Linux下程序编译与优化

目录

GNU编译器

GCC（GNU Compiler Collection，GNU编译器套件）

- 是一套由GNU开发的编程语言编译器。它是一套以GPL及LGPL许可证所发行的自由软件，也是GNU计划的关键部分，亦是自由的类Unix及苹果电脑Mac OS X 操作系统的标准编译器。GCC（特别是其中的C语言编译器）也常被认为是跨平台编译器的事实标准
- GCC可处理C、C++、Fortran、Pascal、Objective-C、Java，以及Ada与其他语言

C++	g++
Fortran77	gfortran
Fortran90/95	gfortran

Intel编译器

Intel Composer XE 2018

- Intel编译器是Intel公司发布的x86平台（IA32/INTEL64/IA64/MIC）编译器产品，支持C/C++/Fortran编程语言
- Intel编译器针对Intel处理器进行了专门优化，性能优异，在其它x86处理器平台上表现同样出色

编程语言	编译器调用名称
C	icc
C++	icpc
Fortran77	ifort
Fortran90/95	ifort

PGI编译器

PGI Accelerator

- PGI编译器是The Portland Group推出的一款编译器产品，支持C、C++和Fortran
- 此外，PGI编译器还支持HPF（High Performance Fortran，Fortran90的并行扩展）编程语言，支持CUDA Fortran
- 已经被NVIDIA收购

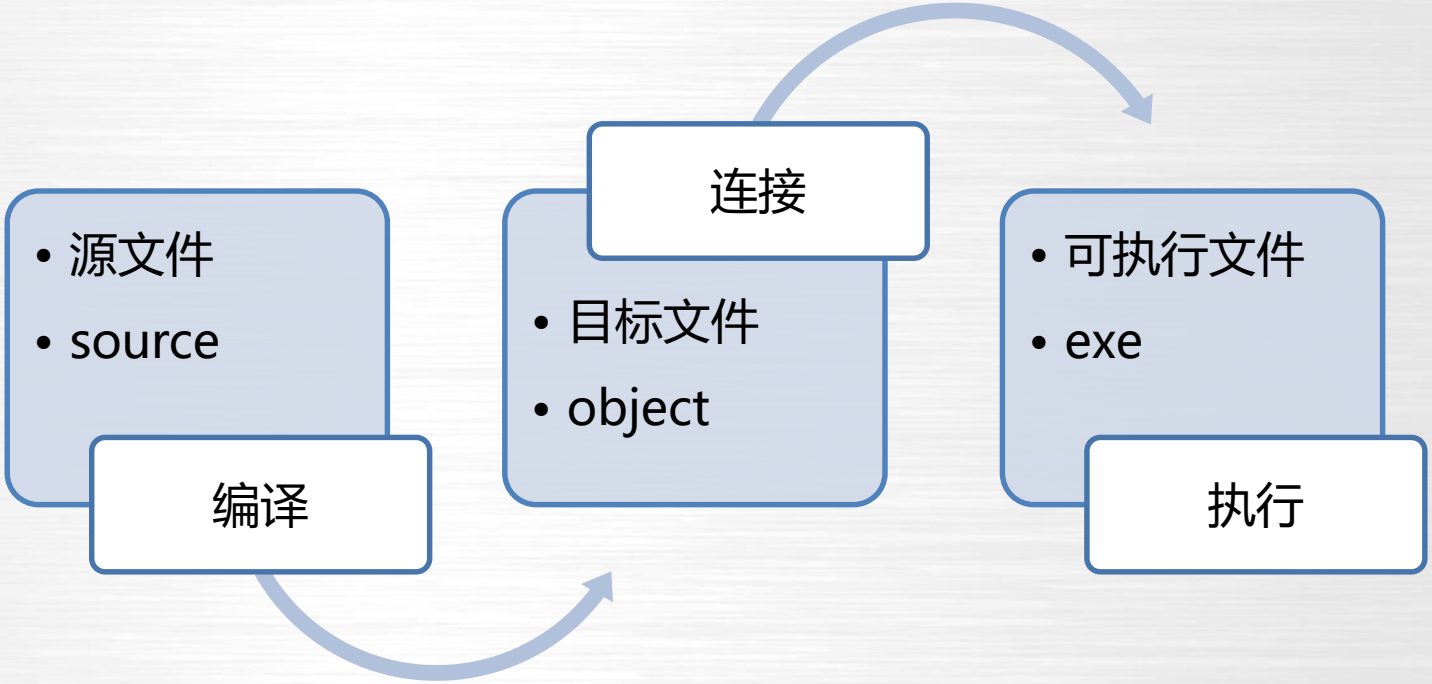
编程语言	编译器调用名称
C	pgcc
C++	pgCC
Fortran77	pgf77
Fortran90/95	pgf90/pgf95
HPF	pghpf

