实验环境搭建及使用

杜忱莹 周辰霏
2021 年 2 月

周浩 李世阳 麦隽韵
2022 年 2 月

目录

1 SIMD 和多线程实验环境

SIMD（Single Instruction Multiple Data），即单指令多数据运算，是一种常见的并行架构，可方便实现数据并行、提高运算效率。SIMD 硬件结构有三种变体：向量体系结构、多媒体 SIMD 指令集扩展和 GPU。多线程是指从软件或者硬件上实现多个线程并发执行的技术，是为了提高计算资源利用率、提高程序运行效率。支持多线程的硬件包括对称多处理机（SMP）、多核 CPU 等。当前的计算机系统，从超级计算机到桌面 PC，甚至移动终端，普遍提供了对 SIMD 和多线程的支持。例如，x86 CPU 的 SSE/AVX 指令集、ARM CPU 的 Neon/SVE 指令集实现了 SIMD 运算；两种 CPU 目前也都采用多核架构、可很好支持多线程运算。考虑到 ARM 逐渐进入云计算领域的趋势，本学期的课程建议同学们尝试使用 ARM 平台进行实验。

除了硬件支持之外，SIMD 和多线程并行还需要操作系统和编程工具的支持。Linux 是一套免费使用和自由传播的类 Unix 操作系统，是云计算、高性能计算等领域占统治地位的操作系统。在我们未来的科研和工作环境，几乎必然会接触 Linux 系统的使用和开发。因此，希望同学们能基于 Linux 系统进行本课程的实验。Linux 传统的开发方式是通过命令行进行操控（虽然现在可视化开发环境也很成熟了），如果同学们以前没有过相关的经验，希望能够认真查阅一下相关资料，难度曲线可能在一开始会稍陡峭一些，但熟悉之后往往能够比鼠标操控的效率高得多。

GCC 全称为 GNU Compiler Collection，将是我们使用的主要编程工具。GCC 提供对 Pthread 多线程编程和 SIMD 编程的良好支持，因此我们 SIMD 和多线程的实验将主要基于 GCC（其实 MPI 和 GPU 的实验也是）。同时，GCC 也是工业级别工具，也希望借此课程增加同学们对它的了解。下面给出 GCC 编译课程所需要并行程序的编译选项示例：

```
gcc -march=corei7-avx   # SSE/AVX
gcc -pthread            # pthread
gcc -fopenmp            # OpenMP
```

git 较大规模的项目往往需要进行版本控制，比如进行代码回溯、分支管理，git 便是最常使用的工具。你可以在这里对它了解更多。同时为了避免实验环境的损坏等原因导致的意外，希望同学们将代码托管至云端 (如github)。此外，commit 记录也是独立完成作业的一种证明。

make 一个大型项目往往有着复杂的文件依赖关系，编译也是一个耗时的工作。make 便是用于辅助管理复杂的编译的依赖，并指定编译指令的程序。当然，为了程序员的方便，它引入了更多特性如伪文件，方便项目的管理与测试。你可以在这个或这个网站上了解更多。

总结一下，对于课程前期的单机 SIMD 和多线程并行实验，可选择如下几种组合：

- x86 平台、Windows 系统、TDM-GCC 编译器 +Code::Blocks 集成开发环境，这是最简单的一种组合，大家只需在熟悉的 Windows 系统中安装、使用两个轻量级开发工具即可。
- x86 平台、Windows 系统 +WSL、GCC 编译器，在 Windows 系统中直接进行 Linux 开发。
- x86 平台、Windows 系统 +WSL+QEMU、GCC 编译器，在 Windows 系统中模拟 ARM 平台 Linux 系统开发，注意，**性能测试结果并非真实 ARM 平台性能表现**。
- ARM 平台（华为鲲鹏服务器）、GCC 编译器，真正的 ARM 平台 Linux 系统。

接下来几节将介绍这些组合的安装和简单使用。

1.1 x86 平台

x86 架构（The x86 architecture）最早是 Intel 提出的微型处理器架构和指令集，经过三十余年的发展，在目前的个人 PC、服务器乃至超算中，x86 架构占据了很大比例。自 1996 年的 MMX 指令集开始，x86 体系不断完善其 SIMD 指令集，而从 2004 年左右 x86 架构开始支持多核，所以 x86 的硬件平台完全支持 SIMD 和多线程。对于 Windows 平台，推荐同学们使用 TDM-GCC 编译器和集成开发环境 Code::Blocks 进行实验。

1.1.1 编译器 TDM-GCC

下载地址：http://tdm-gcc.tdragon.net/download。

Windows 平台下的一个免费的 GCC 编译套件，包含截止到 9.2.0 版的 GCC 稳定版本 +Windows 平台必要的补丁和运行时 API，支持 32 位和 64 位系统，非常小巧的单文件安装包（57.6MB），安装非常简单，且直接支持 SIMD、多线程等编程，可作为相关实验的编译器。

1.1.2 集成开发环境 Code::Blocks

下载地址：http://www.codeblocks.org/downloads/。

Code::Blocks 是免费开源的跨平台集成开发环境，支持多种编译器：GCC、MSVC++、clang 等。具有完善的调试功能，可配合 TDM-GCC 形成完整开发环境。

- **编译器目录设置**

安装好的 Code::Blocks 需要设置编译器目录。如图1.1所示，选择 “Settings-Complier” 菜单项，在 “Toolchain executables” 表单中填入 TDM-GCC 安装地址和编译程序名即可。

