



# 并行计算及计算可视化

李会民

hml@ustc.edu.cn

中国科学技术大学 网络信息中心 超级计算中心

2021-01



# 内容

① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息

李会民（中国科大超算中心）

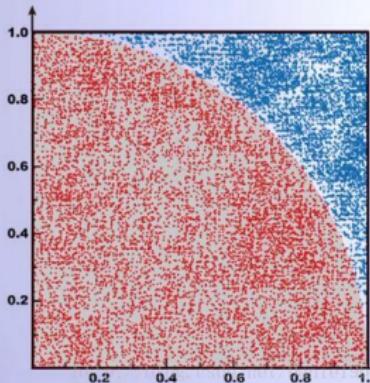
并行计算及计算可视化



# Monte Carlo 算法的缺点



家长帮 jzbg.com



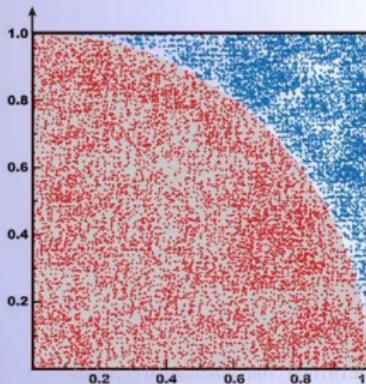


# Monte Carlo算法的缺点

- 对于基础风险因素仍然有一定的假设，存在一定的模型风险



家长帮|jzb.com



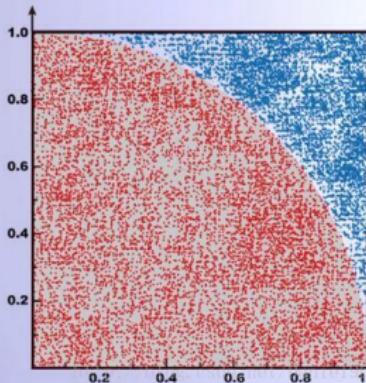


# Monte Carlo算法的缺点

- 对于基础风险因素仍然有一定的假设，存在一定的模型风险
- 如产生的数据序列是伪随机数，可能导致错误结果



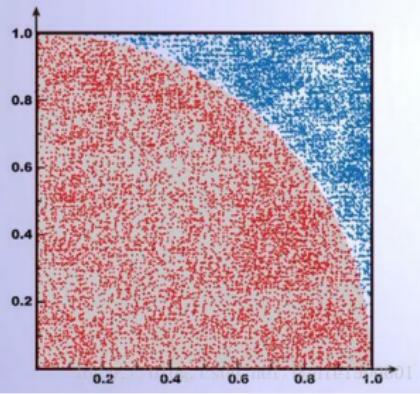
家长帮 jzbg.com





# Monte Carlo算法的缺点

- 对于基础风险因素仍然有一定的假设，存在一定的模型风险
- 如产生的数据序列是伪随机数，可能导致错误结果
- 计算量很大，且准确性的提高速度较慢
  - 如一个因素的准确性要提高10倍，就必须将模拟次数增加100倍以上

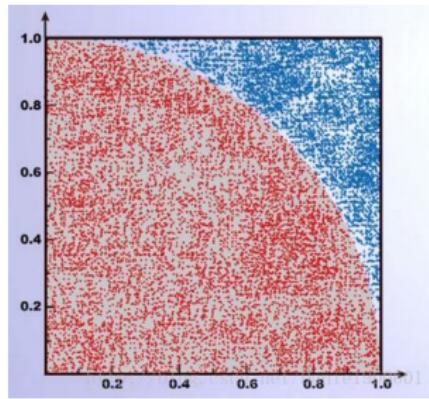




# Monte Carlo算法的缺点

- 对于基础风险因素仍然有一定的假设，存在一定的模型风险
- 如产生的数据序列是伪随机数，可能导致错误结果
- 计算量很大，且准确性的提高速度较慢
  - 如一个因素的准确性要提高10倍，就必须将模拟次数增加100倍以上

需要强大的计算能力





# 理论双精度浮点计算能力

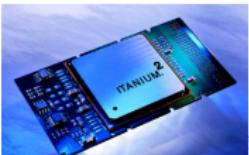
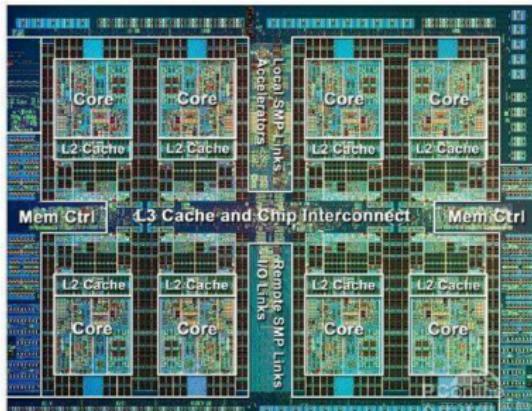
理论双精度浮点计算能力（峰值） = 处理器主频 × 处理器每个时钟周期执行双精度浮点运算的次数 × 系统中处理器核心数

- Intel Core i9 9900k CPU:

- 主频3.6GHz, 8核, 每时钟周期执行浮点运算次数为32
- 理论双精度浮点计算能力:  $3.6\text{GHz} \times 32\text{flops/Hz} \times 8\text{核} = 921.6\text{Gflops}$   
(0.9216万亿次/秒)

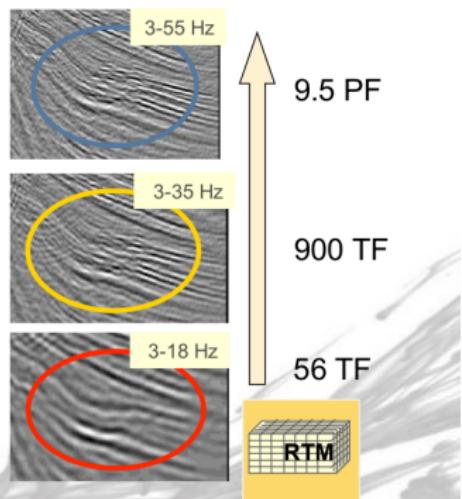
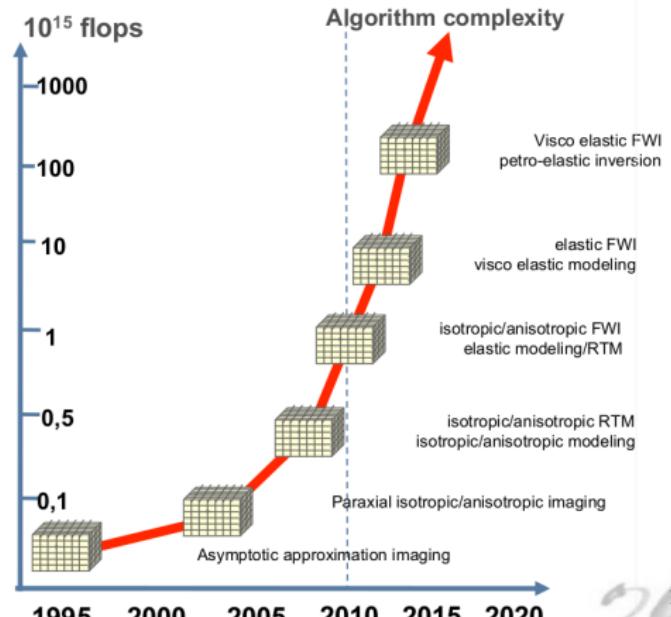
- Intel Xeon Scalable 6248 CPU:

- 主频2.5GHz, 20核, 每时钟周期执行浮点运算的次数为32
- 理论双精度浮点计算能力:  $2.5\text{GHz} \times 32\text{flops/Hz} \times 20\text{核} = 1.6\text{Tflops}$   
(1.6万亿次/秒)



# 计算需求

人类对于计算能力的需求无止境



如果将计算时间固定为8天，RTM下不同频率域细节的展现  
所需要的计算能力

石油勘探：叠前逆时深度偏移(Pre-stack Reverse-time Depth Migration, RTM)



# 计算需求

人类对于计算能力需求是无止境

研究领域	$N \times 10^{15}$ 次/秒		$N \times 10^{18}$ 次/秒
	PetaFLOPS	ExaFLOPS	
生命科学 	微量自由能量计算的分子对接 2000原子化学计算的宏观分子运动 微秒级分辨率	自由能量计算 20,000原子化学计算的宏观分子运动 毫秒级分辨率	
环境 	能源技术结果 100 - 1000世纪长地球系统模型模拟 50千米分辨率	高分辨率地球系统 10个模型配置，每个运行100 - 1000次长世纪模拟 1千米分辨率	
能源 	完全逆时、弹性、各向异性地震模拟	完全逆时、反向的耦合水波纹激发、大褶皱、高分辨率	
纳米科学 	有限几何体、动力学相变的分子运动	相互作用的生物分子、微米级机械 从头计算，数百万个的原子（需要修改算法）	
化学反应 	量子色动力学 等离子体 N - N 反应 12C 16O 3D 超新星	弱电子、光核 40Ca 全3D超新星	



# 计算机能干什么？



- 1996年, Kasparov Vs. Deep Blue(4:2)
- 1997年, Kasparov Vs. Deeper Blue(2.5:3.5)
- 2003年, Kasparov Vs. Deep Junior (3:3)
- 2003年, Kasparov Vs. X3D-Fritz (2:2)



# 超级电脑 “沃森” (Watson)

- 危险边缘:

- 美国的电视智力竞赛节目，比赛问题内容涵盖了历史、文学、艺术、流行文化、科技、体育、地理、文字游戏等多方面。
- 比赛采取一种独特的问答形式：参赛者须根据以答案形式提供的各种线索，以问题的形式作出正确的回答。



# 超级电脑 “沃森” (Watson)

- 危险边缘:

- 美国的电视智力竞赛节目，比赛问题内容涵盖了历史、文学、艺术、流行文化、科技、体育、地理、文字游戏等多方面。
- 比赛采取一种独特的问答形式：参赛者须根据以答案形式提供的各种线索，以问题的形式作出正确的回答。
- 2011年2月17日，由IBM和美国德克萨斯大学联合研制的超级电脑“沃森” (Watson)在美国最受欢迎的智力竞猜电视节目《危险边缘》中击败该节目历史上两位最成功的选手肯-詹宁斯和布拉德-鲁特，成为《危险边缘》节目新的王者。
  - 沃森由90台IBM服务器、360个CPU组成，是一个有10台普通冰箱那么大的计算机系统。它拥有15TB内存(Memory)、2880个CPU、每秒80TFlops运算。





# 为什么以前认为围棋中计算机无法战胜人类？



在2016年以前，人类发明的知名棋类游戏中，只有围棋未曾被人工智能攻克，人类棋手仍把持着黑白纹枰的绝对高峰。

- 围棋的方形棋盘有361个点，每个位点可能出现三种状态：黑棋、白棋或空格。
- 围棋在对局进程的大部分时间里，逻辑性弱于象棋、将棋或国际象棋，除非遇到局部的死活题或者某些“一本道”（围棋术语，意即只有一条正确路径，其余皆为错误）的情况，才是逻辑过程。
- 围棋的变化非常复杂，尤其是置之死地而后生的“扑”、生死循环的“劫”、貌生实死的“征子”和沧海桑田的“转换”将棋局的复杂度提升了若干数量级。
- 特别是“劫争”，可谓是牵一发而动全身，看似是局部的问题，其实关系到全局。这仅仅是围棋复杂性的表现之一。
- 更难的是围棋的形势判断，涉及到“虚”的地方，“厚薄”的判断是非常玄的。人类高手尚未掌握其中规律，各有各的见解，面对同样的局面，几位顶尖高手的判断可能截然相反。因此，根本无法把人类的理解转换成数学表达编入计算机程序，也无法验证其运算结果是否为正解。



# AlphaGo阿尔法围棋

- 阿尔法围棋(AlphaGo)是第一个击败人类职业围棋选手、第一个战胜围棋世界冠军的人工智能机器人，由谷歌(Google)旗下DeepMind公司戴密斯·哈萨比斯领衔的团队开发。其主要工作原理是“深度学习”。
- 2016年3月，阿尔法围棋与围棋世界冠军、职业九段棋手李世石进行围棋人机大战，以4比1的总比分获胜；2016年末2017年初，该程序在中国棋类网站上以“大师”(Master)为注册账号与中日韩数十位围棋高手进行快棋对决，连续60局无一败绩
- 2017年5月，它与排名第一的世界围棋冠军柯洁对战，以3比0的总比分获胜。围棋界公认阿尔法围棋的棋力已超过人类职业围棋顶尖水平，在GoRatings网站公布的世界职业围棋排名中，其等级分曾超过排名人类第一的棋手柯洁。
- AlphaGo团队2017年10月19日在《自然》杂志上发表文章介绍了AlphaGo Zero，这是一个没有用到人类数据的版本，比以前任何击败人类的版本都要强大。通过跟自己对战，AlphaGo Zero经过3天的学习，以100:0的成绩超越了AlphaGo Lee的实力，21天后达到了AlphaGo Master的水平，并在40天内超过了所有之前的版本。

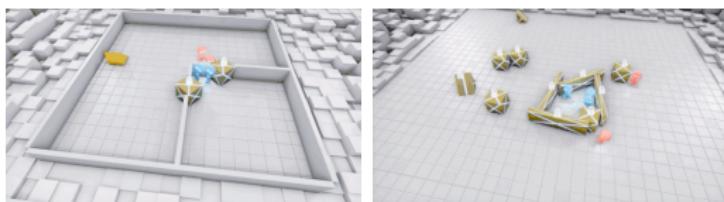




# 跟人形AI玩捉迷藏，你敢吗？

蓝色小人努力隐藏，而红色小人在复杂的地形中苦苦寻找，这场你死我活的对抗，不是CG动画，而是OpenAI的智能体真的在玩捉迷藏。

- 起初，AI们完全不知道自己能做什么，只是出于“本能”地逃跑、追逐
- 在2500万次游戏之后，小蓝人学会了通过移动箱子，建造庇护所，来保护自己不被发现
- 又经过了7500万场比赛，红鬼们会利用坡道闯进庇护所了
- 又吃了1000万次亏后，小蓝人们再建庇护所，知道把坡道也顺走了
- 更厉害的是，AI们不只是会单兵作战，还学会了团队协作
- 什么，你觉得地形太简单？在将近5亿次训练之后，AI们解锁了更复杂的版本



<https://m.huxiu.com/article/318985.html>

李会民（中国科大超算中心）

并行计算及计算可视化



# 科幻成为现实

只有足够强大的计算能力，才能成为可能





# 计算化学领域的黑科技

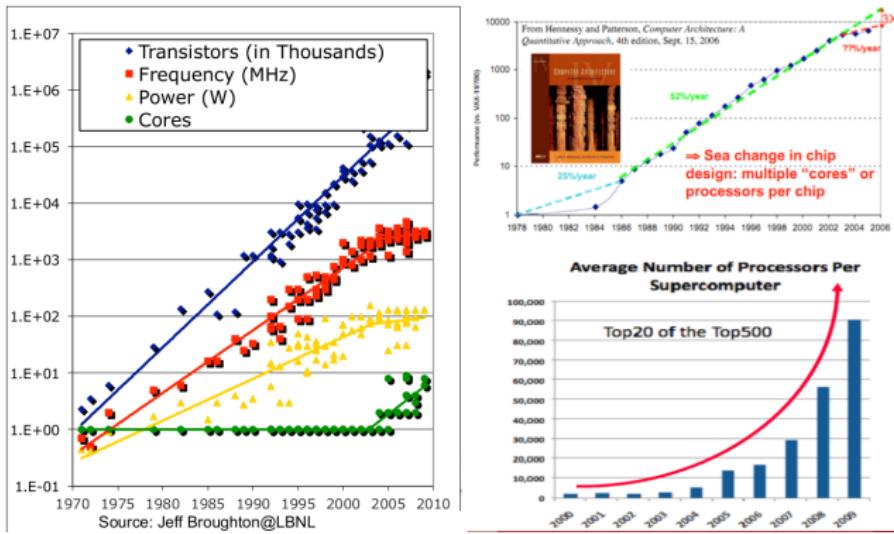
- D.E. Shaw是个学霸，斯坦福大学计算机专业的博士，30岁不到就进入哥伦比亚大学做教授，专门研究超大规模并行计算
- 但是Shaw觉得无聊，哥伦比亚大学地处纽约，遍地暴发的对冲基金男各种花天酒地，游嬉于各种model之间，作为一个同样聪明的教授，却只能坐在冷板凳上写计算机model。总之，Shaw不干了
- 他就开办了自己的对冲基金D.E. Shaw & Co. LP.，专注用计算机自动炒股、债和外汇，利用高速计算机网络和市场瞬间的有效性缺陷来进行高频统计套利
- 作为专门研究超大规模并行计算的顶级专家的Shaw，率先杀入高频交易，完全是流氓会武术，谁也挡不住，剪羊毛速度世界一流，很快人生进入了新的高峰。到2015年，他的个人净值已经41亿美元(David Shaw - Forbes)，杀入全球财富榜前500
- 有钱任性，不依赖国家经费，自己投资，Shaw的大砍刀就落到了萎靡的计算化学上，招了一批最顶尖的计算化学、生物物理、电子工程博士：计算化学的最大黑科技诞生了：它比一般的超级计算机快约10000倍，比最好的超算也快1000倍
- 从2007年起，D.E. Shaw的团队声名鹊起，用这个收割机每年在国际顶尖的学术杂志《自然》和《科学》上灌水，学术声誉不可阻挡



# CPU发展

## Theorem (摩尔定律)

当价格不变时，集成电路上可容纳的晶体管数目，约每隔18个月便会增加一倍，性能也将提升一倍。



近几年CPU在主频方面无大提升，主要向多核发展，比如8核、16核、32核、56核等。



# 高性能计算、超级计算、并行计算

- 一般不区分以下名词
  - 高性能计算: High Performance Computing, HPC
  - 超级计算: Supercomputing
  - 并行计算: Parallel Computing
- 通常指使用很多处理器（作为单个机器的一部分）或某一集群中组织的几台计算机（作为单个计算资源操作）的计算系统和环境
- 有许多类型的系统，其范围从标准计算机的大型集群，到高度专用的硬件
- 大多数基于集群的系统使用高性能网络互连（而不是以太网），如Mellanox InfiniBand/Intel OPA高速网
  - 低延迟: ~ns (纳秒)
  - 高带宽: 40、56、100、200Gbps
  - RDMA直接内存访问，无需经过CPU



# 2020年11月份Top500前十超算系统

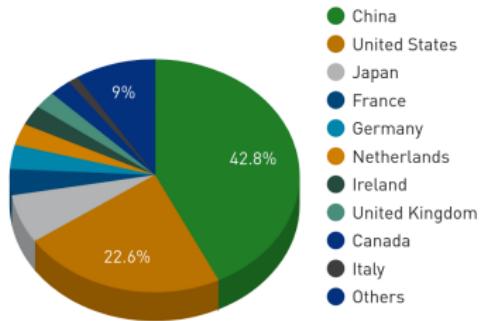
高性能计算系统排名: <http://www.top500.org>

Rank	System	Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu Interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	JUWELS Booster Module - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	HPC5 - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252



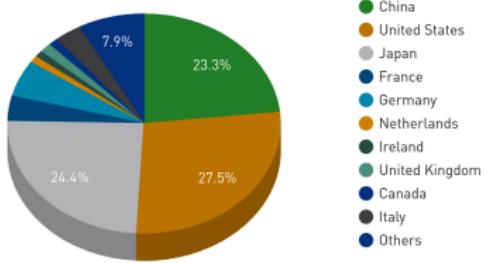
# 2020年11月份Top500国家分布

Countries System Share



(a) 套数

Countries Performance Share

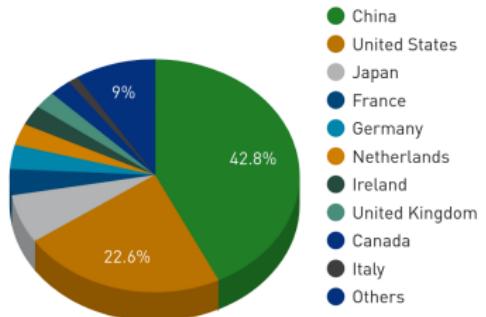


(b) 性能



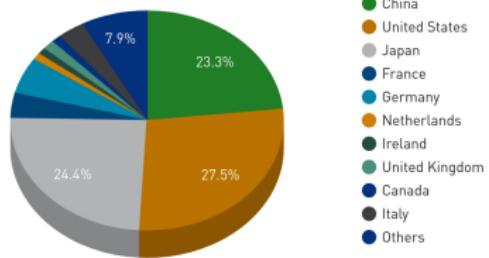
# 2020年11月份Top500国家分布

Countries System Share



(a) 套数

Countries Performance Share



(b) 性能

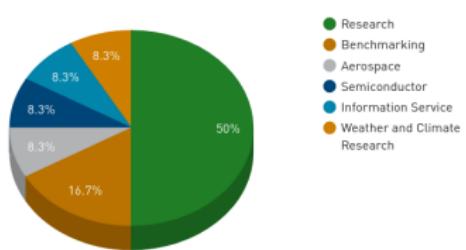
可惜我国

- 很多系统是互联网的，测完就拆分了，没有真正发挥大规模并行计算的价值
- 软件跟不上，主要用国外的
- 国产CPU等硬件存在兼容性及架构难用等生态问题，需要大力发展战略软件适配



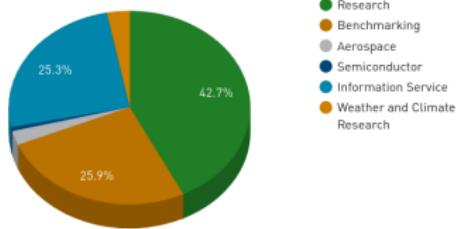
# 2020年11月份Top500应用领域分布

Application Area System Share



(a) 套数

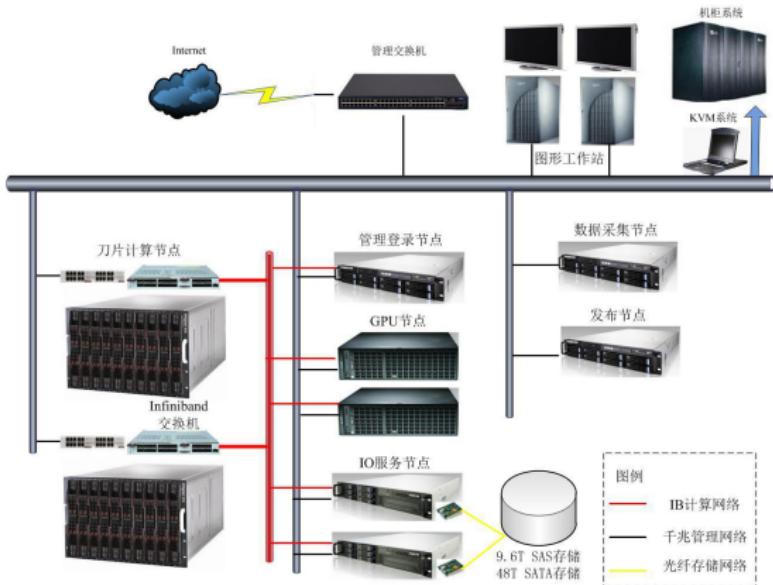
Application Area Performance Share



(b) 性能

- 气象分析和天气预报
- 制药企业的药理分析
- 科研人员的大型科学计算问题
- 石油勘探中对石油储量的分析
- 航空航天企业的设计和模拟
- 化工企业中对分子结构的分析计算
- 制造业中的CAD/CAM系统和模拟试验分析
- 银行和金融业对经济情况的分析
- 生物/生命科学中生物分子研究和基因工程计算
- 人工智能
- .....