其它x86编译器

- Open64（C、C++、Fortran77/90/95）
- PathScale （C、C++、Fortran77/90/95）
- EKOPath（C、C++、Fortran77/90/95）
- Absoft Fortran Compiler
- g95 Fortran Compiler
- Lahey Fortran Compiler
- …

目录

程序编译流程



源代码后缀规范

- 在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分
- 而源代码、目标文件等后缀名最好保持统一的规范，便于识别区分

文件类型	后缀名
C source	.c
C++ source	.C, .cc, .cpp, .cxx, .c++
Fortran77 source	.f, .for
Fortran90/95 source	.f90
汇编source	.s
目标文件	.o
头文件	.h
Fortran90/95模块文件	.mod
动态链接库	.so
静态链接库	.a

最简单的例子

- hello.c源文件（可用vim等文本编辑器编辑）

```c
#include <stdio.h>
int main()
{
        printf("Hello world.\n");
}
```

- 注1：本文都以C语言示例，C++、Fortran等的编译运行流程与C语言类似
- 注2：示例使用的编译器为gcc，其它编译器的使用方法类似，请参见相关文档或man手册

最简单的例子

- 调用gcc编译源代码，默认在当前目录下生成可执行文件a.out

```
$>gcc hello.c    #如果hello.c不在当前目录，需要输入其路径
$>file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for GNU/Linux 2.6.4, dynamically
    linked (uses shared libs), not stripped
```

- 运行可执行文件
  - 可以在终端中输入可执行文件的相对或绝对路径：

```
$>./a.out
Hello world.
$>/home/test/a.out
Hello world.
```

  - 如果可执行文件所在目录加入了PATH环境变量，可以直接使用可执行文件名

```
$>export PATH=$PATH:/home/test
$>a.out
Hello world.
```

最简单的例子

- 编译时，指定生成可执行文件的路径或文件名（-o参数）

```
$>gcc -o hello hello.c
$>file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for GNU/Linux 2.6.4,
    dynamically linked (uses shared libs), not stripped
$>gcc -o /home/test/hello hello.c
```

最简单的例子

前面的例子中，gcc自动执行了编译和连接操作，这两步可以分开进行：

① 只执行编译，不执行连接（-c参数）

```
$>gcc -c hello.c
```

生成目标文件hello.o

```
$>file hello.o
```

hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

② 连接目标文件，生成可执行文件（编译器实际是调用系统的ld连接器）

```
$>gcc -o hello hello.o
$>file hello
```

hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for GNU/Linux 2.6.4, dynamically linked (uses shared libs), not stripped

多个源文件的例子

主程序源文件 main.c

```c
#include <stdio.h>
int main()
{
        int sum=0,r,i;
        for(i=1;i<=10;i++)
        {
        r=function(i);
        sum=sum+r;
        }
        printf("sum is %d\n",sum);
}
```

子函数源文件 function.c

```c
int function(int x)
{
        int result;
        result=x*x;
        return(result);
}
```

多个源文件的例子

- 多个源文件同时编译

  $>gcc -o sum main.c function.c

  生成可执行文件sum

  $>./sum

  sum is 385

- 源文件分别编译，再将目标文件连接成可执行文件

  $>gcc -c main.c

  $>gcc -c function.c

  $>gcc -o sum main.o function.o

使用头文件的例子

主程序源文件 main.c

```c
#include <stdio.h>
#include "myhead.h"
int main()
{
        printf(STRING);
}
```

头文件 myhead.h

```c
#define STRING "Hello World.\n"
```

- 编译时使用 -I 参数指定头文件搜索路径

  $>gcc -c -I/home/test/include main.c

  $>gcc -o program main.o

  或者

  $>gcc -o program -I/home/test/include main.c

Linux下函数库文件介绍

静态库

- 命名规范为libXXX.a
- 库函数会被连接进可执行程序，可执行文件体积较大
- 可执行文件运行时，不需要从磁盘载入库函数，执行效率较高
- 库函数更新后，需要重新编译可执行程序

动态库

- 命名规范为libXXX.so
- 库函数不被连接进可执行程序，可执行文件体积较小
- 可执行文件运行时，库函数动态载入
- 使用灵活，库函数更新后，不需要重新编译可执行程序

使用头文件

- 源文件中如果引用了头文件，编译器会在自动在一些系统头文件目录中搜索
- 默认搜索的头文件目录一般包括（优先级由高到低）：
  1. 源文件所在目录（要求源文件中用#include "..."格式指定）
  2. INCLUDE之类环境变量指定的目录
  3. 编译器自己的头文件目录
  4. /usr/include操作系统头文件目录

- 如果想自定义头文件搜索路径，可以使用 -I<path> 参数
  - 用 -I 指定的目录优先级高于默认搜索路径
  - -I参数也可以指定多个：-I<path1> -I<path2> ...