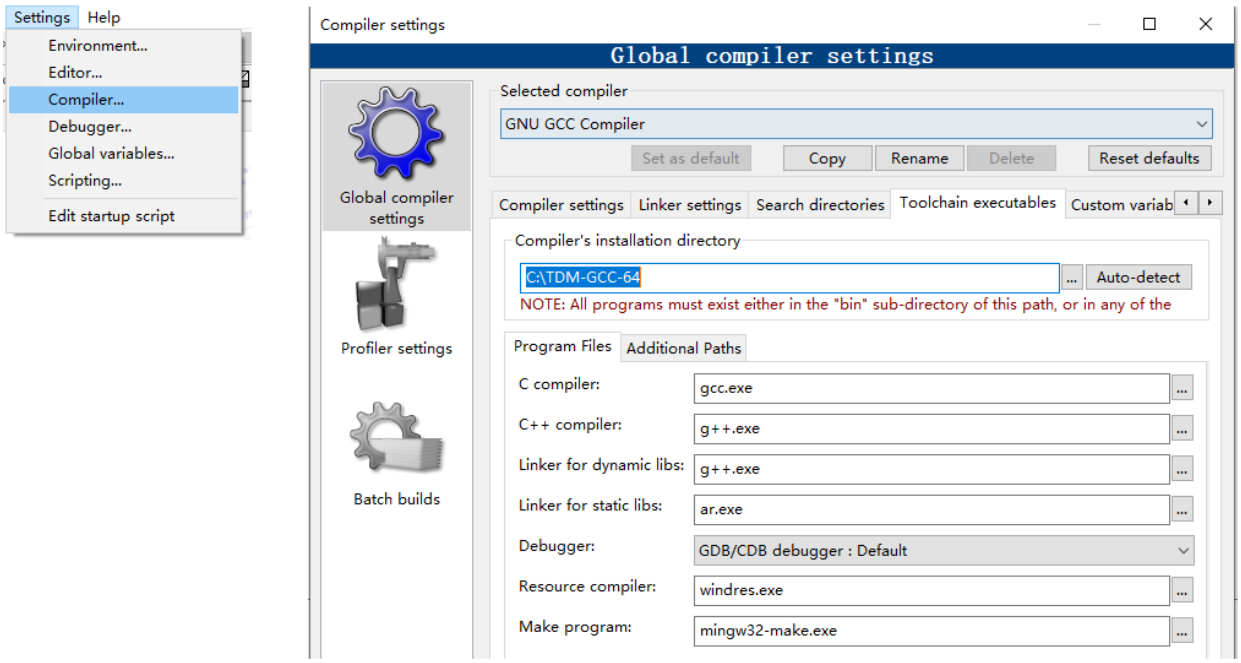

图 1.1: 设置编译器目录

- **创建项目**

创建项目过程如图1.2所示，选择 “File-New-Project” 菜单项，选择创建 “Console Application” 即可（我们的实验基本都是命令行程序），后续过程与 Visual Studio 中创建项目非常相似。

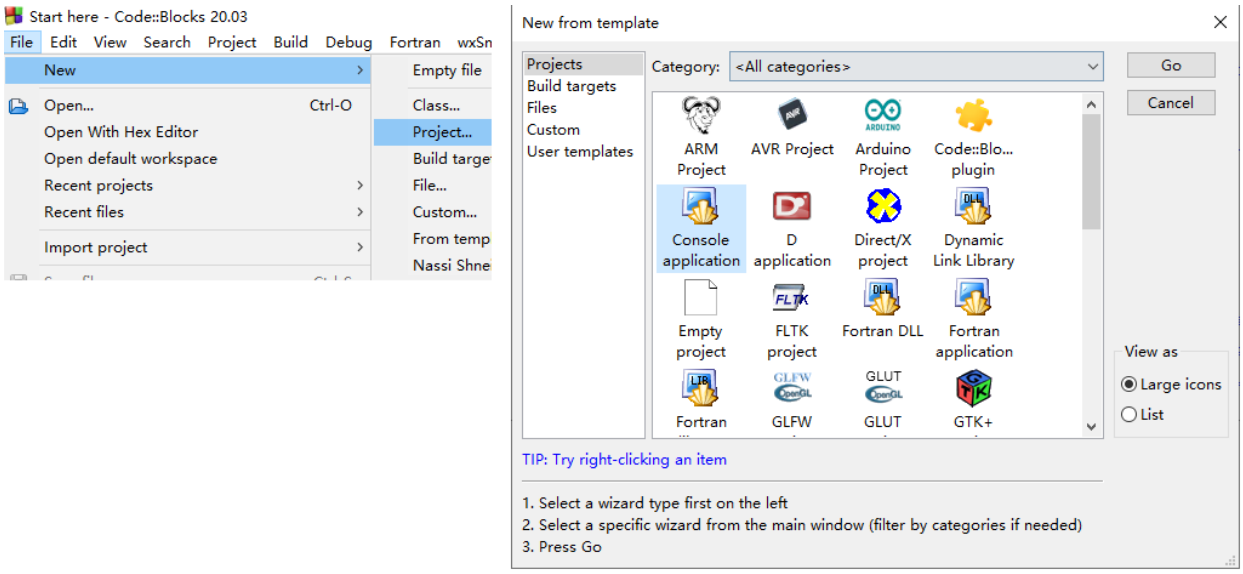

图 1.2: 创建项目

- **编译选项设置**

编译选项设置过程如图1.3所示，选择 “Project-Build options” 菜单项，在打开的对话框中的 “Complier Flags” 表单内勾选（GCC）编译选项。例如，选择 C++ 标准（TDM-GCC 9.2 目前只支持到 C++17）、选择目标程序 32 位或 64 位、选择优化力度、选择体系结构（编译 SSE/AVX 程序需要用到，编译 AVX 程序还需在 “Other Compiler options” 表单的文本框中填写 “-march=native”）等。编译 OpenMP 程序还需在 “Other compiler options” 表单的文本框中填写 “-fopenmp”、并在在 “Linker settings” 表单的 “Other linker options” 文本框中填写 “-fopenmp” 等。