# 超算集群结构



- 主节点：管理、IO、用户登录
- 计算节点：执行计算任务，多个节点
- 各节点间：通过网络连接，如InfiniBand、OmniPath等100Gb/s高速互联
- 操作系统：一般为Linux或Unix，Windows很少
- 用户使用：远程联网使用，比如通过SSH，需要通过作业调度系统提交作业，排队运行

# 校超算中心：瀚海20超级计算系统 I



主页: <http://scc.ustc.edu.cn/2019/1206/c435a407486/page.htm>

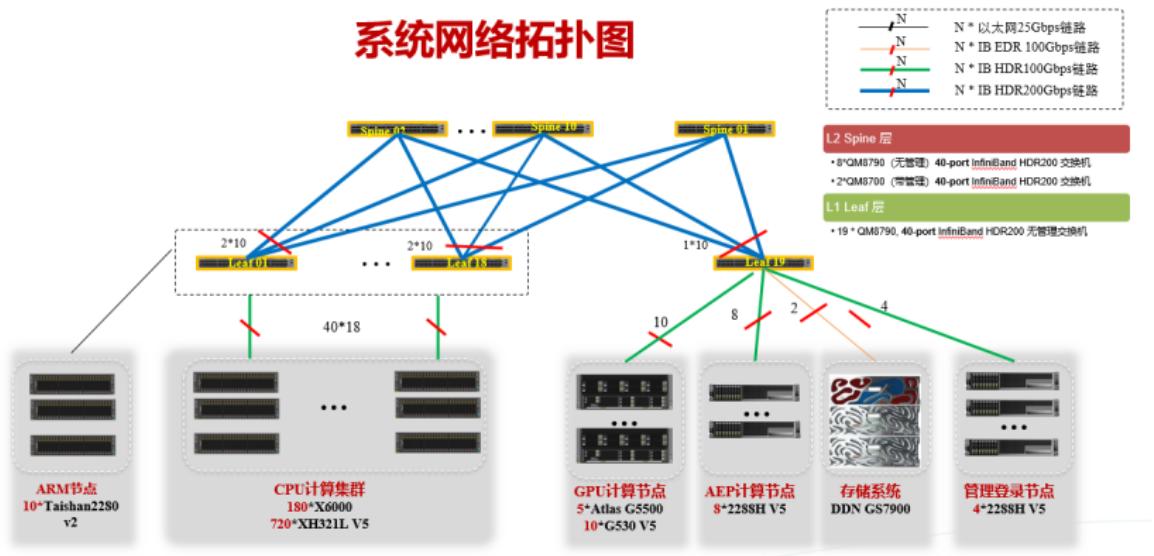


10个液冷机柜+3个风冷机柜

一个液冷机柜等于一辆超跑（约400万元）



## 系统网络拓扑图





- 价值5100万元
- 是目前国内高校最大的单套系统，共752个节点，30640颗CPU核心，20块NVIDIA Tesla V100 GPU卡，理论峰值计算能力达2.52千万亿次/秒，在2019年11月的Top500第371名
- 720个双路CPU计算节点采用华为XH321 V5液冷服务器，具有业界先进的板级高温液冷技术，支持常温水制冷，高效节能，单机柜密度高（72节点/机柜）功耗大（38KW/机柜）
- 具有10个双Nvidia Tesla V100 GPU计算节点，支持对GPU计算资源的需求
- 具有8个2TB Intel AEP大共享内存节点，性价比高于纯普通内存节点，支持大共享内存应用的需求
- 具有20个国产鲲鹏ARM CPU计算节点，部分有Atlas 300 AI计算卡，促进国产超算生态发展
- 采用Mellanox HDR 100Gbps高速计算互联，支持大规模并行



- 采用业界先进的长虹GS7990并行存储(OEM DDN)，可用容量1.5PB，GRIDScaler(GPFS)并行文件系统，持续IO聚合带宽超过20GB/s
- 系统采用无盘集群技术，管理维护及升级方便



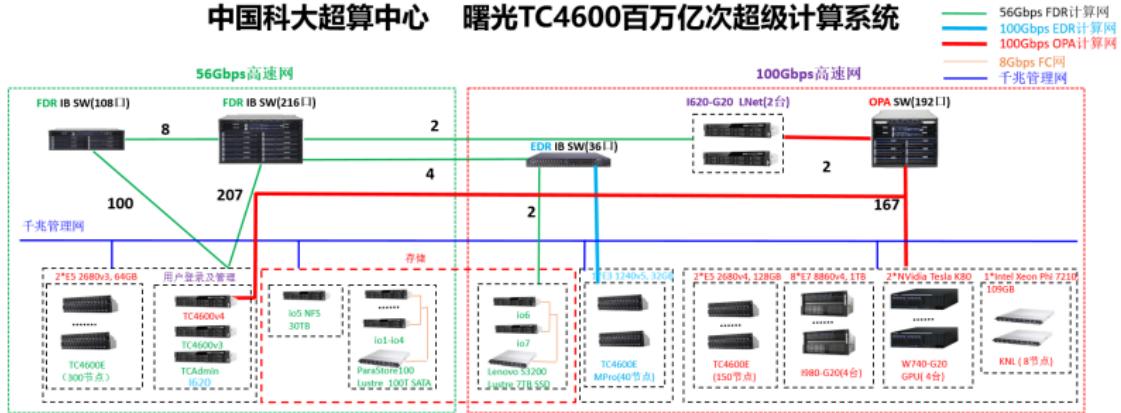
主页：<http://scc.ustc.edu.cn/2014/0930/c435a3035/page.htm>



# 校超算中心：曙光TC4600百万亿次超级计算系统 II



## 中国科大超算中心 曙光TC4600百万亿次超级计算系统



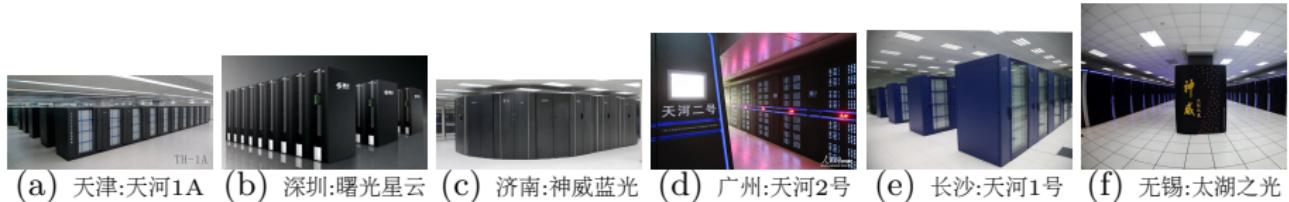


# 校超算中心：曙光TC4600百万亿次超级计算系统 III

- 价值2600万元
- 采用100Gbps和56Gbps高速互联
- 1个管理节点、2个用户登录节点、7个存储节点、2个Lustre LNet路由节点及506个计算节点构成
- 计算节点共：12248颗CPU核心，512颗Intel Xeon Phi融核(MIC) KNL核心和24块GPU卡
- 总双精度峰值计算能力为每秒648.53万亿次
  - CPU: 487.49万亿次/秒
  - GPU: 139.76万亿次/秒
  - Intel Xeon Phi融核: 21.28万亿次/秒



# 国家超级计算中心



- **天津**: “天河-1A”超级计算机系统，由国防科大研制，峰值性能每秒4700万亿次、LINPACK实测值持续性能每秒2507万亿次，配备了14336颗Intel Xeon X5670处理器、7168块基于NVIDIA “Fermi”架构的Tesla M2050计算卡、2048颗国防科大研制的飞腾处理器以及5PB存储设备。
- **深圳**: 曙光“星云”由曙光信息产业（北京）有限公司、中国科学院计算技术研究所、国家超级计算深圳中心共同研制，由曙光集团天津产业基地制造的一款拥有自主知识产权的超千万亿次超级计算机；是国内首台实测性能超千万亿次的超级计算机，其每秒系统峰值达三千万亿次(3PFlops)，每秒实测Linpack值达1271万亿次。
- **济南**: “神威蓝光计算机系统”，由国家并行计算机工程技术研究中心研制，系统采用万万亿次架构，全机装配8704片由国家高性能集成电路（上海）设计中心自主研发的“申威1600”处理器，峰值性能达到1.0706千万亿次浮点运算/秒，持续性能为0.796千万亿次浮点运算/秒，运行（LINPACK）效率达到74.4%。
- **广州**: “天河-2号”超级计算机系统，由国防科大研制，峰值性能每秒5.49万亿次、LINPACK实测值持续性能每秒3.39万亿次，由16000个节点组成，每个节点有2颗基于Ivy Bridge-E的Intel Xeon E5 2692处理器和3个Xeon Phi，累计共有32000颗Xeon E5 2692处理器和48000个Xeon Phi，总计有312万个计算核心。
- **长沙**: “天河-1号”超级计算机系统，全系统峰值计算性能1372万亿次，其中，全系统CPU峰值计算性能317.3万亿次，GPU峰值计算性能1054.7万亿次；全系统共包括4586个CPU，其中8路8核CPU 32个，2路6核和 8核CPU 4538个，4路6核CPU 16个，全系统共包括2048个GPU。
- **无锡**: “神威·太湖之光”高效能计算系统：系统峰值性能125.436PFlops，实测持续运算性能93.015PFlops；处理器：“申威26010”众核处理器；整机处理器个数：40960个；整机处理器核数：1064.96万；系统总内存：1.31PB。商用辅助计算系统：运算性能：1PFlops；980个普通计算节点：2路12核心、主频2.5GHz、内存128GB；32个胖计算节点：8路16核心、主频2.2GHz、内存1TB
- **郑州**: 峰值计算能力达到100Pflops，100PB存储



# 影响高性能计算性能的主要因素

- 硬件:

- CPU: 主频、并发数、Cache
- 内存: 主频、CL延迟 (CAS Latency, 内存存取数据所需的时间)、容量
- IO能力: 缓存、转速、接口速率
- 网络: 带宽、延迟

- 软件:

- 编译器、数值函数库、并行库

- 设置:

- 硬件
- 操作系统
- 软件



# 超算系统CPU特点

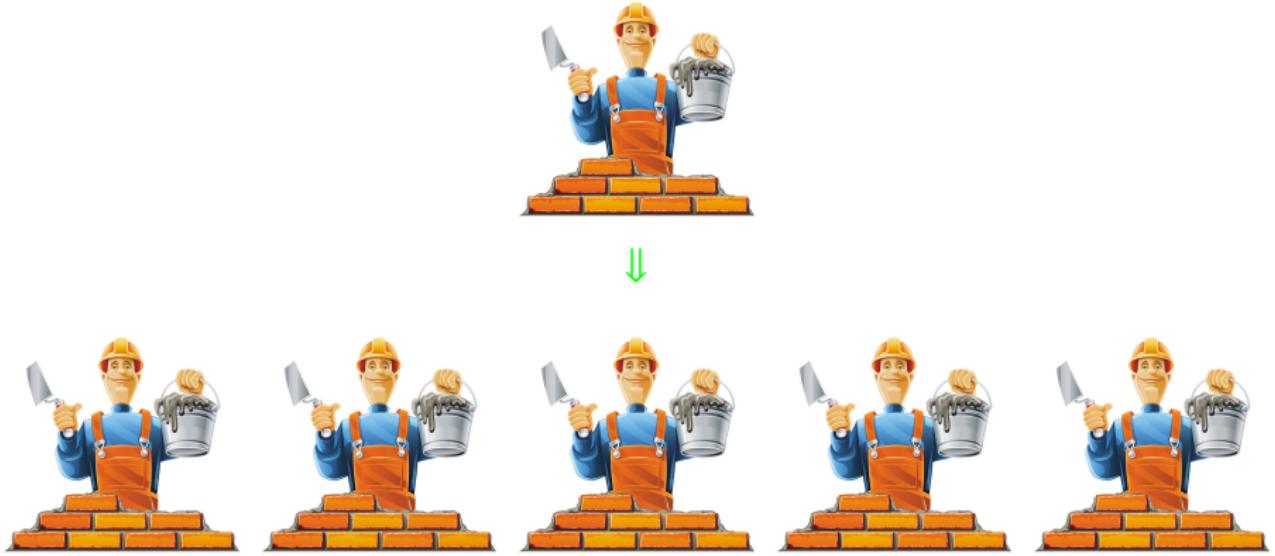
- 单核CPU性能并不高
  - 串行程序运行速度主要依赖于CPU主频
  - 单个串行程序速度不会提高，有可能比自己的计算机运行还慢
- 超算系统CPU核数非常多



# 怎么办?

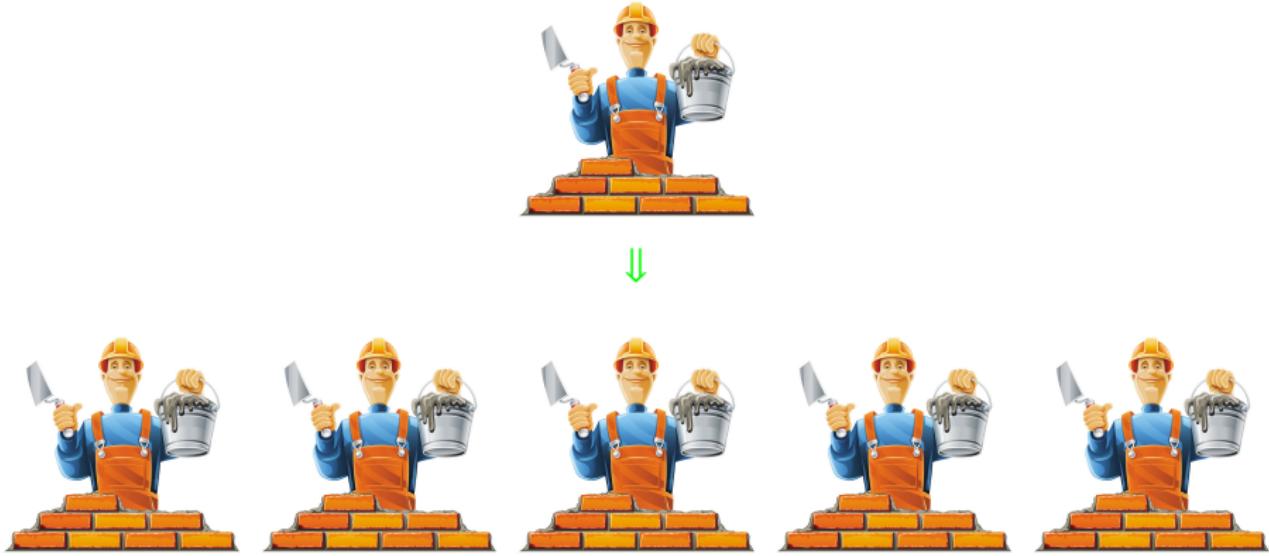
# 怎么办？

- 将任务进行分割，由多个计算核心共同完成 => 并行计算



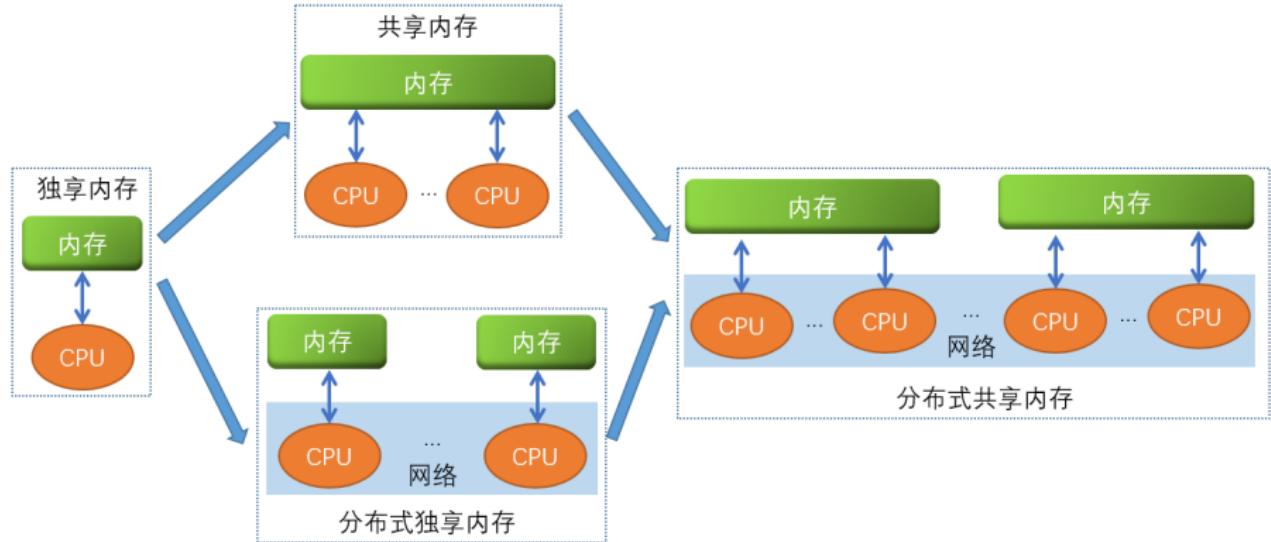
# 怎么办？

- 将任务进行分割，由多个计算核心共同完成 => 并行计算
  - 可以加快速度
  - 可以加大规模，更加宏大或微小的尺度
  - 完成单颗CPU无法完成的任务





# 计算模型



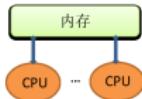


# 并行计算设计的分类 I

- 独享内存:
  - 各处理器使用各自内存



- 共享内存:
  - 多个处理器共享同一内存
  - ccNUMA(Cache-Coherent Non-Uniform Memory Access)、SMP(Symmetric MultiProcessing)
  - 大型共享内存超算系统: 技术要求高, 造价昂贵, 无法做得非常大
  - 适合程序: OpenMP、MPI





# 并行计算设计的分类 II

- 分布式独享内存:

- 每个处理器都有自己的内存
- 处理器之间通过网络传递消息交换信息
- Cluster、MPP(Massively Parallel Processing)
- 分布式内存超算系统: 成本低, 缩放性好, 交换数据时使用通信相比内存延迟大, 通信性能影响可扩展性
- 适合程序: MPI



- 分布式共享内存:

- 节点内部CPU核心共享节点内内存
- 节点之间分布式内存, CPU之间通过传递消息交换信息
- 分布式共享内存超算系统: 当前最多的超算架构
- 适合程序: OpenMP+MPI





# 并行化分解方法

- 任务分解：多任务并发执行
  - 如：计算不同的条件
- 功能分解：分解被执行的计算
  - 如：分别做傅立叶变换、矩阵转秩等不同的计算
- 区域分解：分解被执行的数据
  - 如：计算矩阵不同的部分



① 高性能计算、超级计算、并行计算

② MPI并行编程

- MPI简介
- MPI编程
- MPI点对点通信
- MPI集合通信
- MPI数据的打包与解包
- MPI高级功能
- MPI程序的编译与运行

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息



# MPI简介

- MPI(Message Passing Interface)是1994年5月发布的一种消息传递接口标准
- MPI并不是一种编程语言
- 实际上是一个消息传递函数库的标准说明，以语言独立的形式来定义这个接口库，并提供了与C/C++和Fortran语言的绑定
- 主要站点：MPI-Forum: <http://www.mpi-forum.org/>



# MPI的历史

- MPI初稿：美国并行计算中心工作会议（1992年4月）
- MPI-1提出讨论：第一届MPI大会（1993年1月）
- 标准由MPI Forum发布：
  - MPI-4.0：努力中……
  - MPI-3.1：2015-6
  - MPI-3.0：2012-9
  - MPI-2.2：2009-9
  - MPI-2.1：2008-9
  - MPI-1.3：2008-5
  - MPI-2：1997-7
  - MPI-1.2：1997-7
  - MPI-1.1：1995-6
  - MPI-1.0：1994-5



# MPI的主流实现

- MPICH: 最广泛的MPI实现  
<https://www.mpich.org/>
- MVAPICH和MVAPICH2: 针对InfiniBand网络的MPICH版本  
<http://mvapich.cse.ohio-state.edu/>
- MPICH-MX和MPICH-MX2: 针对Myrinet网络的MPICH版本  
<http://www.myri.com/scs/download-mpichmx.html>
- Open MPI: 建立在FT-MPI、LA-MPI、LAM/MPI等之上  
<http://www.open-mpi.org/>
- Intel MPI: 建立在MPICH2和MVAPICH2之上  
<http://software.intel.com/en-us/intel-mpi-library/>
- LAM(Local Area Multicomputer): Ohio State University开发  
<http://www.lam-mpi.org>



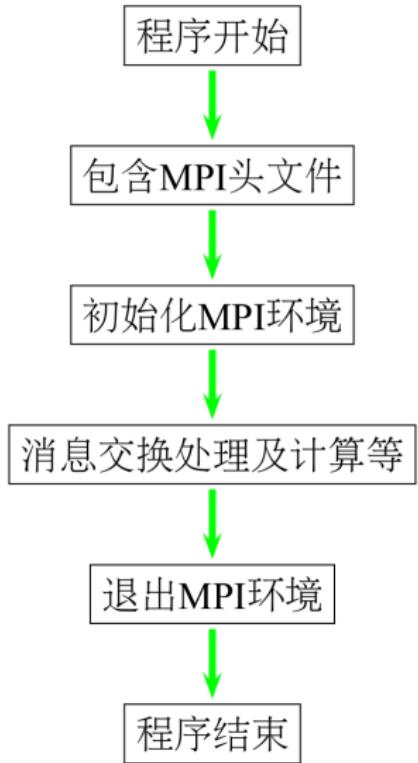
# MPI编程

- MPI为程序员提供一个并行环境库，程序员通过调用MPI的库函数来达到程序员所要的并行目的
- 只使用其中的**六个**最基本的函数就能编写一个完整的MPI程序去求解很多问题
- 这六个基本函数，包括启动和结束MPI环境，识别进程以及发送和接收消息：

MPI_INIT	启动MPI环境
MPI_COMM_SIZE	确定进程数
MPI_COMM_RANK	确定自己的进程标识符
MPI_SEND	发送一条消息
MPI_RECV	接收一条消息
MPI_FINALIZE	结束MPI环境



# MPI程序的一般结构





# 头文件

MPI程序要求所有包含MPI调用的程序应加入MPI文件头:

- C: `#include "mpi.h"`
- Fortran:
  - F77: `include 'mpif.h'`
  - F90及之后: `use mpi`或`include 'mpif.h'`

注意: mpif.h是文件名, 对于文件名

- Fortran虽然不区分大小写
- 但Linux系统区分大小写, 因此这里的mpif.h必须为小写
- 在Windows系统下不存在此问题
- 但一般超算系统为Linux, 因此还是小写为好



# 通信因子和组

- 通讯因子定义了进程组内或组间通讯的上下文（具体就是指明通讯链路的数据结构指针）
- MPI通过指定通信因子和组来对进程进行一种逻辑上的划分
- MPI\_COMM\_WORLD通信因子在MPI环境初始化过程中创建



# 进程号 (rank)

- 在一个通信因子中，每个进程都有一个唯一的整数标识符，称作“进程号”
- 进程号是从0开始的连续整数，编号为：0到进程数-1
- 用进程号可以
  - 控制程序的不同部分在不同的进程中并行运行

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
IF (MYID == 0) THEN
    0号进程运行的程序段
ELSE IF (MYID == 1) THEN
    1号进程运行的程序段
ENDIF
```

- 程序相同部分在不同进程中变量值不一样

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
A=A+MYID
```



# MPI消息

- MPI消息包括信封和数据两部分
- 信封：指出了发送或接收消息的对象及相关信息，含：<源/目进程号，标识，通信域>
- 数据：是本消息将要传递的内容，含：<起始地址，数据个数，数据类型>





# 隐含约定

- MPI函数的命名规则
  - 函数名形式为MPI\_Class\_action\_subset、MPI\_Class\_action、MPI\_Action\_subset或MPI\_Action
  - C语言：MPI和第一个\_后的首字符为大写，其余为小写
  - Fortran语言：
    - 函数名不区分大小写
    - Fortran函数除了MPI\_Wtime、MPI\_Wtick外，比C函数多一个参数IERROR
- 在MPI标准中MPI函数采用语言独立的方式说明，参数用IN、OUT或INOUT标记
  - IN：变量为输入给函数的，调用后其值不变
  - OUT：变量不是输入给函数的，其值被函数调用后更新
  - INOUT：变量既是输入给函数的，调用后也会被更新



# MPI的基本函数

利用以下六条基本函数即可完成MPI并行程序

- MPI\_Init
- MPI\_Comm\_size
- MPI\_Comm\_rank
- MPI\_Send
- MPI\_Recv
- MPI\_Finalize



# MPI\_Init: 初始化MPI环境

- 初始化MPI运行环境
- 必须调用；首先调用；调用一次
- 每个进程都有一个参数表

```
int MPI_Init( int *argc, char ***argv )
```

```
    MPI_INIT( IERROR )
        INTEGER IERROR
```