库函数的生成（静态库）

子函数 fun1.c

```
int fun1(int i)
{
        return(i+i);
}
```

子函数 fun2.c

```
int fun2(int i)
{
        return(i*i);
}
```

- 编译子函数源代码

  $>gcc -c fun1.c

  $>gcc -c fun2.c

- 使用 ar 命令将目标文件打包成静态库.a

  $>ar cr libtest.a fun1.o fun2.o

库函数的生成（动态库）

子函数 fun1.c

```
int fun1(int i)
{
        return(i+i);
}
```

子函数 fun2.c

```
int fun2(int i)
{
        return(i*i);
}
```

- 编译子函数源代码，必须要使用 -fPIC（Position-independent code）参数

  $>gcc -c -fPIC fun1.c

  $>gcc -c -fPIC fun2.c

- 使用编译器 -shared 参数将目标文件连接成动态库.so

  $>gcc -o libtest.so -shared fun1.o fun2.o

库函数的使用

主函数 main.c，调用之前定义的 fun1 和 fun2 子函数

```
#include <stdio.h>
int main()
{
        int i=10, sum, product;
        sum=fun1(i);
        product=fun2(i);
        printf("the sum is %d, the product is %d\n",sum,product);
}
```

生成的库函数可以直接使用，连接时提供即可，可以通过两种方式：

- 方式一：连接时，直接提供库函数路径

  $>gcc -c main.c

  $>gcc -o program main.o /home/test/lib64/libtest.a

  或者

  $>gcc -o program main.o /home/test/lib64/libtest.so

库函数的使用

- 方式二：使用编译器的 -L<path> -lXXX 参数，表示在指定库函数路径下搜索名为 libXXX.so 或 libXXX.a 的库文件
  - 如果在库函数路径下同时有静态库和动态库，会选择动态库
  - -L可以指定多次，-L<path1> -L<path2>
  - -L指定的搜索路径优先级最高
  - 如果在-L指定的搜索路径中没有找到库函数，或者没有指定-L，编译器还会按优先级从高到低搜索以下路径
    1. LIBRARY_PATH（静态库）、LD_LIBRARY_PATH（动态库）环境变量指定路径
    2. 系统配置文件 /etc/ld.so.conf 中指定的动态库搜索路径
    3. 系统的/lib(64)、/usr/lib(64)等操作系统库文件目录
    4. $>gcc -c main.c

$>gcc -o program main.o -L/home/test/lib64 -ltest

程序运行时动态库的搜索路径

- 可执行程序运行时，动态链接的函数库需要从磁盘载入内存，动态库同样有搜索路径
- 搜索路径优先级从高到低：
  - LD_LIBRARY_PATH 环境变量指定的路径
  - 系统配置文件/etc/ld.so.conf中指定的动态库搜索路径
  - 系统的/lib(64)、/usr/lib(64)等库文件目录

```
$>gcc -c main.c
$>gcc -o program main.o -L/home/test/lib64 -ltest
$>export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/test/lib64
$>ldd ./program
    linux-vdso.so.1 =>  (0x00007fffd3bff000)
    libtest.so => /public/users/libin/test/libtest.so (0x00002b0f21f37000)
    libc.so.6 => /lib64/libc.so.6 (0x00002b0f2216b000)
    /lib64/ld-linux-x86-64.so.2 (0x00002b0f21d16000)
```

编译调试选项

常用的一些编译调试选项（以GNU编译器为例）

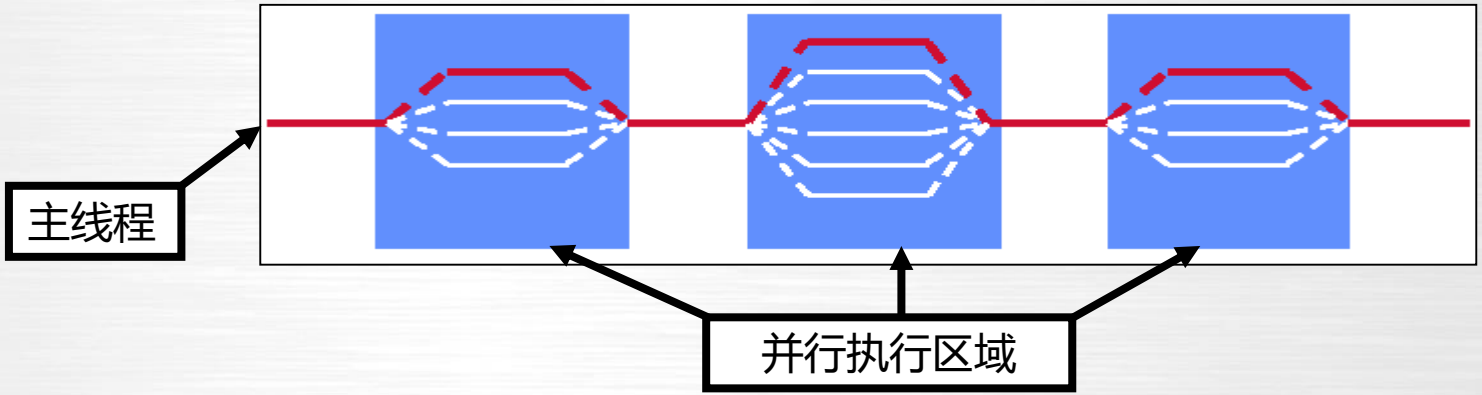
- -Wall，打开编译告警，有助于发现代码不规范的地方和潜在错误
- -g，用于产生调试信息，供 gdb 等调试器使用，-g会自动禁止部分编译优化
- -pg，用于产生profile信息，供 gprof 等分析工具使用