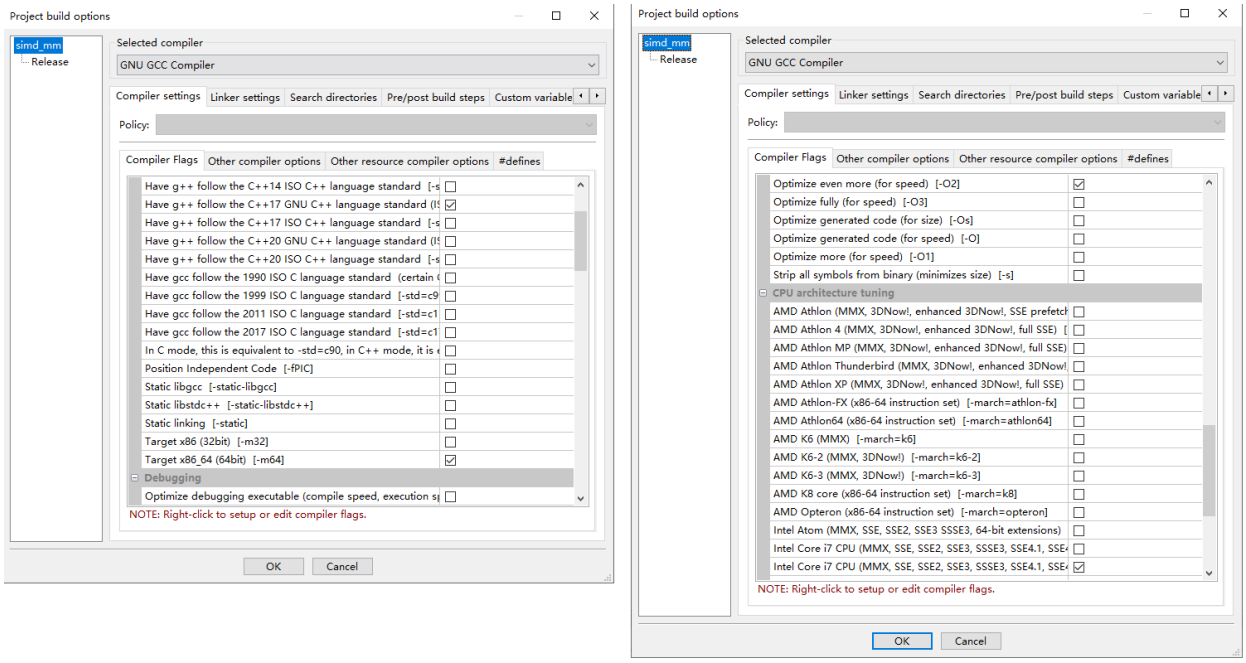

图 1.3: 编译选项

- **编译/运行项目**

接下来即可选择 “Build” 菜单编译和运行项目，以及选择 “Debug” 菜单调试项目，与 Visual Studio 的过程也类似。程序执行结果如图1.4所示。

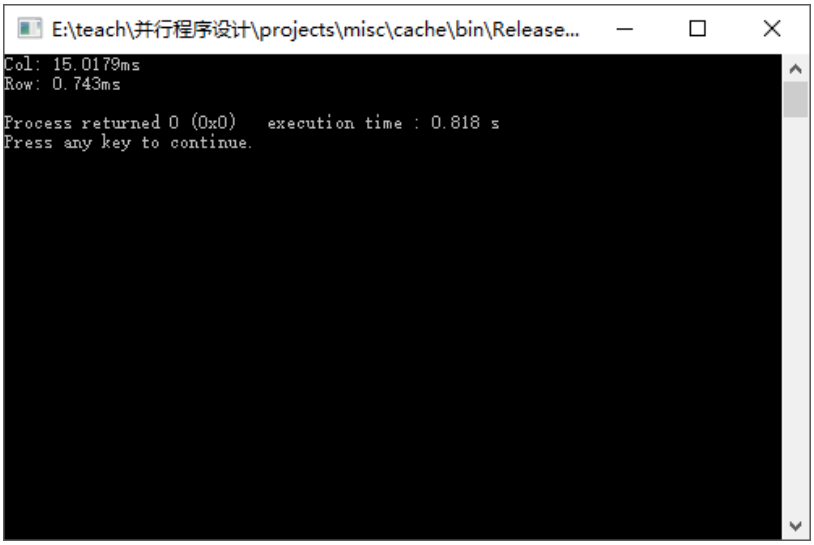

图 1.4: Code::Blocks 运行程序效果

1.2 Linux 系统和 ARM 平台

ARM 架构是进阶精简指令机器 (Advanced RISC Machine) 的简称，最早是一个 32 位精简指令集 (RISC) 处理器架构，目前已发展到 64 位。ARM 由于能耗方面的优势，已广泛地应用于移动终端等嵌入式场景。在保持能耗优势的前提下，ARM 处理器的性能也在不断提升，近年来逐渐进入数据中心领域，取代一部分 x86 平台的应用场景。因此，非常建议同学们在实验中练习 ARM 平台的并行编程，特别是 x86 平台并行效果与 ARM 平台并行效果的对比。

1.2.1 WSL

WSL 是 Windows Subsystem for Linux 的简写，是 Windows 较新版本自带的一个组件，实现了在 Windows 上直接进行 Linux 开发，避免了安装运行虚拟机的复杂性。希望方便地在 Linux 系统中进行 SIMD 和多线程编程实验，即可采用这种方式，并行程序的性能实验结果与在裸机上运行的 Linux 系统中的性能表现差别不大。

WSL 对于系统有一定需求，你可以在这里找到相应说明。你可以通过升级操作系统以满足需求；但若你的机器是 32 位的，则无法使用 WSL。WSL 的安装说明可以在这里找到。你可能会依次遇到这个错误和这个错误。你可能需要升级 WSL kernel，并借助官方工具将 Windows 升级至最新版本。下面的一些软件安装、使用说明都是以在 WSL 上安装了 Unbuntu 20.04 Linux 发布版本为例。

- **WSL 下 SIMD 和多线程程序编译**

```
# 系统更新（定期做即可）和GCC编译器安装
sudo apt-get update                    # 获取Unbuntu系统和软件更新信息
sudo apt-get upgrade                   # 进行系统和软件更新
sudo apt-get install build-essential
sudo apt-get install gcc
sudo apt-get install g++

# 编译SSE/AVX程序，native可以改为本机架构名
g++ -O2 -march=native -o simd_mm simd_mm.cpp
# 编译Pthread程序，新版本GCC中编译选项已改为-pthread
g++ -O2 -pthread -o hello hello.cpp
```

```
# 编译OpenMP程序
g++ -O2 -fopenmp -o hello hello.cpp
```

