# 集群计算

朱虎明
西安电子科技大学
zhuhum@mail.xidian.edu.cn
网安大楼CII 1008

# Contents


1 集群计算基础

2 集群软件

3 多线程

4 OpenMP

5 MPI

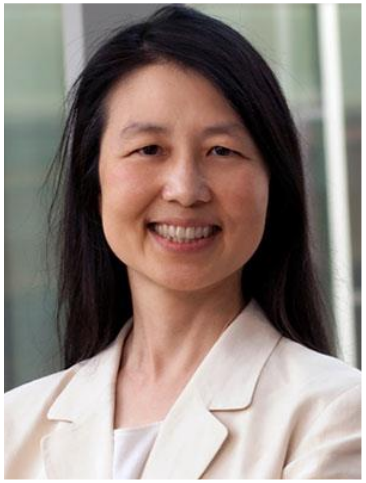
# 高性能计算机--体系结构

并行计算机

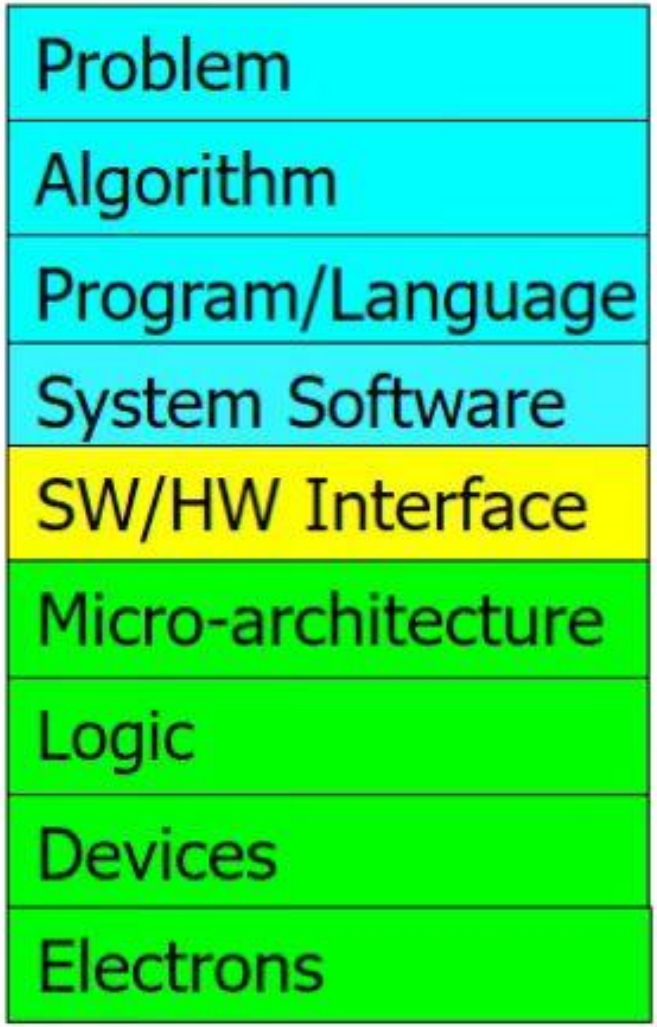
- 多核处理器　　multi-core processor
- 共享存储的对称多处理器系统 SMP
- 分布存储的大规模并行处理系统(MPP)
- 计算机集群系统(Cluster)
- 异构计算集群系统（GPU、AI芯片、FPGA等）

# 高性能计算机—计算思维

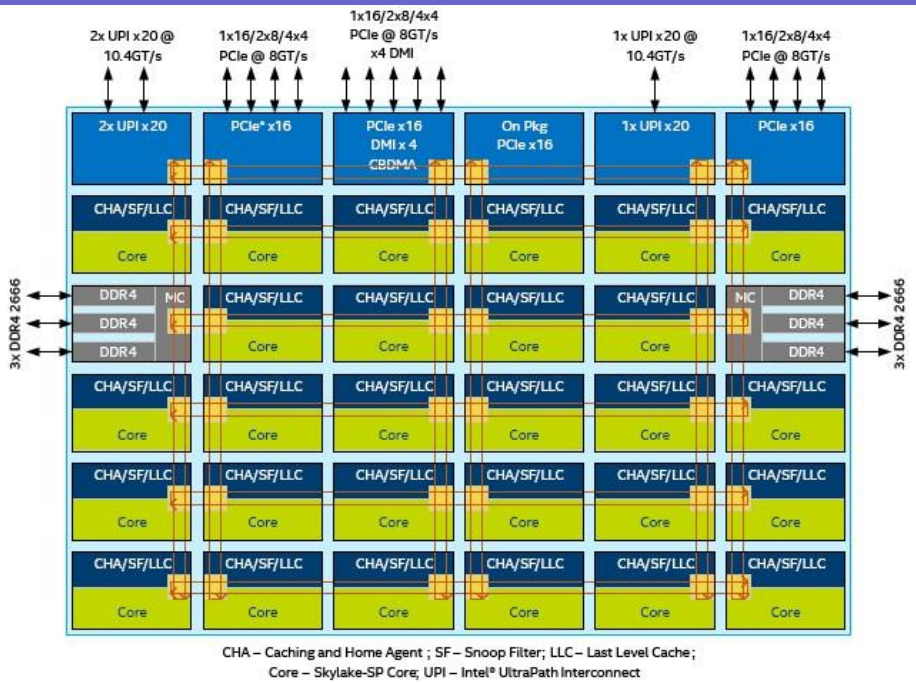
计算思维指的是一种解决问题的思维过程，是能够清晰、抽象地将问题和解决方案用信息处理代理(机器或人)所能有效执行的方式表述出来。



哥伦比亚大学副校长周以真教授



# X86多核处理器






- 9282，14 nm
- 6通道DDR4-2666
- AVX-512 FMA 单元数
- 128 GB/s，56 (112)
- **400W**，$ 10000
- 2.7TFLOPS @ 3.4GHz

- EPYC **9654**
- Transistors:78,840M
- 5nm
- 460 GB/s
- 96
- 360 W
- 5.4T FLOPS
- 11,805 USD

# ARM多核处理器

**1、华为鲲鹏，64核**

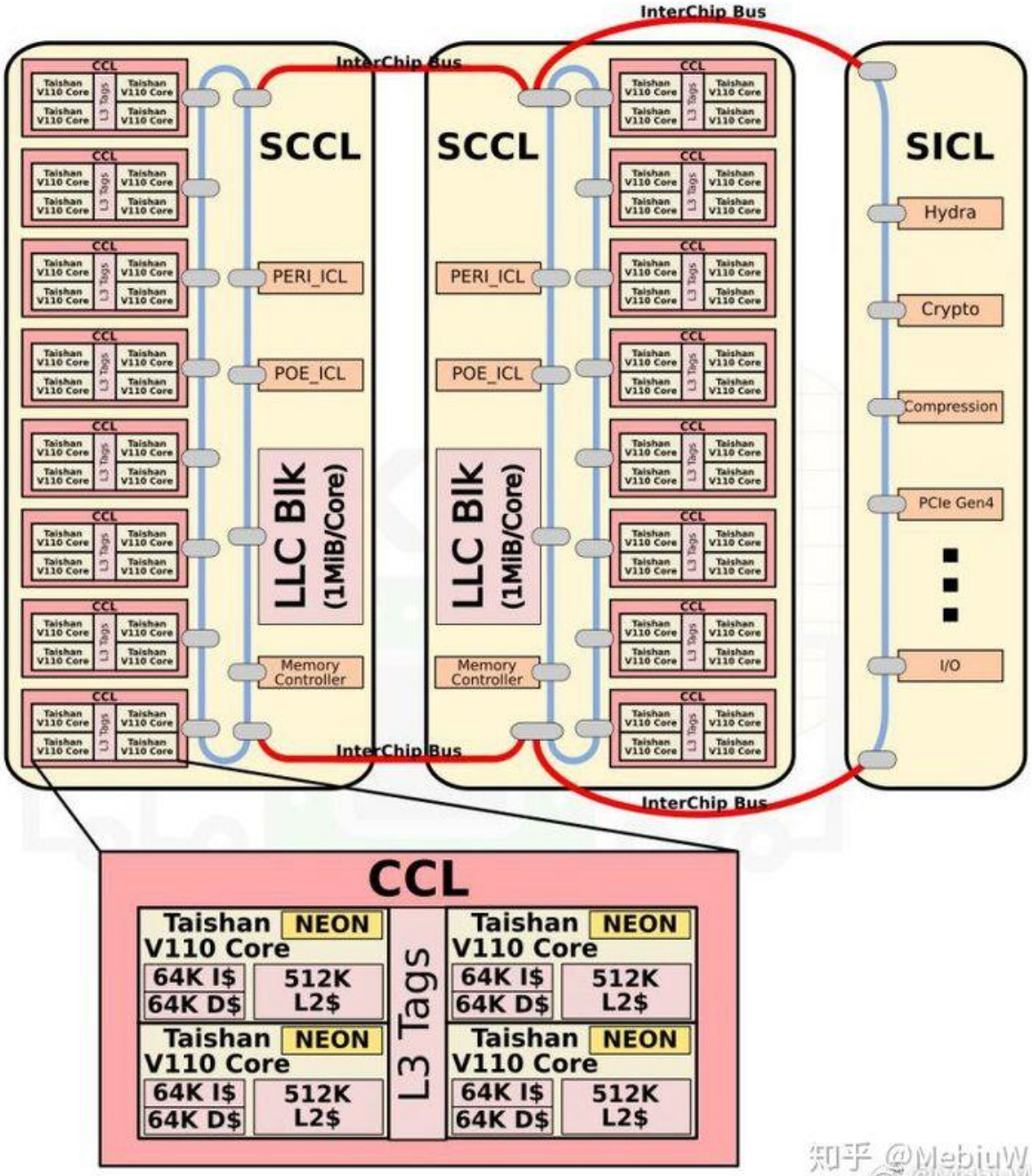
**2、日本富岳，**富士通的48核A64FX SoC，2.7T

**3、国防科大迈创众核处理器（Matrix-2000+）**128 cores， 2.048 T

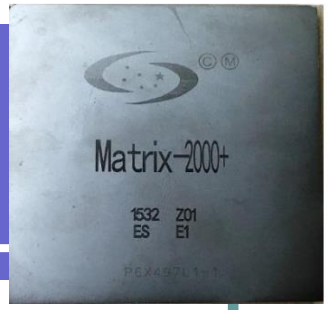
CISC(复杂指令集，比如X86系列)；RISC(精简指令集，比如ARM系列)

4、阿里CPU倚天710

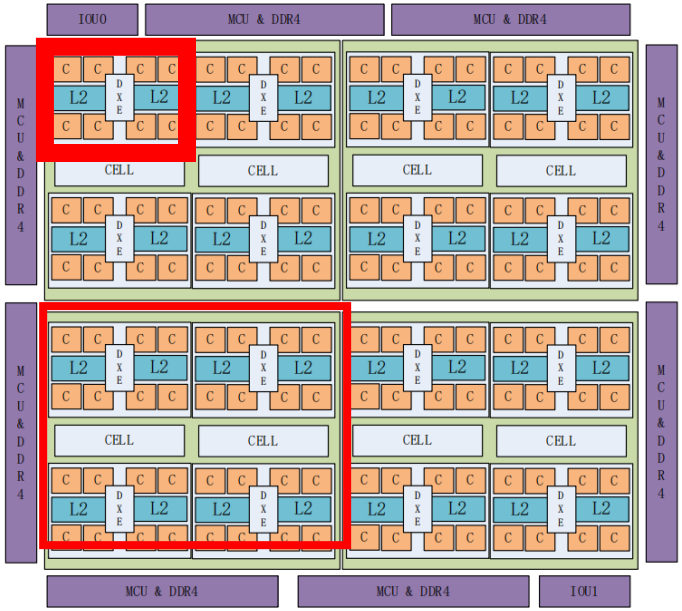
- 服务器芯片鲲鹏920
- TaiShan V110核心
- EulerOS 2.8
- ubuntu18.04
- CentOS7.5
- C
- c++
- golang>=1.5

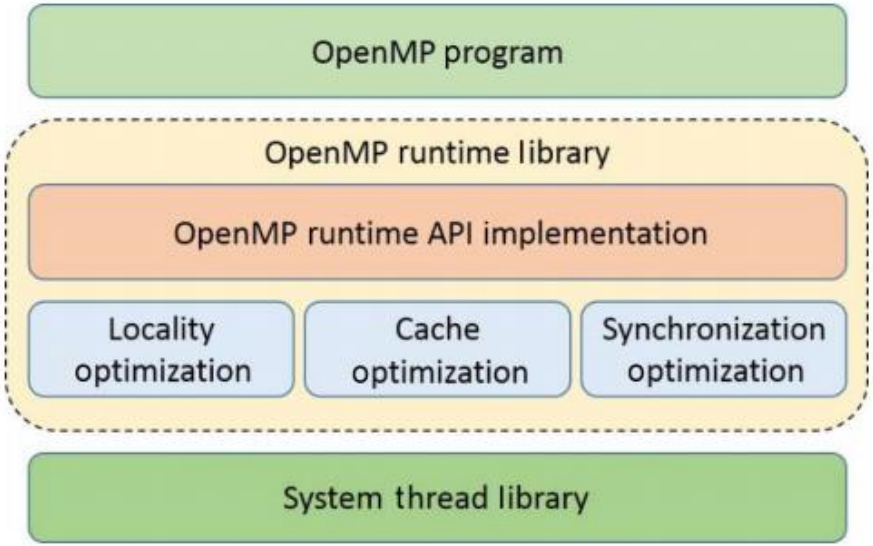


# Matrix-2000+




- 128 compute cores @2 GHz， 4 Super-Nodes (SNs), a scalable on-chip communication network. Each SN has four panels and each panel containing eight cache-coherent compute cores.
- OpenMP 4.5 and OpenCL 1.2
- 4 SNs * 32 cores* 8 flflops *2 GHz = 2.048 Tflflops.