目录

OpenMP简介

OpenMP

- 针对共享式内存的多线程并行编程标准
- 支持C、C++、Fortran等编程语言
- 编译制导（Compiler Directive）型，通常由编译器提供支持



OpenMP程序示例

OpenMP程序 hello_openmp.c

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
        #pragma omp parallel
        printf("Hello World!\n");
```

- 源代码中引用OpenMP头文件
- **#pragma omp** 编译制导语句标识多线程并行执行区域
- OpenMP通过编译器支持，编译时打开OpenMP编译选项即可

编译器	GNU	Intel	PGI
OpenMP编译选项	-fopenmp	-openmp	-mp

OpenMP程序的编译、执行

- 编译（以GNU编译器为例）：

```
$>gcc -o hello -fopenmp hello_openmp.c
$>ldd hello
    linux-vdso.so.1 =>  (0x00007fff789ff000)
    libgomp.so.1 => /usr/lib64/libgomp.so.1 (0x00002b3e5ab8e000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00002b3e5ad97000)
    libc.so.6 => /lib64/libc.so.6 (0x00002b3e5afb4000)
    librt.so.1 => /lib64/librt.so.1 (0x00002b3e5b313000)
    /lib64/ld-linux-x86-64.so.2 (0x00002b3e5a96d000)
```

- 执行，通过环境变量指定线程数运行

```
$>export OMP_NUM_THREADS=2
$>./hello
```

或者

```
$> OMP_NUM_THREADS=2 ./hello
    Hello World!
    Hello World!
```

目录

MPI简介

- MPI（Message Passing Interface）是消息传递函数库的标准规范，不是一种编程语言，支持Fortran和C、C++
- 提供上百个通信函数调用接口，在程序中可以直接调用
- MPI是一种标准或规范的代表，有多个实现版本

MPI实现	支持的网络种类	兼容的MPI规范
MPICH	Ethernet	MPI-1
MPICH2	Ethernet	MPI-2
MPICH-3.x	Ethernet	MPI-3
OpenMPI	Ethernet、InfiniBand、RoCE	MPI-3
MVAPICH	Ethernet、InfiniBand、RoCE	MPI-1，部分MPI-2
MVAPICH2-1.x	Ethernet、InfiniBand、RoCE	MPI-2
MVAPICH2-2.x	Ethernet、InfiniBand、RoCE	MPI-3
Intel MPI	Ethernet、InfiniBand、RoCE	MPI-2

MPI程序示例

MPI 程序 hello_mpi.c

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv)
int argc;
char *argv[];
{
        int rank, size;
        MPI_Init(&argc, &argv); /* start MPI */
        MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get process id */
        MPI_Comm_size(MPI_COMM_WORLD, &size); /* get NO. of processes */
        printf("Hello world from process %d of %d\n", rank, size);
        MPI_Finalize(); /* stop MPI */
        return 0;
}
```

MPI程序的编译

MPI实现版本都会提供相应的编译命令：

编程语言	C	C++	Fortran77	Fortran90/95
编译器调用	mpicc	mpicxx	mpif77	mpif90

- 这些编译命令只是一层封装，实际会调用系统编译器，编译时自动添加MPI头文件搜索路径，连接时自动连接MPI库文件，方便使用

  以 OpenMPI 的 mpicc为例：

```
$>mpicc -show
gcc -I/public/software/openmpi-gnu/include -pthread -L/public/software/openmpi-gnu/lib -lmpi
-lopen-rte -lopen-pal -ldl -Wl,--export-dynamic -lnsl -lutil -lm -ldl
```