在 Linux 系统下我们可以使用 QEMU 模拟多种架构，这是一款开源的硬件模拟器。QEMU 支持多种模式，我们的课程中主要用到的是 User mode 和 System mode。在 User mode 即用户模式下，QEMU 可运行不同架构下的可执行文件，功能类似于一个中间层，接受程序的指令并翻译为当前系统能够识别的指令、系统调用等。而在 System mode 即系统模式下，QEMU 会模拟一个完整的计算机系统，包括外围设备等。QEMU 模拟器对 ARM 架构有着良好的支持，借助 QEMU 就可以很好的调试和研究 ARM 环境。在 QEMU 模拟器上实现 ARM 环境主要需要进行如下步骤。

- **安装 QEMU 模拟器**

```
sudo apt-get install qemu
sudo apt-get install qemu-system
sudo apt-get install qemu-user
# whatever you need
# 有可能缺少依赖和库文件，根据提示安装即可
```

- **安装 ARM 的交叉编译工具链**

```
# 实际上我们可以看到有gcc-arm-linux-gnueabi、gcc-arm-linux-gnueabihf和gcc-aarch64-linux-gnu三种格式
# 具体的差异同样推荐同学们自行查阅了解，其中第一种不支持模拟SIMD
sudo apt-get install gcc-arm-linux-gnueabi
sudo apt-get install g++-arm-linux-gnueabi
```

- **WSL 模拟 ARM 平台 SIMD 和多线程程序编译**

```
# 编译Neon程序，cortex-a76可替换成实际架构名
# 如模拟aarch64架构，不用-mfpu选项。模拟方式只能验证程序正确性，不能反映ARM SIMD真实性能
arm-linux-gnueabihf-g++ -O2 -o neon_mm -mcpu=cortex-a76 -mfpu=neon-vfpv4 neon_mm.cpp
# 编译Pthread程序
arm-linux-gnueabi-g++ -O2 -pthread -o hello hello.cpp
# 编译OpenMP程序
arm-linux-gnueabi-g++ -O2 -fopenmp -o hello hello.cpp
# 运行程序
qemu-aarch64 neon_mm    # 模拟64位ARM架构下执行程序
qemu-arm hello          # 模拟32位ARM架构下执行程序
hello                   # 隐含调用了qemu-arm模拟器
```

1.2.2 华为鲲鹏服务器

华为鲲鹏计算产业是基于鲲鹏处理器构建的全栈 IT 基础设施、行业应用及服务。鲲鹏通用计算平台适配各行业多样性计算、绿色计算需求，致力于打造最强算力平台。鲲鹏 HPC（High Performance Computing）提供了超高浮点计算能力解决方案，可用于解决计算密集型、海量数据处理等业务的计算需求，如科学研究、气象预报、计算模拟、军事研究、CAD/CAE、生物制药、基因测序、图像处理等，缩短需要的大量计算时间，提高计算精度。如希望在实际 ARM 平台上评测真实 ARM 并行性能，可采用鲲鹏平台。

在鲲鹏服务器上，SIMD 和多线程程序的编译运行与 WSL+QEMU 环境下相似。

```
# 编译Neon程序
# 在真正ARM机器上运行，实验结果是ARM SIMD真实性能
gcc -O3 -o neon_mm -mcpu=cortex-a76 -mfpu=neon-vfpv4 neon_mm.cpp
# 编译Pthread程序
gcc -O3 -pthread -o hello hello.cpp
# 编译OpenMP程序
gcc -O3 -fopenmp -o hello hello.cpp
```

华为鲲鹏服务器可采用 putty/psftp、MobaXterm（Windows 推荐使用）等工具以 ssh 方式远程登录服务器、传输程序和数据等。**服务器的 ip 为 nbjlservers.icu，端口号为 9001**。每位同学的账户名为 **s+ 学号，例如，s2011111**，初始密码与账户名相同，同学们登录账户后可自行修改密码。

- **bisheng(毕升) 编译器**

bisheng(毕升) 编译器是华为编译器实验室针对鲲鹏等通用处理器架构场景，打造的一款高性能、高可信及易扩展的编译器工具链，增强和引入了多种编译优化技术，支持 C/C++/Fortran 等编程语言。在鲲鹏服务器上已经部署了 bisheng 编译器，在终端输入命令 “clang -v”，就能看到 bisheng 相关的版本提示信息。由于 bisheng 编译器也是基于 clang 开发的，所以使用起来与 clang 几乎没有区别，大多数默认的编译选项也与 clang 相同。经过我们的实测，在鲲鹏服务器上，使用 bisheng 编译出的程序比使用 gcc 编译的程序运行速度更快，在不同优化级别时，bisheng 相比于 gcc 取得的加速比不同，优化级别 O2 时加速比最高。不过我们的测试也可能还不够全面，同学们可以在实验过程当中对两种编译器自己进行尝试和对比。

关于 bisheng 编译器更加详细的使用方法和介绍可以在在这里找到。

基于鲲鹏服务器搭建的集群和基于金山云搭建的虚拟集群都是公共实验环境，通过作业管理系统实现同学们的实验程序编译、提交、资源分配和运行。因此，**登录节点只能进行简单的编译、任务提交等工作，禁止在登录节点直接执行程序**。同学们应通过作业管理系统将自己的实验程序提交，等价作业管理系统分配资源、调度执行完毕后再获取结果，详见下一节 MPI 实验环境的介绍。建议同学们现在本地台式机、笔记本（x86 平台和模拟 arm 平台）进行程序的编写和正确性调试，进行性能评测时再登录两个平台提交任务。

2 MPI 实验环境

MPI 是一种由学术界工业界联合设计的标准化可移植消息传递标准，可在多种并行计算体系结构上运行。MPI 提供了在并行计算机中进行节点通信的协议，支持点对点通信和组通信，其目标是高性能、可伸缩性和可移植性。MPI 直到现在仍是高性能计算中使用的主导编程模型。

2.1 自行搭建

MPI 适用于各种并行计算架构，你可以自行搭建不同系统下的 MPI 实验环境。

2.1.1 MS-MPI

MS-MPI 是 MPI 标准的 Microsoft 实现，用于在 Windows 平台上开发和运行 MPI 程序。它在 Windows 系统上具有着高性能，同时很容易移植使用 MPICH 的现有代码。

