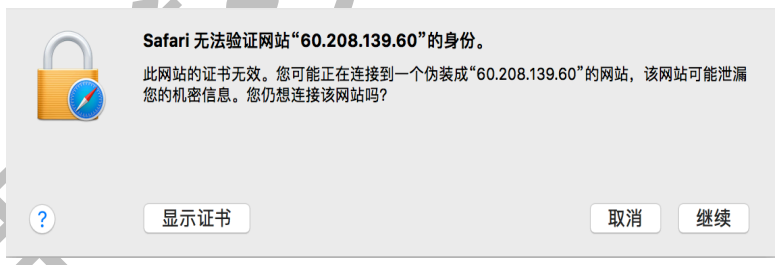


4) 右击 Easy Connect 小图标选择系统设置选项，点击生成桌面快捷方式，可更快捷的登录 VPN 系统，如上图所示。

B.3 MacOS 连接注意事项

1) 使用 Safari 浏览器访问国家超级计算无锡中心官网：http://www.nsccwx.cn/；

2) 当出现提示“Safari 无法验证网站…”，请选择“继续”：



3) 出现提示“访问被拒绝”，如下图，选择“下载安装控件”