MPI程序的编译示例

- 使用 OpenMPI 的 mpicc 编译示例程序 hello_mpi.c

```
$>mpicc -o hello hello_mpi.c
$>ldd hello
linux-vdso.so.1 =>  (0x00007fff20dff000)
libmpi.so.0 => /public/software/openmpi-gnu/lib/libmpi.so.0 (0x00002b7397ef7000)
libopen-rte.so.0 => /public/software/openmpi-gnu/lib/libopen-rte.so.0 (0x00002b739819c000)
libopen-pal.so.0 => /public/software/openmpi-gnu/lib/libopen-pal.so.0 (0x00002b73983e8000)
libdl.so.2 => /lib64/libdl.so.2 (0x00002b739868f000)
libnsl.so.1 => /lib64/libnsl.so.1 (0x00002b7398893000)
libutil.so.1 => /lib64/libutil.so.1 (0x00002b7398aab000)
libm.so.6 => /lib64/libm.so.6 (0x00002b7398caf000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00002b7398f05000)
libc.so.6 => /lib64/libc.so.6 (0x00002b7399122000)
/lib64/ld-linux-x86-64.so.2 (0x00002b7397cd6000)
```

MPI程序的运行

MPI实现版都会提供**MPI程序的进程启动器**，不同进程启动器的名称和语法略有不同：

MPI实现	启动器名	指定进程数	指定运行节点
MPICH	mpirun	-np	-machinefile
OpenMPI	mpirun	-np	-machinefile -hostfile
MVAPICH	mpirun_rsh	-np	-hostfile
MPICH2 MVAPICH2 Intel MPI	mpiexec.hydra	-np -n	-f

MPI程序的运行示例

- 以OpenMPI为例，在单节点运行示例MPI程序：

```
$>mpirun -np 4 ./hello
Hello world from process 2 of 4
Hello world from process 0 of 4
Hello world from process 1 of 4
Hello world from process 3 of 4
```

- 多节点运行

```
$>mpirun -np 4 -machinefile ./hosts.list ./hello
$>cat hosts.list
node1
node1
node2
node2
```

（本例中，启动4个进程运行hello程序，每个节点2个进程）

目录

编译级优化的范围

确认你要优化的应用是否可以获得源代码。

物理化学材料：大部分可获得，除了商业应用gaussian和MS。

CAE：主流应用基本均为商业应用。

生命科学：除少量商业应用，基本均为开源应用。

气象环境科学：基本均为开源应用。

地震资料处理：基本均为商业应用。

动漫渲染：基本均为商业应用。

编译优化的原理

- 找出而执行文件生成的各个环节。
- 从每个环节尽可能的提高代码的执行效率。
- 对于高性能应用，编译优化一般从以下几个环节入手
  - 编译器及编译选项的选择
  - 依赖的数学库的选择
  - MPI的选择

常用编译优化选项

- 选择合适的编译器及优化选项,尽量使用最新版本编译器

- Intel编译器，
  通用的性能优化-O2/-O3,-fast（相当于-ipo,-O3,-no-prev-div,-static和-xHost)
  快速除法和开方-fp-model fast=2 –no-prev-div/-no-prev-sqrt
  实现指令级向量化操作
  xHOST,-xCORE-AVX2,-xCORE-AVX512
  降低Cache Miss Rate:-fno-alias..
  高级优化选项：-ip/-ipo,-prof-gen/-prof-use
  其他常用选项：-static -unroll(n) -recursive

常用编译优化选项

- 选择合适的编译器及优化选项,尽量使用最新版本编译器

Level	Description
-O0	No Optimization
-O1	Optimization without code size increase
-O2	Most common optimization, vectorization, loop unrolling, function call inclined
-O3	Advanced optimization, loop fusion, interchange, cache blocking, loop split

- -O2为默认编译优化选项，-O3为-O2基础上增加更激进的优化，比如循环和内存读取转换和预取等，但是在某些情况下会导致程序运行速度变慢，建议在具有大量的浮点计算和大数据处理的循环时的程序使用。

快速除法和开方

- –no-prec-div/sqrt -fp-model fast=2

如果不要求高精度，可使用编译选项 -fp-model fast=2 –no-prev-div –no-prev-sqrt替代-ft-model precise进行编译，大胆使用浮点模型，从而进一步优化浮点数（不太安全）。编译器实施更快速、精度较低的平方根和除法运算。

针对天气预报模式WRF（X86或者X86_64平台），使用 -fp-model fast=2 –no-prev-div –no-prev-sqrt相比默认编译选项，性能可优化3-5%