注：

- 蓝色表示Fortran语法，第一行为函数，后面的行为变量声明
- 之上的表示C语法



# MPI\_Comm\_size: 取得进程数

- 返回与该组通信因子相关的进程数，存在SIZE变量中
- 通讯因子必须是组内通讯因子

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

```
MPI_COMM_SIZE( COMM, SIZE, IERROR )  
INTEGER COMM, SIZE, IERROR
```



# MPI\_Comm\_rank: 取得进程号

- 返回该进程在指定通信因子中的进程号（0~进程数-1），存在RANK变量中
- 一个进程在不同通信因子中的进程号可能不同

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

```
MPI_COMM_RANK( COMM, RANK, IERROR )
```

```
INTEGER COMM, SIZE, IERROR
```



# MPI\_Send: 发送消息

发送buf缓冲区中的count个datatype数据类型的数据发送到dest目的进程，且带有tag标记

`MPI_SEND(buf, count, datatype, dest, tag, comm)`

IN buf 发送缓存的起始地址（选择型）

IN count 发送缓存的元素的个数（非负整数）

IN datatype 每个发送缓存元素的数据类型（句柄）

IN dest 目的地进程号（整型）

IN tag 消息标志（整型）

IN comm 通信子（句柄）

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

`MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)`

`<type> BUF(*)`

`INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR`



# MPI\_Recv: 接收消息

- 从指定的source进程接收带有tag标记消息，按datatype数据类型存到buf缓冲区
- 收到的消息所包含的数据元素的个数最多不能超过count

`MPI_RECV(buf, count, datatype, source, tag, comm, status)`

OUT buf 接收缓存的起始地址（选择型）

IN count 接收缓存中元素的个数（整型）

IN datatype 每个接收缓存元素的数据类型（句柄）

IN source 发送操作的进程号（整型），可为MPI\_ANY\_SOURCE

IN tag 消息的标识（整型），可为MPI\_ANY\_TAG

IN comm 通信组（句柄）

OUT status 状态对象（状态）

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm  
comm, MPI_Status *status)
```

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)  
<type>BUF(*  
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), I
```



# 注意避免死锁

下面例子各进程都在等待对方进程先发来再发送，导致死锁

```
if (MyID == 0) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,STATUS,iErr)
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
elseif (MyID == 1) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,iErr)
endif
```

下面两个例子不存在死锁

```
if (MyID == 0) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,STATUS,iErr)
elseif (MyID == 1) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,iErr)
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
endif
```

都主动发

```
if (MyID == 0) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,STATUS,iErr)
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
elseif (MyID == 1) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,iErr)
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
endif
```

至少一个主动发



# MPI\_Finalize: 结束MPI环境

## 结束MPI执行环境

- 该函数一旦被应用程序调用时，就不能调用MPI的其它例行函数（包括MPI\_Init）
- 必须保证在进程调用MPI\_Finalize之前完成与进程有关的所有通信

```
int MPI_Finalize( void )
```

```
MPI_FINALIZE( IERROR )
```

```
INTEGER IERROR
```



# 参数说明

- 缓冲区 (buffer)
- 数据个数 (count)
- 数据类型 (type)
- 目的地 (dest)
- 源 (source)
- 标识符 (tag)
- 通信因子 (comm)
- 状态 (status)



# 缓冲区 (buffer)

- 应用程序定义的用于发送或接收数据的消息缓冲区
- 可以理解为变量地址



# 数据个数 (count)

- 发送或接收指定数据类型的个数
- 用户实际传递的消息长度 = 数据类型的长度\*数据个数



# 数据类型 (type)

MPI定义了一些缺省的数据类型，用户也可根据需要建立自己的类型

## MPI定义的与C数据类型的对应关系

MPI数据类型	C数据类型	MPI数据类型	C数据类型
MPI_CHAR	char	MPI_C_BOOL	_Bool
MPI_SHORT	signed short int	MPI_INT8_T	int8_t
MPI_INT	signed int	MPI_INT16_T	int16_t
MPI_LONG	signed long int	MPI_INT32_T	int32_t
MPI_LONG_LONG_INT	signed long long int	MPI_INT64_T	int64_t
MPI_LONG_LONG (as a synonym)	signed long long int	MPI_UINT8_T	uint8_t
MPI_SIGNED_CHAR	signed char	MPI_UINT16_T	uint16_t
MPI_UNSIGNED_CHAR	unsigned char	MPI_UINT32_T	uint32_t
MPI_UNSIGNED_SHORT	unsigned short int	MPI_UINT64_T	uint64_t
MPI_UNSIGNED	unsigned int	MPI_C_COMPLEX	float_Complex
MPI_UNSIGNED_LONG	unsigned long int	MPI_C_FLOAT_COMPLEX (as a synonym)	float_Complex
MPI_UNSIGNED_LONG_LONG	unsigned long long int	MPI_C_DOUBLE_COMPLEX	double_Complex
MPI_FLOAT	float	MPI_C_LONG_DOUBLE_COMPLEX	long_double_Complex
MPI_DOUBLE	double	MPI_BYTE	8 binary digits
MPI_LONG_DOUBLE	long double	MPI_PACKED	MPI_Pack打包和 MPI_Unpack解包的数据
MPI_WCHAR	wchar_t (<std-def.h>中定义)		



# MPI定义的与Fortran数据类型的对应关系

## MPI定义的与Fortran数据类型的对应关系

MPI数据类型	Fortran数据类型
MPI_CHARACTER	character(1)
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack MPI_Unpack



# 目的地 (dest)

发送进程指定的接收该消息的目的进程，也就是接收进程的进程号



- 接收进程指定的发送该消息的源进程，也就是发送进程的进程号
- 如该值为MPI\_ANY\_SOURCE表示接收任意源进程发来的消息



# 标识符 (tag)

- 由程序员指定的为标识一个消息的唯一非负整数值 (0-32767)
- 发送操作和接收操作的标识符一定要匹配
- 对于接收操作来说，如tag指定为MPI\_ANY\_TAG则可与任何发送操作的tag相匹配



# 通信因子 (comm)

- 包含源与目的进程的一组上下文相关的进程集合
- 除非用户自己定义（创建）了新的通信因子，否则一般使用系统预先定义的全局通信因子MPI\_COMM\_WORLD



# 状态 (status)

- 对接收操作，包含接收消息的源进程（source）和标识符（tag）
- 在C程序中，是个包含三个成员的结构体：
  - status.MPI\_SOURCE: 发送数据的进程标识
  - status.MPI\_TAG: 发送数据使用的tag标识
  - status.MPI\_ERROR: 接收操作返回的错误代码
- 在Fortran程序中，是包含MPI\_STATUS\_SIZE个整数的数组：
  - status(MPI\_SOURCE): 发送数据的进程标识
  - status(MPI\_TAG): 发送数据使用的tag标识
  - status(MPI\_ERROR): 接收操作返回的错误代码
- 相当于一种接收方对消息的监测机制，并且以其为依据对消息作出不同的处理（当用通配符接受消息时）



# Fortran版本简单例子

```
use mpi !使用MPI模块
integer myid, ierr, count1, tag1, tag2, mysize, status1(400)
real buf1(10),buf2(10),buf3(1)
count1=5
tag1=1
tag2=2
buf1=1.0
buf2=2.0

call mpi_init(ierr) !初始化MPI环境
call mpi_comm_rank(mpi_comm_world, myid, ierr) !获取进程号，存储在myid中
call mpi_comm_size(mpi_comm_world, mysize, ierr) !获取进程数，存储在mysize中
if (myid .eq. 0) then      !下行向1号进程发送buf1中count1个浮点数的数据
    call mpi_send(buf1, count1, mpi_real, 1, tag1, mpi_comm_world, ierr)
    buf1=10.0                !下行向1号进程发送buf1中count1个浮点数的数据
    call mpi_send(buf1, count1, mpi_real, 1, tag2, mpi_comm_world, ierr)
else if (myid .eq. 1) then   !下行从0号进程接收count1个浮点数的数据存储在buf2中
    call mpi_recv(buf2, count1, mpi_real, 0, tag2, mpi_comm_world, status1, ierr)
    print*, 'recv',buf2        !下行从0号进程接收count1个浮点数的数据存储在buf2中
    call mpi_recv(buf2, count1, mpi_real, 0, tag1, mpi_comm_world, status1, ierr)
    print*, 'recv',buf2
end if
call mpi_barrier(mpi_comm_world,ierr) !各进程间同步
call mpi_finalize(ierr) !结束MPI环境
end
```



# C版本简单例子

```
#include <stdio.h>
#include "mpi.h" //包含MPI头文件
main(int argc, char **argv)
{
    int NumProcs,MyID,i,j,k;
    MPI_Status status;
    char msg[20];
    MPI_Init(&argc, &argv); //初始化MPI环境
    MPI_Comm_size(MPI_COMM_WORLD, &NumProcs); //获取进程数，存储在NumProcs中
    MPI_Comm_rank(MPI_COMM_WORLD, &MyID); //获取进程号，存储在MyID中
    if(MyID == 0){
        strcpy(msg,"HelloWorld"); //下行向1号进程发送msg中strlen(msg)+1个字符串数据
        MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }else if(MyID ==1){ //下行从0号进程接收20个字符串数据存储在msg中
        MPI_Recv(msg, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("Receive message=%s\n",msg);
    }
    MPI_Finalize(); //结束MPI环境
}
```

注意，因为有些是变量地址，某些变量前需要有&



# 点对点通信：阻塞发送

- MPI\_SEND(buf, count, datatype, dest, tag, comm): 标准模式
- MPI\_RECV(buf, count, datatype, source, tag, comm, status)
- MPI\_BSEND(buf, count, datatype, dest, tag, comm): 缓存模式
- MPI\_SSEND(buf, count, datatype, dest, tag, comm): 同步模式
- MPI\_RSEND(buf, count, datatype, dest, tag, comm): 准备好模式



# 三种附加通信模式

- 无论一个匹配接收是否已登入，能开始一个“缓存模式”的发送操作。它可以在一个匹配接收登入以前完成。但是，不象标准发送，这个操作是局部的，它的完成不依赖一个匹配接收的发生。因此，如执行一个发送而没有匹配接收登入，那么MPI必须缓存正发出的消息，以便允许发送调用完成。如没有充足的缓存空间，一个错误将发生。可得到的缓存空间数量由用户控制。为使缓存模式有效可以要求用户分配缓存。

**只要一可能，消息就被发送。**

- 无论一个匹配接收是否已登入，能开始“同步模式”的一个发送操作。但是，只有一个匹配接收登入，接收操作已开始接收同步发送的消息时，发送操作将成功完成。所以，一个同步发送的完成不表示发送缓存能被再使用，但是表明接收者已到达执行的某一点，即它已开始执行匹配接收。如发送和接收都是阻塞操作，那么同步模式的使用提供同步语义：两个进程在通信时聚会以前，一个通信不能完成。在这个模式下执行的一个发送是非局部的。

**发送者发一个“请求发送”的消息。接收者存储这个请求。当一个匹配接收登入时，接收者发回一个“允许发送”的消息，这时接收者发送消息。**

- 只要匹配接收已登入，可以开始一个“准备好通信”模式的发送。否则，这个操作是错误的，其结果是无定义的。在某些系统，这允许移出一个信号交换式操作，并导致提高性能。发送操作的完成不依赖一个匹配接收的状态，只表明发送缓存能被再使用。准备好模式的一个发送操作，与一个标准发送操作或一个同步发送操作有相同的语义；只是发送者给系统提供附加的信息（即：一个匹配接收已登入），能节省一些额外开销。因此，在一个正确的程序中，一个准备好发送能被一个标准发送替代，对程序的动作无影响而对性能有影响。

**发送者把消息拷贝到一个缓存，然后以非阻塞方式发送它（使用与标准发送同样的协议）。**



# 非阻塞通信

- 通过重叠通信和计算在许多系统能提高性能
- 当数据已被从发送缓存拷出时，发送完成调用返回
- MPI\_ISEND(buf, count, datatype, dest, tag, comm, request): 标准模式
- MPI\_IRecv(buf, count, datatype, source, tag, comm, request): 接收，只有一种标准模式
- MPI\_IBSEND(buf, count, datatype, dest, tag, comm, request): 缓存模式
- MPI\_ISSEND(buf, count, datatype, dest, tag, comm, request): 同步模式
- MPI\_IRSEND(buf, count, datatype, dest, tag, comm, request): 准备好模式

**注：**多了个请求句柄（request参数），在C语言中为MPI\_Request类型，在Fortran中为INTEGER类型。



# 等待通信完成和检测通信是否完成

- MPI\_WAIT(request, status): 等待检测的通信完成
- MPI\_WAITANY(count, array\_of\_requests, index, status): 等待任意一个检测的通信完成
- MPI\_WAITALL(count, array\_of\_requests, array\_of\_statuses): 等待所有检测的通信都完成
- MPI\_WAITSOME(incount, array\_of\_requests, outcount, array\_of\_indices, array\_of\_statuses): 等待有些检测的通信完成
- MPI\_TEST(request, flag, status): 测试检测的通信是否完成
- MPI\_TESTANY(count, array\_of\_requests, index, ag, status): 测试是否任意一个检测的通信完成
- MPI\_TESTALL(count, array\_of\_requests, ag, array\_of\_statuses): 测试是否所有要检测的通信都完成
- MPI\_TESTSOME(incount, array\_of\_requests, outcount, array\_of\_indices, array\_of\_statuses): 测试是否有些检测的通信完成
- MPI\_REQUEST\_GET\_STATUS(request, ag, status): 获取检测的通信的状态, 与WAIT和TEST不同, 并不释放句柄
- MPI\_REQUEST\_FREE(request): 释放某个通信

WAIT是要等待满足某个条件后完成, TEST是检测以后即完成。



# 探测和取消

- MPI\_IProbe(source, tag, comm, flag, status): 非阻塞探测进程source是否发标记为tag的消息来，状态存储在flag
- MPI\_Probe(source, tag, comm, status): 阻塞探测进程source是否发标记为tag的消息来
- MPI\_Cancel(request): 取消通信
- MPI\_Test\_Canceled(status, flag): 探测通信是否已取消



# 坚持式通信请求

当重复发送接收同样参数的信息时，可以利用下面坚持式通信方式得到更高的通信性能

- 初始化发送和接收

- MPI\_SEND\_INIT(buf, count, datatype, dest, tag, comm, request)
- MPI\_BSEND\_INIT(buf, count, datatype, dest, tag, comm, request)
- MPI\_SSEND\_INIT(buf, count, datatype, dest, tag, comm, request)
- MPI\_RSEND\_INIT(buf, count, datatype, dest, tag, comm, request)
- MPI\_RECV\_INIT(buf, count, datatype, dest, tag, comm, request)
- MPI\_START(request): 开始某个通信
- MPI\_STARTALL(count, array\_of\_requests): 开始所有通信
- MPI\_REQUEST\_FREE(request): 释放某个通信



# 发送接收和空进程

- MPI\_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status): 发送接收使用不同的缓冲区
- MPI\_SENDRECV\_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status): 发送接收使用同样缓冲区
- MPI\_PROC\_NULL: 空进程 (虚拟进程), 对此进程的发送或接收立即返回, 在某些情况下编程采用空进程会更加方便

**注意:** 目的地与源、数据个数、数据类型等都可以不一样, 可以用普通的MPI\_SEND与MPI\_RECV来接收, 只要匹配集合, 没必要非也得用MPI\_SEDNRECV等



# 集合通信

- 所有组成员间的栅障同步 (barrier synchronization)
- 一个成员到组内所有成员的广播 (broadcast) 通信
- 一个成员从所有组成员收集 (gather) 数据
- 一个成员向组内所有成员分散 (scatter) 数据
- 组内所有成员都接收结果 (allgather)
- 组内所有成员到所有成员间的分散/收集数据操作 (alltoall)
- 全局归约 (global reduction) 操作如求和 (sum)，求极大值 (max)，或用户自定义的操作，结果返回给所有的组成员或仅返回给其中的一个成员
- 组合归约 (combined reduction) 和分散操作
- 组内所有成员上的搜索 (scan) 操作 (也称前置操作)

**注意：**对所有集合通讯，每个进程都需要执行到这里才能执行，千万不要用进程号或其他方式限制到某个进程来执行，也不需要再采用 MPI\_SEND和MPI\_RECV等来发送接收，否则会导致死锁



# 集合通信函数列表

- MPI\_BARRIER(comm): 同步
- MPI\_BCAST(buffer, count, datatype, root, comm): 广播
- MPI\_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm): 收集
- MPI\_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm): 带有偏移的收集
- MPI\_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm): 散发
- MPI\_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm): 带有偏移的散发
- MPI\_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm): 所有进程都收集
- MPI\_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm): 带有偏移的所有进程都收集
- MPI\_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm): 全交换
- MPI\_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm): 规约
- MPI\_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm): 全部进程都进行规约
- MPI\_REDUCE\_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm): 规约后散发
- MPI\_SCAN(sendbuf, recvbuf, count, datatype, op, comm): 扫描, 又称前置规约, 对排在此进程前面的进程和此进程的数据进行规约
- MPI\_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm): 排它扫描, 对排在此进程前面的进程的数据进行规约

MPI\_REDUCE、MPI\_ALLREDUCE、MPI\_REDUCE\_SCATTER和MPI\_SCAN的规约运算op可为:

- MPI\_MAX: 最大值
- MPI\_MIN: 最小值
- MPI\_SUM: 求和
- MPI\_PROD: 乘积
- MPI\_LAND: 逻辑与
- MPI\_BAND: 位与
- MPI\_LOR: 逻辑或
- MPI\_BOR: 位或
- MPI\_LXOR: 逻辑异或
- MPI\_BXOR: 位异或
- MPI\_MAXLOC: 最大值及其位置
- MPI\_MINLOC: 最小值及其位置
- 自定义运算



# 数据的打包与解包

一些通信库为发送不连续数据提供打包/解包函数，用户在发送前显式地把数据包装到一个连续的缓冲区，在接收之后从连续缓冲区中解包。在多数情况下允许有一个派生数据类型不显式打包和解包，用户指明要发送的和接收的数据的分布，通信库直接访问一个不连续的缓冲区。

- MPI\_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm): 打包
- MPI\_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm): 解包
- MPI\_PACK\_SIZE( incount, datatype, comm, size ): 获取打包数据占用空间的上界

一个打包单元可用MPI\_PACKED类型发送。任何点到点或收集式通信功能可被用来从一个进程移动构成打包单元的字节序列到另一个进程中。该打包单元这时可用任何接收操作使用任意数据类型来接收：类型匹配的规定对于以MPI\_PACKED类型发送的消息不再起作用。

以任何类型发送的消息（包括MPI\_PACKED类型）都可用MPI\_PACKED类型接收后被调用MPI\_UNPACK来解包。



# 打包解包例子

将浮点数数组和字符数组打包发送与接收后解包

```
USE MPI
INTEGER MYID,IERR,MYSIZE,STATUS(400),POSTION
REAL BUF1(10)
CHARACTER*20 BUF2
CHARACTER PSEND,PRECV

CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
IF (MYID==0) THEN
    BUF1=1.0
    BUF2="ABCDEFGHIJKLMNPQRST"
    POSTION=0
    CALL MPI_PACK(BUF1, 10, MPI_REAL, PSEND, 100, POSTION, MPI_COMM_WORLD, IERR)
    CALL MPI_PACK(BUF2, 20, MPI_CHARACTER, PSEND, 100, POSTION, MPI_COMM_WORLD, IERR)
    CALL MPI_SEND(PSEND, POSTION, MPI_PACKED, 1, 1, MPI_COMM_WORLD, IERR)
ELSE IF (MYID==1) THEN
    CALL MPI_RECV(PRECV, 100, MPI_PACKED, 0, 1, MPI_COMM_WORLD, STATUS, IERR)
    CALL MPI_UNPACK(PRECV, 100, POSTION, BUF1, 10, MPI_REAL, MPI_COMM_WORLD, IERR)
    CALL MPI_UNPACK(PRECV, 100, POSTION, BUF2, 20, MPI_CHARACTER, MPI_COMM_WORLD, IERR)
    PRINT*, 'BUF1:',BUF1
    PRINT*, 'BUF2:',BUF2
END IF
CALL MPI_FINALIZE(IERR)
END
```



# MPI高级功能

- 组, 上下文及通信子
- 进程拓扑
- MPI环境管理
- 派生数据类型
- 动态进程管理
- 远程存储访问
- 并行I/O



# MPI组，上下文及通信子

- 通信子：将所有这些观点都封装起来以便为MPI中的所有操作提供适当的机会。通信子可分为两种：组内通信子用于一组进程内的操作；组间通信子用于两组进程间的点对点通信。
- 缓冲区：通信子提供了一个缓冲机制允许人们将与MPI固有特征等价的新的属性联系到通信子上。高级用户可利用这一机制进一步修饰通信子并通过MPI实现一些通信子函数。
- 组：组定义了一个进程的有序集合，每一进程具有一个序列号，而且为组间进程通信定义低级名字也是由组完成的。这样组在点对点通信中为进程名字定义了一个范围。另外组还定义了集合操作的范围。在MPI中可从通信子中对组进行分别维护，但是只有通信子才能用于通信操作。
- 上下文：为MPI提供了拥有分离安全的消息传送空间的能力。



# MPI进程拓扑

拓扑是加在内部通信子上的额外、可选的属性，它不能被加在组间通信子上。对于一组进程（通信子内部），拓扑能够提供一种方便的命名机制，另外，可以辅助运行时间系统，将进程映射到硬件上。

MPI中的进程组是n个进程的集合，组中的每一进程被赋予一个从0到n-1的标识数。在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型（通常由基本问题几何和所用的数字算法所决定），进程经常被排列成二维或三维网格形式的拓扑模型，而且，通常用一个图来描述逻辑进程排列，我们指这种逻辑进程排列为“虚拟拓扑”。

在虚拟进程拓扑和底层的物理硬件拓扑之间有一个清晰的差别。进程分配到物理处理器上时，虚拟拓扑可以由系统开发，假设这会帮助提高所给机器的通信性能，然而，这种映射是如何来完成的，已超出了MPI的范围。另外，虚拟拓扑的描述仅依赖于应用，并且独立于机器。



# MPI环境管理

用于获取和在合适的地方设置相关于MPI实现及执行环境（例如错误处理）的不同参数的例程。这里描述了用于进入和离开MPI执行环境的过程。



# MPI派生数据类型

到此为止，所有的点对点通信只牵涉含有相同数据类型的相邻缓冲区，这对两种用户限制太大。一种是经常想传送含有不同数据类型值的消息的用户；另一种是经常发送非连续数据的用户。一种解决的办法是在发送端把非连续的数据打包到一个连续的缓冲区，在接收端再解包。这样做的缺点在于在两端都需要额外的内存到内存拷贝操作，甚至当通信子系统具有收集分散数据功能的时候也是如此。而MPI提供说明更通用的，混合的非连续通信缓冲区的机制。直到执行（implementation）时再决定数据应该在发送之前打包到连续缓冲中，还是直接从数据存储区收集。

这里提供的通用机制允许不需拷贝，而是直接传送各种形式和大小的目标。我们并没有假设MPI库是用本地语言描述的连续目标。因此，如用户想要传送一个结构或一个数组部分，则需要向MPI提供一个通信缓冲区的定义，该定义用问题模仿那个结构和数组部分的定义。这些工具可以用于使库设计者定义能够传送给用本地语言定义的目标的通信函数：通过对可获得的符号表或虚拟向量（dope vector）的定义解码即可。这种高级通信功能不是MPI的部分。



# MPI动态进程管理

- 组间通信域
- 动态创建新的MPI进程
- 独立进程间的通信
- 基于socket的通信



# MPI远程存储访问

- 远程存储访问即直接对非本地的存储空间进行访问
- 一个进程对另外一个进程的存储区域进行直接访问



# MPI并行I/O

- 显式偏移的并行文件读写
- 多视口的并行文件并行读写
- 共享文件读写
- 分布式数组文件的存取



# MPI程序编译

各种不同MPI实现的编译命令不一样，常见的：

- Fortran77: mpif77、mpifort、mpiifort
- Fortran90+: mpif90、mpifort、mpiifort
- C: mpicc、mpiicc
- C++: mpicxx、mpic++、mpiCC、mpiicpc