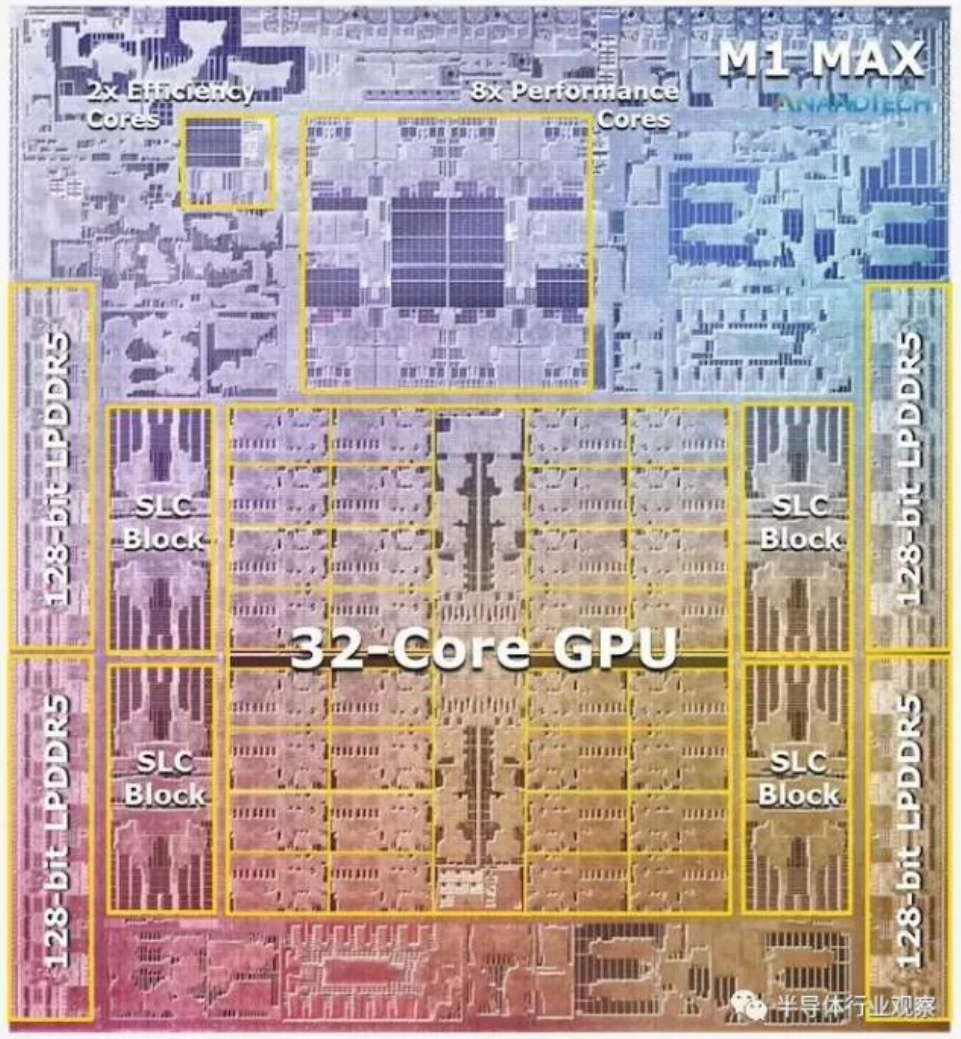

Brief Introduction of TianHe Exascale Prototype System，Tsinghua Science and Technology, June 2021, 26(3): 361–369

# 阿里云芯片倚天710

- **5nm工艺，单芯片容纳高达600亿晶体管；**
- **ARMv9架构，内含128核CPU，主频3.2GHz**
- **在内存和接口方面，DDR5、PCIe5.0等技术**
- **云端AI推理芯片含光800**

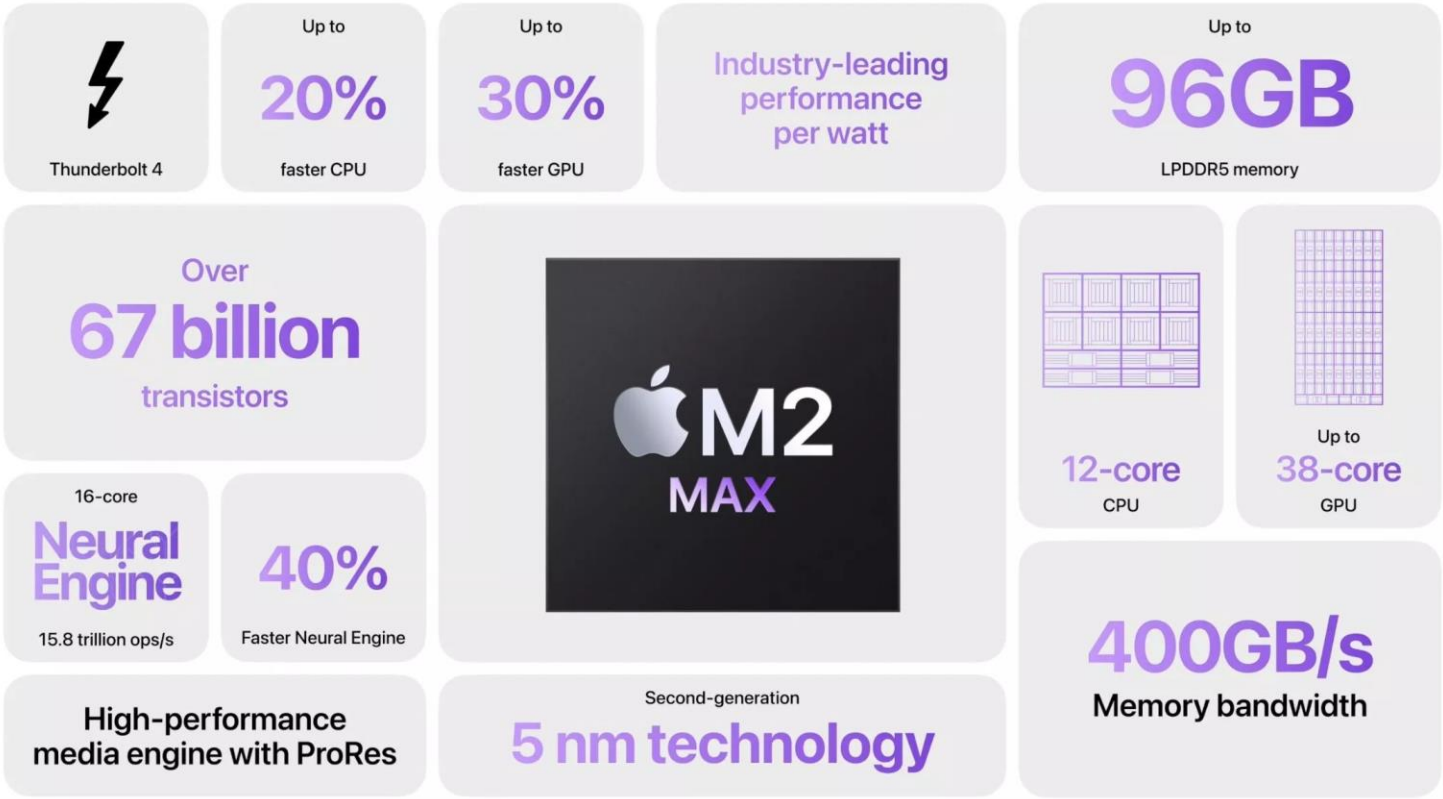
# 苹果 M1芯片

- **M1 Max：100W**
- **5nm**
- **570 亿晶体管**
- **32 核 GPU**
- **512 位 LPDDR5**
- **408GB/s 带宽**
- **10.6 TFLOPS GPU**
- 16 Neural Engine cores 15TOPS



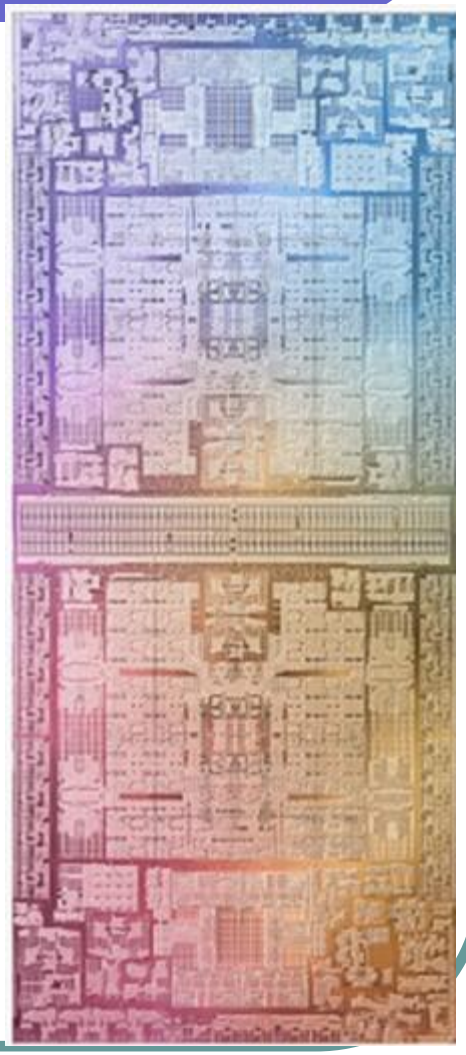
# 苹果 M2 Ultra芯片

**M2 Ultra 由两个 M2 Max 芯片组成，通过定制封装技术 UltraFusion 连接而成，将超10000 多个信号连接起来，提供超过 2.5TB/s 的低延迟带宽**



# 苹果 UltraFusion

- combined two M2 Max
- most powerful SoC
- supports 192GB of unified RAM
- memory bandwidth of 800GB/s



# 应用迁移

**程序代码（C/C++）：**

```
int main()
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
    return c;
}
```

编译

**鲲鹏处理器指令（RISC）**

| 指令 | 汇编代码 | 说明 |
|---|---|---|
| b9400fe1 | ldr x1, [sp,#12] | 从内存将变量a的值放入寄存器x1 |
| b9400be0 | ldr x0, [sp,#8] | 从内存将变量b的值放入寄存器x0 |
| 0b000020 | add x0, x1, x0 | 将x1(a)中的值加上x0(b)的值放入x0寄存器 |
| b90007e0 | str x0, [sp,#4] | 将x0寄存器的值存入内存（变量c） |

**x86处理器指令（CISC）**

| 指令 | 汇编代码 | 说明 |
|---|---|---|
| 8b 55 fc | mov -0x4(%rbp),%edx | 从内存将变量a的值放入寄存器edx |
| 8b 45 f8 | mov -0x8(%rbp),%eax | 从内存将变量b的值放入寄存器eax |
| 01 d0 | add %edx,%eax | 将edx(a)中的值加上eax(b)的值放入eax寄存器 |
| 89 45 f4 | mov %eax,-0xc(%rbp) | 将eax寄存器的值存入内存（变量c） |



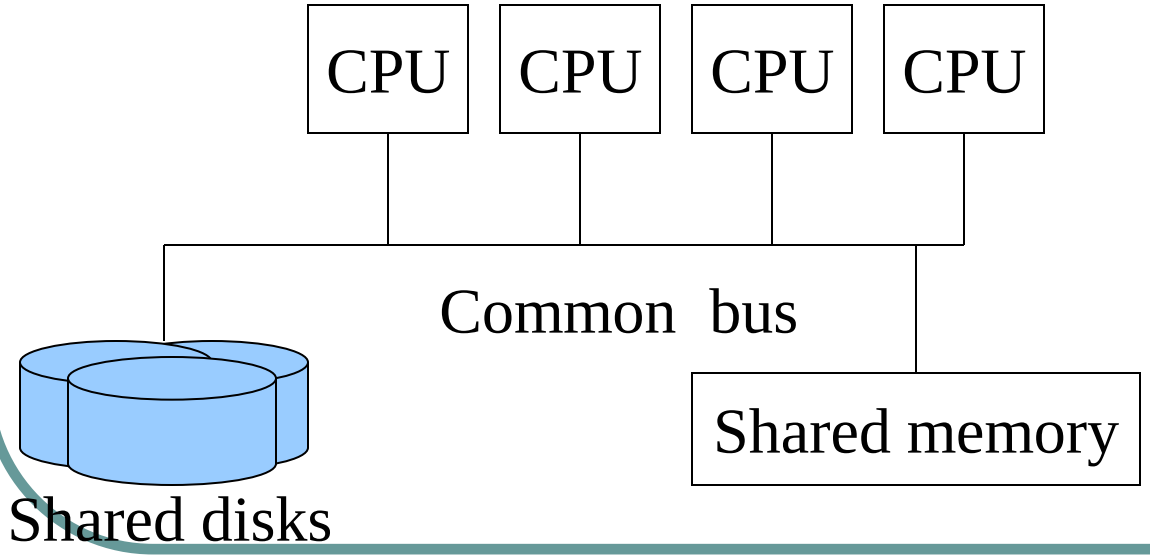
- 绝大部分云平台应用都跑在于X86架构CPU上

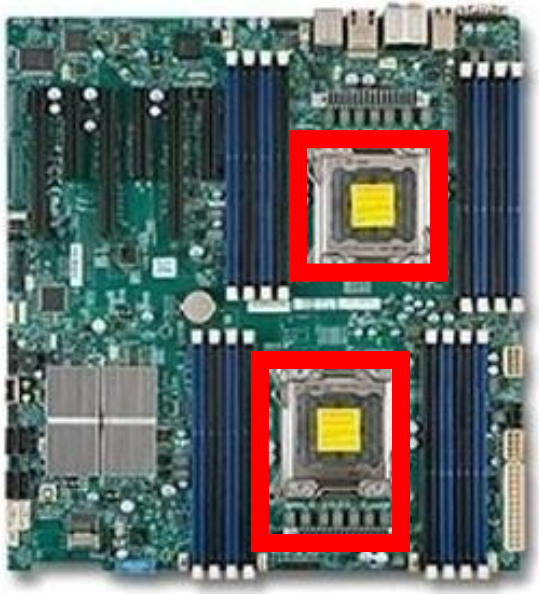
# 应用迁移


JAVA, Pyhton,
等解释型语言
语言支持跨平台，运行环境OK即可
X86
平台
C/C++/GO
等编译型语言
开源软件
社区协同
自研软件
商业合作
获取源代码
重新编译
商用闭源软件
商业合作
选取可替代的其它软件
无法获取源代码且无替代软件
其他平台
开发语言
开源生态
鲲鹏
计算
平台
与x86
混合
部署
华为云社区

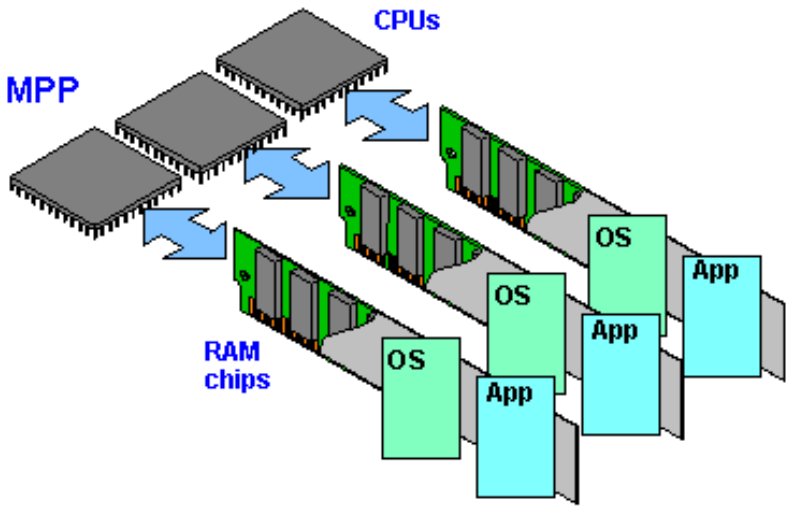
# Symmetric Multi-Processor -- SMP

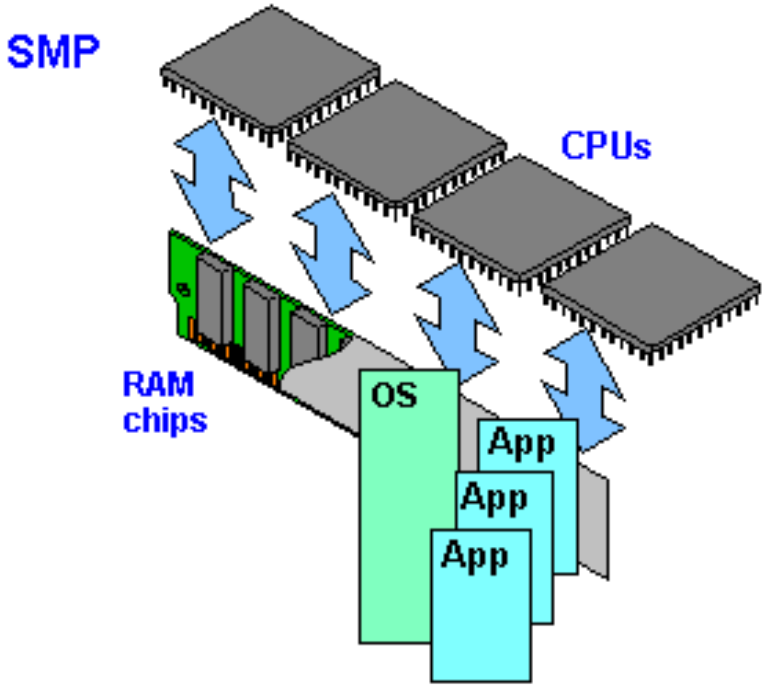
- Communication by shared memory
- Disk controllers accessible to all CPUs
- Proven technology



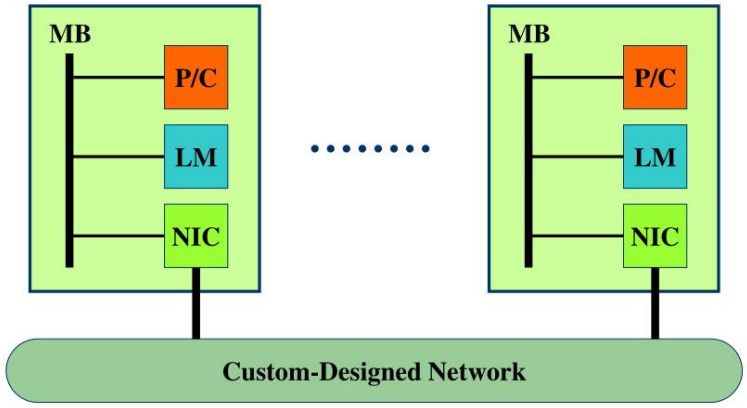



# MPP-- **M**assively **P**arallel **P**rocessor




MB : Memory Bus NIC : Network Interface Circuitry



# 集群系统-- cluster

由节点和集群互连网络组成，再配置上全局软件，是一种松散耦合的多机系统. 通过各节点的并行运行，可以实现高性能的并行计算.