实现指令级向量化操作

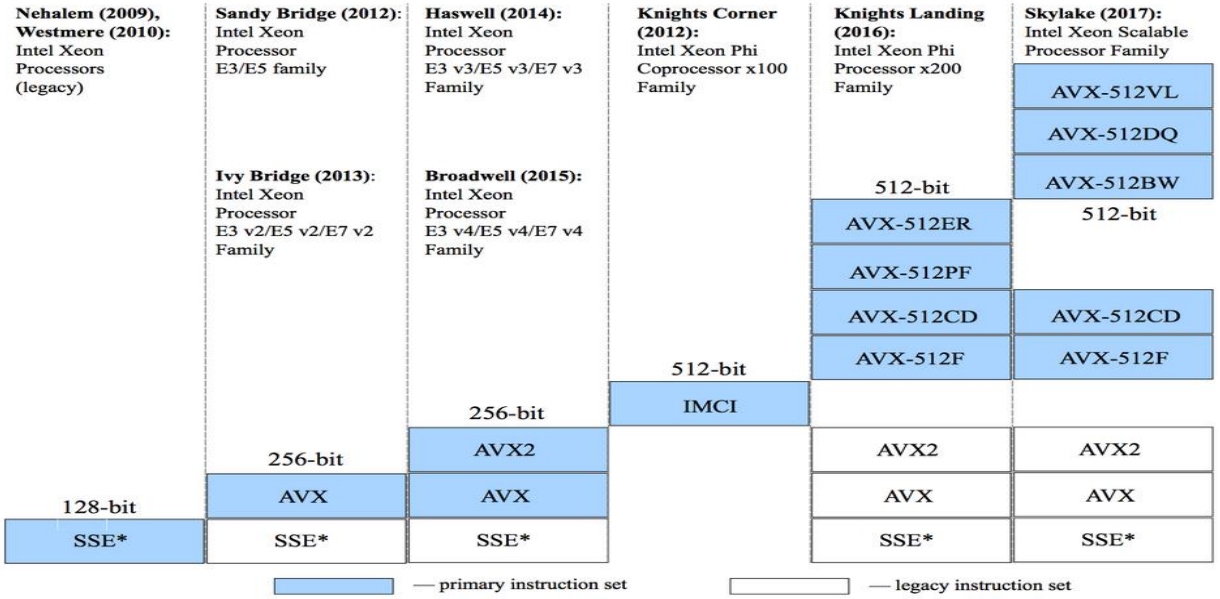


Figure 1: Instruction sets supported by different architectures.

Compiler	Intel compilers 17.x	Intel compilers 18.x
Cross-platform	`-xCommon-AVX512`	`-xCommon-AVX512`
SKL processors	`-xCore-AVX512`	`-xCore-AVX512 -qopt-zmm-usage=high`
KNL processors	`-xMIC-AVX512`	`-xMIC-AVX512`

过程间分析

- -IP：在单个文件中进行过程间优化（Interprocedural Optimizations ）
- -IPO[n]:在多个文件中进行过程间优化，n为可产生的目标文件数，为非负整数

导向优化（Profile Guided Optimization）

提供运行时反馈，以指导数据和代码布局策略，从而提高了指令缓存效率、分页和分支预测效率

- 使用编译选项-prof-gen -prof-dir=/tmp/profdata：其输出为辅助性可执行文件，在可执行文件中插入了相关的代码，-prof-dir是存储性能分析文件的目录
- 运行编译好的程序，运行过程中，产生dyn动态信息文件（dynamic profile information)
- perfmerge –prof_dir /tmp/profdata生成汇总文件,profmerge 工具将多个dyn工具合并到一个dpi文件中
- 重新编译程序，使用选项-prof-use=nomerge –prof-func-groups –prof-dir=/tmp/profdata，生成的文件即为优化后的可执行程序

依赖的数学库的优化

- 常用高性能应用软件依赖的数学库。
  - Blas、lapack、blacs，scalapack数学库

    线性方程组相关库函数，在物理化学材料应用领域广泛使用。在分子动力学应用领域也有使用
  - FFT数学库。

    傅里叶变化相关库，是分子动力学等领域主要使用的库函数。
  - 其它数学库。

    netcdf、jasper及其它数学库。

Blas相关库的选择

- BLAS库，基本线性代数库(Basic Linear Algebra Subroutines），提供最基本的线性代数函数接口。BLAS分为三级：BLAS 1（Level 1）向量与向量操作、BLAS 2（Level 2）：矩阵与向量操作、BLAS 3（Level 3）：矩阵与矩阵操作。
- BLAS库为高性能计算最为基础的库函数，BLAS库的选择的好坏，甚至可以影响应用20%-50%的性能。
- 目前一般编译的过程中，使用优化的blas库，主要有如下几种。
  - **MKL**
  - ACML
  - **Gotoblas**
  - Atlas

基本线性方程组相关库的选择

- 对于Intel平台，MKL依然是最佳性能。
- 对于这一代的AMD平台，gotoblas依然是性能最佳。
- 数学库的选择和CPU的架构、应用的具体类型息息相关，具体的最佳性能还要具体的测试。