如：

- 编译C程序：

`mpicc -o prog-mpi prog-mpi.c`

- 编译C++程序，MPICH1、2：

`mpiCC -o prog-mpi prog-mpi.cpp`

- 编译C++程序，MPICH2：

`mpicxx -o prog-mpi prog-mpi.cpp`

- 编译F77程序：

`mpif77 -o prog-mpi prog-mpi.f`

- 编译F90程序：

`mpif90 -o prog-mpi prog-mpi.f90`



# MPI程序运行

运行命令也与具体的MPI实现有关，常见的为mpirun及mpiexec，如加载n个进程运行：

- mpiexec -np n prog-mpi
- mpirun -np n prog-mpi



# 影响MPI程序效率的主要因素

- 并行程序由于进程/线程之间有时候需要相互依赖，某个进程/线程需要等另外的进程/线程完成才可以进行下一步计算，那么会产生等待，导致效率无法100%。
- MPI利用通信进行数据交换，影响MPI程序的主要因素为：
  - 网络延迟
  - 网络带宽
- 解决措施：
  - 根据不同的网络情况决定发送接收的频率及通信包的大小等，比如降低通讯频率，将几次发送接收合并为一次等
  - 改进并行模型及算法，有时候比从所用函数等代码方面单纯优化程序对效率影响更大



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

- OpenMP简介
- OpenMP语法
- OpenMP内在控制变量（ICV）
- OpenMP指令
- OpenMP数据共享环境
- OpenMP运行库子程序
- OpenMP与MPI混合并行编程
- OpenMP程序在Linux/Unix下的编译与运行

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

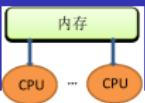
⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息



# 什么是OpenMP



- OpenMP是通过在源代码中添加一些OpenMP编译指令、调用OpenMP库函数来实现在**共享内存**的系统上并行运行的一种标准
- OpenMP提供给共享内存并行程序员一种简单灵活的用于开发并行应用的接口模型，使程序既可以运行在台式机，又可以运行在超级计算机上，并具有良好可移植性和可缩放性等：
  - 未打开OpenMP编译选项编译时将忽略OpenMP编译指令而直接编译成串行可执行程序
  - 打开OpenMP编译选项编译时将对OpenMP编译指令进行处理编译成OpenMP并行可执行程序
  - 运行时的并行线程数可在程序启动时利用环境变量等动态设置
  - 支持与MPI的混合编程，在节点内部运行OpenMP并行，节点之间运行MPI并行
- OpenMP支持C/C++和Fortran语言
- OpenMP程序可以运行在Linux、Unix和Windows等操作系统上



# OpenMP参与的主要公司、机构和人员

AMD Greg Rodgers	Argonne National Laboratory Kalyan Kumaran	ARM Graham Hunter	ASCI/Lawrence Livermore National Laboratory Bronis R. de Supinski	Barcelona Supercomputing Center Xavier Martorell	NEC Shin-ichi Okano	NVIDIA Jeff Larkin	Oak Ridge National Laboratory Oscar Hernandez	Red Hat Torvald Riegel	WTH Aachen University Dierer an Mey
Bristol University Simon McIntosh-Smith	Brookhaven National Laboratory Vivien Kalu	cOMFunTy Yonghong Yan	CRAY Deepak Echampati a Hewlett-Packard Enterprise company	epcc Edinburgh Parallel Computing Centre Mark Bull	Sandia National Laboratory Stephen Oliver	Stony Brook University Dr. Barbara Chapman	SLUSE Michael Mitz	TACC Kent Miller	Texas Instruments Gaurav Mitra
Fujitsu Naoki Sugiyama	IBM Kelvin Li	INRIA Oliver Beaumont	Intel Xianmin Tian	Lawrence Berkeley National Laboratory Helen He	The University of Manchester Antoniu Pop	University of Delaware Sundar Chandrasekaran	University of Tennessee Peter Luszczek		
Leibniz Supercomputing Centre Volker Weisberg	Los Alamos National Laboratory Jamel Mofid-Yusef	Masai High Performance Computing Center Alice Konges	Micron Randy Meyer	NASA Henry Jin					

官方站点: <http://www.openmp.org>



# OpenMP发布历史

- Version 5.0 Complete Specifications (November 2018)
- Version 4.5 Complete Specifications (November 2015)
- Version 4.0 Complete Specifications (July 2013)
- Version 3.1 Specification Released (July 2011)
- Version 3.0 Complete Specifications (May 2008)
- Version 2.5 (May 2005, combined C/C++ and Fortran)
- C/C++ version 2.0 (March 2002)
- C/C++ version 2.0 with change bars reflecting changes from 1.0 (March 2002)
- Fortran version 2.0 (November 2000)
- Fortran version 2.0 with change bars reflecting changes from 1.1 (November 2000)
- C/C++ version 1.0 (October 1998)
- Fortran version 1.1 (November 1999 - incorporates April 1999 Interpretations and Errata)
- Fortran version 1.0 (October 1997)



# 支持OpenMP的编译器以及对应编译参数

公司或组织	编译器	对应的编译参数
GNU	gcc( $\geq 4.3.2$ )	-fopenmp
Intel	C/C++/Fortran	Linux: -qopenmp, Win: /Qopenmp
Portland Group	C/C++/Fortran	-mp
IBM	XL C/C++/Fortran	-qsmp=omp
HP	C/C++/Fortran	+Oopenmp
Microsoft	Visual Studio 2008 C++	/openmp
Oracle Corporation/Sun Microsystems	C/C++/Fortran	-xopenmp
Absoft Pro FortranMP	Fortran	-openmp
Lahey/Fujitsu Fortran95	C/C++/Fortran	-openmp -threadstack -threadheap
PathScale	C/C++/Fortran	-apo -mp



# 一维矢量点乘的串行代码

两个一维矢量的点乘:  $\text{sum} = \mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{n-1} A[i] * B[i]$

```
int main(argc,argv)
int argc;
char *argv[];
{
    double sum;
    double a[256], b[256];
    int i,n;
    n = 256;
    for (i = 0; i < n; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;
    for (i = 0; i < n; i++) {
        sum = sum + a[i] * b[i];
    }
    printf ("sum=%f\n", sum);
}
```



# 一维矢量点乘的MPI并行代码

两个一维矢量的点乘:  $\text{sum} = \mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{n-1} A[i] * B[i]$

```
#include "mpi.h"
int main(argc,argv)
int argc; char *argv[];
{
    double sum, sum_local;
    double a[256], b[256];
    int i, n, numprocs, myid, my_first, my_last;
    n = 256;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    my_first = myid * n / numprocs; /*利用进程号及进程数使得每个进程的my_first与my_last不一样*/
    my_last = (myid + 1) * n / numprocs;
    for (i = 0; i < n; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum_local = 0;
    for (i = my_first; i < my_last; i++) { /*每个进程的my_first与my_last不一样, */
        sum_local = sum_local + a[i] * b[i]; /*每个进程计算for循环的不同部分实现并行*/
    }
    MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    if (myid == 0) printf ("sum=%f\n", sum);
    MPI_Finalize();
}
```



# 一维矢量点乘的OpenMP并行代码

两个一维矢量的点乘:  $\text{sum} = \mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{n-1} A[i] * B[i]$

```
int main(argc,argv)
int argc;
char *argv[];
{
    double sum;
    double a[256], b[256];
    int status;
    int i,n=256;
    for (i = 0; i < n; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;
#pragma omp parallel for reduction(+:sum) /*仅需该行代码即完成OpenMP并行*/
    for (i = 1; i <= n; i++) {
        sum = sum + a[i]*b[i];
    }
    printf ("sum=%f\n", sum);
}
```



# OpenMP组成

OpenMP是以下的集合：

- 编译指令
- 库函数
- 环境变量



# OpenMP格式: C/C++

C/C++中利用**pragma**预处理指令做为OpenMP的指令，语法格式如下：

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

- 每个指令以**#pragma omp**开始
- 指令其余部分按照C/C++编译指令标准，并区分大小写
- **#**之前和之后可有空白，且有时必须用空白来分隔指令中的文字
- **#pragma omp**之后的预处理目标在编译时将被宏替换
- 每个OpenMP执行指令必须应用于至少一个随后的语句，并且必须是一个结构块
- 每行只能有一个OpenMP指令
- 参数的位置无前后之分，可重复

**注:**[ ]表示内部的参数是可选的



# OpenMP格式: Fortran

Fortran下的OpenMP指令语法格式如下:

```
sentinel directive—name [clause[, clause]...]
```

OpenMP指令需要满足以下条件:

- 所有OpenMP编译指令必须以标志符开始
- 固定格式和自由格式的OpenMP标识符不同
- OpenMP指令不区分大小写
- OpenMP指令不能内嵌在连续语句中，语句也不能内嵌在指令中
- 每行只能有一个OpenMP指令
- 参数的位置无前后之分，可重复

**注:**对于Fortran，今后除非特殊说明，都以自由格式表述



# OpenMP格式: Fortran固定格式

固定格式Fortran的OpenMP指令标识符为**!\$omp**、**c\$omp**或**\*\$omp**，需满足：

- 标识符必需在第一列开始，并且做为一个单词，之间不得有其它字符
- 固定格式**Fortran**的行宽、空白、续行和列规则对指令行一样需遵守
- 初始指令行必须有一个空格或0在第六列，指令续行时必须有一个非空格或0的字符在第六列
- 注释可以与指令同一行，在第六列之后的**!**表明注释开始
- 如指令标志符之后跟的第一个字符是非空格字符，或者指令续行是一个**!**，那么此行被忽略

基本格式：

```
!$omp directive-name [ clause [[ ,] clause]. . . ] new-line
c$omp directive-name [ clause [[ ,] clause]. . . ] new-line
*$omp directive-name [ clause [[ ,] clause]. . . ] new-line
```

以下三种表示方式等价：

c23456789表示列号

**!\$omp parallel do shared(a,b,c)** !方式一

**c\$omp parallel do** !方式二

**c\$omp+shared(a,b,c)** !方式二

**\*\$omp parallel do shared(a,b,c)** !方式三



# OpenMP格式：Fortran自由格式

自由格式Fortran的OpenMP指令标识符为`!$omp`，需满足：

- 标识符可在任何位置开始，但之前的字符必须只能为空白而不能为其它字符
- 必须做为一个单独的单词，之间不能有其它字符
- 自由格式Fortran的行宽、空白、续行和列规则对指令行一样需遵守
- 初始指令行必须在标识符后有一个空格
- 需要续行的行的最后，必须为`&`，其续行可以在标识符之后有一个`&`，且`&`前后可有空白
- 注释可以与指令同一行，`!`表明注释开始
- 如指令标志符之后跟的第一个字符是非空字符，或指令续行仅是一个`!`，那么此行被忽略
- 一个或更多的空格或制表符用于分割邻近指令中的关键词，以下情形，空格是可选的：  
`end critical`、`end do`、`end master`、`end ordered`、`end parallel`、`end sections`、`end single`、`end task`、  
`end workshare`、`parallel do`、`parallel sections`、`parallel workshare`

基本格式：

```
!$omp directive-name [clause[, clause]. . . ] new-line
```

`!23456789`表示列号

```
!$omp parallel do &
 !$omp shared(a,b,c)
```

```
!$omp parallel &
 !$omp do shared(a,b,c)
```

```
!$omp parallel do shared(a,b,c)
```



# 条件编译

- 对于支持预处理的OpenMP实现，宏`_OPENMP`的值被定义为`yyyymm`格式，其中`yyyy`和`mm`分别为此编译器使用的OpenMP标准发布的4位年和2位月
- 尽量不要自己定义`_OPENMP`：如另有`#define`和`#undef`声明，那么编译行为将无法预料



# 条件编译：Fortran固定格式

标志符可为!\$、\*\$或c\$之一，并需要满足以下条件：

- 标记符必需从第一列开始，并且是作为一个整体
- 标记符被替换为两个空格之后，初始行必须在第六列有一个空格或者0，并且在第一到第五列之间只能为空白或数字
- 标记符被替换为两个空格之后，续行必须在第六列有一个非空白或者0的字符，并且在第一到第五列之间只能为空白

如上述满足，那么编译时此行的标识符将被替换为两个空格，否则将不做处理



# 条件编译：Fortran固定格式

!23456789表示列号

```
!$      interval=l*omp_get_thread_num() / &
!$          (omp_get_num_threads()-1)
```

!下行的 !\$ 由于的前面有非空格字符而不表示OpenMP条件编译，而是注释

```
Do i=1,100, !$ OMP_get_num_threads()
```

以下两种方式等价：

!23456789表示列号

```
!$ 10 iam = omp_get_thread_num() +
!$   &           index
```

```
#ifdef _OPENMP
 10 iam = omp_get_thread_num() +
   &           index
#endif
```



# 条件编译：Fortran自由格式

标志符为 `!$`，并需满足以下条件：

- 以 `!$` 而不以 `!$omp` 为开始标记，且前面没有空格以外的其它字符
- 标记符是作为一个之间没有空白的整体
- 初始行必须在标识符之后有一个空格
- 需要续行的行的最后必须为 `&`，续行可在标识符之后有一个 `&`，`&` 前后可有空白

如上述满足，则编译时此行的标识符将被替换为两个空格，否则将不做处理



# 条件编译：Fortran自由格式

```
!$ interval=l*omp_get_thread_num() / &
!$ (omp_get_num_threads()-1)
!下行的 !$ 由于的前面有非空格字符而不表示OpenMP条件编译，而是注释
Do i=1,100, !$ OMP_get_num_threads()
```

以下两种方式等价：

```
c23456789
 !$ iam = omp_get_thread_num() + &
 !$&    index

#endif _OPENMP
 iam = omp_get_thread_num() + &
       index
#endif
```



# 内在控制变量 (ICV)

内置的用于控制OpenMP程序行为的变量，如存储线程数、线程号等

- 影响并行块的内在控制变量：

- dyn-var: 控制影响的并行区域是否允许动态调整线程数，每任务一个ICV副本
- nest-var: 控制影响的并行区域是否允许嵌套并行，每任务一个ICV副本
- nthreads-var: 控制影响的并行区域的线程数，每任务一个ICV副本
- thread-limit-var: 控制程序最大允许参与的线程数，整个任务共用一个ICV副本
- max-active-levels-var: 控制嵌套的激活的并行区域的最大数，整个任务共用一个ICV副本

- 影响循环区域操作的内在控制变量：

- run-sched-var: 控制循环区域的实时调度策略，每任务一个ICV副本
- def-sched-var: 控制循环区域的默认实时调度策略，整个任务共用一个ICV副本

- 影响程序执行的内在控制变量：

- stacksize-var: 控制OpenMP实现产生的线程的堆栈 (stack) 的大小，整个任务共用一个ICV副本
- wait-policy-var: 控制等待线程的期望行为，整个任务共用一个ICV副本



# 更改和获取内在控制变量的值

内在控制变量	作用范围	变量值的设置方式	获取值	初始值
dyn-var	单个任务	OMP_DYNAMIC omp_set_dynamic()	omp_get_dynamic()	参看注解
nest-var	单个任务	OMP_NESTED omp_set_nested()	omp_get_nested()	false
nthreads-var	单个任务	OMP_NUM_THREADS omp_set_num_threads()	omp_get_max_threads()	执行时定义
run-sched-var	单个任务	OMP_SCHEDULE omp_set_schedule()	omp_get_schedule()	执行时定义
def-sched-var	全局	(none)	(none)	执行时定义
stacksize-var	全局	OMP_STACKSIZE	(none)	执行时定义
wait-policy-var	全局	OMP_WAIT_POLICY	(none)	执行时定义
thread-limit-var	全局	OMP_THREAD_LIMIT	omp_get_thread_limit()	执行时定义
max-active-levels-var	全局	OMP_MAX_ACTIVE_LEVELS omp_set_max_active_levels()	omp_get_max_active_levels()	参看注解

注解：

- 如OpenMP的执行支持动态调整线程，dyn-var的初始值由执行时定义，否则为false
- max-active-levels-var的初始值是执行时支持的并行级别数目

程序开始运行时，变量初始值将被赋予，并在OpenMP结构或API子程序中执行，用户设置的OpenMP环境变量的值被读取并赋予对应的内部控制变量，之后任何OpenMP环境变量的改变都不再影响内部控制变量。OpenMP结构的参数不会影响任何内部控制变量。



# 各任务的内在控制变量的工作方式

- 各任务区域具有初始变量dyn-var、nest-var、nthreads-var和run-sched-var自己的副本
- 当一个任务结构或并行结构在一个任务中执行时，其产生的子任务继承这些变量
- 调用omp\_set\_num\_threads()、omp\_set\_dynamic()、omp\_set\_nested()和omp\_set\_schedule()仅会更新与其相关的任务区域的内在控制变量
- 当执行到指定schedule(runtime)的循环工作共享区域时，所有组成绑定parallel区域的任务区域的run-sched-var必须有同样的值，否则行为将无法预料



# 控制并行的因素的优先级

结构参数	API子程序	环境变量	内在控制变量
(none)	omp_set_dynamic()	OMP_DYNAMIC	dyn-var
(none)	omp_set_nested()	OMP_NESTED	nest-var
num_threads	omp_set_num_threads()	OMP_NUM_THREADS	nthreads-var
schedule	omp_set_schedule()	OMP_SCHEDULE	run-sched-var
schedule	(none)	(none)	def-sched-var
(none)	(none)	OMP_STACKSIZE	stacksize-var
(none)	(none)	OMP_WAIT_POLICY	wait-policy-var
(none)	(none)	OMP_THREAD_LIMIT	thread-limit-var
(none)	omp_set_max_active_levels()	OMP_MAX_ACTIVE_LEVELS	max-active-levels-var

优先级：左边高于右边



# 并行结构: parallel Construct

指明随后区域中的代码将并行执行，并且并行是可以嵌套的

- C/C++:

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
structured-block
```

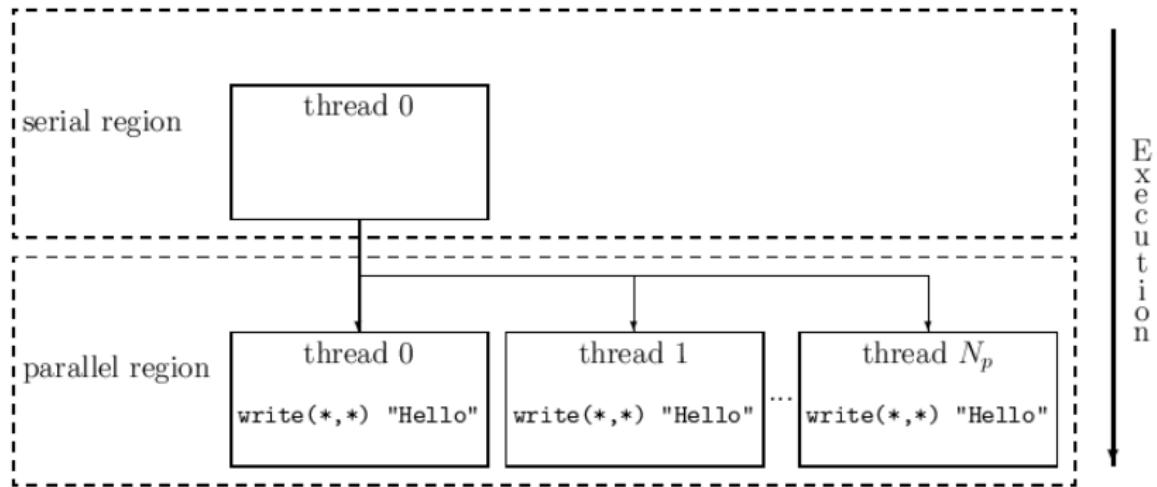
- Fortran:

```
!$omp parallel [clause[[,] clause]...]
structured-block
!$omp end parallel
```



# 并行结构图示

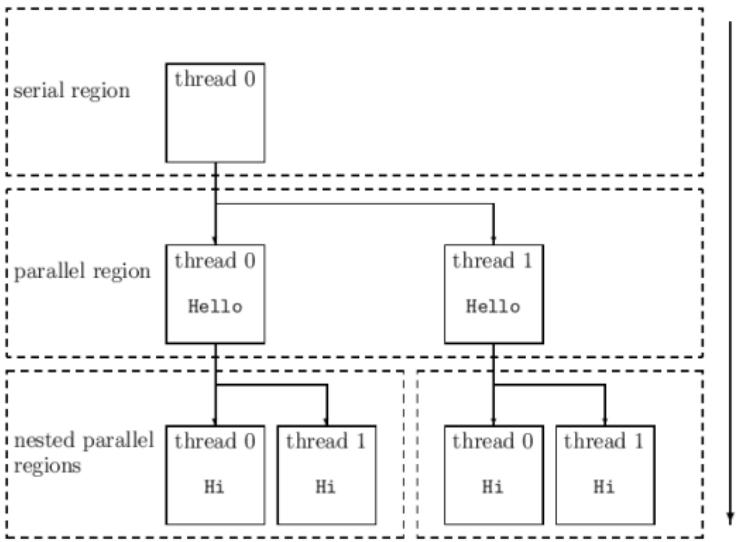
```
!$omp parallel  
    write(*,*) "Hello"  
!$omp end parallel
```





# 并行嵌套图示

```
!$omp parallel
    write(*,*) "Hello"
    !$omp parallel
        write(*,*) 'Hi'
    !$omp end parallel
 !$omp end parallel
```





# 工作共享构造：Worksharing Constructs

- 工作共享构造会在执行到此结构的线程中分配与之相关区域的到各线程执行
- 线程会执行在每个正在执行任务的情况下隐含的部分区域
- 在每个工作共享构造的入口处无需进行同步，但在工作共享区域的结束处，除非有nowait参数，否则默认会进行同步
- 如有nowait，那么执行到此处时将会忽略此工作共享区域的同步，早执行完此结构的线程会直接执行下面的指令而不需等待其它进程执行到此，也不需要进行刷新（flush）操作



OpenMP定义了以下工作共享构造:

- 循环结构: loop structure
- 分块结构: sections structure
- 单执行结构: single structure
- 工作共享结构: workshare structure

限制:

- 工作共享区域必须为所有组内线程都执行到或都不执行到
- 一组内每个线程执行到工作共享区域和同步区域的顺序必须一致



# 循环结构：loop Constructs

指明一个或多个与之相关的循环迭代将被一组线程执行

- C/C++:

```
#pragma omp for [clause[ [, clause] ... ] new-line
    for-loops
```

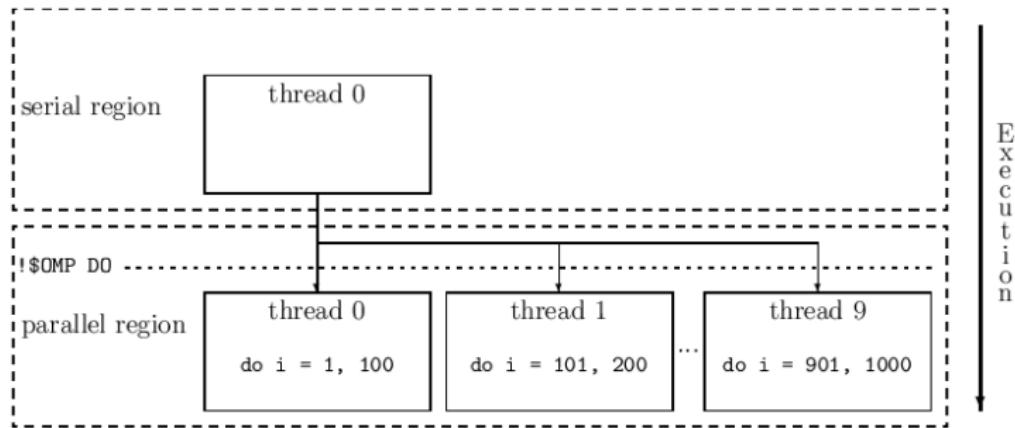
- Fortran:

```
!$omp do [clause[ [, clause] ... ]
    do-loops
[ !$omp end do [nowait] ]
```



# 循环结构图示

```
!$omp do,clause ...
do i = 1, 1000
...
enddo
 !$omp end do
```





# 分块结构： sections Construct

指明不同的线程执行不同区域内的代码

- C/C++:

```
#pragma omp sections [clause[, clause] ...] new-line
{
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block ]
    ...
}
```

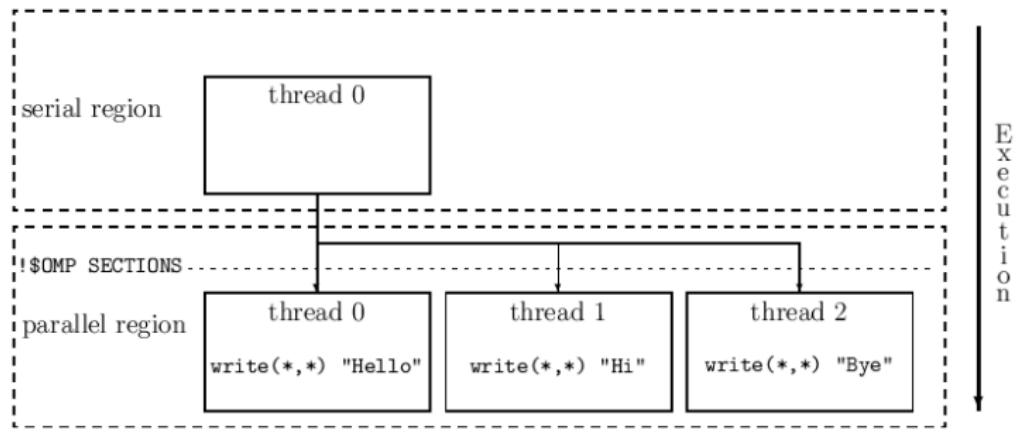
- Fortran:

```
!$omp sections [clause[, clause] ...]
[ !$omp section
    structured-block
 !$omp section
    structured-block ]
...
 !$omp end sections [nowait]
```



# 分块结构图示

```
!$omp sections
  !$omp section
    write(*,*) "hello"
  !$omp section
    write(*,*) "hi"
  !$omp section
    write(*,*) "bye"
 !$omp end sections
```





# 单执行结构: single Structure

指明相关代码只能有一个线程（并不必须是雇主主线程，0号线程）执行，一般为先到的线程执行

- C/C++:

```
#pragma omp single [clause[ [, clause] ...] new-line
structured-block
```

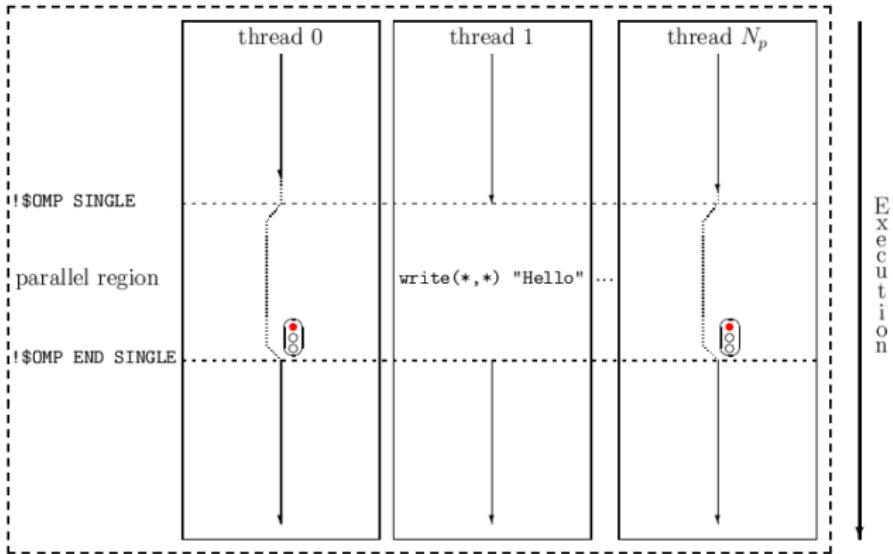
- Fortran:

```
!$omp single [clause[ [, clause] ...]
structured-block
!$omp end single [end_clause[ [, end_clause] ...]
```



# 单执行结构图示

```
!$omp single clause1 clause2 ...
...
!$omp end single end_clause
```





# 工作共享结构: workshare Construct

指明相关的代码被分割成不同的单元以供不同线程执行，不同单元只能被某个线程执行一次

- C/C++: 不支持此OpenMP指令
- Fortran:

```
!$omp workshare  
structured-block  
!$omp end workshare [nowait]
```

structured-block需要满足以下条件之一:

- 数组赋值
- 标量赋值
- forall语句
- forall结构
- where语句
- where结构
- atomic结构
- critical结构
- parallel结构

限制:

- critical结构中封入的语句受此限制
- parallel结构中封入的语句不受此限制



# 联合并行工作共享结构

- 实际上是在并行结构中嵌套共享结构的一个快捷方式，这些指令将同时指明一个工作共享结构具有并行性，而不需其它语句另行指明
- 允许一些并行结构和工作共享结构都允许的特定参数
- 如程序含有对并行结构或工作共享结构具有不同行为的参数，那么其行为将不可预测
- 包含以下结构：
  - 并行循环结构：parallel loop constructs
  - 并行分块结构：parallel sections construct
  - 并行工作共享结构：parallel workshare construct



# 并行循环结构: parallel loop Constructs

指明一个或多个相关的循环迭代将被一组线程并行执行

- C/C++:

```
#pragma omp parallel for [clause[, clause] ... ] new-line
for-loops
```

- Fortran:

```
!$omp parallel do [clause[, clause] ... ]
do-loops
[ !$omp parallel end do [nowait] ]
```



# 并行分块结构: parallel sections Construct

指明与之相关的不同区域的代码将被不同线程并行执行

- C/C++:

```
#pragma omp parallel sections [clause[, clause] ...] new-line
{
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block ]
    ...
}
```

- Fortran:

```
!$omp parallel sections [clause[, clause] ...]
[ !$omp section
    structured-block
[ !$omp section
    structured-block ]
...
 !$omp end parallel sections [nowait]
```



# 并行工作共享结构: parallel workshare Construct

指明相关的代码被分割成不同的单元以供不同线程并行执行，不同单元只能被某一线程执行一次

- C/C++: 不支持此OpenMP指令
- Fortran:

```
!$omp parallel workshare  
structured-block  
!$omp parallel end workshare [nowait]
```

structured-block需要满足以下条件:

- 数组赋值
- 标量赋值
- forall语句
- forall结构
- where语句
- where结构
- atomic结构
- critical结构
- parallel结构

限制:

- critical结构中封入的语句受此限制
- parallel结构中封入的语句不受此限制



# 任务结构: task Construct

定义一个明确的任务

- C/C++:

```
#pragma omp task [clause[, clause] ...] new-line
structured-block
```

- Fortran:

```
!$omp task [clause[, clause] ...]
structured-block
!$omp end task
```



# 雇主结构和同步结构

雇主结构和同步结构主要包含以下结构

- 雇主结构: master construct
- 临界结构: critical construct
- 栅栏结构: barrier construct
- 任务等待结构: taskwait construct
- 原子结构: atomic construct
- 刷新结构: flush construct
- 有序结构: ordered construct



# 雇主结构: master Construct

指明结构块需要雇主线程（主线程，0号线程）执行

- C/C++:

```
#pragma omp master new-line
structured-block
```

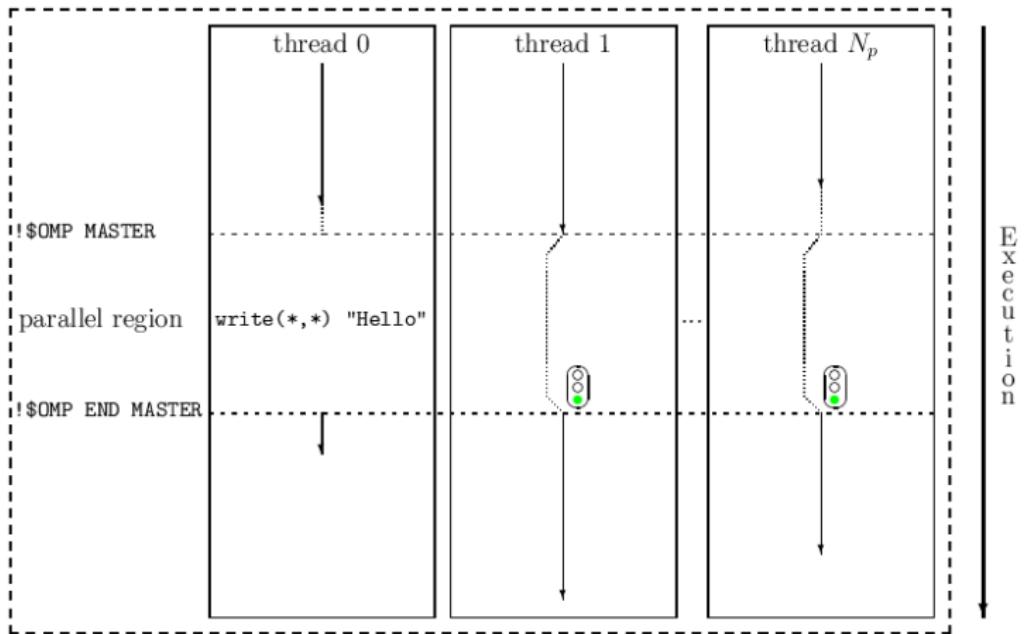
- Fortran:

```
!$omp master
structured-block
!$omp end master
```



# 雇主结构图示

```
!$omp master  
    write(*,*) "Hello"  
 !$omp end master
```





# 临界结构: critical Construct

指明结构块在同一时间只能有一个线程执行，一个线程执行完毕，其它线程才能执行该处代码

- C/C++:

```
#pragma omp critical [(name)] new-line
structured-block
```

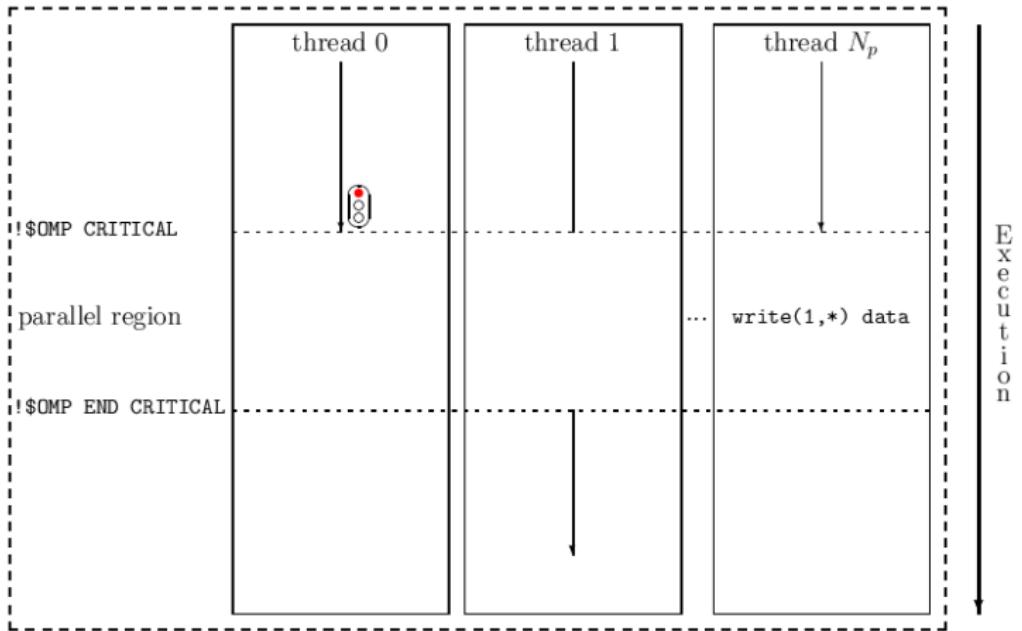
- Fortran:

```
!$omp critical [(name)]
structured-block
!$omp end critical [(name)]
```



# 临界结构图示

```
!$omp critical write_file  
    write(1,*) data  
 !$omp end critical write_file
```





# 栅栏结构: barrier Construct

指明在此处有一个栅栏，需进行显式同步，即需本组内所有线程都要运行到此后才执行后续代码

- C/C++:

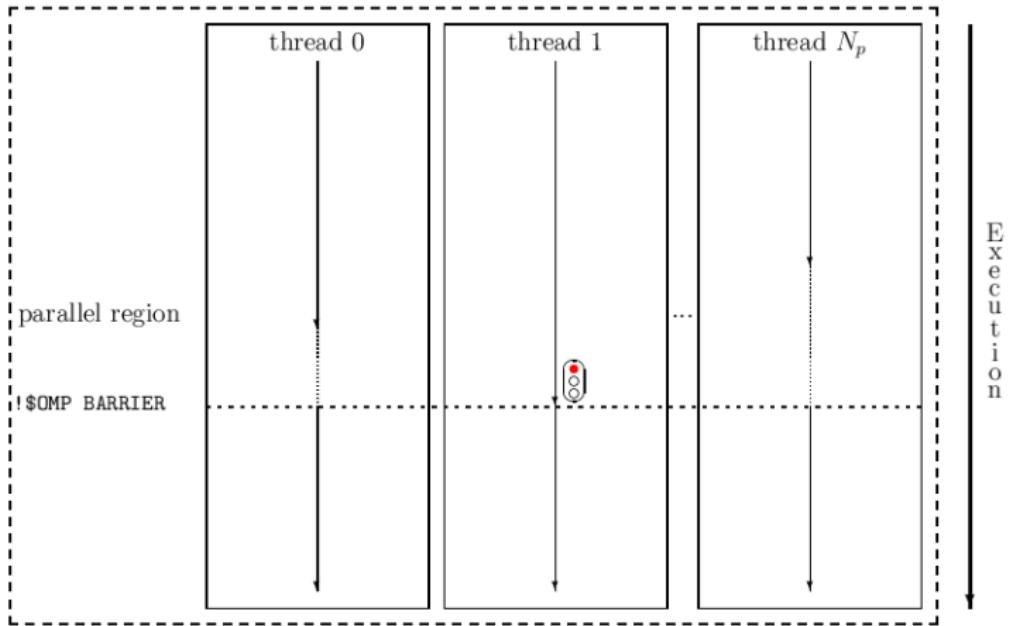
```
#pragma omp barrier new-line
```

- Fortran:

```
!$omp barrier
```



# 栅栏结构图示





# 栅栏结构常见错误举例

注意：栅栏结构必须在所有线程的同样顺序中

常见错误举例：

```
...  
!$omp critical
```

```
...  
!$omp barrier
```

```
...  
!$omp end critical  
...
```

```
...  
!$omp single  
...  
!$omp barrier  
...  
!$omp end single  
...
```

```
...  
!$omp master
```

```
...  
!$omp barrier
```

```
...  
!$omp end master  
...
```

```
...  
!$omp sections
```

```
...  
!$omp section
```

```
...  
!$omp barrier
```

```
...  
!$omp end section  
...  
!$omp end sections  
...
```



# 任务等待结构: taskwait Construct

指明需要自当前任务开始就需等待产生的子任务完成

- C/C++:

```
#pragma omp taskwait newline
```

- Fortran:

```
!$omp taskwait
```



# 原子结构: atomic Construct

用于确保指明的存储区域对某项操作进行原子更新，而不使其暴露给多个同步进行写的线程，避免同时多个线程对这些变量进行更新

- C/C++:

```
#pragma omp atomic new-line  
expression-stmt
```

- Fortran:

```
!$omp atomic  
statement
```

举例:

```
!$omp do  
do i = 1, 1000  
 !$omp atomic +  
   a = a + i  
 enddo  
 !$omp end do
```



# 刷新结构: flush Construct

指明对指定的变量执行内存刷新操作以保证线程此时看到这些变量在内存中的值一致

- C/C++:

```
#pragma omp flush [(list)] new-line
```

- Fortran:

```
!$omp flush [(list)]
```



# 有序结构: ordered Construct

指明循环结构中结构块的执行按照循环的迭代顺序，将排序此结构块中的执行顺序，并允许结构块之外的代码可以并行执行

- C/C++:

```
#pragma omp ordered new-line
structured-block
```

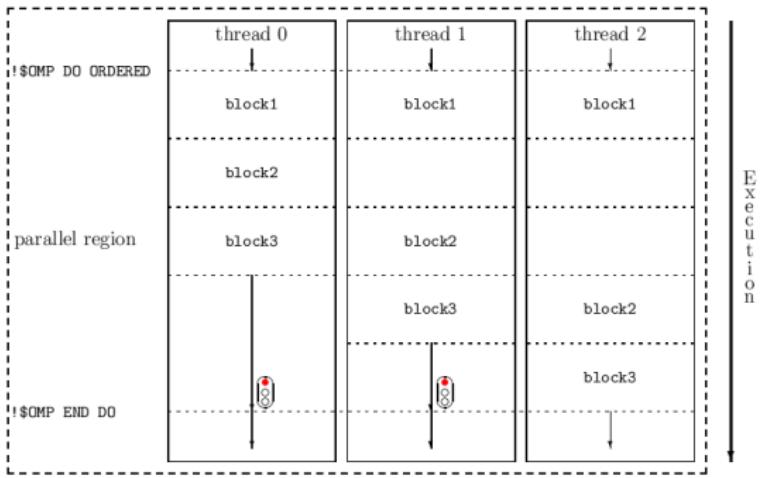
- Fortran:

```
!$omp ordered
structured-block
!$omp end ordered
```



# 有序结构图示

```
!$omp do ordered
do i = 1, 100
    block1
    !$omp ordered
        block2
    !$omp end ordered
    block3
enddo
 !$omp end do
```





# 结构内引用的变量的数据共享属性法则

结构中引用变量的数据共享属性可为预先决定的、显示决定的或隐式决定的之一。封闭结构中的firstprivate、lastprivate或reduction参数声明的变量会导致此结构中变量采用隐式的引用方式，并遵守以下规则：

- C/C++:

- threadprivate指令指明的变量是线程私有的
- 结构中某个范围内申明的具有自动存储周期的变量是私有的
- 具有堆分配存储的变量是共享的
- 静态数据成员是共享的
- 关联的for循环或 parallel for循环结构中的循环迭代变量是私有的
- 不具有易变成员的保留常数变量是共享的
- 结构中某个范围内声明的静态变量是共享的

- Fortran:

- threadprivate指令指明的变量和common块是线程私有的
- 关联的do循环或 parallel do循环结构中的循环迭代变量是私有的
- parallel 或task结构中的顺序循环的循环迭代变量在封闭此循环的最内部结构中是私有的
- 隐式do和 forall 的索引是私有的
- Cray指针继承其关联存储的数据共享属性



# 数据共享属性法则

除了在以下情形中，具有预定义数据共享属性的变量也许未在数据共享属性参数中列出，对这些例外，在数据共享属性参数中允许列出一些预定义的变量，且将取代这些变量的预先定义数据共享属性。

- C/C++:
  - `for` 循环或 `parallel for` 循环结构中的循环迭代变量可以在 `private` 或 `lastprivate` 参数中被列出
- Fortran:
  - `do` 循环或 `parallel do` 循环结构中的循环迭代变量可以在 `private` 或 `lastprivate` 参数中被列出
  - `parallel` 或 `task` 结构中的顺序循环的循环迭代变量可以在 `private`、`firstprivate`、`lastprivate`、`shared`、或 `reduction` 参数中列出，并且在封闭结构中受其它的限制
  - 假定大小的数组可以在 `share` 参数中被列出



# 数据共享法则

具有显式决定或隐式决定的数据共享属性的变量：

- 具有显式决定的数据共享属性的变量：在一个给定的结构中被引用并且在此结构的数据共享参数中被列出的变量
- 具有隐式决定的数据共享属性的变量：在一个给定的结构中被引用，但没有被预先决定数据共享属性，且没有在此结构的数据共享属性参数中列出的变量



# 隐式数据共享法则

隐式决定的数据共享属性规则如下：

- 在parallel或task结构中，如存在default参数，那么这些变量的数据共享属性由default参数决定
- 在parallel结构中，如没有default参数，那么这些变量是共享的
- 对于不是task的结构来说，如没有default参数，那么这些变量将从此封闭上下文中继承数据共享属性
- 在task结构中，如没有default参数，那么在所有封闭结构并一直到最内部的封闭parallel结构中变量被决定为共享的
- 在task结构中，如没有default参数且数据共享属性没有被上述规则决定，那么变量是firstprivate的



# 区域而非结构中的数据共享属性规则

- C/C++:

- 在被调用子程序中声明的静态变量在此区域中是共享的
- 不具有易变成员且在被调用子程序中声明的保留常数变量是共享的
- 除非是在threadprivate指令中出现，否则区域中在被调用子程序中引用的文件范围或者名字空间范围变量是共享的
- 具有堆分配存储的变量是共享的
- 除非是在threadprivate指令中出现，否则静态数据成员是共享的
- 区域中被调用子程序的通过引用传递的正式参数继承关联的实际参数数据的数据共享属性
- 区域中被调用子程序中的其它变量是私有的

- Fortran:

- 区域中被调用子程序中声明的本地变量如有save属性，或数据是被初始化的，且未在threadprivate指令中出现的变量是共享的
- 区域中被调用子程序中引用的common块或module中的变量是共享的，除非是在threadprivate指令中出现
- 区域中被调用子程序中的哑元参数如通过引用传递，那么将继承与之关联的实际参数的数据共享属性
- Cray指针继承其关联存储的数据共享属性
- 区域中被调用子程序中的隐式do和forall索引及声明的其它本地变量是私有的



# threadprivate指令

threadprivate指令指明变量是线程私有的

- C/C++:

```
#pragma omp threadprivate(list) new-line
```

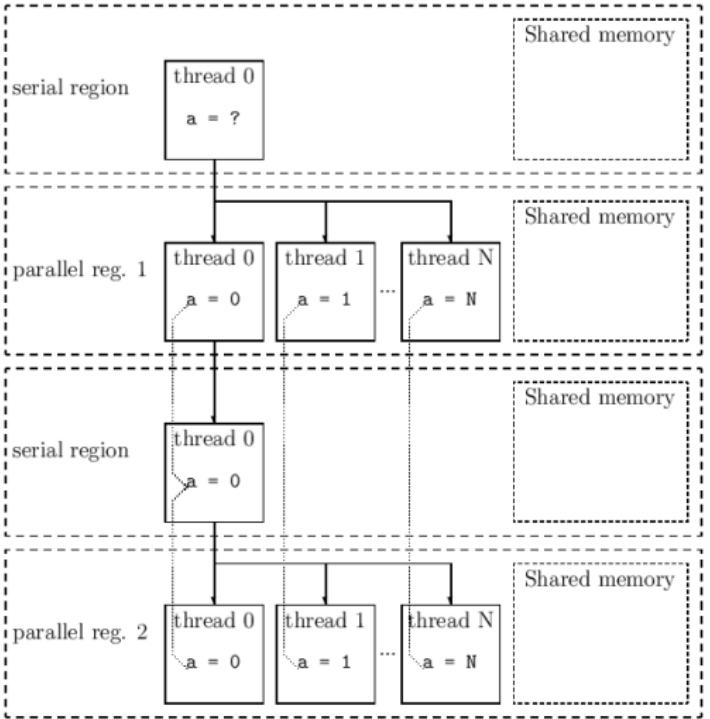
- Fortran:

```
!$omp threadprivate(list)
```



# threadprivate指令图示

```
real(8), save :: A(100), B  
integer, save :: C  
...  
!$omp threadprivate(a, b, c)
```





# 数据共享属性参数

- default: 设置默认的共享方式

- C/C++:

```
default(shared | none)
```

- Fortran:

```
default(private | firstprivate | shared | none)
```

- 其中:

- shared: 线程共享
    - private: 线程私有
    - firstprivate: 线程私有, 其值为开始此结构时从原始变量继承的值
    - lastprivate: 线程私有, 并在此区域结束时更新原始变量的值

- reduction: 声明对变量进行规约操作

- C/C++:

```
reduction(operator:list)
```

- Fortran:

```
reduction({operator | intrinsic_procedure_name}:list)
```



# 数据复制参数

- copyin: 在执行parallel结构时，将主线程的线程私有变量的值复制给所有其它线程

```
copyin(list)
```

- copyprivate: 提供一种机制可以将隐式任务数据环境的某个私有变量的值广播给此parallel结构中的其它数据环境