- 优点： 可靠性、扩展性和性价比
- 高性能计算
- 大数据
- 深度学习

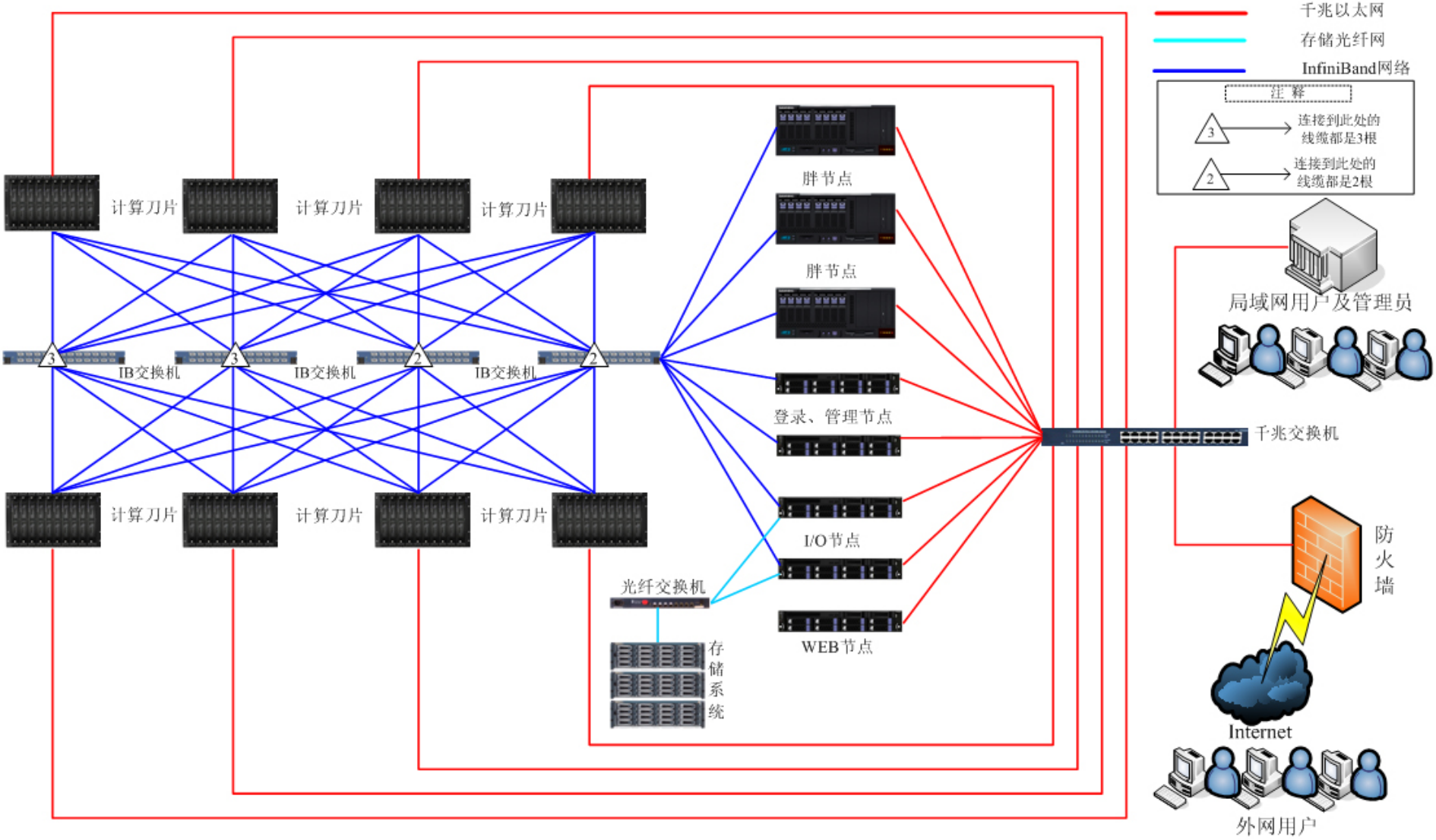
# 集群

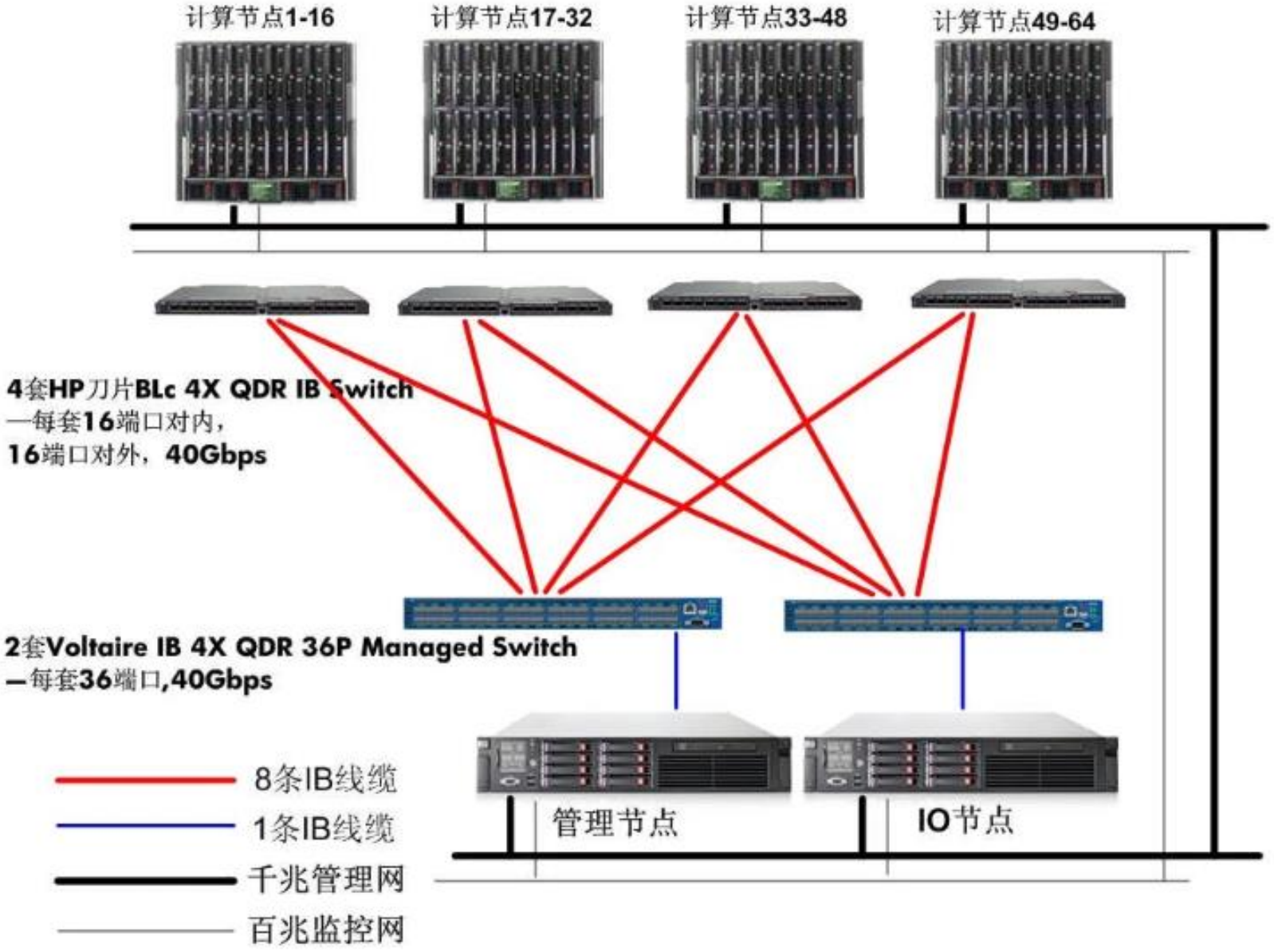
节点:机架式或者刀片式

网络:**InfiniBand**和千兆以太网.

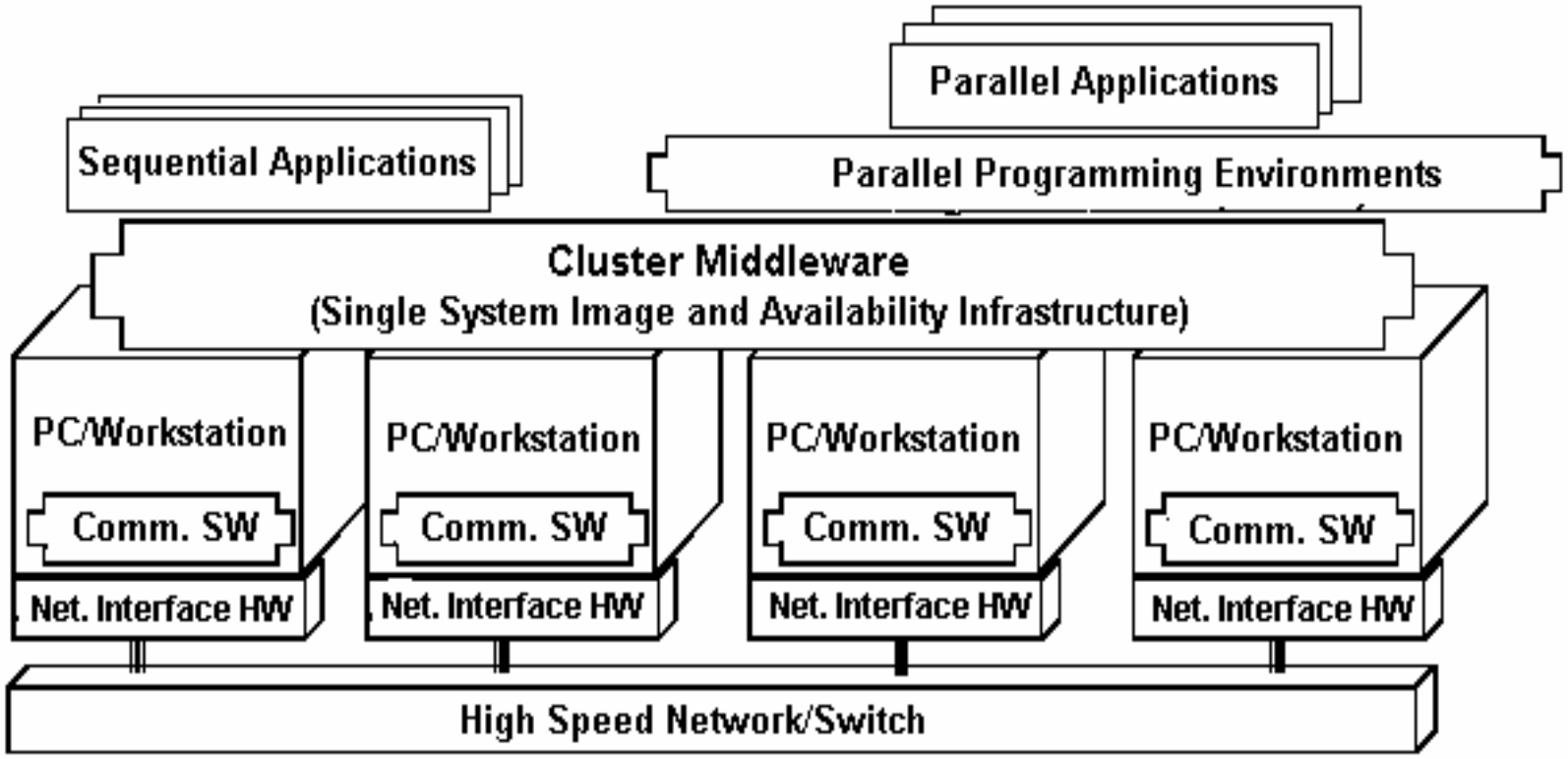
# 中南大学高性能计算平台网络拓扑图

中南大学高性能计算平台体系结构为混合式的Cluster（集群）架构，采用78片四路刀片计算节点、3台八路SMP胖节点、2台登录和管理节点、1台WEB节点、2台I/O节点。存储能力为20TB。计算网络采用20GB全线速IB高速交换机联结，双精度浮点运算次数理论峰值为10 TFlops（十万亿次）。其综合性能在2009年中国高性能计算机性能TOP100排行榜中位列高校第8，全国位列第80。



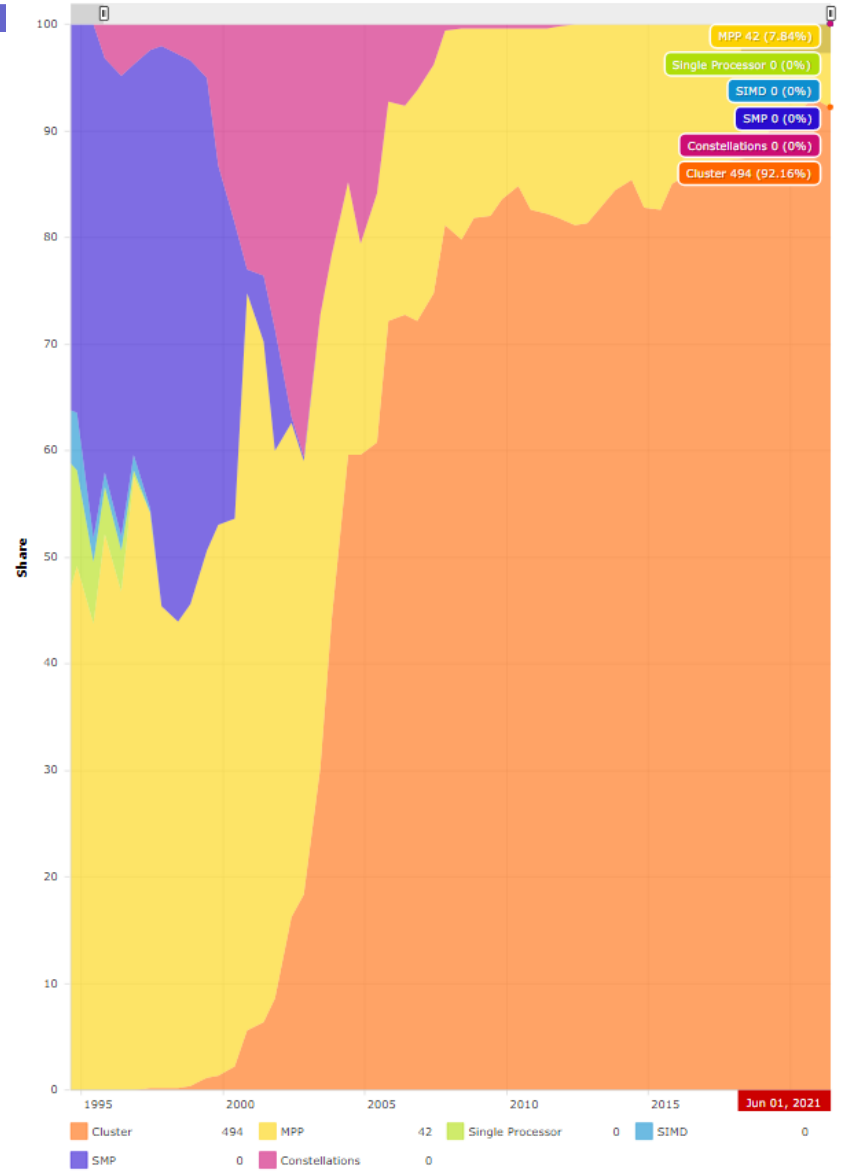

计算节点1-16
计算节点17-32
计算节点33-48
计算节点49-64
4套HP刀片BLc 4X QDR IB Switch
—每套16端口对内，
16端口对外，40Gbps
2套Voltaire IB 4X QDR 36P Managed Switch
—每套36端口,40Gbps
8条IB线缆
1条IB线缆
千兆管理网
百兆监控网
管理节点
IO节点