你可以在这里下载最新 MS-MPI 代码，并在这里找到安装说明和运行简单测试程序的方法。

2.1.2 OpenMPI

OpenMPI 是一个广泛使用的 MPI 实现，在 Linux 系统或是 WSL 中，你可以直接自行部署 OpenMPI 环境。这里提供了最新版本的 OpenMPI 安装包，参考部署指南进行部署和验证。

2.1.3 MPICH

MPICH 是 MPI 标准的一个重要实现。Windows 版本的 MPICH 由于某些原因不再更新，所以 Win7 及以后的版本都很难成功安装 MPICH，如果你需要使用 MPICH 进行实验，那么强烈推荐你在 Linux 环境或是 WSL 环境下进行部署配置。

你可以在这里获取到适用于各种系统的最新版本。你可以根据这里的安装指南进行安装和使用。

2.2 金山云

2.2.1 作业提交

我们还将基于金山云搭建虚拟机集群供同学们进行 MPI 编程实验，其架构如图2.5所示。

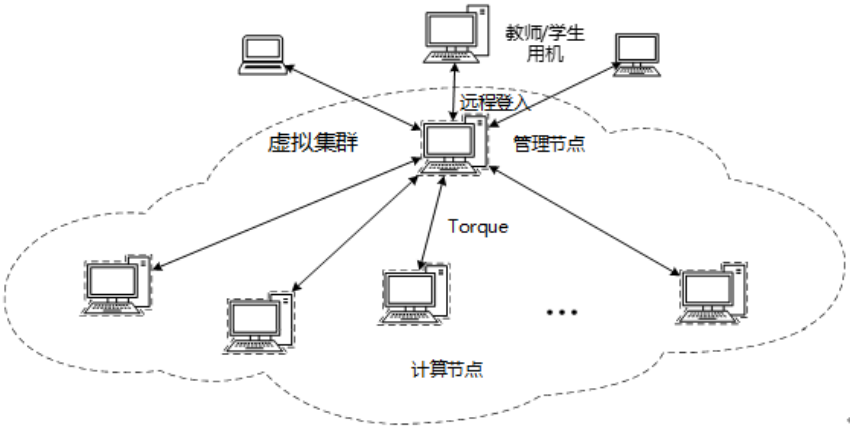

图 2.5: 虚拟机集群

集群的详细信息以及登录方式待搭建完成后更新。

登录管理节点后（**登录方式待更新**），将程序编译成可执行文件（单节点实验使用 gcc 或毕升；MPI 实验使用 mpicc，与自行搭建的 OpenMPI/MPICH 系统中类似），在可执行文件所在目录下输入命令（**禁止在管理节点直接执行程序**）：

```
pbsmpirun 程序名 线程数 程序参数1 程序参数2 ....
```

可以自动生成 pbs 脚本在当前目录下并自动提交，pbs 脚本名称为 “程序名.pbs.sh”。提交后会得到一个作业编号。输入 qstat 可以看到目前队列中作业的状态：

```
Q–正在排队  R–正在运行  C–运行完毕
```

当运行完成后，目录下会多两个文件：“程序名.o 作业编号”，“程序名.e 作业编号”，分别代表标准输出和错误输出。

2.2.2 提交示例

以 MPI 自带的 cpi 程序为例，输入命令

```
pbsmpirun cpi 4
qstat
qstat
```

可以看到如图2.6所示结果：

提交后得到作业 129.master。开始状态为 R，表示正在运行，之后状态为 C，表示运行结束。在程序运行结束后，在当前目录下出现两个文件：cpi.o129、cpi.e129。分别表示 129 作业的标准输出和错误输出。打开 cpi.o129 可以看到程序的标准输出 (图2.7)。

打开 cpi.e129，可以看到程序的错误输出如图2.8。这些是节点间文件同步操作的输出信息，和 cpi 程序无关，是 cpi.pbs.sh 脚本中 pssh 的输出。

```
[pbs@master test]$ pbsmpirun cpi 4
Submit job (excute file = cpi, number of process = 4).
129.master
[pbs@master test]$ qstat
Job ID                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
127.master                 cpi              pbs             00:00:00 C batch
128.master                 cpi              pbs             00:00:00 C batch
129.master                 cpi              pbs                    0 R batch
[pbs@master test]$ qstat
Job ID                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
127.master                 cpi              pbs             00:00:00 C batch
128.master                 cpi              pbs             00:00:00 C batch
129.master                 cpi              pbs             00:00:00 C batch
[pbs@master test]$ ls
cpi  cpi.e129  cpi.o129  cpi.pbs.sh
```

图 2.6: 作业提交

```
Process 0 of 4 is on node4
Process 2 of 4 is on node2
Process 1 of 4 is on node3
pi is approximately 3.1415926544231239, Error is 0.0000000008333307
wall clock time = 0.001947
Process 3 of 4 is on node1
```

图 2.7: cpi.o129

2.2.3 pbs 脚本说明

输入命令 pbsmpirun 后可以得到名为“程序名.pbs.sh”的脚本，并自动提交该脚本。同学们也可以自行编写 pbs 脚本并提交，以 cpi.pbs.sh 为例：

```
#PBS -N cpi          # 任务名称为cpi
#PBS -l nodes=4      # 需要四个节点来计算

pssh -h $PBS_NODEFILE mkdir -p /home/pbs/test 1>&2                              # 在分配到的4个计算节点上创建对应的路径
scp master:/home/pbs/test/cpi/home/pbs/test
pscp .pssh -h $PBS_NODEFILE/home/pbs/test/cpi/home/pbs/test 1>&2                # 获取master节点中的cpi程序，并分发到每个计算节点
/usr/local/bin/mpiexec -np 4 -machinefile $PBS_NODEFILE /home/pbs/test/cpi      # 在4个计算节点上运行mpi程序cpi
```

然后提交任务：

```
qsub cpi.pbs.sh
```

注意到本次虚拟机集群每个子节点最多可计算 2 线程，4 线程使用 2 个子节点即可。

2.3 鲲鹏服务器

2.3.1 pbs 脚本配置

提交任务前需配置脚本，配置内容如下（如单节点非 MPI 实验节点数设置为 1，无须使用 mpiexec，直接执行程序即可）：

```
[1] 21:17:59 [SUCCESS] node4
[2] 21:17:59 [SUCCESS] node3
[3] 21:17:59 [SUCCESS] node2
[4] 21:17:59 [SUCCESS] node1
[1] 21:18:00 [SUCCESS] node4
[2] 21:18:00 [SUCCESS] node2
[3] 21:18:00 [SUCCESS] node1
[4] 21:18:00 [SUCCESS] node3
```