```
copyprivate(list)
```



# 线程嵌套

线程嵌套的规则如下

- 工作共享区域：可没被嵌套封闭在工作共享、显式task、critical、ordered或master区域
- barrier区域：可没被嵌套封闭在工作共享、显式task、critical、ordered或master区域
- master区域：可没被嵌套封闭在工作共享或显式task区域
- ordered区域：可没被嵌套封闭在critical或显式task区域
- ordered区域：必须被嵌套封闭在具有ordered参数的循环区域或并行循环区域中
- critical区域：可没被嵌套（封闭或其它）在具有同样名字的critical区域中，此限制并不充分能防止一个死锁



# 运行库定义

- C/C++:

头文件omp.h中定义了：

- 所有OpenMP子程序的原型
- `omp_lock_t`类型
- `omp_nest_lock_t`类型
- `omp_sched_t`类型

- Fortran:

F77头文件omp\_lib.h和F90的module文件omp\_lib定义了：

- 所有OpenMP子程序的接口
- `integer parameter omp_lock_kind`
- `integer parameter omp_nest_lock_kind`
- `integer parameter omp_sched_kind`
- `integer parameter openmp_version`



# 执行环境子程序

- `omp_set_num_threads`: 设置线程数
- `omp_get_num_threads`: 获取线程数
- `omp_get_max_threads`: 获取最大进程数
- `omp_get_thread_num`: 获取进程号
- `omp_get_num_procs`: 获取处理器数
- `omp_in_parallel`: 判断是否在parallel区域
- `omp_set_dynamic`: 设置是否允许动态调整线程数
- `omp_get_dynamic`: 获取是否允许动态调整线程数
- `omp_set_nested`: 设置是否允许嵌套
- `omp_get_nested`: 获取是否允许嵌套
- `omp_set_schedule`: 设置调度策略
- `omp_get_schedule`: 获取调度策略
- `omp_get_thread_limit`: 获取程序允许的最大线程数
- `omp_set_max_active_levels`: 设置允许最大嵌套激活的层数
- `omp_get_max_active_levels`: 获取允许最大嵌套激活的层数
- `omp_get_level`: 获取当前嵌套parallel区域层级
- `omp_get_ancestor_thread_num`: 获取父线程或者当前线程号
- `omp_get_team_size`: 获取某嵌套层级的父或当前线程允许的组线程数
- `omp_get_active_level`: 获取嵌套的活跃的parallel区域层级



# 锁子程序

- OpenMP运行库包含一系列专门用途的锁子程序，可以用于同步等。这些锁子程序操作那些用OpenMP锁变量表示的OpenMP锁。OpenMP锁只能被这些子程序存取，其余方式存取OpenMP锁的方式不被确认。
- OpenMP锁可有uninitialized、unlocked或locked状态。如一个锁在unlocked状态，一个任务可设置该锁为locked状态。设置此锁的任务将拥有此锁，拥有此锁的任务可设置此锁为unlocked状态。一个任务复位属于另一个任务的锁是无法保证的。
- OpenMP支持两种类型的锁：简单锁和可嵌套锁。可嵌套锁可被同一个任务在被复位之前多次设置；简单锁在属于的任务尝试再次设置的时候将不被设置。简单锁变量与简单锁相关联，且只能被传递给简单锁子程序。可嵌套锁变量与可嵌套锁相关联，并且只能传递给可嵌套锁子程序。
- 每个锁子程序存取的锁状态和所有者的约束在此子程序中描述。如这些约束没有被执行，这些子程序的行为是非特定的。
- OpenMP锁子程序存取锁变量通过以下方式进行：它们总是读取和更新锁变量的最当前值。锁子程序暗含更新（flush）过程；对这些锁变量值的读取和更新必须按照具有原子性（atomic）更新操作来实现，并不需要OpenMP包含显示更新指令来保证锁变量在不同任务中的一致性。



# 简单锁子程序

- `omp_init_lock`: 初始化简单锁
- `omp_destroy_lock`: 销毁简单锁
- `omp_set_lock`: 设置简单锁
- `omp_unset_lock`: 复位简单锁
- `omp_test_lock`: 测试简单锁, 如存在则设置



# 嵌套锁子程序

- `omp_init_nest_lock`: 初始化嵌套锁
- `omp_destroy_nest_lock`: 销毁嵌套锁
- `omp_set_nest_lock`: 设置嵌套锁
- `omp_unset_nest_lock`: 复位嵌套锁
- `omp_test_nest_lock`: 测试嵌套锁, 如存在则设置



# omp\_init\_lock和omp\_init\_nest\_lock

分别初始化简单锁和嵌套锁

- C/C++:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```



# omp\_destroy\_lock和omp\_destroy\_nest\_lock

分别销毁简单锁和嵌套锁

- C/C++:

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_destroy_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_destroy_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```



# omp\_set\_lock和omp\_set\_nest\_lock

分别设置简单锁和嵌套锁

- C/C++:

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_set_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```



# omp\_unset\_lock和omp\_unset\_nest\_lock

分别复位简单锁和嵌套锁

- C/C++:

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_unset_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```



# omp\_test\_lock和omp\_test\_nest\_lock

分别测试简单锁和嵌套锁，如存在则设置

- C/C++:

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

- Fortran:

```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar
integer function omp_test_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```



# 时间子程序

## 获取计时器相关变量

- `omp_get_wtime`: 获取以秒为单位的当前时间
  - C/C++:

```
double omp_get_wtime(void);
```

- Fortran:

```
double precision function omp_get_wtime()
```

- `omp_get_wtick`: 获取计时器的精度

- C/C++:

```
double omp_get_wtick(void);
```

- Fortran:

```
double precision function omp_get_wtick()
```



# 时间子程序举例

利用时间程序可以计算程序中某段计算花费的时间

- C/C++:

```
double start, end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

- Fortran:

```
double precision start, end
start = omp_get_wtime()
... work to be timed ...
end = omp_get_wtime()
print *, "Work took", end - start, "seconds"
```



# 环境变量

利用在程序运行前设置的环境变量可以控制OpenMP程序的运行

- **OMP\_SCHEDULE**: 设置run-sched-var内在控制变量以控制运行调度类型和并行块大小，可设为static、dynamic、guided或auto
- **OMP\_NUM\_THREADS**: 设置nthreads-var内在控制变量以设置线程数，最常用的参数之一
- **OMP\_DYNAMIC**: 设置dyn-var内在控制变量以设置是否允许动态调整线程
- **OMP\_NESTED**: 设置nest-var内在控制变量以设置是否允许嵌套
- **OMP\_STACKSIZE**: 设置stacksize-var内在控制变量以设置堆栈大小
- **OMP\_WAIT\_POLICY**: 设置wait-policy-var内在控制变量以设置线程等待策略
- **OMP\_MAX\_ACTIVE\_LEVELS**: 设置max-active-levels-var内在控制变量以设置允许的最大嵌套的激活的并行区域
- **OMP\_THREAD\_LIMIT**: 设置thread-limit-var内在控制变量以设置在OpenMP程序中允许的最大线程分割数



# 环境变量设置方法

需要在程序运行前设置，程序运行后再设置将不起作用，以下为几种常见shell下的设置方法，以设置OMP\_SCHEDULE为例：

- sh、bash、ksh:

```
export OMP_SCHEDULE=dynamic
```

- csh:

```
setenv OMP_SCHEDULE dynamic
```

- DOS:

```
set OMP_SCHEDULE=dynamic
```



# OpenMP与MPI混合并行编程

- MPI并行进程内部可以再执行OpenMP并行线程以联合进行并行计算
- 一般用于节点内部为共享内存，节点间为分布式内存的超算系统

```
program main
use omp_lib
use mpi
implicit none
integer iErr, MyID, NumNodes
integer i, j

call mpi_init(ierr)
call mpi_comm_rank(MPI_COMM_WORLD, MyID, iErr)
call mpi_comm_size(MPI_COMM_WORLD, NumNodes, iErr)
print*, 'MPI: My ID:', MyID, ', Number of Processes:', NumNodes
!$omp parallel
i=omp_get_num_threads()
j=omp_get_thread_num()
print*, 'OMP: Thread Number:', j, ', Number of Threads:', i
!$omp end parallel
call mpi_finalize(ierr)
end
```



# OpenMP程序在Linux/Unix下的编译

在Linux/Unix平台下常见GCC、Intel、PGI编译器的编译方式

- C:

- GCC: `gcc -fopenmp -o prog-omp prog-omp.c`
- Intel: `icc -qopenmp -o prog-omp prog-omp.c`
- PGI: `pgcc -mp -o prog-omp prog-omp.c`

- C++:

- GCC: `g++ -fopenmp -o prog-omp prog-omp.cpp`
- Intel: `icpc -qopenmp -o prog-omp prog-omp.cpp`
- PGI: `pgCC -mp -o prog-omp prog-omp.cpp`

- Fortran:

- GCC: `gfortran -fopenmp -o prog-omp prog-omp.f90`
- Intel: `ifort -qopenmp -o prog-omp prog-omp.f90`
- PGI: `pgf90 -mp -o prog-omp prog-omp.f90`

- 编译器是从某个版本开始支持OpenMP的，如GCC从4.3.2开始



对OpenMP与MPI混合编程的程序，只要将串行编译命令换成对应的MPI编译命令以及对应的OpenMP编译选项即可

- C:

- GCC: mpicc -fopenmp -o prog-omp prog-omp.c
- Intel: mpicc -qopenmp -o prog-omp prog-omp.c
- PGI: mpicc -mp -o prog-omp prog-omp.c

- C++:

- GCC: mpiCC -fopenmp -o prog-omp prog-omp.cpp
- Intel: mpiCC -qopenmp -o prog-omp prog-omp.cpp
- PGI: mpiCC -mp -o prog-omp prog-omp.cpp

- Fortran:

- GCC: mpif90 -fopenmp -o prog-omp prog-omp.f90
- Intel: mpif90 -qopenmp -o prog-omp prog-omp.f90
- PGI: mpif90 -mp -o prog-omp prog-omp.f90

注：上述mpicc、mpiCC和mpif90等命令在不同MPI实现下有可能不一样



# Linux/Unix下的运行

运行：

- bash、ksh、zsh:

```
export OMP_NUM_THREADS=8  
prog-omp
```

- csh、tcsh:

```
setenv OMP_NUM_THREADS 8  
prog-omp
```

- 对OpenMP与MPI混合编程的程序的执行，只要设置OpenMP环境变量后用MPI程序运行方式运行即可，如以下启动四个MPI进程，每个MPI进程启动8个OpenMP线程：

```
export OMP_NUM_THREADS=8  
mpiexec -np 4 prog-omp
```

- 超算系统一般采用作业调度系统管理用户作业，需通过作业调度系统来运行作业，请参看对应的作业调度系统手册



# MPI与OpenMP对比

- OpenMP和MPI是并行编程的两种方式:
  - MPI:
    - 进程级
    - 分布式内存
    - 显式
    - 可扩展性好
  - OpenMP:
    - 线程级（并行粒度）
    - 共享内存
    - 隐式（数据分配方式）
    - 可扩展性差
- OpenMP采用共享内存，只适应SMP、DSM架构，不适合集群架构
- MPI适合于广泛架构，但编程模型复杂:
  - 需要分析及划分应用程序问题，并将问题映射到分布式进程集合
  - 需要解决通信延迟大和负载不平衡两个主要问题
  - 调试麻烦
  - 程序可靠性差，一个进程出问题，整个程序将错误



# 影响OpenMP并行程序的因素

OpenMP程序为共享内存的，各线程可以直接读写其它线程的内存，无需通过网络来进行数据交换：

- 避免频繁设置内存私有或共享
- 降低写同样内存的操作，为了保证正确，各线程写同样内存时，需要排队，导致效率降低



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

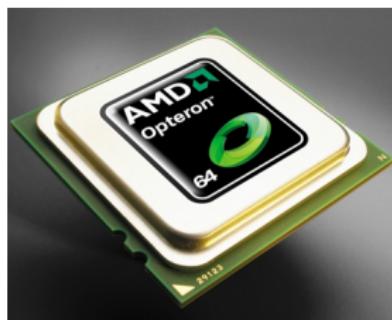
⑩ 杂七杂八

⑪ 资料书籍

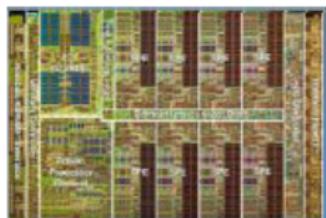
⑫ 联系信息

# 多核时代

- 核心时钟频率基本不变
- 多个适当复杂度、相对低功耗内核并行工作
  - 配置并行硬件资源提高处理能力



(a) AMD Quad-core  
Opteron



(b) IBM Cell Broadband  
Engine

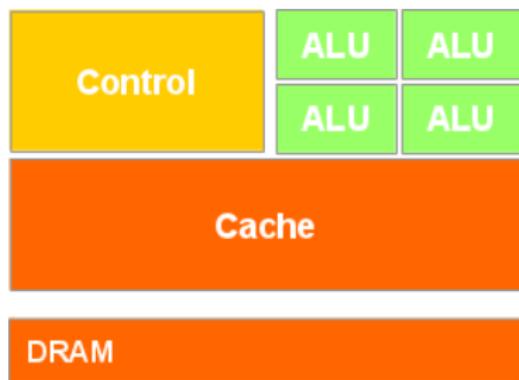


(c) NVIDIA GT200



# GPU与CPU硬件架构对比

- CPU: 更多资源用于缓存及流控制



(a) CPU



# GPU与CPU硬件架构对比

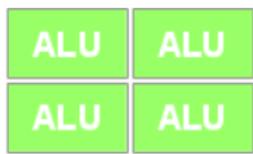
- CPU: 更多资源用于缓存及流控制
- GPU: 更多资源用于数据计算
  - 适合具备可预测、针对数组的计算模式



Cache

DRAM

(a) CPU



DRAM

(b) GPU



# GPU与CPU性能对比

## NVIDIA Tesla V100 GPU与Intel Xeon Gold 6140 CPU对比

- 双精度浮点运算性能优势明显
  - 6140 CPU:  $2.30\text{GHz} \times 18\text{核} \times 32\text{flops/Hz} = 1324.8\text{GFlops} = 1.32\text{TFlops}$
  - V100 GPU: CUDA 5120核(1.38GHz/1.53GHz), Tensor 640核
    - 双精度计算能力: 7/7.8 TFlops(PCIe/SXM2)
    - 单精度计算能力: 14/15.7 TFlops
    - Tensor计算能力: 112/125 TFlops
- 很多应用V100实测性能远优于CPU

### 1 GPU Node Replaces Up To 54 CPU Nodes

Node Replacement: HPC Mixed Workload



CPU Server: Dual Xeon Gold 6140@2.30GHz, GPU Servers: some CPU server w/ 4x V100 PCIe | CUDA Version: CUDA 9.1| Dataset: NAMD (STMD), GTC (mpiproc.inl), MILC (APEX Medium), SPECFEM3D [four\_material\_simple\_model] | To arrive at CPU node equivalence, we use measured benchmark with up to 8 CPU nodes. Then we use linear scaling to scale beyond 8 nodes.



# 应用范围

CPU: control processor

- 不规则数据结构
- 不可预测存取模式
- 递归算法
- 分支密集型算法
- 单线程程序



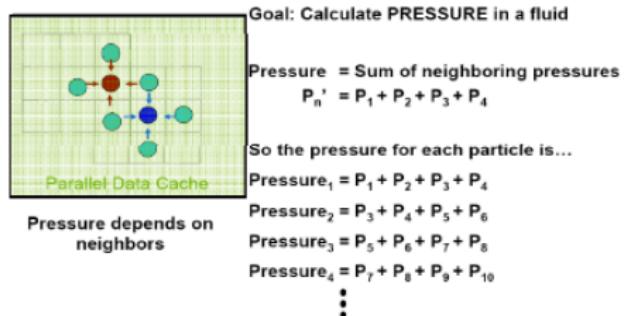
# 应用范围

CPU: control processor

- 不规则数据结构
- 不可预测存取模式
- 递归算法
- 分支密集型算法
- 单线程程序

GPU: data processor

- 规则数据结构
- 可预测存取模式
- 适合领域: 油气勘探、金融分析、医疗成像、有限元、基因分析、地理信息系统、……



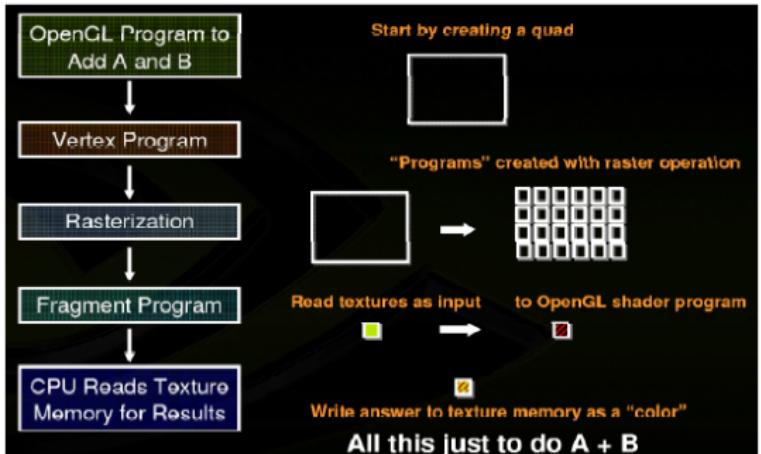


## ● 核心思想

- 用图形语言描述通用计算问题
- 把数据映射到vertex或者fragment处理器

## ● 缺点

- 硬件资源使用不充分
- 存储器访问方式严重受限
- 难以调试和查错
- 高度图形处理和编程技巧





# GPGPU编程语言：CUDA、OpenCL和OpenACC

- CUDA(Compute Unified Device Architecture):

- 主要针对NVIDIA GPU，也支持OpenCL
- CUDA C语言比OpenCL更高一个级别，相当于高级开发环境
- 较新的PGI Fortran编译器支持CUDA Fortran
- 出现较早，现有应用多
- [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html)

- OpenCL(Open Computing Language):

- 由OpenGL标准指定组织Khronos Group负责
- 开放标准，不仅支持GPU（NVIDIA、AMD和Intel等），有些CPU也支持
- OpenCL是一个API，相当于底层开发环境
- 不直接支持Fortran，开源项目ForOpenCL是Fortran语言环境下，对OpenCL的C函数库封装的层次模块
- 近几年才出现，应用较少
- <http://www.khronos.org/opencl/>

- OpenACC(Directives for Accelerators):

- OpenACC指令在加速科学代码方面是一种简单而又可移植的方式，也支持众核等
- 只要在Fortran或C语言代码中插入编译器提示，编译器即可将代码中计算量繁重的部分自动交由GPU处理，以实现更高的性能
- <http://www.openacc.org/>

- 需特定级别之上显卡才支持，很多现在主流的台式机和笔记本显卡已支持



# CUDA程序 I

```
// Kernel definition, see also section 4.2.3 of NVIDIA Cuda Programming Guide
__global__ void vecAdd(float* A, float* B, float* C) //此处__global__声明的函数在GPU端执行
{
    // threadIdx.x is a built-in variable provided by CUDA at runtime
    int i = threadIdx.x;
    A[i]=0;
    B[i]=i;
    C[i] = A[i] + B[i];
}

#include <stdio.h>
#define SIZE 10
int main()
{
    int N=SIZE;
    float A[SIZE], B[SIZE], C[SIZE];
    float *devPtrA;
    float *devPtrB;
    float *devPtrC;
    int memsize= SIZE * sizeof(float);

    cudaMalloc((void**)&devPtrA, memsize); //申请GPU中的内存
    cudaMalloc((void**)&devPtrB, memsize);
    cudaMalloc((void**)&devPtrC, memsize);
```



# CUDA程序 II

```
cudaMemcpy(devPtrA, A, memsize, cudaMemcpyHostToDevice); //从CPU复制GPU
cudaMemcpy(devPtrB, B, memsize, cudaMemcpyHostToDevice); //Device为GPU, Host为CPU
// __global__ functions are called: Func<<< Dg, Db, Ns >>>(parameter);
vecAdd<<<1, N>>>(devPtrA, devPtrB, devPtrC); //<<<>>>调用在GPU上执行的函数
cudaMemcpy(C, devPtrC, memsize, cudaMemcpyDeviceToHost); //从GPU复制CPU

for (int i=0; i<SIZE; i++) printf("C[%d]=%f\n", i, C[i]);

cudaFree(devPtrA); //释放GPU内存
cudaFree(devPtrA);
cudaFree(devPtrA);
}
```



# OpenCL程序 I

```
#include <stdlib.h>
#include <stdio.h>
#include <CL/cl.h> //OpenCL头文件

#define LEN(arr) sizeof(arr) / sizeof(arr[0])
//设备端kernel源程序，以字符串数组的方式保存，在某些论坛中提示每个语句最好以回车结束;
//在运行时从源码编译成可执行在GPU上的kernel代码
const char* src[] = {
    "__kernel void vec_add(__global const float*a,__global const float*b,__global float*c)\n",
    "{\n",
    "    int gid=get_global_id(0);\n",
    "    c[gid]=a[gid]+b[gid];\n",
    "}"
};

int main()
{
    //创建OpenCL context, 该context与GPU设备关联, 详见OpenCL规范4.3节
    cl_context context=clCreateContextFromType(NULL,CL_DEVICE_TYPE_GPU,NULL,NULL,NULL);

    //获取context中的设备ID, 详见OpenCL规范4.3节
    size_t cb;
    clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
    cl_device_id *devices = malloc(cb);
```



# OpenCL程序 II

```
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

//create a command-queue, 详见OpenCL规范5.1节
cl_command_queue cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

//创建kernel, 详见OpenCL规范5.4节和5.5.1节
cl_program program = clCreateProgramWithSource(context, LEN(src),src,NULL,NULL);
cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "vec_add", NULL);

//Host端输入初始化
size_t n = 5;
float srcA[] = {1, 2, 3, 4, 5};
float srcB[] = {5, 4, 3, 2, 1};
float dst[n];

//设置kernel的输入输出参数, 详见OpenCL规范5.2.1节
cl_mem memobjs[3];
memobjs[0]=clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(float)*n,srcA,NULL);
memobjs[1]=clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(float)*n,srcB,NULL);
memobjs[2]=clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(float)*n,NULL,NULL);
```



# OpenCL程序 III

```
//set "a", "b", "c" vector argument, 详见OpenCL规范5.5.2节
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&memobjs[2]);

size_t global_work_size[1] = {n};
//execute kernel, 详见OpenCL规范6.1节
err=clEnqueueNDRangeKernel(cmd_queue,kernel,1,NULL,global_work_size,NULL,0,NULL,NULL);

//read output array, 详见OpenCL规范5.2.2节
err=clEnqueueReadBuffer(cmd_queue,memobjs[2],CL_TRUE,0,n*sizeof(cl_float),dst,0,NULL,NULL);

for (int i=0; i<n; ++i) {
    printf("->%f\n", dst[i]);
}
return 0;
}
```



# OpenACC程序

微积分分割圆法求 $\pi$ :

$$f(x) = \frac{4}{1+x^2}$$
$$\pi = \frac{1}{N} \sum_{i=1}^N f\left(\frac{i-0.5}{N}\right) = \frac{1}{N} \sum_{i=0}^{N-1} f\left(\frac{i+0.5}{N}\right)$$

OpenACC加速计算 $\pi$ 的Fortran程序