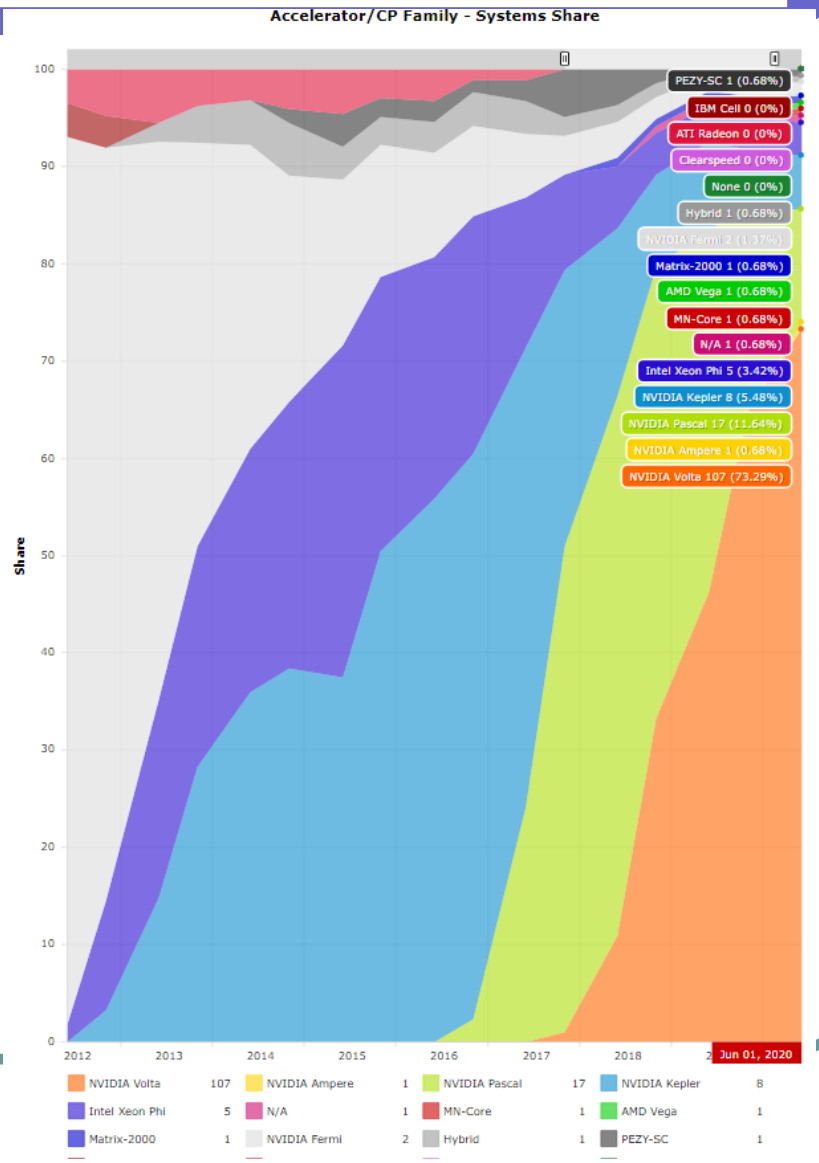
# 集群系统




- 硬件+软件（管理软件，开发软件）

# 世界500强计算机体系结构发展趋势图



# 性能评测

- 集群系统理性能：浮点计算能力 FLOPS，带宽 GB/S

理论峰值＝主频×每个时钟周期执行浮点运算的次数×计算核心个数

- HPL(High Performance Linpack)是衡量高性能计算机浮点计算能力指标
- 国际上 TOP500 高性能计算机系统性能评价的标准。
- HPL 的数学原理就是用矩阵的 LU 分解并行算法求解一个线型方程组，即：

$$\mathbf{A}\,\mathbf{x} = \mathbf{l},\ (\mathbf{A} \in R^{n*n};\ \mathbf{x}, \mathbf{b} \in R^{n})$$

问题规模 $N$，根据线性方程组求解过程中消元和回代部分的耗时 t 就可以计算出集群的性能参数，即每秒执行的浮点运算次数：$(\frac{2N^3}{3} + \frac{3N^2}{2})/t$。

# LU分解

- 用高斯消去法系数矩阵分解为一个单位下三角矩阵和一个上三角矩阵相乘的形式。这样，求解这个线性方程组就转化为求解两个三角矩阵的方程组。

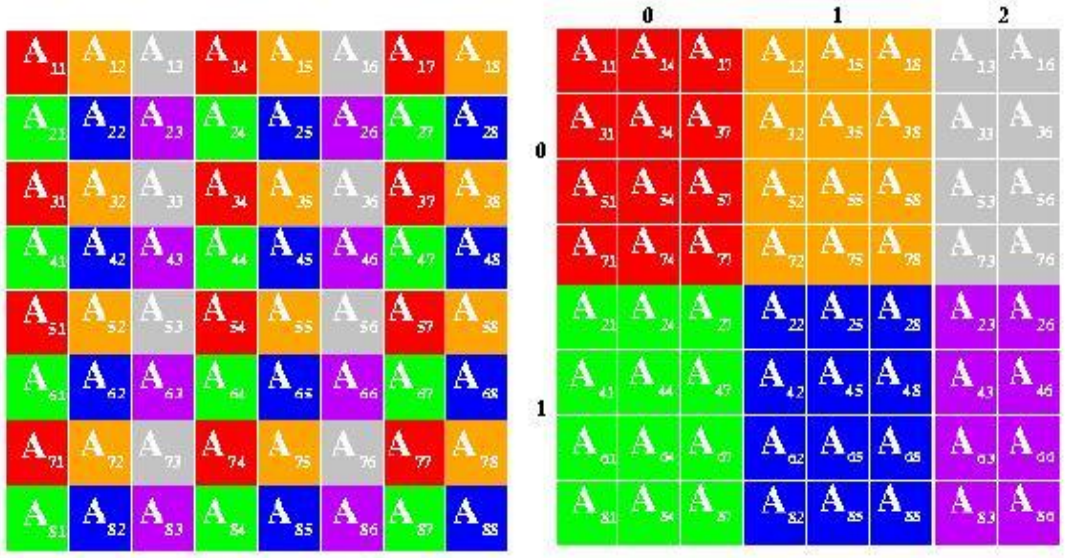
$$A = A^{(0)} = \begin{bmatrix} 8 & 1 & 6 \\ 4 & 9 & 2 \\ 0 & 5 & 7 \end{bmatrix}.$$

$$A = LU \approx \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0.5882 & 1 \end{bmatrix} \begin{bmatrix} 8 & 1 & 6 \\ 0 & 8.5 & -1 \\ 0 & 0 & 7.5882 \end{bmatrix}$$

# 性能评测

- 理论峰值＝每秒浮点运算次数FLOPS
- GOPS （giga operations per second ）每秒十亿次运算数
- HPL(High Performance Linpack)

$$\mathbf{A}\mathbf{x}= \mathbf{l}_{,(}\mathbf{A}\in R^{n*n}{}_{;}\mathbf{x}_{,}\mathbf{b}\in R^{n}{}_{)}{}_{。}$$




$$(\frac{2N^3}{3}+\frac{3N^2}{2})/t$$

# Green500

- 每瓦功耗所获取的运算性能（FLOPS/W）


| Rank | TOP500 Rank | System | Cores | Rmax (PFlop/s) | Power (kW) | Energy Efficiency (GFlops/watts) |
|---|---|---|---|---|---|---|
| 1 | 29 | **Frontier TDS** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 120,832 | 19.20 | 309 | 62.684 |
| 2 | 1 | **Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 8,730,112 | 1,102.00 | 21,100 | 52.227 |
| 3 | 3 | **LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>EuroHPC/CSC<br>Finland | 1,110,144 | 151.90 | 2,942 | 51.629 |
| 4 | 10 | **Adastra** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES)<br>France | 319,072 | 46.10 | 921 | 50.028 |
| 5 | 326 | **MN-3** - MN-Core Server, Xeon Platinum 8260M 24C 2.4GHz, Preferred Networks MN-Core, MN-Core DirectConnect, Preferred | 1,664 | 2.18 | 53 | 40.901 |

# HPCG

- 高性能共轭梯度( Gradient,HPCG)基准测试
- High Performance Conjugate Gradients (HPCG) Benchmark
- Linpack关注线性方程的计算性能
- 使用复杂的微分方程计算方式
- 内存系统和网络延迟要求也更高
- Sparse matrix-vector multiplication.
- Vector updates.
- Global dot products.
- Local symmetric Gauss-Seidel smoother.
- Sparse triangular solve (as part of the Gauss-Seidel smoother).

| Rank | TOP500 Rank | System | Cores | Rmax (PFlop/s) | HPCG (TFlop/s) |
|---|---|---|---|---|---|
| 1 | 2 | **Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu<br>RIKEN Center for Computational Science<br>Japan | 7,630,848 | 442.01 | 16004.50 |
| 2 | 4 | **Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 2,414,592 | 148.60 | 2925.75 |
| 3 | 3 | **LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>EuroHPC/CSC<br>Finland | 1,110,144 | 151.90 | 1935.73 |
| 4 | 7 | **Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE<br>DOE/SC/LBNL/NERSC<br>United States | 761,856 | 70.87 | 1905.44 |
| 5 | 5 | **Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox<br>DOE/NNSA/LLNL<br>United States | 1,572,480 | 94.64 | 1795.67 |

# High performance Linpack

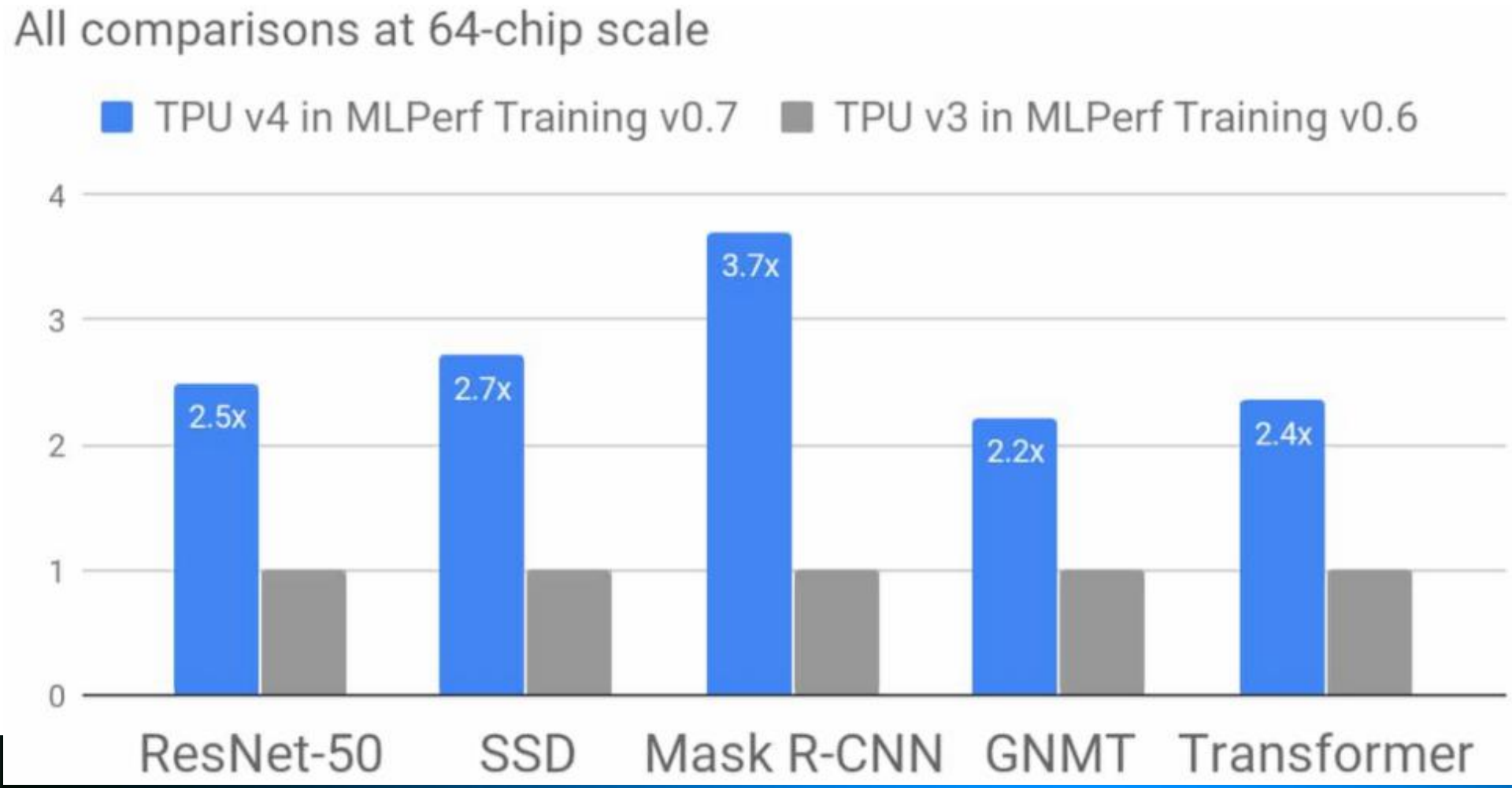
- 1、操作系统 LINUX
- 2、并行软件OpenMP，MPICH，CUDA
- 3、数学库：线性代数程序——支持向量、向量-矩阵、矩阵-矩阵运算；