MPI的选择

- **MPI类型**
  - OpenMPI
  - Intel mpi
  - mpich2
  - Mvapich2
- **网络类型**
  - TCP
  - IB only
  - IB+sm(shared memory)
  - IPoIB
- **IMB测试函数**
  - 单点通信
    - PingPong
  - 集合通信
    - Bcast
    - Gather
    - Scatter
    - Allgather
    - Alltoall
    - Reduce
    - Reduce_scatter
    - Allreduce
    - Barrier

测试结果——IMB

- **结果总结：**
  - 单点通信：
    - 相比以太网络，IB网络在通信的延时和带宽上均提升较大，而IPoIB仅提升了带宽，延时未明显变化
    - Intel MPI配IB网络在在所有单点测试中性能最佳，带宽达到3196 MB/sec
  - 集合通信：
    - Intel MPI在配IB网络加共享内存时，各MPI函数测试能达到最佳性能，mvapich2和openmpi配IB+sm性能次之。Openmpi的tcp网络性能较差。
    - 共享内存通信方式保证单个节点内的所有进程都通过内存来进行网络通信，由于内存的访问性能高于目前的网络设备，同时，内存的通讯降低了网络设备的负载，从而提高了应用的性能。

不同MPI选择通信网络

MPI类型		MPI命令形式
Intel MPI	IB+shared memory	mpirun … -env **I_MPI_DEVICE** rdssm
	IB	mpirun … -env **I_MPI_DEVICE** rdma
	TCP	mpirun … -env **I_MPI_DEVICE** sock
	IPoIB	mpirun … -env **I_MPI_DEVICE** sock –env **I_MPI_NETMASK** ib
Open MPI	IB+shared memory	mpirun … --mca btl self,sm,openib
	IB	mpirun … --mca btl self,openib
	TCP	mpirun … --mca btl self,tcp **--mca btl_tcp_if_include eth0**(若不指定会使用以太网+IPoIB的混合方式)
	IPoIB	mpirun … --mca btl self,tcp --mca btl_tcp_if_include ib0

不同MPI选择通信网络

MPI类型		MPI命令形式
MPICH2	TCP	修改hosts文件，各节点使用以太网IP地址或对应hostname
	IPoIB	修改hosts文件，各节点使用IB的IP地址或对应hostname
MVAPICH2	IB	mpirun_rsh -ssh -np # -hostfile hosts ./program

MPI进程绑定方式-OPENMPI

目前有两种进程绑定策略供选择：命令行形式和rankfile形式。

1. 命令行形式

常用的进程绑定选项如下：

- --bind-to-socket　　按socket绑定进程
- --bind-to-core　　按core绑定进程
- --bind-to-none　　不设进程绑定（默认）
- --bysocket　　尽量利用连续的socket（与--bind-to-socket配合使用）
- --bycore　　尽量利用连续的core（与--bind-to-core配合使用）
- --report-bindings　　报告进程绑定信息

MPI进程绑定方式-OPENMPI

2. rankfile形式

OpenMPI可以通过rankfile对进程绑定进行更加详细的设置，rankfile规定了每个进程（即rank，从0开始编号）在节点及CPU上的分布情况，文件格式如下：

```
$ cat rankfile
rank 0=node0 slot=2
rank 1=node1 slot=4-7,0
rank 2=node2 slot=1:0
rank 3=node3 slot=1:2-3
$ mpirun -np 4 -hostfile hostfile --rankfile rankfile ./program
```

- rankfile说明：

一行一个rank，后面跟执行该rank的节点名及slot-list

slot-list为执行该rank的CPU基本单元编号，从0开始。该编号由BIOS和操作系统共同决定。也可以按socket:core形式给出。

- 详情参考：http://www.open-mpi.org/doc/v1.4/man1/orterun.1.php

MPI进程绑定方式-MPICH2

MPICH2提供了丰富的进程绑定选项，既有基于命令行的选项设定，也有基于hostfile文件的设置。

1. 绑定选项

最简单的方法是利用-binding选项，其中又包含了基于CPU和Cache的两种情况。基于CPU的绑定最常用：

- -binding cpu　　#绑定到cpu处理单元
- -binding cpu:sockets　#绑定到cpu sockets
- -binding cpu:cores　　#绑定到cpu cores
- -binding cpu:threads　#绑定到cpu线程

MPI进程绑定方式-MPICH2

修改hostfile文件

MPICH2通过在hostfile里的绑定选项（binding），可以控制进程在节点内的分布情况。如下所示：

- $ cat hostfile

```
node0:4  binding=rr                    #round-robin方式绑定
node1:4  binding=user:0,-1,-1,3        #绑定0和3进程，1和2进程不绑定
node2:4  binding=user:0,1,2,3          #绑定所有4个进程
```

binding=rr表示系统按照round-robin方式在CPU处理核心上分布进程，

binding=user表示用户自定义，user后面跟着绑定的进程编号，不写或-1表示不绑定。

- 详情参考：
  http://wiki.mcs.anl.gov/mpich2/index.php/Using_the_Hydra_Process_Manager