```
program picalc
implicit none
integer, parameter :: n=1000000
integer :: i
real(kind=8) :: t, pi
pi = 0.0
 !$acc parallel loop
 do i=0, n-1
   t = (i+0.5)/n
   pi = pi + 4.0/(1.0 + t*t)
 end do
 !$acc end parallel loop
print *, 'pi=', pi/n
end program picalc
```



# FPGA：现场可编程门阵列

- FPGA(Field-Programmable Gate Array)，即现场可编程门阵列，它是在PAL、GAL、CPLD等可编程器件的基础上进一步发展的产物。它是作为专用集成电路（ASIC）领域中的一种半定制电路而出现的，既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。
- 开发FPGA的开发相对于传统PC、单片机的开发有很大不同。FPGA以并行运算为主，以硬件描述语言来实现；相比于PC或单片机（无论是冯诺依曼结构还是哈佛结构）的顺序操作有很大区别，也造成了FPGA开发入门较难。FPGA开发需要从顶层设计、模块分层、逻辑实现、软硬件调试等多方面着手。
- 2010年12月30日消息，美英两国科学家联合开发了一款运算速度超快的电脑芯片，使当前台式机的运算能力提升20倍。



# FPGA优缺点

## • 优点

- FPGA由逻辑单元、RAM、乘法器等硬件资源组成，通过将这些硬件资源合理组织，可实现乘法器、寄存器、地址发生器等硬件电路
- FPGA可通过使用框图或者Verilog HDL来设计，从简单的门电路到FIR或者FFT电路
- FPGA可无限地重新编程，加载一个新的设计方案只需几百毫秒，利用重配置可以减少硬件的开销
- FPGA的工作频率由FPGA芯片以及设计决定，可以通过修改设计或者更换更快的芯片来达到某些苛刻的要求（当然，工作频率也不是无限制的可以提高，而是受当前的IC工艺等因素制约）

## • 缺点

- FPGA的所有功能均依靠硬件实现，无法实现分支条件跳转等操作
- FPGA只能实现定点运算

总结：FPGA依靠硬件来实现所有的功能，速度上可以和专用芯片相比，但设计的灵活度与通用处理器相比有很大的差距。



# DSP：数字信号处理

- DSP(Digital Signal Processing): 数字信号处理就是用数值计算的方式对信号进行加工的理论和技术
- 是一种特别适合于进行数字信号处理运算的微处理器，其主要应用是实时快速地实现各种数字信号处理算法，DSP芯片一般具有如下主要特点：
  - 在一个指令周期内可完成一次乘法和一次加法
  - 程序和数据空间分开，可以同时访问指令和数据
  - 片内具有快速RAM，通常可通过独立的数据总线在两块中同时访问
  - 具有低开销或无开销循环及跳转的硬件支持
  - 快速的中断处理和硬件I/O支持
  - 具有在单周期内操作的多个硬件地址产生器
  - 可以并行执行多个操作
  - 支持流水线操作，使取指、译码和执行等操作可以重叠执行
  - 与通用微处理器相比，DSP芯片的其他通用功能相对较弱些



# 其他协处理器

- AMD: AGU
- 中科院计算所: 寒武纪人工智能(AI)芯片
- 华为: 昇腾AI芯片
- 曙光: DCU



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数

假设 $f_{par} = 80\%$ , 那么16、32核及理论最大的并行效率是多少?



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数

假设 $f_{par} = 80\%$ , 那么16、32核及理论最大的并行效率是多少?

- 16核: 4



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数

假设 $f_{par} = 80\%$ , 那么16、32核及理论最大的并行效率是多少?

- 16核: 4
- 32核: 4.4



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数

假设 $f_{par} = 80\%$ , 那么16、32核及理论最大的并行效率是多少?

- 16核: 4
- 32核: 4.4
- 理论: 5



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数

假设 $f_{par} = 80\%$ , 那么16、32核及理论最大的并行效率是多少?

- 16核: 4
- 32核: 4.4
- 理论: 5

怎么办?



# 并行效率

Theorem (Amdahl定律)

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

- $f_{par}$ : 可并行的部分串行时占用的计算时间的百分比
- P: 并行的核数

假设 $f_{par} = 80\%$ , 那么16、32核及理论最大的并行效率是多少?

- 16核: 4
- 32核: 4.4
- 理论: 5

怎么办?

尽可能多地并行化代码

- 是耗时部分
- 不是代码长的部分



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

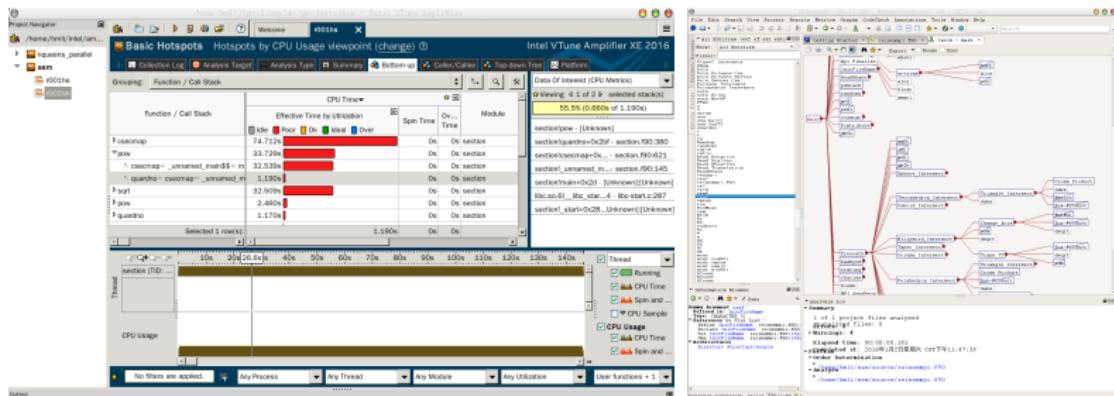
⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息

# 辅助软件工具

- 程序性能分析: Intel VTune Amplifier  
<http://software.intel.com/en-us/intel-vtune/>
- 程序源码结构分析: Understand  
<http://www.scitools.com/products/understand/>



(a) Intel Vtune Amplifier

(b) Understand



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息

李会民（中国科大超算中心）

并行计算及计算可视化



# 为什么要使用数值函数库

- 避免重复开发，缩短开发周期
- 可靠性高
- 性能高
- 程序可以直接调用



# 常见数值函数库

- 一般数值函数库
  - 基本线性代数库(BLAS)
  - 线性代数库(LAPACK)
  - 可扩展性线性代数库(ScalAPACK)
  - 傅立叶变换程序(Fourier Transform functions, FFT)
  - Netlib(<http://www.netlib.org>): 收集了大量常见的数学函数库
- 数值函数库的集合产品
  - MKL: Intel Math Kernel Library
  - AMD CPU库
  - IMSL: Rogue Wave International Mathematics and Statistics Library



# Intel MKL

Intel Math Kernel Library(MKL)主要针对Intel CPU进行了优化，包含：

- 基本线性代数子系统库(BLAS)
- 离散基本线性代数库(Sparse BLAS)
- 线性代数库(LAPACK)
- 可扩展性线性代数库(ScalAPACK)
- 离散求解程序(Sparse Solver routines)
- 向量数学库函数(Vector Mathematical Library functions)
- 向量统计库函数(Vector Statistical Library functions)
- 傅立叶变换程序(Fourier Transform functions, FFT)
- 集群版傅立叶变换程序(Cluster FFT)
- 区间求解程序(Interval Solver routines)
- 三角变换程序(Trigonometric Transform routines)
- 泊松、拉普拉斯和哈密顿求解程序(Poisson, Laplace, and Helmholtz Solver routines)
- 优化（信赖域）求解程序(Optimization (Trust-Region) Solver routines)

主页: <https://software.intel.com/en-us/mkl>

李会民 (中国科大超算中心)

并行计算及计算可视化



# IMSL数学库内容

<https://www.roguewave.com/products-services/imsl-numerical-libraries>

- 线性系统(Linear Systems)
- 本征系统分析(Eigenvalue Analysis)
- 插值与近似(Interpolation and Approximation)
- 积分(Quadrature)
- 微分方程(Differential Equations)
- 变换(Transforms)
- 非线性方程(Nonlinear Equations)
- 优化(Optimization)
- 特殊函数(Special Functions)
- 统计和随机数生成器(Statistics and Random Number Generator)
- 打印函数(Printing Functions)
- 实用程序(Utilities)



# IMSL统计库内容

- 基本统计(Basic Statistics)
- 回归(Regression)
- 相关性与协方差(Correlation and Covariance)
- 变量分析和设计实验(Analysis of Variance and Designed Experiments)
- 绝对和离散数据分析(Categorical and Discrete Data Analysis)
- 非参数统计(Nonparametric Statistics)
- 拟合良好度测试(Tests of Goodness-of-Fit)
- 时间序列和预测(Time Series and Forecasting)
- 多变量分析(Multivariate Analysis)
- 生存和健壮性分析(Survival and Reliability Analysis)
- 几率分布函数和其反换式(Probability Distribution Functions and Inverses)
- 随机数生成器(Random Number Generator)
- 神经网络(Neural Networks)
- 打印函数(Printing Functions)
- 实用程序(Utility)



AMD CPU Libraries主要针对AMD CPU进行了优化，包含：

- BLIS: 基本线性代数库(BLAS)
- libFLAME: 线性代数库(LAPACK)
- AMD Math Library (LibM): 标准C99数学库
- AMD Random Number Generator Library: 随机数产生器
- AMD Secure RNG Library: 安全随机数产生器
- 傅立叶变换程序(Fourier Transform functions, FFT)



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息



- 大量类似数据需要处理，如画1000副图，需更改全部图的某条曲线的颜色等
- 需要对数据进行进一步处理，如存在坏的数据
- 保证生成的图像风格一致



# 数据处理、计算可视化软件

- Origin
- Mathematica
- Maple
- MATLAB
- IDL
- GNUPLOT



# Origin

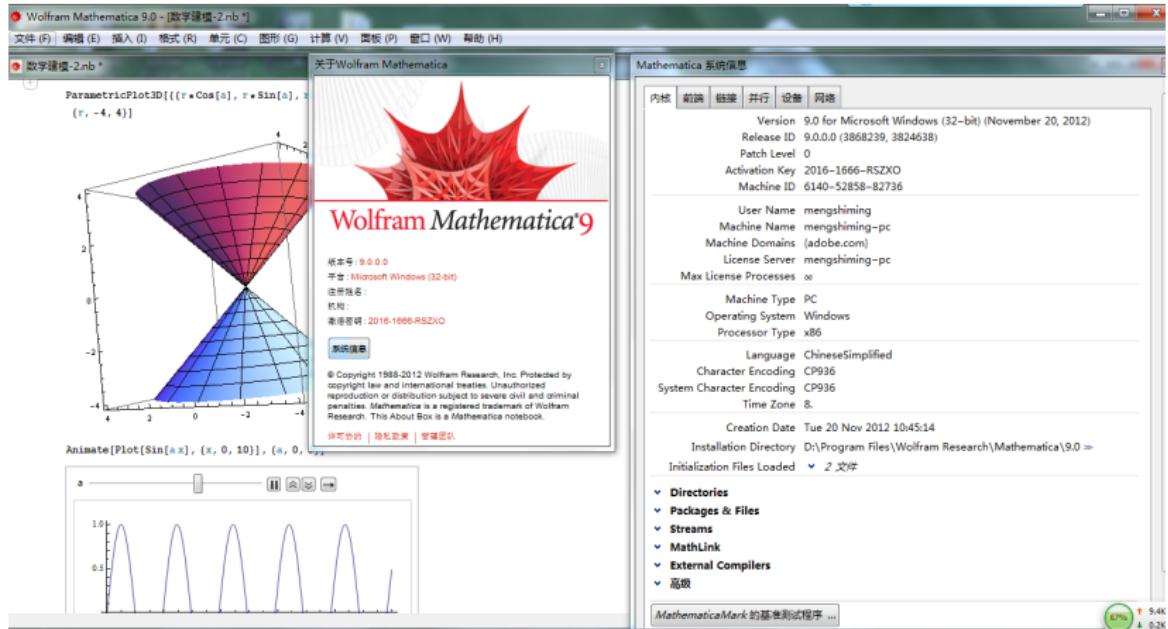
- OriginLab总裁及创办人之一杨超平：1977年考入我校地球与空间科学系，1982年通过CUSPEA到卡耐基梅农大学并于1988年获取生物物理学博士学位
- 我校已购买供全校师生（2017.1.1-2022.12.31）免费使用的Origin Pro，参见正版化软件主页：<http://zbh.ustc.edu.cn>
- 主页：<https://www.originlab.com/>
- Origin 7.0简明用法

The screenshot displays the Origin software interface with several windows open, illustrating its features:

- Left Panel:** A large blue panel titled "Publication-quality Graphing for Scientists and Engineers" lists features:
  - ✓ Over 100 Built-in Graph Templates
  - ✓ Point-and-click Customization of All Elements
  - ✓ Create Custom Graph Templates & Themes
  - ✓ Push to PowerPoint or Copy-Paste to Word
  - ✓ Export Image to 12+ Vector and Raster Formats
  - ✓ Perform Batch Plotting Across Columns/Sheets/Books
- Top Right Window:** A graph titled "Wortable Gulf - Fitted vs Measured" comparing experimental data (red dots) with a fitted curve (orange line). It includes subplots for Agarose and Sphingomyelin.
- Middle Left Window:** A histogram titled "Particle Number and Volume Concentrations in a Closed Photochemical System" showing particle size distributions for 40 nm, 60 nm, and 120 nm.
- Middle Right Window:** A 3D surface plot titled "Jan.27.2011 US Fed Inquiry Committee released" showing data over time and space.
- Bottom Left Window:** A scatter plot titled "US CPI vs Electricity Price (MWh)" showing a strong positive correlation.
- Bottom Right Window:** A 2D contour plot titled "Jan.27.2011 US Fed Inquiry Committee released" showing spatial distribution.



# Mathematica



- 我校已购买50个浮动用户的从2020年1月1日起的永久授权
- 主页: <http://www.wolfram.com/mathematica/>
- 数学百科: <http://mathworld.wolfram.com/>



# Mathematica

Mathematica是一款科学计算软件，很好地结合了数值和符号计算引擎、图形系统、编程语言、文本系统、和与其他应用程序的高级连接。

- 覆盖范围

- 符号式语言
- 数学计算
- 数值方法
- 可视化
- 代数操作
- 数论
- 数据分析
- 图形计算
- 交互式计算
- 图像计算
- 几何计算
- ...

- Mathematica的用户群

- 数学中的许多计算是非常繁琐的，特别是函数的作图费时又费力，而且所画的图形很不规范，所以现在流行用Mathematica符号计算系统进行学习，很多问题便迎刃而解
- 最主要的是科技工作者和其它专业人士，还被广泛地用于教学中



# Mathematica例子

- 解方程:

$$\begin{cases} \frac{x^2}{3} + \frac{y^2}{4} = 1 \\ y = x \end{cases}$$

- 在Mathematica中输入:

In [1]: Solve[{x^2/3+y^2/4==1,y==x},{x,y}]

- Mathematica输出结果:

Out[1]= $\left\{\left\{x \rightarrow -2 \sqrt{\frac{3}{7}}, y \rightarrow -2 \sqrt{\frac{3}{7}}\right\}, \left\{x \rightarrow 2 \sqrt{\frac{3}{7}}, y \rightarrow 2 \sqrt{\frac{3}{7}}\right\}\right\}$

- 不定积分:

$$\int \frac{1}{x^3 + 1} dx$$

- 在Mathematica中输入:

In [1]: Integrate[1/(x^3+1),x]

- Mathematica输出结果:

Out[1]= $\frac{\text{ArcTan}\left[\frac{-1+2x}{\sqrt{3}}\right]}{\sqrt{3}} + \frac{1}{3} \log[1+x] - \frac{1}{6} \log[1-x+x^2]$



# Maple

Maple 11 - Y:\MKRDEV\Jackalope\screenshots\what's new\worksheets\contextplot.mw - [Server 6]

File Edit View Insert Format Table Drawing Plot Spreadsheet Tools Window Help

Favorites

π ∞ ±  $\int_a^b f dx$  θ  
 $\phi \lambda \Delta \frac{d}{dx} f$   
 $\frac{\partial}{\partial x} f \sqrt{a} \lim_{x \rightarrow a} f$   
 $\llbracket kg \rrbracket \llbracket m \rrbracket \llbracket s \rrbracket$

Units (SI)  
Units (FPS)

Expression

$\int f dx$   $\int_a^b f dx$   $\sum_{i=k}^n f$   
 $\prod_{i=k}^n f$   $\frac{d}{dx} f$   $\frac{\partial}{\partial x} f$   
 $\lim_{x \rightarrow a} f$   $a^b$   $a_n$   
 $a_s \sqrt{a}$   $\sqrt[n]{a}$   
 $a! |a| e^a$   
 $\ln(a) \log_{10}(a)$   
 $\log_b(a) \sin(a) \cos(a)$   
 $\tan(a) \begin{pmatrix} a \\ b \end{pmatrix} f(a)$

Plot of  $x^2 \sin(x)$

Local minimum  
 $x \approx \frac{6\pi}{4}$

Curve 1

2 x sin(x) +  $\frac{x^2 \cos(x)}{\pi}$  integrate w.r.t x  $\frac{2 (\sin(x) - x \cos(x))}{\pi} + \frac{x^2 \sin(x) - 2 \sin(x) + 2 x \cos(x)}{\pi}$

Cut Ctrl+X  
Copy Ctrl+C  
Copy full precision  
Paste Ctrl+V  
Numeric Formatting...  
Apply a Command  
Approximate  
Assign to a Name  
Collect  
Combine  
Differentiate  
Evaluate at a Point  
Expand  
Factor  
Integrate  
Series  
Simplify  
Solve  
Complex Maps  
Constructions  
Conversions  
Integer Functions  
Integral Transforms  
Language Conversions  
Optimization  
Plots  
Simplifications  
Units

Fourier Cosine Transform  
Fourier Sine Transform  
Fourier Transform  
Hilbert Transform  
Inverse Fourier Transform  
Inverse Hilbert Transform  
Inverse Laplace Transform  
Laplace Transform

try: 4.93M Time: 0.54s Math Mode

主页: <https://www.maplesoft.com/products/Maple/>



## 强大的求解器：数学和分析软件的领导者

- 内置超过5000个符号和数值计算命令，覆盖几乎所有的数学领域，如微积分、线性代数、方程求解、积分和离散变换、概率论和数理统计、物理、图论、张量分析、微分和解析几何、金融数学、矩阵计算、线性规划、组合数学、矢量分析、抽象代数、泛函分析、数论、复分析和实分析、抽象代数、级数和积分变换、特殊函数、编码和密码理论、优化等
- 各种工程计算：优化、统计过程控制、灵敏度分析、动力系统设计、小波分析、信号处理、控制器设计、集总参数分析和建模、各种工程图形等
- 提供世界上最强大的符号计算和高性能数值计算引擎，包括世界上最强大的微分方程求解器（ODEs、PDEs、高指数DAEs）
- 智能自动算法选择
- 强大、灵活、容易使用的编程语言，让您能够开发更复杂的模型或算法
- 与多学科复杂系统建模和仿真平台MapleSim紧密集成



# MATLAB

The screenshot shows a MATLAB interface with several windows open:

- Current Folder Browser:** Shows files like cell.eps, comsphry-hml.m, comsphry-hml.mlog, comsphry-hml.mnw, comsphry-hml.pdf, comsphry-hml.sem, comsphry-hml.tex, comsphry-hml.toc, comsphry-hml.vrb, gppu.eps, ganglia1.eps, ganglia2.eps, gppu1.eps, gppu2.eps, heatmaps.3png, heatmaps.eps, idl.eps, j2c.eps, j3c.eps, lenovo1800.eps, lenovo700g3.eps, Maple.eps, maple.eps, moreeps, moreL1.png, moreZ2.png, note, opteron.eps, pressure.eps, pressure1.eps, prox.eps, randn1.eps, sinnew, sugar.eps, sunway.eps, surface23.eps, th1-la.eps, top-1.eps.
- Editor - /home/hml/tx/comsphry.m:** Displays the following MATLAB code:

```

1 % Load raw data
2 fid = fopen('raw.dat','r');
3 fprintf(fid,'%.10f %.10f %.10f\n',alpha,beta,delta);
4 % Start writing output
5 for i=1:418
6    for j=1:418
7       for k=1:418
8          fprintf(fid,'%.10e %.10e %.10e\n',E_2(i,j,k));
9       end
10    end
11  end
12  fclose(fid);
13
14 % Read xyz data
15 fid=fopen('xyz.dat','r');
16 for i=1:418
17    fprintf(fid,'%.10e %.10e %.10e\n',x(i),y(i),z(i));
18 end
19 whos
20 fclose(fid);

```
- Figure 1:** A 3D surface plot showing a bell-shaped distribution over a 3D grid. The axes range from -5 to 5.
- Workspace:** Shows variables X, Y, Z, T, I, x, y, z, xlin, ylin, and zlin.
- Command History:** Displays the command history for the session, including various file operations and data processing steps.

主页: <https://www.mathworks.com/>



# MATLAB

- MATLAB和Mathematica、Maple并称为三大数学软件，它在数学类科技应用软件中在数值计算方面首屈一指
- 在MATLAB中创建的组是矩阵，MATLAB的名字取自矩阵实验室(MATrix LABoratory)
- MATLAB是一个可视化的计算程序，被广泛地使用于从个人计算机到超级计算机范围内的各种计算机上
- MATLAB包括命令控制、可编程，有上百个预先定义好的命令和函数。这些函数能通过用户自定义函数进一步扩展
- MATLAB有许多强有力的命令。例如，MATLAB能够用一个单一的命令求解线性系统，能完成大量的高级矩阵处理
- MATLAB有强有力的二维、三维图形工具
- MATLAB能与其他程序一起使用。例如，MATLAB的图形功能，可以在一个FORTRAN程序中完成可视化计算
- 上百种不同的MATLAB工具箱可应用于特殊的应用领域

学校已购买供全校师生免费使用的MATLAB，参见正版化软件主页：  
<http://zbh.ustc.edu.cn>



# MATLAB应用领域

- 工业研究与开发
- 数学教学，特别是线性代数，所有基本概念都能涉及在数值分析和科学计算方面的教学与研究，能够详细地研究和比较各种算法
- 在诸如电子学、控制理论和物理学等工程和科学学科方面的教学与研究
- 在诸如经济学、化学和生物学等有计算问题的所有其他领域中的教学与研究

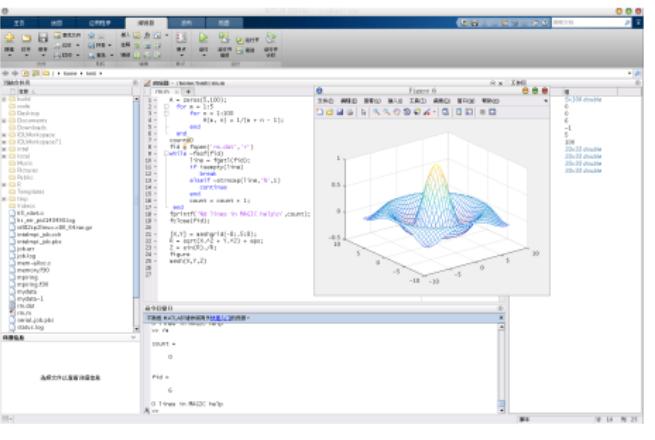


# MATLAB代码

```
A = zeros(5, 100);
for m = 1:5
    for n = 1:100
        A(m, n) = 1 / (m + n - 1);
    end
end

count=0
fid = fopen('rm.dat', 'r')
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line)
        break
    elseif ~strcmp(line, '%', 1)
        continue
    end
    count = count + 1;
end
fprintf('%d lines in MAGIC.help\n', count);
fclose(fid);