  Gotoblas，BLAS（Basic Linear Algebra Subprograms）

  cuBLAS，cuDNN
- 4、 HPL or other applications

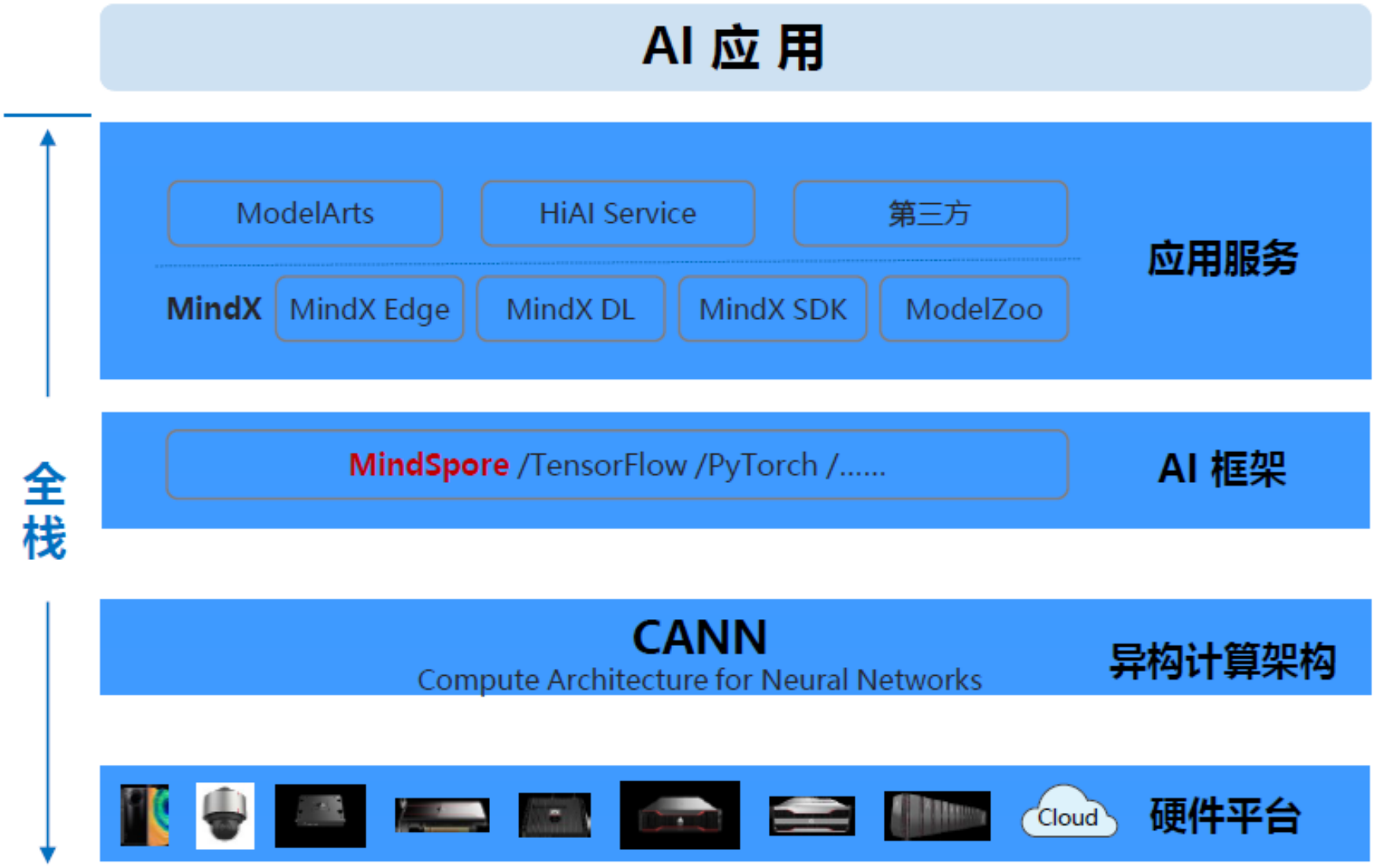
# AI 三要素算力——AI芯片

- 华为、寒武纪、TPU

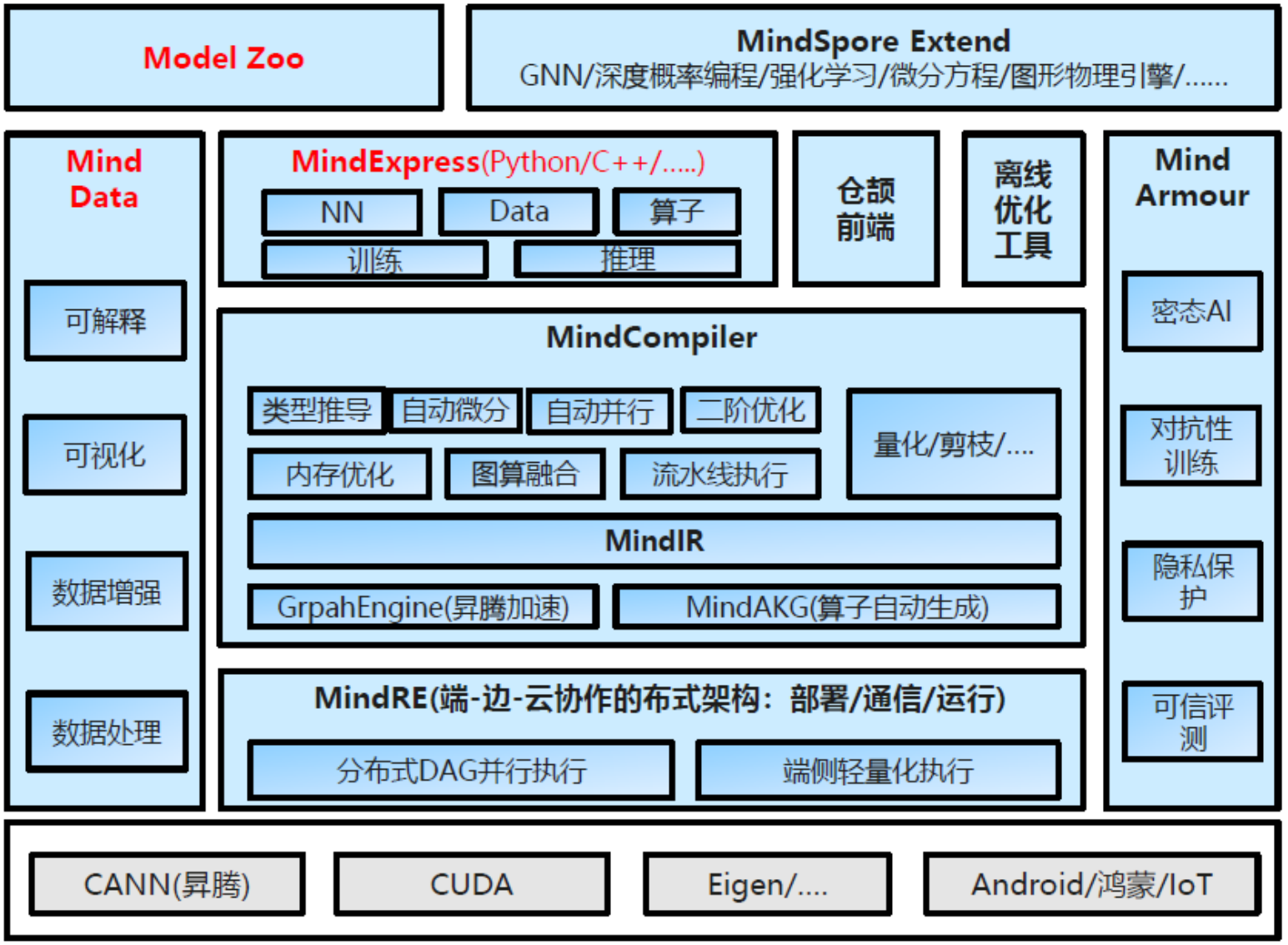


| Feature | TPUv1 | TPUv2 | TPUv3 | Volta |
|---|---|---|---|---|
| Peak TeraFLOPS/ Chip | 92 (8b int) | 46 (16b) 3 (32b) | 123 (16b) 4 (32b) | 125 (16b) 16 (32b) |
| Network links x Gbits/s/Chip | -- | 4 x 496 | 4 x 656 | 6 x 200 |
| Max chips/supercomputer | -- | 256 | 1024 | Varies |
| Peak PetaFLOPS/supercomputer | -- | 11.8 | 126 | Varies |
| Bisection Terabits/supercomputer | -- | 15.9 | 42.0 | Varies |
| Clock Rate (MHz) | 700 | 700 | 940 | 1530 |
| TDP (Watts)/Chip | 75 | 280 | 450 | 450 |
| TDP (Kwatts)/supercomputer | -- | 124 | 594 | Varies |
| Die Size ($mm^2$) | <331 | <611 | <648 | 815 |
| Chip Technology | 28nm | >12nm | >12nm | 12nm |
| Memory size (on-/off-chip) | 28MiB/8GiB | 32MiB/16GiB | 32MiB/32GiB | 36MiB/32GiB |
| Memory GB/s/Chip | 34 | 700 | 900 | 900 |
| MXUs/Core, MXU Size | 1 256x256 | 1 128x128 | 2 128x128 | 8 4x4 |
| Cores/Chip | 1 | 2 | 2 | 80 |

# AI 三要素算力——AI芯片

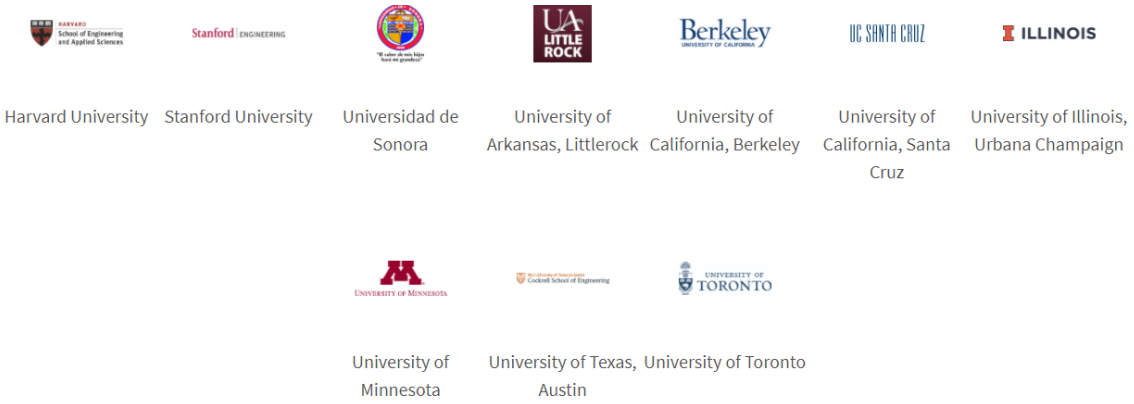


# AI 三要素算力——AI芯片



# MLPerf

- Training： measures how fast a system can train ML models.
- Inference： measures how fast a system can perform ML inference using a trained model.

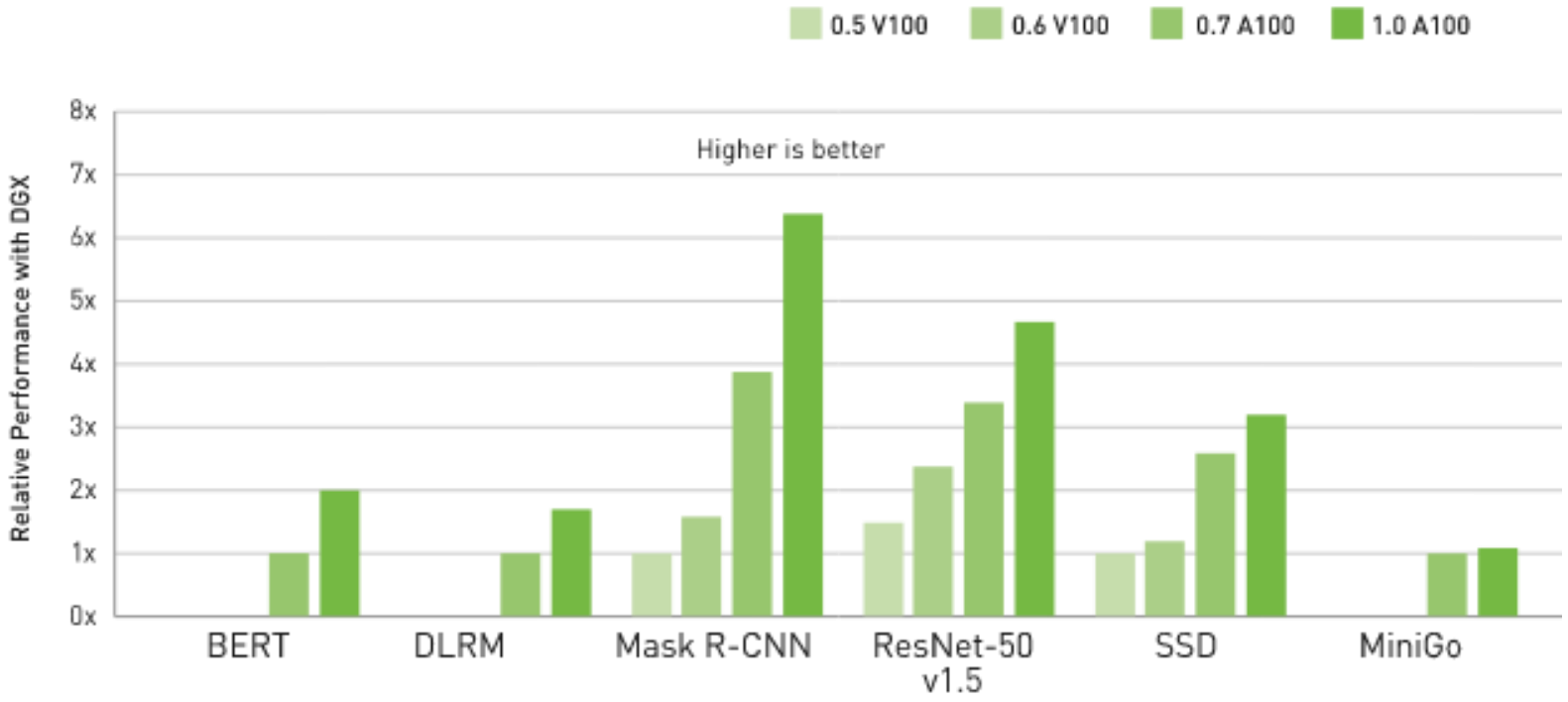


# MLPerf

| Area | Task | Model | Dataset |
|---|---|---|---|
| Vision | Image classification | Resnet50-v1.5 | ImageNet (224x224) |
| Vision | Object detection (large) | SSD-ResNet34 | COCO (1200x1200) |
| Vision | Medical image segmentation | 3D UNET | BraTS 2019 (224x224x160) |
| Speech | Speech-to-text | RNNT | Librispeech dev-clean (samples < 15 seconds) |
| Language | Language processing | BERT-large | SQuAD v1.1 (max_seq_len=384) |
| Commerce | Recommendation | DLRM | 1TB Click Logs |

# OPTIMIZED SOFTWARE

- OVER 6.5X PERFORMANCE IN 2.5 YEARS OF MLPERF
- NVIDIA's Full-Stack Innovation Delivers Continuous Improvements




https://developer.nvidia.com/blog/mlperf-v1-0-training-benchmarks-insights-into-a-record-setting-performance/

# Contents


1 集群计算

2 集群软件

3 集群计算

4 高性能计算应用

5 结论

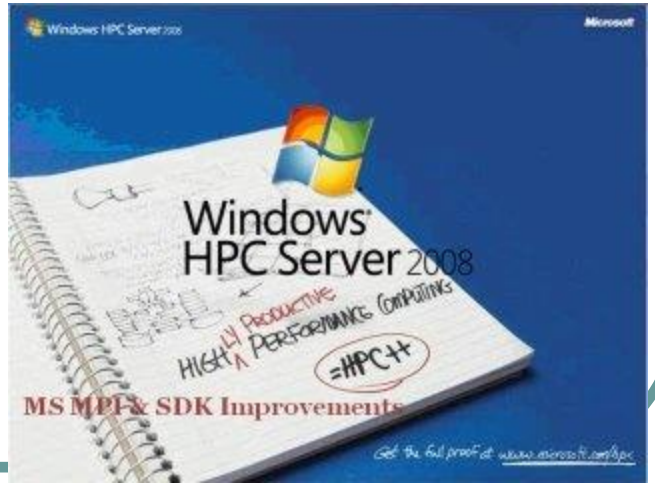
# 软件环境组成（HPC+AI）

- 操作系统
- 作业调度系统
- 管理和监控软件
- 并行开发软件
- 客户端软件

Platform

redhat.

PBS Professional



# Linux

- **RedHat Enterprise Linux VS Ubuntu**
- cd directory 进入指定的目录
  cd .. 进入上一级目录
  ls　查看目录下的文件
- mkdir　新建目录
- mv　将文件重命名或移动到新目录

  Vi 编辑器
- 进入vi：**vi** myfile.c

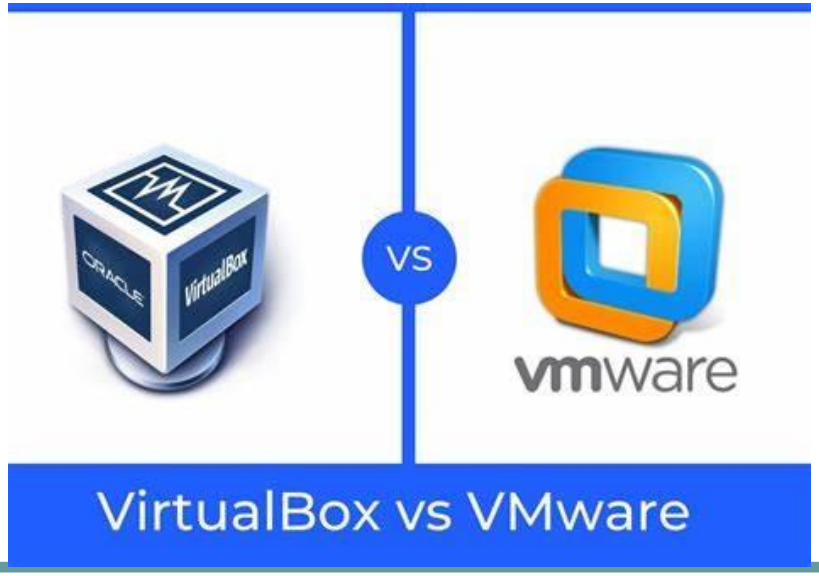
# 环境建立

- Cygwin

  一个在windows平台上运行的类UNIX模拟环境

- 虚拟机

*VirtualBox* 与 *VMware WorkStation*



# GNU Make

- make和makefile。
- make只对上次编译后修改过的部分进行编译。
- Info　make

- makefile文件
- 1、创建的目标体(target)，通常是目标文件或可执行文件。
- 2、要创建的目标体所依赖的文件(dependency_file)。
- 3、创建每个目标体时需要运行的命令(command)。

# 例子：

- prog程序由三个C源文件filea.c、fileb.c和filec.c以及库文件LS编译生成，这三个文件还分别包含自己的头文件a.h 、b.h和c.h 。
- 目标文件filea.o、fileb.o和filec.o。
- 假设filea.c和fileb.c都要声明用到一个名为defs的文件，但filec.c不用。即在filea.c和fileb.c里
  都有这样的声明：

```
#include "defs"
```

# makefile

- #It is a example for describing makefile

```
prog : filea.o fileb.o filec.o
cc filea.o fileb.o filec.o -LS -o prog
filea.o : filea.c a.h defs
cc -c filea.c
fileb.o : fileb.c b.h defs
cc -c fileb.c
filec.o : filec.c c.h
cc -c filec.c
```