图 2.8: cpi.e129

```
# test.sh
# !/bin/sh
# PBS -N test        # 任务名称为test
# PBS -l nodes=4     # 需要四个节点来计算
pssh -h $PBS_NODEFILE mkdir -p /home/s2011111/test 1>&2 # 在分配到的4个计算节点上创建对应的路径
pscp -h $PBS_NODEFILE /home/s2011111/test/mpihelloworld /home/s2011111/test 1>&2 # 将程序分配给每个计算节点，第一个路径是你的可执行文件
mpiexec -np 4 -machinefile $PBS_NODEFILE /home/s2011111/test/mpihelloworld  # 在4个计算节点上运行mpi程序mpihelloworld
```

注意，对单节点非 MPI 实验，每个任务只能使用 1 个节点（核心）、运行一个进程。

2.3.2 作业提交

我们利用鲲鹏服务器搭建虚拟机集群供同学们进行 MPI 编程实验。登录管理节点，登录用户名和密码均为 s+ 学号，例学号为 2011111，则用户名和密码均为 s2011111。

登陆管理节点后，先使用 mpicc 或 mpic++ 将程序编译成可执行文件（与自行搭建的 Open-MPI/MPICH 系统类似；非 MPI 的单节点实验使用 gcc 或毕升），然后在可执行文件所在目录下输入命令：

```
qsub test.sh
```

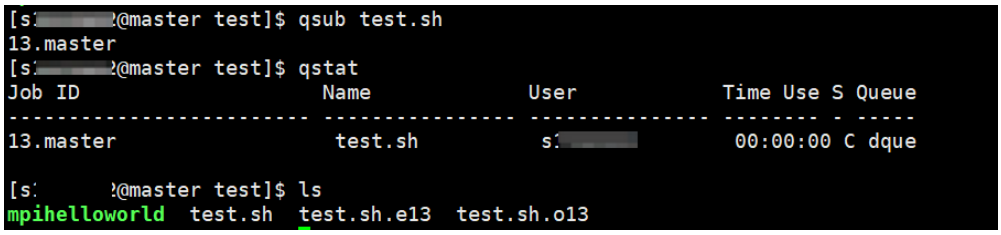

图 2.9: 鲲鹏服务器作业提交

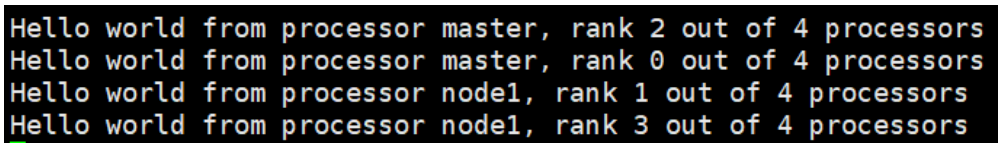

图 2.10: test.sh.o13

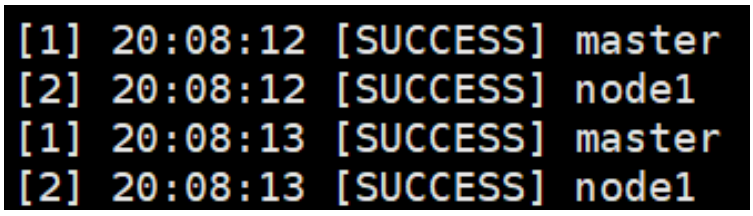

图 2.11: test.sh.e13

提交后会得到一个作业编号，如图2.9, 比如 13.master。输入 qstat 可以看到目前队列中作业的状态：

```
Q-正在排队 R-正在运行 C-运行完毕
```

执行完成后，在当前目录下会生成两个文件，一个是标准输出，文件名为 test.sh.o13（13 是作业编号），如图2.10；另一个是错误输出，包括提示或报错，文件名为 test.sh.e13，如图2.11。

3 GPU 实验环境

3.1 英伟达 CUDA

CUDA（Compute Unified Device Architecture）是显卡厂商 NVIDIA 推出的运算平台，是一种通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。它包含了 CUDA 指令集架构以及 GPU 内部的并行计算引擎。开发人员现在可以使用 C 语言来为 CUDA 架构编写程序。下面介绍在个人电脑上如何进行环境搭建和云平台。

3.1.1 个人电脑环境搭建

在个人电脑上安装 CUDA 首先要满足的硬件条件是：电脑必须要有 NVIDIA 独立显卡。满足此条件之后，按照此方法可以查询电脑 CUDA 支持的版本，然后去官网下载工具包，并按照官网文档中步骤进行安装。官网链接：https://developer.nvidia.com/zh-cn/cuda-toolkit。

3.1.2 英伟达公司提供的 CUDA 云端学习平台

- **网址为：**

待更新

- **课程报名：**如图3.12，在打开的页面点击“Enroll Now”。

图 3.12: 开始页面

- **注册账号：**如图3.13，点击“CREATE AN ACCOUNT”。
- **用优惠码支付：**创建完账户后，在“购物车和付款信息”页点击“输入促销码”，输入优惠码 DLITEACH0221_16_BRMH_58（**待更新**），点击“应用”，即可免去费用。注意：此处虽然不实际支付，但账单地址还是要填，并不真正邮寄。过程如图3.14-3.16所示。