[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R). / R;
figure
mesh(X, Y, Z)
```



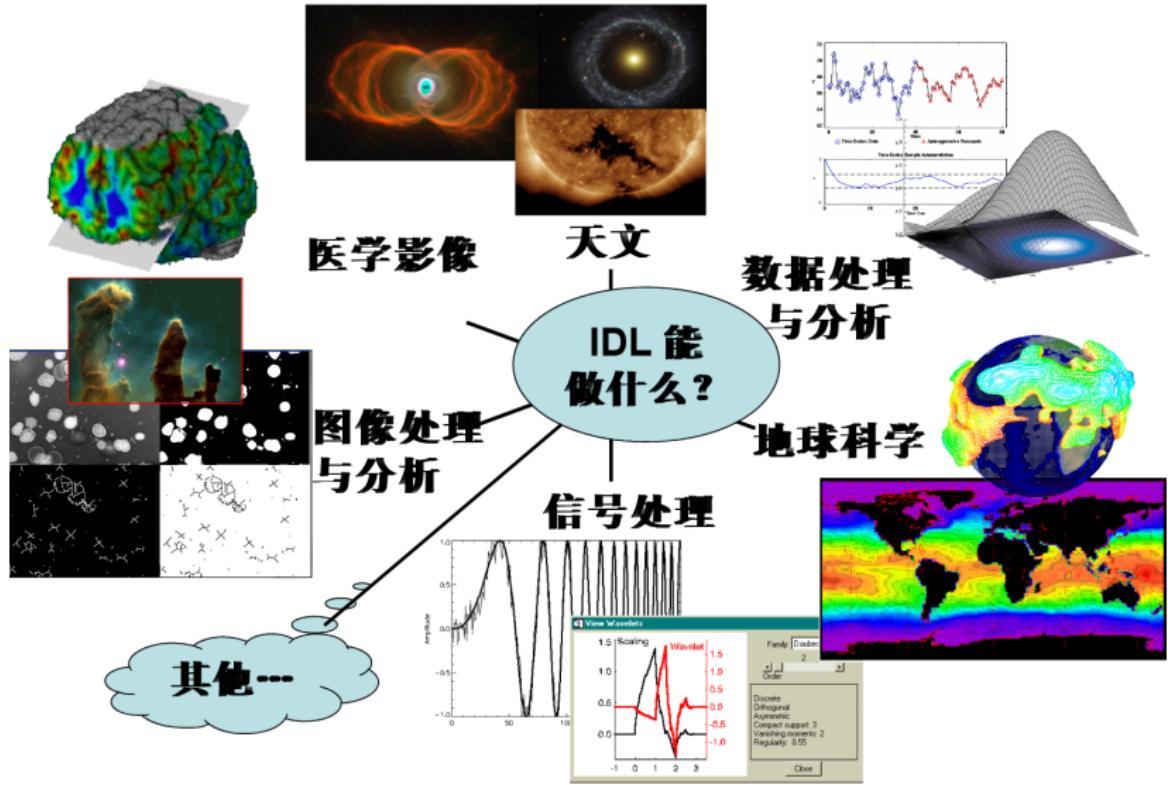


# Mathematica、Maple、MATLAB比较

- MATLAB算是最流行、最通用的数学软件了，它的数值运算可说是最强的，而符号运算方面稍差，用的是Maple的内核
- Maple和Mathematica是互相竞争的对手
  - Mathematica把数值和符号运算二者结合的比较好，两方面都很强
  - Maple主攻符号运算，这方面可说是最强



# IDL能做什么





# IDL是什么

- IDL(Interactive Data Language, 交互式数据语言)是美国Harris Geospatial Solutions, Inc (之前叫Exelisvis、ittvis) 公司的产品，进行二维及多维数据可视化表现、分析及应用开发的软件，面向矩阵、语法简单的第四代可视化计算机语言，是进行数据分析、可视化表达与跨平台应用开发的理想工具
- IDL用户涵盖NASA、ESA、NOAA、Siemens、EMedical等
- 科学家利用IDL对“勇气号”和“机遇号”的数据进行数据分析和图像处理
- 主页：<https://www.harrisgeospatial.com/Software-Technology/IDL>

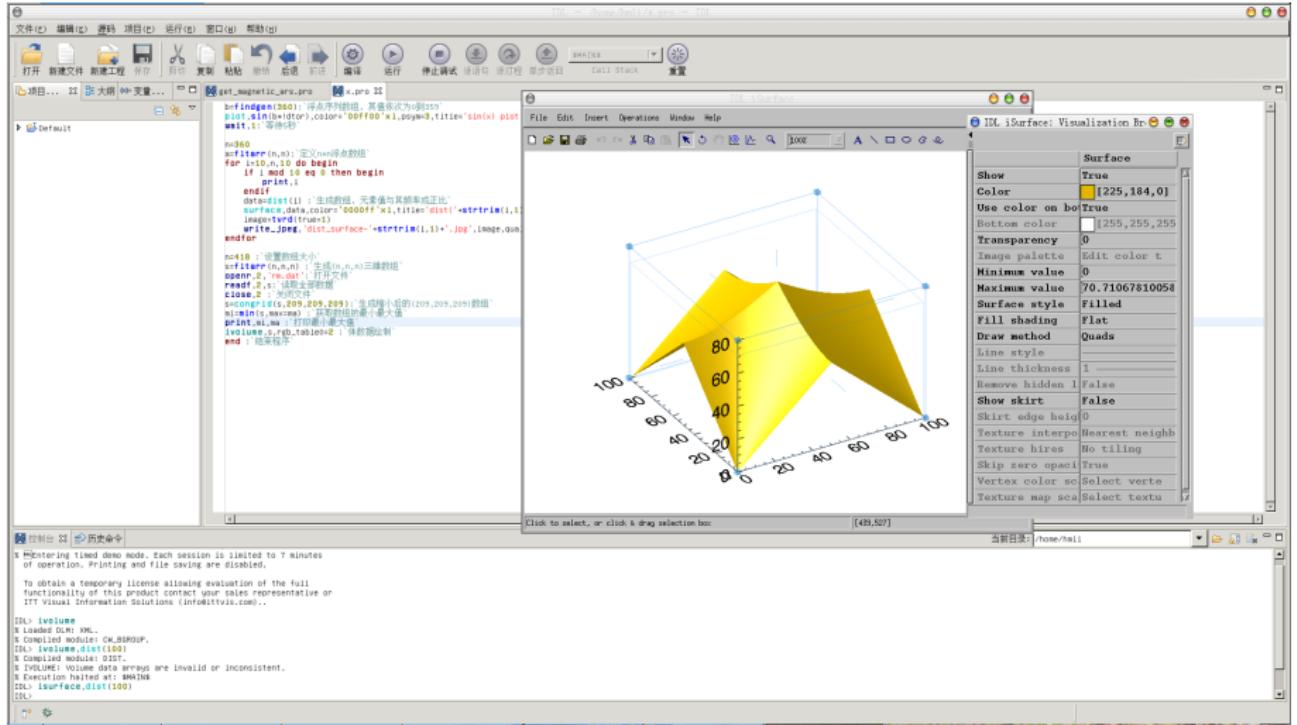


# IDL的特点

- IDL=VC 菜单的定制、消息传递
- IDL=VB 可视化界面的设计、语言通俗易懂、编程入门容易
- IDL=JAVA 具有良好的跨平台能力，方便移植，面向对象特性
- IDL=FORTRAN+C
  - 语言风格绝大部分继承自Fortran，少量来源于C
  - 面向矩阵，执行效率高，代码量比C和Fortran少得多，简洁而不失灵活性
- IDL=MATLAB 提供了大量封装和参数化了的数学函数及各种信号处理的方法
- IDL=OPENGL 提供了丰富的二维、三维图形图像操作类，能高效快速地对数据进行可视化



# IDL集成开发环境

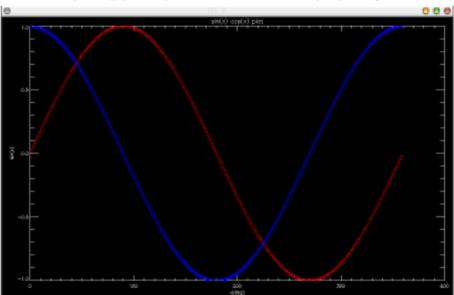




# IDL程序

; 分号 ; 后为注释

```
b=findgen(360)*!dtor ;生成浮点序列数组，其值依次为0到 $2\pi$ ，下行绘制坐标轴  
plot,sin(b),color='ffffff' xl,title='sin(x) plot',xtitle='x(deg)',ytitle='sin(x)',/nodata  
oplot,sin(b),color='00ff00' xl,psym=4 ;绘制红色sin曲线  
oplot,cos(b),color='ff0000' xl,psym=5 ;增加蓝色cos曲线  
wait,5 ;等待5秒
```



n=360

```
for i=10,n,10 do begin  
    if i mod 10 eq 0 then begin  
        print,i  
    endif
```

data=dist(i) ;生成数组，元素值与其频率成正比

```
surface,data,color='ff00ff' xl,title='dist ('+strtrim(i,1)+')',charsize=2 ;绘制表面图  
image=tvrdf(true=1) ;将屏幕图像存入到图像数组
```

```
write_jpeg,'dist_surface'+strtrim(i,1)+'.jpg',image,true=1 ;存储图像到 dist_surface-i.jpg
```

endfor

n=418 ;设置数组大小

s=fltarr(n,n,n) ;生成(n,n,n)三维数组

openr,2,'rm.dat' ;打开文件

readf,2,s ;读取全部数据

close,2 ;关闭文件

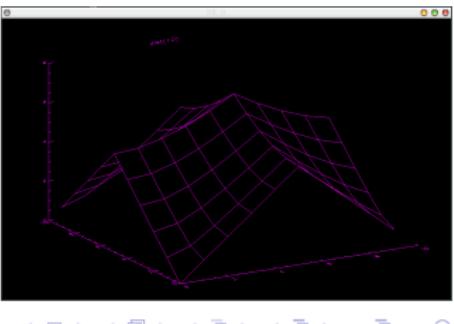
s=congrid(s,209,209,209);生成缩小的(209,209,209)数组

mi=min(s,max=ma) ;获取数组的最小最大值

print,mi,ma ;打印最小最大值

ivolume,s,rgb\_table0=2 ;体数据绘制

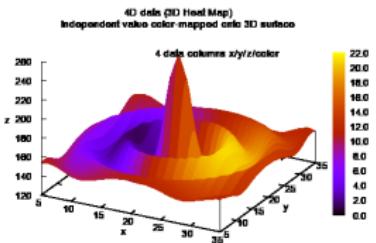
end ;结束程序





# GNUPLOT

- GNUPLOT是一个命令行的交互式绘图工具。用户通过输入命令，可以逐步设置或修改绘图环境，并以图形描述数据或函数，使我们可以借由图形做更进一步的分析。
- GNUPLOT是由Colin Kelly和Thomas Williams于1986年开始开发的科学绘图工具，支持二维和三维图形。它的功能是把数据资料和数学函数转换为容易观察的平面或立体的图形，它有两种工作方式，交互式方式和批处理方式，它可以让使用者很容易地读入外部的数据结果，在屏幕上显示图形，并且可以选择和修改图形的画法，明显地表现出数据的特性。



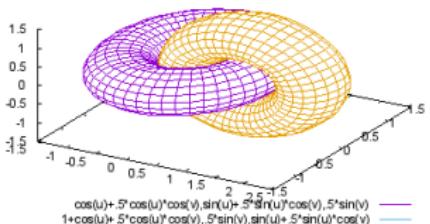
主页: <http://gnuplot.info/>



# GNUPLOT举例

```
# set terminal pngcairo transparent enhanced font "arial,10" fontsize 1.0 size 500, 350
# set output 'surface2.9.png'
set dummy u,v
set key bmargin center horizontal Right noreverse enhanced autotitles nobox
set parametric
set view 50, 30, 1, 1
set isosamples 50, 20
set hidden3d back offset 1 trianglepattern 3 undefined 1 altdiagonal bentover
set ticslevel 0
set title "Interlocking Tori"
set urange [ -3.14159 : 3.14159 ] noreverse nowriteback
set vrangle [ -3.14159 : 3.14159 ] noreverse nowriteback
splot cos(u)+.5*cos(u)*cos(v),sin(u)+.5*sin(u)*cos(v),.5*sin(v) with lines,
      1+cos(u)+.5*cos(u)*cos(v),.5*sin(v),sin(u)+.5*sin(u)*cos(v) with lines
```

Interlocking Tori





① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息



# 作业调度系统

- 在一个大型系统内部，通常需要处理一些自动化运行的任务，通常会采用系统自带的crontable的定时任务完成
- 但是，很多情况下，是多个作业，彼此先后执行，共同完成任务。在这样的情况下，定时任务存在两个明显的问题：
  - 浪费了大量的系统等待时间
  - 假设两个作业，第一个作业必须在第二个作业前运行，如第二个作业先运行，就会有灾难性的后果，对于定时任务而言，解决任务这样两个作业优先级的问题是只能把任务一的运行时间安排在二之前，不能完全满足前面的假设，但是对于作业调度器而言，安排作业的优先级，是最基本的功能，简直是小Case
- 常见作业调度系统：
  - Condor
  - LSF(IBM Platform Load Sharing Facility)
  - LoadLeveler(IBM Tivoli Workload Scheduler)
  - Maui
  - PBS(PBS Pro、OpenPBS、TORQUE)
  - SGE(Oracle Grid Engine)
  - SLURM(Simple Linux Utility for Resource Management)



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息



# 什么是T<sub>E</sub>X/L<sub>A</sub>T<sub>E</sub>X?



T<sub>E</sub>X由Donald E. Knuth(高德纳)在1977-1982年间为撰写The Art of Computer Programming而设计，是非常优秀的排版软件，而L<sub>A</sub>T<sub>E</sub>X则是基于T<sub>E</sub>X之上的一一个宏包集，使人们更加容易使用T<sub>E</sub>X，目前大部分使用的T<sub>E</sub>X系统都是L<sub>A</sub>T<sub>E</sub>X这个宏集。

Donald E. Knuth作为斯坦福大学The Art of Computer Programming的名誉退休教授是算法和程序设计技术的先驱者，是计算机排版系统T<sub>E</sub>X和METAFONT的发明者，他因这些成就和大量创造性的影响深远的著作（19部书和160篇论文）而誉满全球。

<http://www-cs-faculty.stanford.edu/~knuth/>



# 为什么要用 TeX/LaTeX?

- 优点:

- 排版的效果非常整齐漂亮
- 排版的效率非常高
- 非常稳定, 从1995年到现在, TeX系统只发现过一个bug
- 排版科技文献, 尤其是含有许多数学公式的文献特别方便、高效。现今没有一个排版软件在排版数学公式上面能和TeX/LaTeX相媲美
- 纯文本, 可用任何编辑器编辑

- 缺点:

- 不是WYSIWYG, 需编译后才知道效果
- 国内站点: <http://www.ctex.org/>



# 例子

```
\documentclass[12pt]{article}
\usepackage{fontspec,xltxtra,xunicode}
\defaultfontfeatures{Mapping=tex-text}
\setromanfont{AR PL UMing CN}
\setsansfont[Scale=MatchLowercase,Mapping=tex-text]{FreeSans}
\setmonofont[Scale=MatchLowercase]{FreeMono}
\begin{document}
关于$\sigma$的积分公式
\begin{equation}
\int_a^b \sigma(x) dx
\end{equation}
\end{document}
```

关于 $\sigma$ 的积分公式

$$\int_a^b \sigma(x) dx \quad (1)$$



# 高效编辑：两大编辑器VIM与Emacs

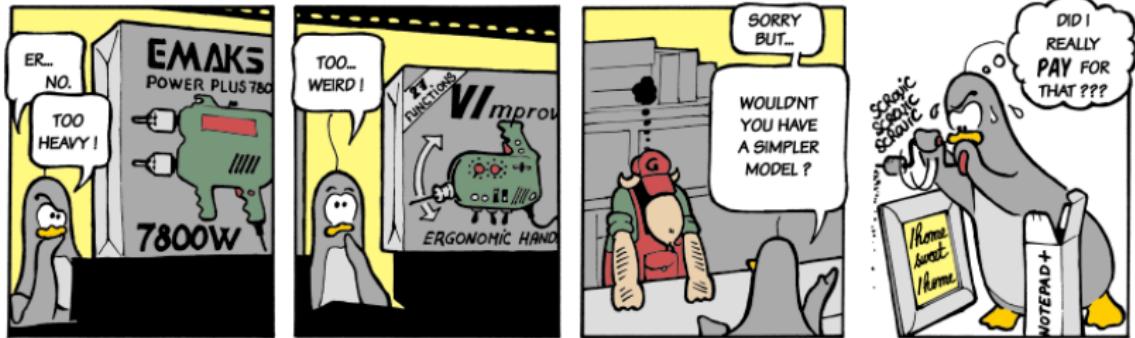
- 有人说：世界上的程序员分三种，一种使用Emacs，一种使用VIM（VI的改良版），剩余的是其它（完全无视其他编辑器的存在……）
- VIM号称编辑器的神，Emacs号称神的编辑器，用户群都非常广
- 由于区分了模式，导致VI的命令非常简洁，而无模式编辑器比如Emacs，所有的命令都需要加上控制键Ctrl或Alt，所以有个笑话说Emacsers们最希望计算机备一个脚踏板，这样就可以用脚踩Ctrl和Alt键了（编辑器圣战——在VI和Emacs之间有很多口水战，自然也引出非常多的幽默）
- VI继承了ed的理念，VI追求的是快捷——启动程序迅速，编辑文本高效，功能专注
- Emacs追求的是功能的丰富强大以及集成带来的方便，在Emacs里头可以发邮件、上新闻组、听mp3、浏览网页、玩游戏，几乎可以login -> Emacs -> logout了:-)
- VI和Emacs都是程序员的编辑器，相比而言，Emacs更是提供了一种程序员的生活氛围

建议看看VIM作者Bram Moolenaar写的Seven Habits of Effective Text Editing，下载地址：

[http://scc.ustc.edu.cn/\\_upload/article/files/7d/f9/](http://scc.ustc.edu.cn/_upload/article/files/7d/f9/)

[033cd3b84a9d8a16b2b2eb9987e6/W020150417520333865377.pdf](http://scc.ustc.edu.cn/_upload/article/files/7d/f9/033cd3b84a9d8a16b2b2eb9987e6/W020150417520333865377.pdf)

# 有趣图片



- Emacs: 太重量级了
- Vim: 太奇怪了
- notepad+: 太累了



# 文件比较：vimdiff

vim -d多个文件或 vimdiff 多个文件，可以对多个文件进行比较编辑

```
# LAPACK, simplest use vasp.5.lib/lapack
LAPACK= ..../vasp.5.lib/lapack_double.o

# use the mkl Intel lapack
#LAPACK= /opt/intel/composerxe/mkl/1

#-----#
LIB = -L../vasp.5.lib -ldm\y \
      ..../vasp.5.lib/lapack_double.o $(BLAS)

# options for linking, nothing is required
LINK =

#-----#
# fft libraries:
# VASP.5.2 can use fftw.3.1.X (http://fftw.org)
# since this version is faster on P4 machines
#-----#
#FFT3D = fft3dfurth.o fft3dlib.o

Makefile          145,1           39% <kefile.linux_ifc_P4 131,1           37%
```

在差异行处，利用对应命令（如do、dp）可以消除文件对应部分的差异。



① 高性能计算、超级计算、并行计算

② MPI并行编程

③ OpenMP并行编程

④ GPGPU并行编程

⑤ 并行效率

⑥ 辅助软件工具

⑦ 数值函数库

⑧ 数据处理、计算可视化

⑨ 作业调度系统

⑩ 杂七杂八

⑪ 资料书籍

⑫ 联系信息

李会民（中国科大超算中心）

并行计算及计算可视化



# 资料书籍

中国科大超算中心主页资料手册: <http://scc.ustc.edu.cn/389/list.htm>



中国科学院大学 大学超级计算中心  
Supercomputing Center of UCAS

帮助科学发展 建造人才摇篮

首页 系统平台 新闻公告 业界动态 培训信息 业务服务 成果展示 运行监控 用户申请 资料手册 联系方式

常见问题手册

常见使用问题

- 常见使用问题
- 基于Google Authenticator二级密码密钥SSHA登录用户操作指南

培训讲座

- 2019年11月Omega项目资料
- 2019年3月GPU资源
- 2018年11月Omega培训
- 2018年10月NVIDIA培训
- 2017年11月Omega培训海报

支持海量数据处理的高IO性能计算模拟系统

- 支持海量数据处理的高IO性能计算模拟系统用户使用指南

曙光TC4000百万亿次超级计算机系统

- 曙光TC4000百万亿次超级计算机系统用户使用指南
- 曙光TC4000百万亿次超级计算机系统用户使用指南-部署篇
- 曙光TC4000百万亿次超级计算机系统新系统测试
- Intel 2013.1.14编译器、调试器、MKL及MPI安装资料
- PGI 2016.9编译器资料

LSF作业调度系统

- LSF作业调度系统(8.3)
- LSF作业调度系统(6.0)

计算软件

- VASP并行效率测试
- OpenFOAM 5.0 安装说明
- WIEN2k14.2编译安装
- VASP 5.4.1+VAST编译安装
- 超算平台上MATLAB使用简介

程序语言

- IDL科学计算可视化基础
- CUDA 程序设计
- Fortran
- OpenMP
- MPI

操作系统

- Linux系统
- IBM AIX系统
- HP-UX系统

more



# 计算实训平台

中国科大计算实训平台: <http://training.ustc.edu.cn/>

The screenshot shows a web-based development environment for the Chinese University of Science and Technology (USTC). The title bar indicates the user is logged in as 'Administrator' with a session ID of '102小时 59分钟'. The main area displays a code editor for 'RL\_net.py' under the 'lesson\_18' directory. The code implements a Deep Q-Network (DQN) architecture for the Five-in-a-Row (AlphaZero) game. It includes three convolutional layers (conv1, conv2, conv3) followed by a fully connected layer (FC\_V1). The code uses TensorFlow operations like tf.placeholder, tf.transpose, and tf.layers.conv2d. A terminal window below the code editor shows the command 'python lesson\_18/player0.py' and its execution output. On the right side, there is a file tree showing the project structure, including files like 'RL\_net.py', 'VS.py', 'completed', 'gomoku.py', 'player0.py', 'playemp.py', 'playerp.py', and 'policyTF.model2'. The bottom of the screen features a standard Windows-style navigation bar.

```
# 17 构建价值函数网络
# 输入:
self.input_states = tf.placeholder(
    tf.float32, shape=[None, 4, board_height, board_width])
self.input_state = tf.transpose(self.input_states, [0, 2, 3, 1])
# 卷积层 conv 32@3x3 -> 64@3x3 -> 128@3x3 -> 401X1
self.conv1 = tf.layers.conv2d(inputs=self.input_state,
    filters=32, kernel_size=[3, 3],
    padding="same", data_format="channels_last",
    activation=tf.nn.relu)
self.conv2 = tf.layers.conv2d(inputs=self.conv1, filters=64,
    kernel_size=[3, 3], padding="same",
    data_format="channels.last",
    activation=tf.nn.relu)
self.conv3 = tf.layers.conv2d(inputs=self.conv2, filters=128,
    kernel_size=[3, 3], padding="same",
    data format="channels last".
from player0 import M_Player
File "lesson_18/player0.py", line 44
def update(self, leaf_value):
    ^
IndentationError: expected an indented block
admin@c613ba5f64af:~$
```



# 作业

## 设计一个MPI程序

- 各进程随机生成一个 $10 \times 10$ 二维单精度浮点数组，且各进程中的数组对应元素值的不全一样
  - 调用集合通信MPI\_Reduce函数实现对数组中对应元素求最大值的归约操作
    - 结果是一个 $10 \times 10$ 数组，比如其(1,3)元素存储的是各进程自己的 $10 \times 10$ 数组中对应(1,3)元素最大的
    - 不是各进程求自己的单个数组中最大的
  - 利用点对点通讯实现上述归约操作并与之进行验证

结果类似：

The results of MPI\_Reduce and MPI\_Send+MPI\_Recv are equal.



# 编译及运行

编译、测试可使用<https://training.ustc.edu.cn/>中的《MPI编译环境的使用》

The screenshot shows a web-based MPI compilation environment. On the left, there's a sidebar with '步骤二: Intel MPI简介' and '步骤三: 编译命令'. The main area has a code editor with MPI C code:

```
1 user mpi
2 implicit none
3 real*8 send_data(1000000),recv_data(1000000)
4 integer ierr,mynode,nnodes,status(1000)
5 integer i,j,k
6
7 send_data=0.0D0
8 call mpi_init(ierr)
9 call mpi_comm_rank(MPI_COMM_WORLD, mynode, ierr)
10 call mpi_finalize(ierr)
```

Below the code editor is a terminal window showing the output of a command:

```
[P-3200200] [~]# mpif90 -o mpirun -o mpirun.f90
[P-3200200] [~]# mpiccc -o mpirun -o mpirun.c
[P-3200200] [~]# ./mpirun -n 2 ./mpirun
1 -> 0 -> 1
0 -> 1 -> 0
1
1 -> 0 -> 1
0 -> 1 -> 0
2
```

On the right is a file browser showing a directory structure under '我的文件' (My Files):

- P-3200200
- Desktop
- all.f90
- a.out
- mpiring.f
- lsfadmin
- memdata
- memt
- mpf
- proc
- root
- run
- srv
- sys
- tmp
- usr
- var
- export
- software
- home
- 3200200
- mpirun
- mpirun
- boot

- 上载文件：类似右边选择/export/home/3200200301（各人的不一样）点右键，选择上载文件，上载自己的MPI程序等文件
- 中间黑色终端中切换目录：`cd /export/home/3200200301`（各人的不一样）
- 编译：
  - Fortran: `mpif90 -o prog-mpi` `prog-mpi.f90` （文件名改成自己的）
  - C: `mpicc -o prog-mpi` `prog-mpi.c` （文件名改成自己的）
- 运行：`mpirun -n 4 ./prog-mpi`



# 联系信息

李会民:

- 办公室: 科大东区新科研楼A座网络信息中心二楼204室
- 办公电话: 0551-63600316
- 电子信箱: [hmli@ustc.edu.cn](mailto:hmli@ustc.edu.cn)
- 个人主页: <http://hmli.ustc.edu.cn>
- 中国科大超算中心主页: <http://scc.ustc.edu.cn>