# Contents


1 集群计算基础

2 集群软件

3 多线程

4 OpenMP

5 MPI

# 多线程--windows

- **Windows**
- **线程创建函数 CreateThread**
- **HANDLE WINAPI CreateThread(**
- **_In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,**
- **_In_ SIZE_T dwStackSize,**
- **_In_ LPTHREAD_START_ROUTINE lpStartAddress,**
- **_In_opt_ LPVOID lpParameter,**
- **_In_ DWORD dwCreationFlags,**
- **_Out_opt_ LPDWORD lpThreadId**
- **);**

# 参数含义

1. 第一个参数 **lpThreadAttributes** 表示线程内核对象的安全属性，一般传入**NULL**表示使用默认设置。
2. 第二个参数 **dwStackSize** 表示线程栈空间大小。传入**0**表示使用默认大小（**1MB**）。
3. 第三个参数 **lpStartAddress** 表示新线程所执行的线程函数地址，多个线程可以使用同一个函数地址。
4. 第四个参数 **lpParameter** 是传给线程函数的参数。
5. 第五个参数 **dwCreationFlags** 指定额外的标志来控制线程的创建，为**0**表示线程创建之后立即就可以进行调度，如果为**CREATE_SUSPENDED**则表示线程创建后暂停运行，这样它就无法调度，直到调用**ResumeThread()**。
6. 第六个参数 **lpThreadId** 将返回线程的**ID**号，传入**NULL**表示不需要返回该线程**ID**号。
7. 返回值　线程创建成功返回新线程的句柄，失败返回**NULL**

# 多线程--windows

```
#include<Windows.h>        #include<stdio.h>
 DWORD WINAPI ThreadFunc(LPVOID);
 void main()
{    HANDLE hThread;    DWORD  threadId;
     hThread = CreateThread(NULL, 0, ThreadFunc, 0, 0, &threadId);
    printf("我是主线程， pid = %d\n", GetCurrentThreadId());  //主pid
    Sleep(2000);
}
 DWORD WINAPI ThreadFunc(LPVOID p)
{   printf("我是子线程， pid = %d\n", GetCurrentThreadId());   //子pid
    return 0;
}
```

# 多线程—原子操作

```
const unsigned int THREAD_NUM = 10;   DWORD WINAPI  ThreadFunc(LPVOID);
 int main()
{
   printf("我是主线程，  pid = %d\n", GetCurrentThreadId());
   HANDLE hThread[THREAD_NUM];
   for (int i = 0; i < THREAD_NUM; i++)
   {  hThread[i] = CreateThread(NULL, 0, ThreadFunc, &i, 0, NULL);   }
      WaitForMultipleObjects(THREAD_NUM,hThread,true, INFINITE);
 //一直等待，到所有子线程全部返回
   return 0;
}
 DWORD WINAPI  ThreadFunc(LPVOID p)
{    int n = *(int*)p;
   Sleep(1000*n);  //第 n 个线程睡眠 n 秒
   printf("我是，  pid = %d 的子线程\n", GetCurrentThreadId());   //输出子线程pid
   printf(" pid = %d 的子线程退出\n\n", GetCurrentThreadId());   //延时10s后输出
    return 0;
}
```

# 多线程—原子操作

```
int main()
{  for (int i = 0; i < THREAD_NUM; i++)
    {  hThread[i] = CreateThread(NULL, 0, ThreadFunc, &i, 0, NULL);   }
       WaitForMultipleObjects(THREAD_NUM,hThread,true, INFINITE);
       return 0;
}
 DWORD WINAPI  ThreadFunc(LPVOID p)
{    int n = *(int*)p;
    Sleep(1000*n);  //第 n 个线程睡眠 n 秒
    printf("我是，  pid = %d 的子线程\n", GetCurrentThreadId());
    printf(“ pid = %d 的子线程退出\n\n”, GetCurrentThreadId());
    return 0;
}
```

# 多线程—原子操作

```
int main()
{  for (int i = 0; i < THREAD_NUM; i++)
    {  hThread[i] = CreateThread(NULL, 0, ThreadFunc, &i, 0, NULL);   }
       WaitForMultipleObjects(THREAD_NUM,hThread,true, INFINITE);
       return 0;
}
 DWORD WINAPI  ThreadFunc(LPVOID p)
{    int n = *(int*)p;
    Sleep(1000*n);  //第 n 个线程睡眠 n 秒
    printf("我是，  pid = %d 的子线程\n", GetCurrentThreadId());
    printf(“ pid = %d 的子线程退出\n\n”, GetCurrentThreadId());
    return 0;
}
```

# 多线程—原子操作

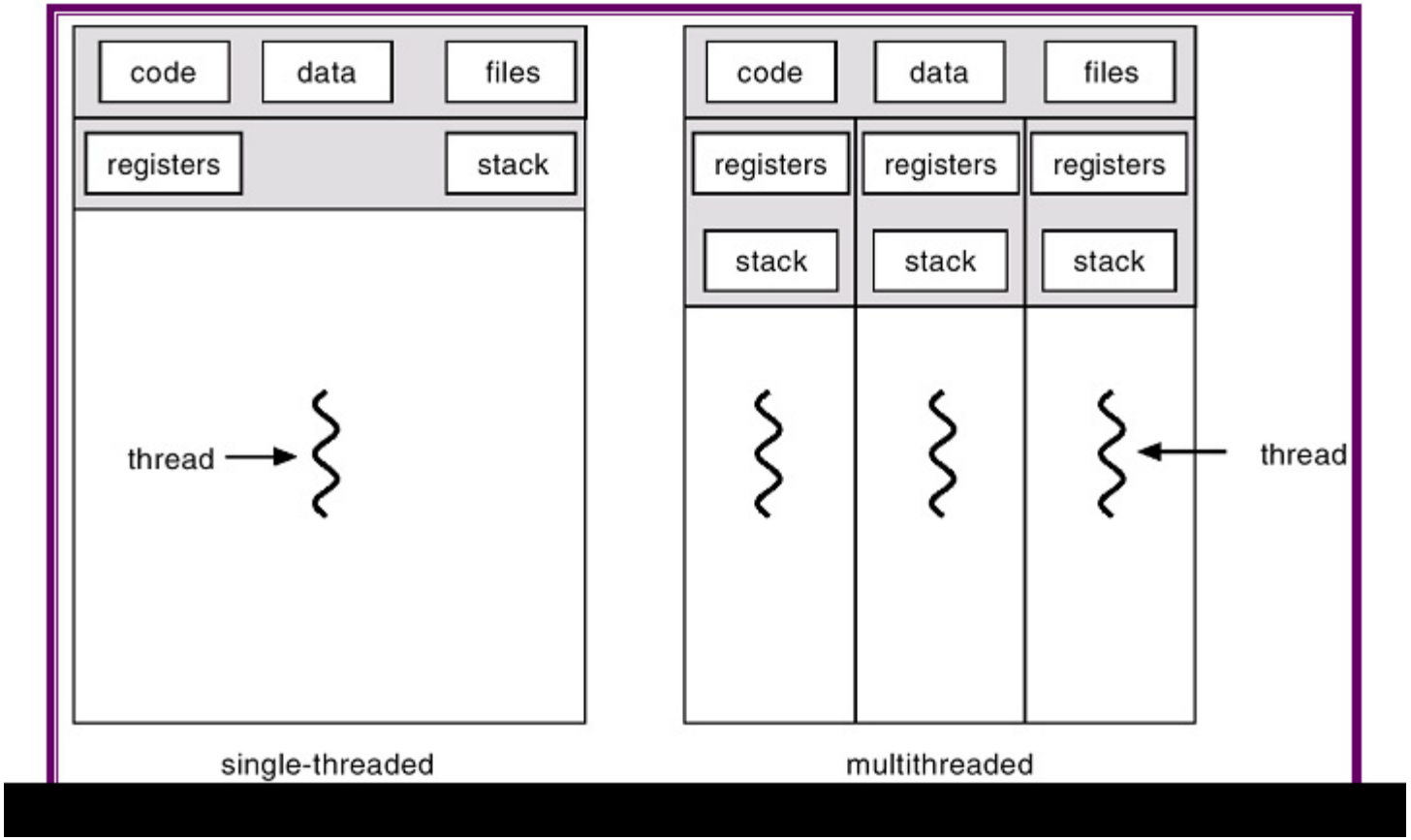
```
const unsigned int THREAD_NUM = 50;  unsigned int g_Count = 0;
DWORD WINAPI ThreadFunc(LPVOID);
int main()
{  for (int i = 0; i < THREAD_NUM; i++)
   {  hThread[i] = CreateThread(NULL, 0, ThreadFunc, &i, 0, NULL);   }
      WaitForMultipleObjects(THREAD_NUM,hThread,true, INFINITE);
      return 0;
}
 DWORD WINAPI ThreadFunc(LPVOID p)
{
Sleep(50);       g_Count++;          Sleep(50);
return 0;
```

3.DWORD WINAPI ThreadFunc(LPVOID);
4.

# 多线程—原子操作

- Sleep(50);
- 0004140E mov esi,esp
- 00041410 push 32h
- 00041412 call dword ptr ds:[49000h]
- 00041418 cmp esi,esp
- 0004141A call __RTC_CheckEsp (041145h)
- g_Count++;
- 0004141F mov eax,dword ptr ds:[00048130h]
- 00041424 add eax,1
- 00041427 mov dword ptr ds:[00048130h],eax
- Sleep(50);
- 0004142C mov esi,esp
- 0004142E push 32h
- 00041430 call dword ptr ds:[49000h]
- 00041436 cmp esi,esp
- 00041438 call __RTC_CheckEsp (041145h)
- return 0;

# 多线程


code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

# 多线程操作

- 线程调度与原子操作

1. DWORD WINAPI ThreadFunc(LPVOID p) {
2. Sleep(50);
3. g_Count++;
4. Sleep(50);
5. return 0; }

```
    g_Count++;
0117140F  mov         eax,dword ptr ds:[01178130h]
01171414  add         eax,1
01171417  mov         dword ptr ds:[01178130h],eax
    Sleep(50);
```

P1、P2顺序执行

| 进程P1 | 进程P2 |
| --- | --- |
| R1:=COUNT | R2:=COUNT |
| R1:=R1+1 | R2:=R2+1 |
| COUNT:=R1 | COUNT:=R2 |
| R1= R2= | COUNT=0 |

P1、P2并发执行

| 进程P1 | 进程P2 |
| --- | --- |
| R1:=COUNT | R2:=COUNT |
| R1:=R1+1 | R2:=R2+1 |
| COUNT:=R1 | COUNT:=R2 |
| R1= R2= | COUNT=0 |

- 1、把 g_Count 的值给寄存器 eax，2、寄存器 eax 的值加一，3、eax 的值给 g_Count，完成一次 ++ 操作。
- 在这几条汇编语句执行的过程中发送了线程切换×

# Contents


1 集群计算基础

2 集群软件

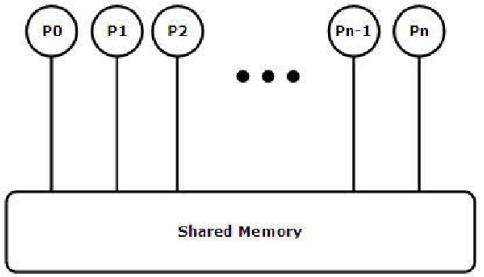
3 多线程

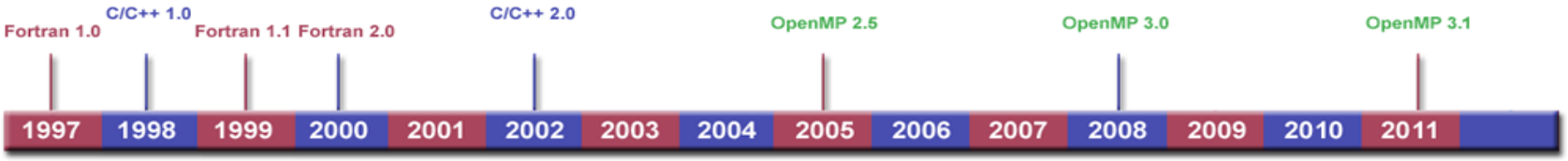
4 OpenMP

5 MPI

# Open Multiprocessing (OpenMP)

- 支持C、C++ 和 Fortran
- 编译指导语句
- GCC 4.4 支持OpenMP 3 标准
- Microsoft Visual Studio 支持
- 适用于共享存储架构



# Fork-Join执行模式

- 在开始执行的时候，只有主线程的运行线程存在
- 主线程在运行过程中，当遇到需要进行并行计算的时候，派生出（Fork，创建新线程或者唤醒已有线程）线程来执行并行任务
- 在并行执行的时候，主线程和派生线程共同工作
- 在并行代码结束执行后，派生线程退出或者挂起，不再工作，控制流程回到单独的主线程中（Join，即多线程的会和）。

# 并行Hello world

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv)
{
  int nthreads, thread_id;
   printf("I am the main thread.\n");
  #pragma omp parallel private(nthreads, thread_id)
 {
    nthreads = omp_get_num_threads();
    thread_id = omp_get_thread_num();
     printf("Hello. I am thread %d out of a team of %d\n",  thread_id, nthreads);
}
printf("Here I am, back to the main thread.\n");
 return 0;
}
```

# 几点说明

- #pragma omp parallel，
- 创建并行区：并行区里每个线程都会去执行并行区中的代码。

- 控制并行区中线程的数量
- 并行区内线程数=系统中核的个数。
- omp_set_num_threads(5);
- omp_get_max_threads()
- **gcc** : gcc -fopenmp hello.c -o hello

# 循环并行化

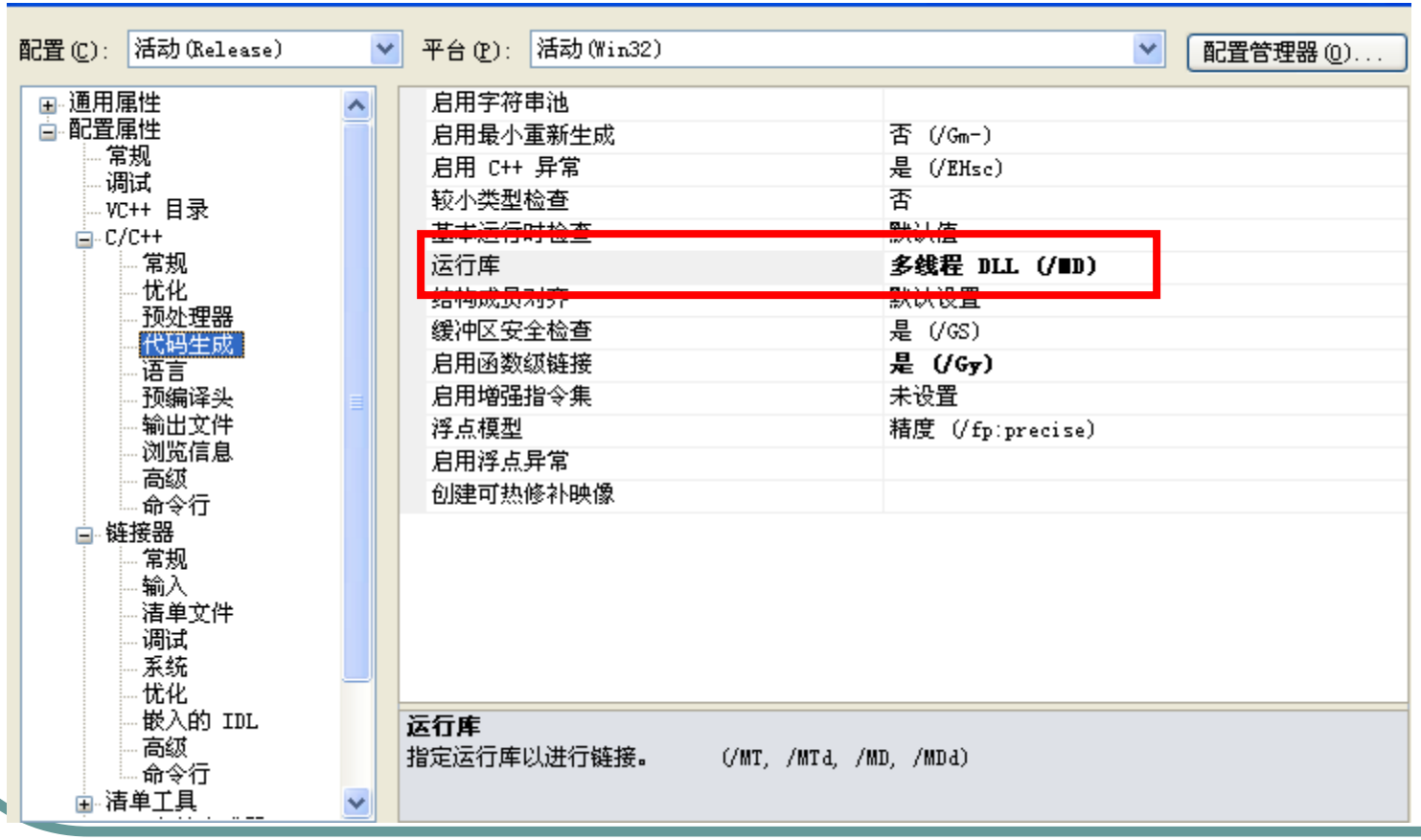
- 循环并行化是是把一个循环分成多部分在不同的进程上执行。