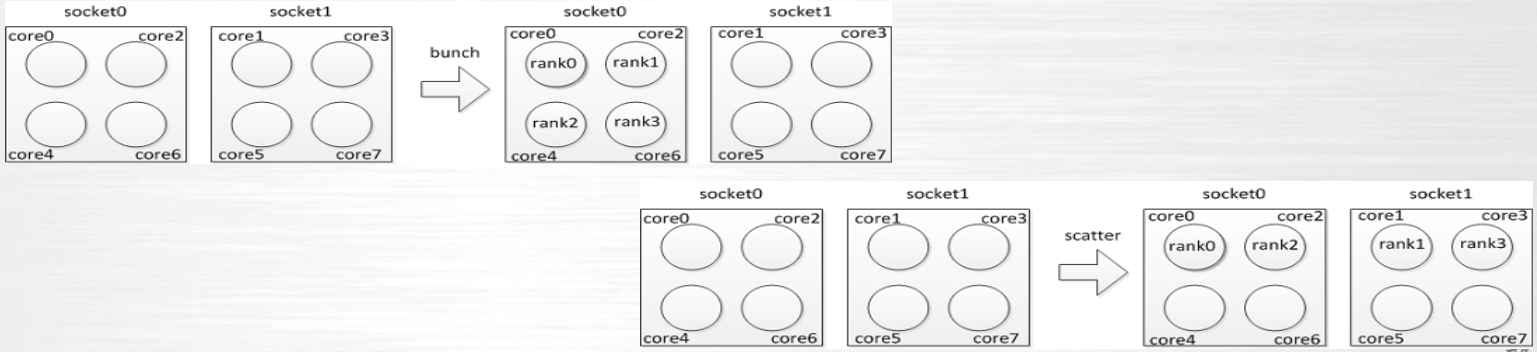
MPI进程绑定方式-MVAPICH2

- MVAPICH2的进程绑定是通过HWLOC（Portable Hardware Locality）程序包实现，HWLOC默认已经安装。有两种设置方式：
- **1. MV2_CPU_BINDING_POLICY变量设置**

MV2_CPU_BINDING_POLICY=bunch：聚集绑定，进程尽可能在一个物理CPU

MV2_CPU_BINDING_POLICY=scatter：分散绑定，进程尽可能在多个物理CPU

下面两幅图形象地说明了bunch（上）和scatter（下）两种绑定方式在一个节点跑4个进程时的区别：



MPI进程绑定方式-MVAPICH2

2. MV2_CPU_MAPPING变量设置

- MV2_CPU_MAPPING变量指定参与计算的CPU核心编号，如：

$ cat hostfile

node0:4

node1:4

- $ mpirun_rsh -ssh -np 8 -hostfile hosts MV2_CPU_MAPPING=0:1:2:3 ./program
- 在每个节点上，4个进程都绑定到编号为0，1，2，3的四个CPU核心上。
- 此外，MVAPICH2也可以使用hydra管理器中的进程绑定策略，参见MPICH2。更多运行方式参考：
- http://mvapich.cse.ohio-state.edu/support/user_guide_mvapich2-1.7rc1.html

MPI进程绑定方式-IntelMPI

IntelMPI默认启动了进程绑定（I_MPI_PIN=enable）,有两种常用的设置方法：

1. I_MPI_PIN_DOMAIN变量

- -env I_MPI_PIN_DOMAIN core　　绑定CPU核心
- -env I_MPI_PIN_DOMAIN sock　　绑定CPU socket
- -env I_MPI_PIN_DOMAIN node　　绑定节点

此外，还可以按照cache进行绑定。

MPI进程绑定方式-IntelMPI

2. I_MPI_PIN_PROCESS_LIST变量

I_MPI_PIN_PROCESS_LIST可以提供比I_MPI_PIN_DOMAIN更多的控制选项，用来指定节点内部进程在CPU上的分布顺序。I_MPI_PIN_PROCESS_LIST又有三种指定形式：

- 显式声明绑定列表（推荐使用），如：

I_MPI_PIN_PROCESS_LIST=0,1,2,3，则节点上的0，1，2，3进程分别绑定到0，1，2，3的CPU核心上。

- 进程映射策略，如：

I_MPI_PIN_PROCESS_LIST=allcores:grain=core将按核心数依次循环绑定进程，该方法设置复杂，适合特别细化的进程绑定情况，一般很少用到。

- 预设场景策略，包括scatter和bunch两种情况，见MVAPICH2进程绑定说明。
- 详情参考Intel MPI Reference Manual。

计算 决定未来

谢谢