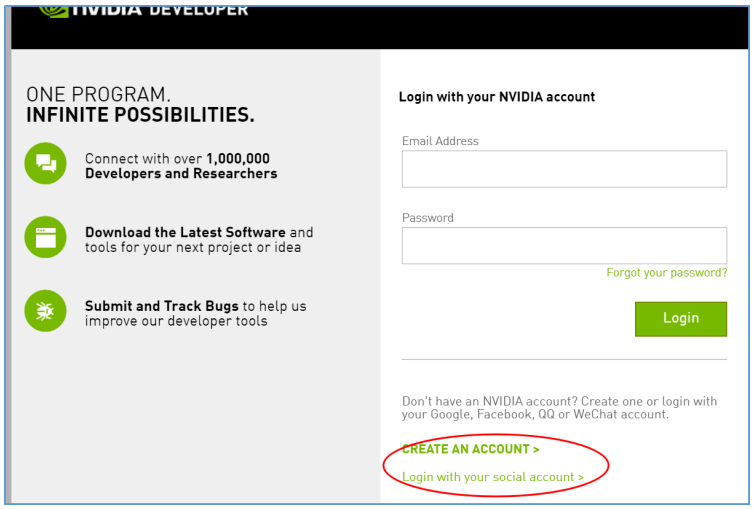

图 3.13: 注册账号

图 3.14: 购物车与付款信息

图 3.15: 购物车与付款信息

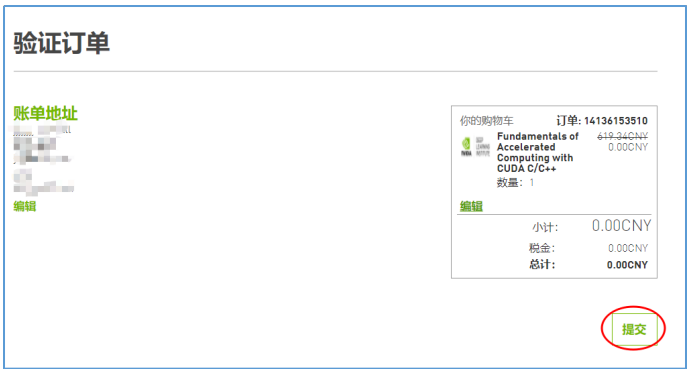

图 3.16: 验证订单

- **进入课程：**在“收据”页 (图3.17)，点击“PROCESS TO COURSES”，进入课程页 (图3.18)。

图 3.17: 收据页面

图 3.18: 课程

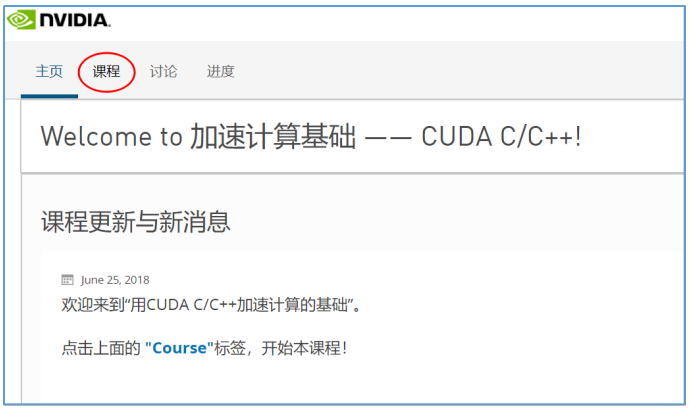

图 3.19: 课程

- **课程学习：**在“课程”标签下，会发现“加速计算基础—CUDA C/C++”课程一共有 3 个小节，需要依次完成每个小节。点击各小节标题可进入相应小节。每个小节进入后，有详细的说明，请大家仔细阅读。有一个类似“播放”的“START”按钮，点击后可加载相应的实验环境，需要等 5-10 分钟左右时间才能加载成功，成功后会显示“2 小时倒计时”和“Launch Task”按钮，点击“Launch Task”按钮，开始本课程的学习。依次学习完 3 个小节后，最后有一个作业任务，完成后可获得相应的学习证书。过程如图3.20-3.22所示。

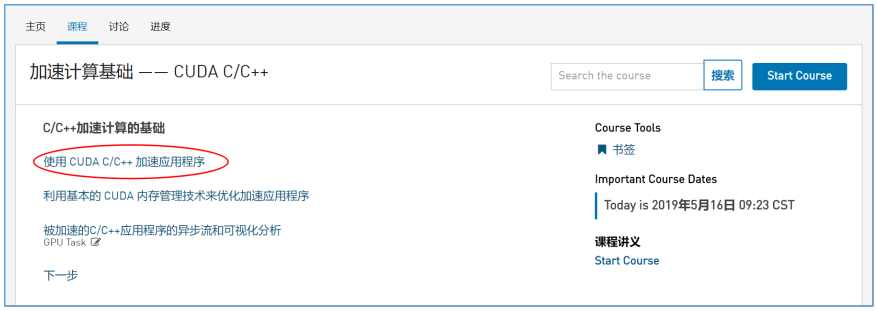

图 3.20: 课程

图 3.21: 课程

图 3.22: 课程

- **编译、执行程序：**点击 "Launch Task"，在新开的窗口中进行学习 (图3.23)，请仔细阅读文中的内容，里边有程序如何编译、执行的说明。

图 3.23: 编译、执行说明

4 Intel DevCloud 平台

4.1 注册

打开注册网址在这里，填写注册信息（Intel 为高校学生学习目的提供 120 天的免费使用时间）。

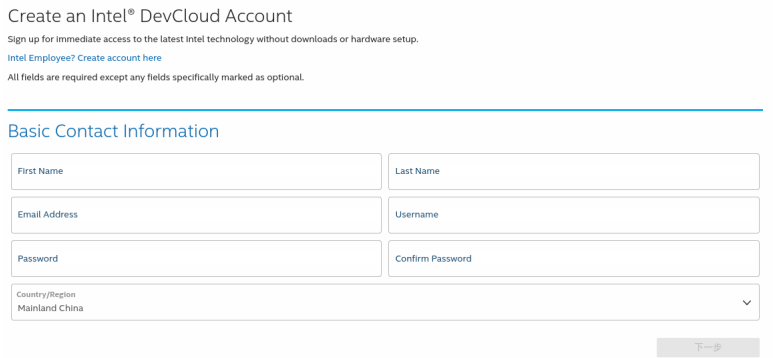

图 4.24: 注册流程：填写个人信息

图 4.25: 注册流程：选择机构

4.2 使用

4.2.1 连接并配置环境变量

Intel 提供了 OneAPI 的学习课程，也提供了与我们的华为鲲鹏实验环境类似的使用方式，前几次 SIMD（包括 AVX-512）、Pthread/OpenMP、MPI 实验使用后一种方式。详细使用介绍的官方文档点击这里。选择合适的连接方式，连接后运行以下代码，配置环境变量：

```
source /opt/intel/inteloneapi/setvars.sh
```