```
#pragma omp parallel for
        for（int i=0; i<n; i++）
              z[i] = x[i]+y[i];
```

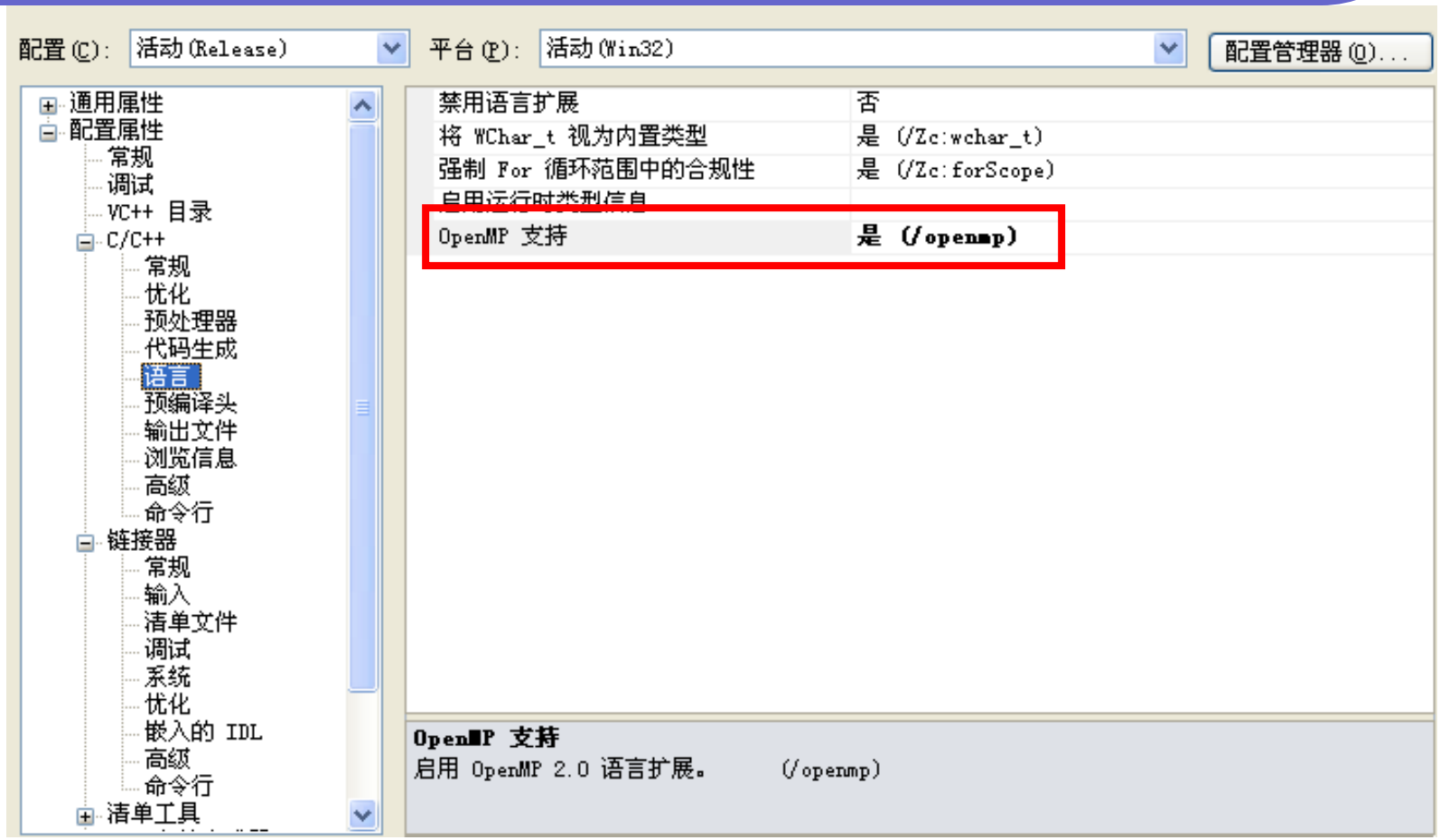
- 工作原理：将for循环中的工作分配到一个线程组中，线程组中的每一个线程将完成循环中的一部分内容；

# Windows openMP

- #include "omp.h"



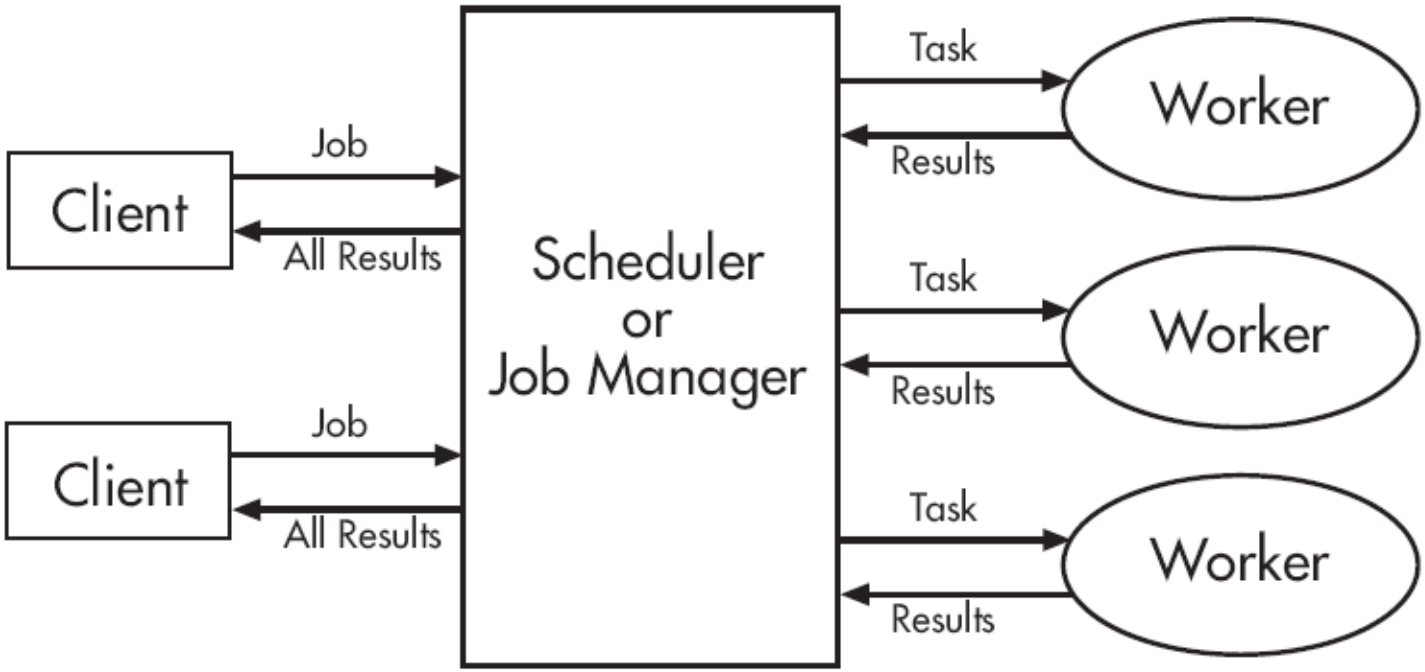
# Windows openMP



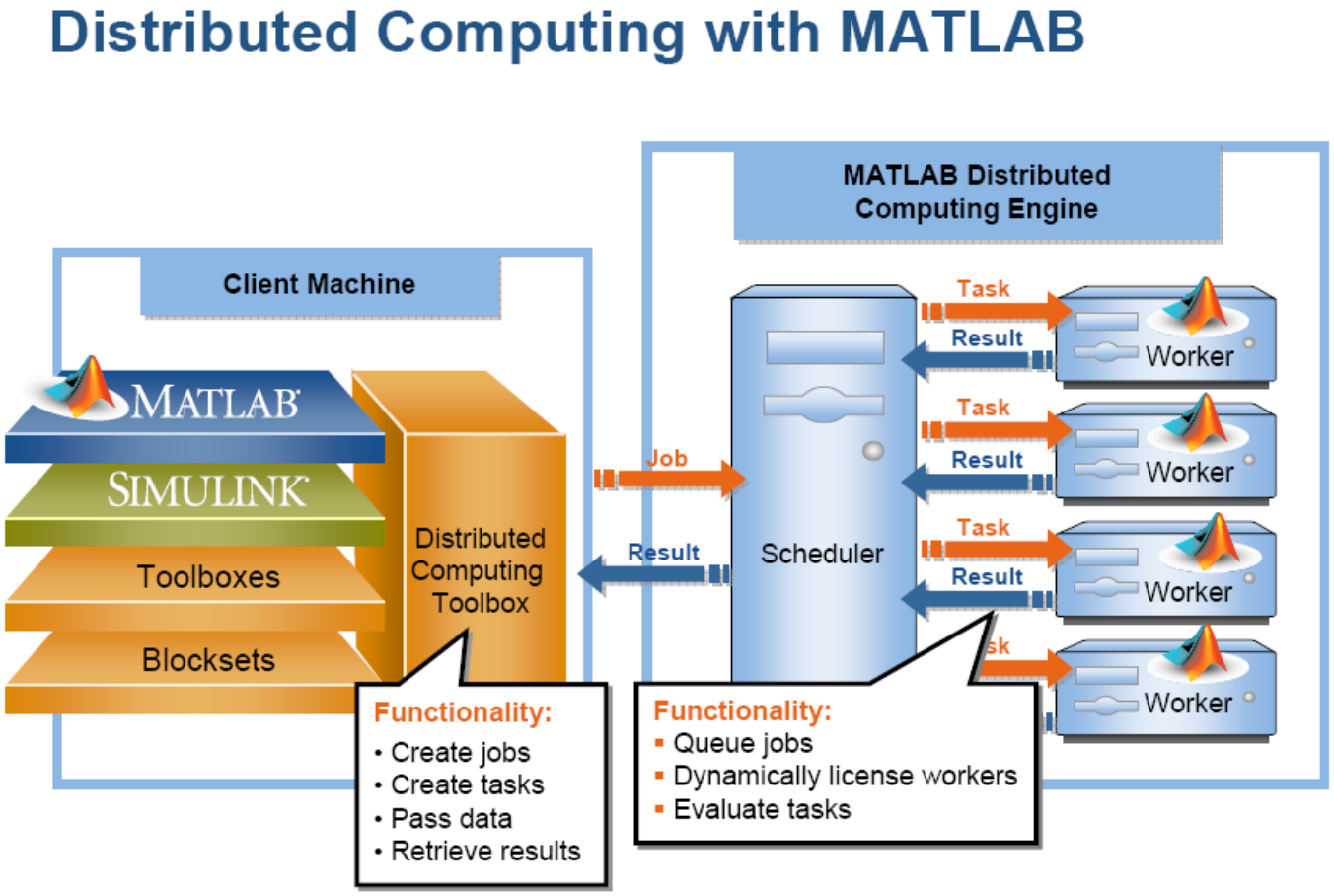
# 矩阵乘法

```
void matrixMulti()
{
for (int row = 0; row < MatrixOrder; row++){
  for (int col = 0; col < MatrixOrder; col++){
   matrixMultiResult[row][col] = calcuPartOfMatrixMulti(row, col);
}
}
}
void matrixMultiOMP()
{ #pragma omp parallel for num_threads(4)
  for (int row = 0; row < MatrixOrder; row++){
     for (int col = 0; col < MatrixOrder; col++){
       matrixMultiResult[row][col] = calcuPartOfMatrixMulti(row, col);
}   }     }
```

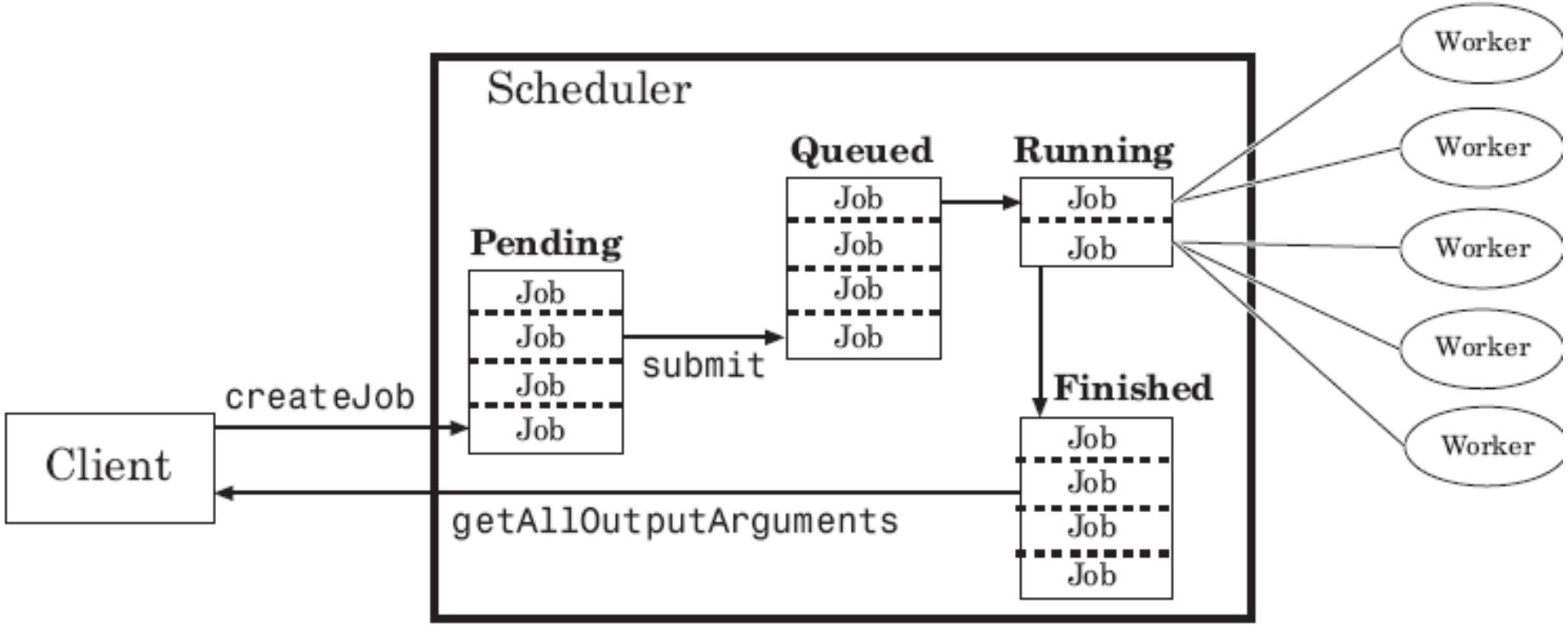
# Matlab并行计算


Client
Job
All Results
Client
Job
All Results
Scheduler
or
Job Manager
Task
Results
Task
Results
Task
Results
Worker
Worker
Worker

# 粗粒度并行计算

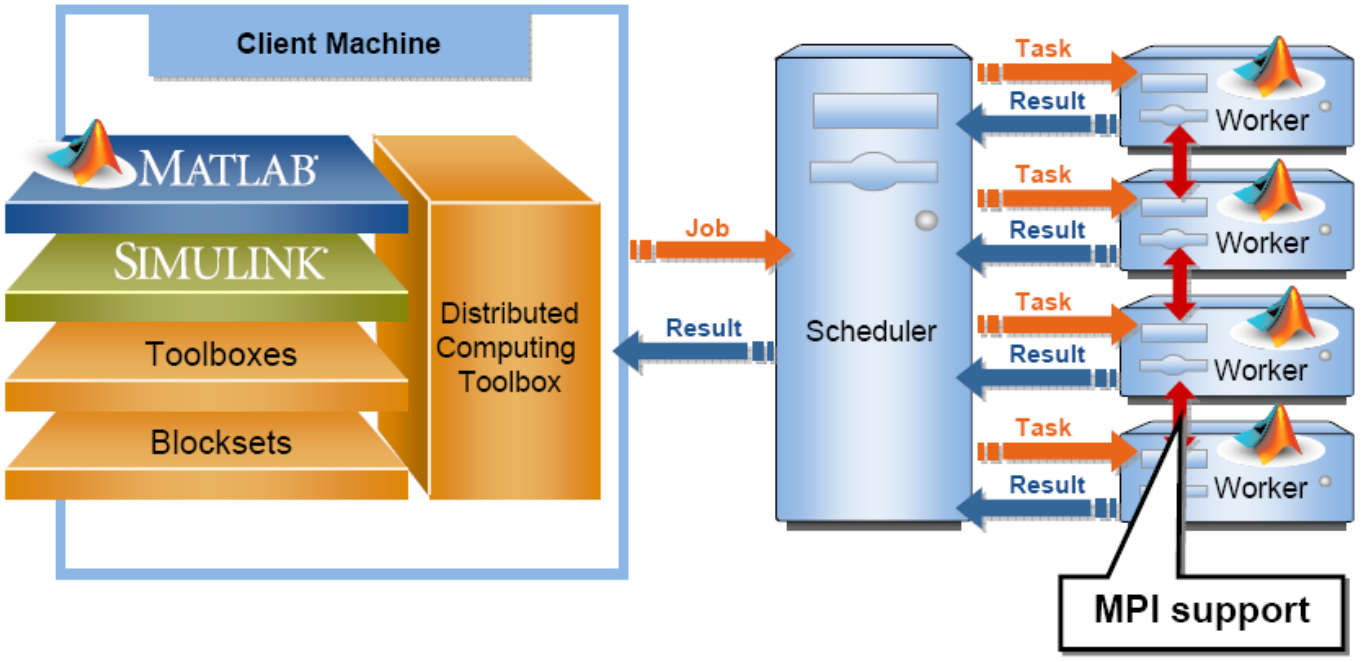

Distributed Computing with MATLAB
MATLAB Distributed Computing Engine
Client Machine
MATLAB
SIMULINK
Toolboxes
Blocksets
Distributed Computing Toolbox
Job
Result
Scheduler
Task
Result
Worker
Task
Result
Worker
Task
Result
Worker
Worker
Functionality:
• Create jobs
• Create tasks
• Pass data
• Retrieve results
Functionality:
▪ Queue jobs
▪ Dynamically license workers
▪ Evaluate tasks

# 生命周期


Scheduler
Queued
Running
Pending
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
Job
submit
createJob
Client
getAllOutputArguments
Finished
Worker
Worker
Worker
Worker
Worker

# 细粒度并行计算

## Support for Parallel Applications



# Parallel Computing Toolbox™ 5