4.2.2 编译

```
# 编译AVX
icpc -O3 -march=native -o avx512_test avx512_test.cpp
# 编译Pthread程序
icpc -O3 -pthread -o pthread_test pthread_test.cpp
# 编译OpenMP程序
icpc -O3 -fopenmp -o omp_test omp_test.cpp
# 编译 MPI 程序
mpiicpc -O3 -o mpi_test mpi_test.cpp
```

4.2.3 运行

提交作业的方式仍然采用 pbs 的方式提交，可以在登录节点上编译代码，但是不要在登录节点上直接运行，pbs 的运行方式如下。

pbs 脚本的内容：

```
# AVX
./avx512_test
# Pthread程序
./pthread_test
# OpenMP程序
./omp_test
# MPI 程序
./mpi_test
```

qsub 提交的方式：

```
# AVX
qsub -l nodes=1:iris_xe_max:ppn=1 -d . job.sh

Note: -l nodes=1:iris_xe_max:ppn=1 (lower case L) is used to assign one full GPU node to the job.
(iris_xe_max 一定不要改动)
```

```
Note: The -d . is used to configure the current folder as the working directory for the task.
Note: job.sh is the script that gets executed on the compute node.

# Pthread程序和OpenMP程序
qsub -l nodes=1:gpu:ppn=2 -d . job.sh

# MPI 程序
qsub -l nodes=4:gpu:ppn=2 -d . job.sh
```

5 性能测试方法

测试程序的性能，我们需要得到程序实际需要的空间和时间，其中编译所需空间时间可以不考虑。计时的机制有：

- 跨平台：clock() / CLOCKS_PER_SEC，精度不高
- Linux：gettimeofday()
- Windows：QueryPerformance

测试方法：(1) 给定问题规模 n; (2) 按需求设计测试数据。

5.1 问题规模

问题规模的确定因素为执行时间和执行次数，即对于一个程序，我们究竟需要测试多少次，每次测试多少数据。以插入排序为例子，其中最坏的情况为 $\Theta(n^2)$，平方函数至少需要 3 个点，并且在测试时需要 3 个以上的 n，n 需要渐进的取到，同时当 n 取值较小时，同一个 n 需要测试多次来取到精确的结果。一种可能的规模设计为：对于 $n = 0 - 100$，精细测试 0, 10, 20，..., 100；对 n>100，稀疏测试 200, 300, 400，...。

5.2 测试数据

测试数据的设计需要仔细思考，如果测试数据过于特别，将无法得到程序真正的性能。以输入排序为例，最坏以及最好的情况下测试数据分别为递减序列和递增序列。但是如果我们需要测试程序的平均情况，这时候需要随机产生大量的测试数据。

5.3 样例

以插入排序为例子，如果我们需要测试插入排序的最坏情况，测试数据就需要是递减序列。

```
void main(void) {
   int a[1000], step = 10;
   clock_t start, finish;
   for (int n = 0; n <= 1000; n += step) {
      // get time for size n
      for (int i = 0; i < n; i++)
         a[i] = n - i; // initialize
      start = clock();
      InsertionSort(a, n);
      finish = clock();
      cout << n << ' '
      << (finish - start)/float(CLOCKS_PER_SEC) << endl;
      if (n == 100) step = 100;
    }
}
```

表 1: 测试结果

n	时间 (s)	n	时间 (s)
0	0	100	0.06
10	0	200	0
20	0	300	0
30	0	400	0
40	0	500	0
50	0	600	0.05
60	0	700	0.06
70	0	800	0.05
80	0	900	0.06
90	0	1000	0.11

测试规模我们取 $n = 10, 20, \ldots, 100, 200, \ldots, 1000$，测试结果如表1所示。

同时我们需要重复测试，从而提高精度。程序可以改为:

```
void main(void)
{
   int a[1000], n, i, step = 10;
   long counter;
   float seconds;
   clock_t start, finish;
   for (n = 0; n <= 1000; n += step) {
      // get time for size n
      start = clock(); counter = 0;
      while (clock() - start < 10) {
         counter++;
         for (i = 0; i < n; i++)
            a[i] = n - i; // initialize
         InsertionSort(a, n);
         }
      finish = clock();
      seconds = (finish - start)/float(CLOCKS_PER_SEC);
      cout << n << ' ' << counter << ' ' << seconds
           << ' ' << seconds / counter << endl;
      if (n == 100) step = 100;}
}
```

这样得到的测试结果中，当 n 取值较小时，重复多次来获得更加精确的结果，测试结果如表2所示。

表 2: 测试结果

n	重复次数	总时间 (s)	每次排序时间 (s)
0	65	0.06	0.000923077
10	294	0.05	0.000170068
20	348	0.06	0.000172414
30	145	0.05	0.000344828
40	120	0.06	0.0005
50	179	0.05	0.00027933
60	141	0.06	0.000425532
70	70	0.05	0.000714286
80	99	0.06	0.000606061
90	60	0.05	0.000833333
100	60	0.06	0.001
200	12	0.05	0.00416667
300	9	0.06	0.00666667
400	4	0.05	0.0125
500	3	0.06	0.02
600	2	0.05	0.025
700	1	0.06	0.06
800	1	0.05	0.05
900	1	0.11	0.11

5.4 Windows 下精确计时

windows 下精确计时可使用 QueryPerformance 系列函数进行计时（需包含 windows.h 头文件），具体如下：

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

using namespace std;

const int N = 10240;              // matrix size

Double b[N][N], col_sum[N];

Void init(int n)          // generate a N*N matrix
{
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      b[i][j] = i + j;
}
int main()
{
    long long head, tail, freq;         // timers
    init(N);
    // similar to CLOCKS_PER_SEC
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    // start time
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for (int i = 0; i < 1000; i++) {
        col_sum[i] = 0.0;
        for (int j = 0; j < 1000; j++)
        col_sum[i] += b[j][i];
     }
    // end time
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout << "Col: " << (tail - head) * 1000.0 / freq
    << "ms" << endl;
}
```