```
clear A
for i = 1:10
    A(i) =fun(i);
End
A
```

```
clear A
parfor i = 1:10
    A(i) =fun(i);
end
A
```

# Programming Distributed Jobs

- jm = findResource('scheduler', 'type', 'jobmanager','name','myjobm','LookupURL', 'xdcn048');
- job = createJob(jm);
- createTask(job, @algorithm, 1, {dataset1});
- createTask(job, @algorithm, 1, {dataset2});
- createTask(job, @algorithm, 1, {dataset3});
- submit(job);
- results = getAllOutputArguments(job);

# Programming Parallel Jobs

- jm = findResource('scheduler', 'type', 'jobmanager','name','myjobm','LookupURL', 'xdcn048');
- job = createParallelJob(jm);
- set(pjob, 'MaximumNumberOfWorkers', 4)
- set(pjob, 'MinimumNumberOfWorkers', 4)
- createTask(job, @algorithm, 1, {dataset});
- submit(job);
- results = getAllOutputArguments(job);

# Programming Parallel Jobs

- labSend(data, destination, tag)
- data = labReceive(source,tag)
- shared_data = labBroadcast(senderlab, data)
- n = numlabs
- id = labindex

# 区别

| Distributed Job | Parallel Job |
| --- | --- |
| ***Workers*** **do not communicate with each other.** | ***labs,*** **communicate with each other** |
| **You define any number of tasks in a job.** | **You define only one task in a job. Duplicates of that task run on all labs running the parallel job.** |
| **Tasks need not run simultaneously.Tasks are distributed to workers as the workers become available, so a worker can perform several of the tasks in a job.** | **Tasks run simultaneously, so you can run the job only on as many labs as are available at run time. The start of the job might be delayed until the required number of labs is available.** |

# 对GPU的支持

- Transferring data between the MATLAB workspace and the GPU
- Evaluating built-in functions on the GPU
- Running MATLAB code on the GPU
- Creating kernels from PTX files for execution on the GPU

# Contents


1 集群计算基础

2 集群软件

3 多线程

4 OpenMP

5 MPI

# 消息传递接口（ MPI ）

- MPI库函数
- 一个MPI并行程序由一组运行在相同或不同计算机上的进程构成。
- 进程组（process group）
- 进程号（rank）

# MPI

MPI 目前支持 C、C++和 FORTRAN 语言[1]，下面给出基于 C 语言的并行程序框架。

```
#include "并行库"
int main(int    argc , char **argv )
{
    使用 MPI_Init 函数初始化并行计算环境；
    使用 MPI_Comm_size 函数返回指定通信器中进程的数目；
    使用 MPI_Comm_rank 函数返回指定通信器中本进程的进程号；
    并行程序代码和进程通信代码；
    使用 MPI_Finalize 退出并行计算环境，并行程序结束；
    Return 0;
}
```

# DEMO

```
#include "mpi.h"
int main( argc, argv )
int  argc;     char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

# 通信模式

- 点对点通信（point to point communication）
  - 阻塞型：MPI_Send和MPI_Recv。
  - 非阻塞型：MPI_Isend和MPI_Irecv。

- 聚合通信(collective communication)
  - 一对多通信: 广播MPI_Bcast
  - 多对一通信：数据收集MPI_Gather，
  - 多对多通信：全交换MPI_Alltoall

# /* first.c */

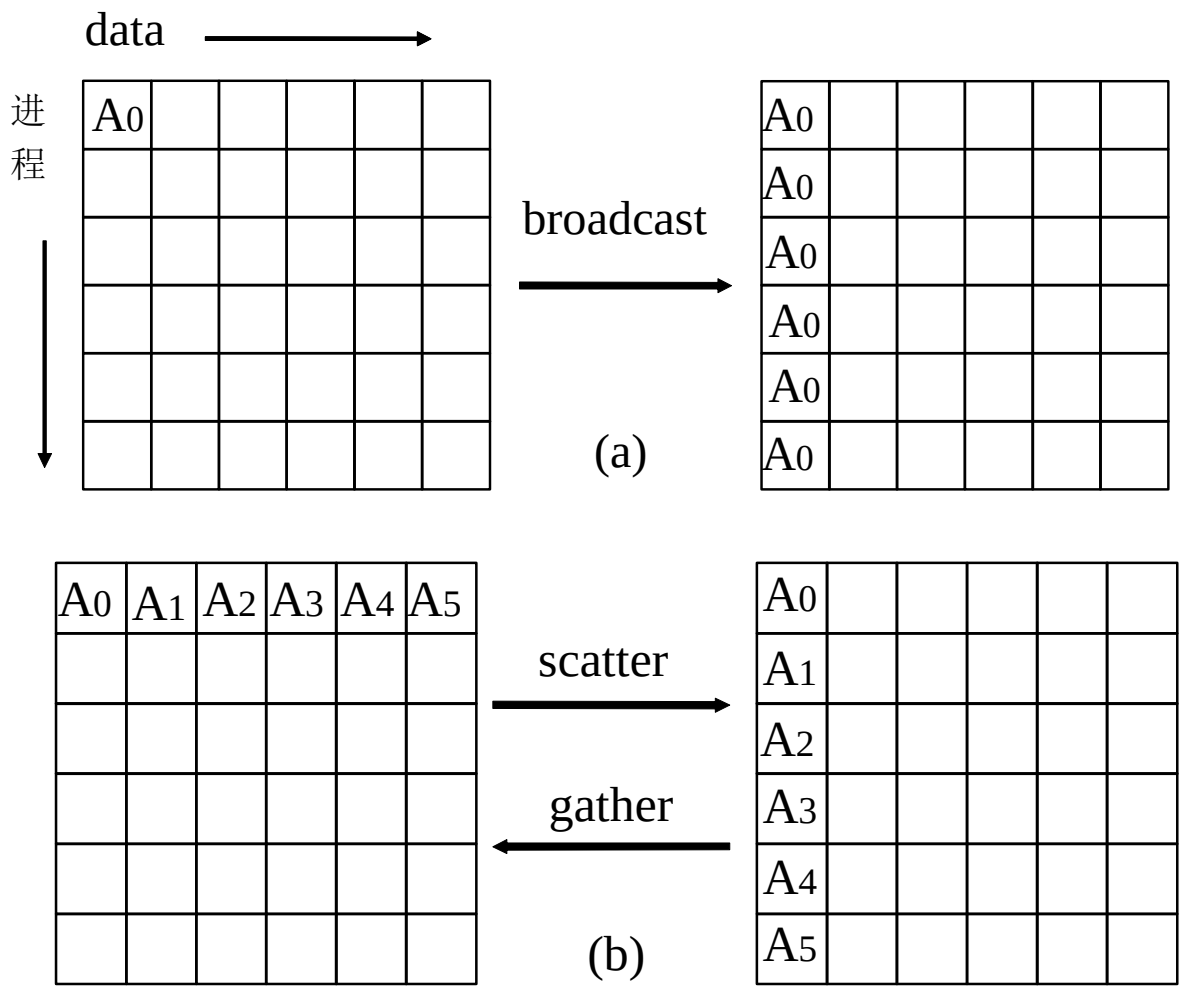
```
#include “mpi.h”        /*MPI的头函数，
int main(int   argc, char **  argv )
{
int rank, size, tag=333;
int buf[20]
MPI_Status status
MPI_Init( &argc, &argv );        /*MPI的初始化函数*/
MPI_Comm_rank( MPI_COMM_WORLD, &rank );      /*该进程的编号*/
MPI_Comm_size( MPI_COMM_WORLD, &size );      /*总的进程数目*/
if (rank==0)
    MPI_Send( buf, 20, MPI_Int, 1, tag, MPI_COMM_WORLD);
if (rank==1)
    MPI_Recv( buf, 20, MPI_Int, 0, tag, MPI_COMM_WORLD, &status);

MPI_Finalize();        /*MPI的结束函数*/
return 0;
}
```

# 聚合通信 (Collective Communication )

- 从一个进程到本组内的所有进程的播送 broadcast
- 从本组所有进程收集数据到一个进程gather
- 从一个进程分散数据到本组内的所有进程 sactter
- 求和，最大值，最小值及用户定义的函数等的 reduce操作

# 图例


data
进程
A0
broadcast
A0
A0
A0
A0
A0
A0
(a)
A0 A1 A2 A3 A4 A5
scatter
gather
A0
A1
A2
A3
A4
A5
(b)

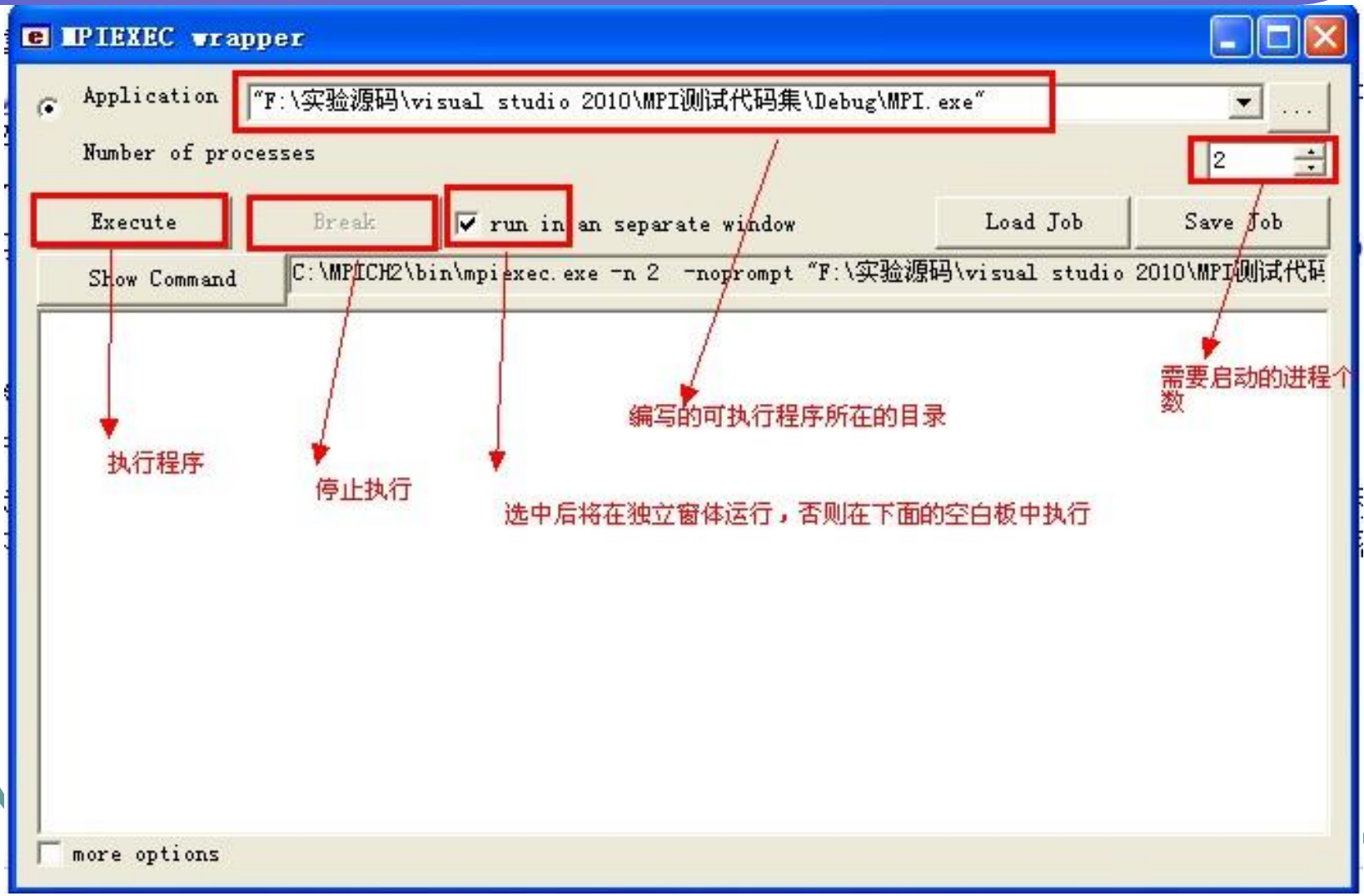
# 编译链接

- mpicc cpi.c -o cpi
- mpirun -np 2 -machinefile hostfile cpi
- np为运行mpi作业的CPU个数，hostfile为运行mpi作业的节点名
- xdio01 xdcn001

# Windows mpich

- 1、http://www.mpich.org/downloads/
- 2、路径包含，项目属性-配置属性-VC++目录，在包含目录和库目录中，分别添加MPI安装目录中的include和lib文件夹。

# 运行


MPIEXEC wrapper
Application
"F:\实验源码\visual studio 2010\MPI测试代码集\Debug\MPI.exe"
...
Number of processes
2
Execute
Break
run in an separate window
Load Job
Save Job
Show Command
C:\MPICH2\bin\mpiexec.exe -n 2 -noprompt "F:\实验源码\visual studio 2010\MPI测试代码
more options
执行程序
停止执行
选中后将在独立窗体运行，否则在下面的空白板中执行
编写的可执行程序所在的目录
需要启动的进程个数