线程绑定相关代码：



## 1.线程映射来优化机器学习算法

### Serpa M D S, Krause A M, Cruz E H M D, et al. Optimizing Machine Learning Algorithms on Multi-core and Many-core Architectures using Thread and Data Mapping[C]// Euromicro International Conference on Parallel, Distributed and Network-Based Processing. 2018.

线程间数据共享，因此存在线程片内和片间的迁移问题。一方面，通信密集的线程间在片上距离越近通信延迟越低；另一方面，带宽共享，故线程争用带宽，因此通信多的线程片上放置距离应该更远，以避免争用互连的带宽。NUMA架构的系统中，数据存储的距离取决于其存储的节点和管理线程的物理CPU.

用线程和数据的放置算法，优化机器学习算法。

数据映射对存储访问性能的影响：需要知道每个线程对每个页的访存次数。使用exclusivity level矩阵，排他性级别高代表访问私有数据比共享数据多。将具有高排他性的页面映射到最多访问他们的NUMA节点线程中。具有高排他性的页面更容易从共享感知的数据映射方式中获益，因为远程访存会减少很多。同时，若一个页面具有低排他性，远程访存水平相当。

共享模式下的线程映射：共享数据的访问次数以及共享的线程是哪个。需调查应用线程如何共享数据。共享模式矩阵sharing matrices，pin工具收集，监测了所有访存情况。All-to-all模式可以映射优化而不规律的模式不能映射优化。

## 2.基于共享缓存和缓存一致性的动态线程映射

### Cruz E H M, Diener M, Alves M A Z, et al. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols[J]. Journal of Parallel & Distributed Computing, 2014, 74(3):2215-2228.

***引用7.1，7.2，8***

线程之间的通信性能因内存结构的不同而不同。在并行应用程序运行中，通信情况不好检测。在程序执行时需要以一个低开销找到合适的线程映射和迁移方法。通过监测缓存一致性，提出结合了通信检测机制以及动态映射机制的启发式方法。

1）背景：通信意味着数据在核心之间移动。考虑不同线程之间的通信量，将通信频繁的线程映射到邻近的核心之中。多核架构系统中，一些缓存行在多个核中有数据复制，需要协议保持一致性。经过有效映射后，缓存行中的失效核复制就会减少，访存得到优化。缓存一致性协议，我们考虑的一种是使用消息传递机制，监测通信情况需要monitor消息发送的起始和终止位置。需要动态。对同一缓存行的访问代表了牵涉线程之间的通信。我们的监测机制使用一致性协议中的失效消息invalidation message估算通信量。

2）通信监测方法：

共享缓存行的一次访问代表一次线程间通信。读操作虽然执行在共享缓存行上，但是读操作不可能决定有哪些cache共享了相应的缓存行：因为在多数协议实现中，高级别的内存层只保留有限的共享信息，且共享缓存行的读操作不会生成访问目录。相比之下写操作时，一致性协议要求cache给具有相应缓存行的所有缓存发送失效命令。因此可以把每个失效信息当做通信事件。

通常在最低一级的私有cache中确定哪个核心访问的数据变成了失效，为了保存通信量，需要为每个LPCL缓存提供一个向量。每接收到一个失效信息，LPCL中的cache通信向量在执行写操作的相应位置自增。自增操作可以并行执行，同时需要指令让操作系统读取、清除通信向量。为了映射，需要知道哪些线程在通信，操作系统知道每个核心中哪个线程在执行，因此可以在内存中为每个线程都保留一个通信向量的复制。如果存储层次中没有私有缓存，也可以在共享缓存级的监测机制实现。

多线程核心中，需要一个方法识别具体哪个虚拟核心访问了缓存行，执行通信监测时需要给每个虚拟核心一个通信向量。在非LPCL级的cache上做通信监测时，需要额外的硬件：每个虚拟核在每个缓存行设置一个访问位。例如在LLC级缓存中，设置一个访问位，无论何时有写操作或者失效信息到达，都可以根据对应的访问位上的值进行自增操作。在一些给予目录的协议中，联系在LLC上的目录包含了一个访问位，追踪高级别cache中具有相同缓存行的信息。

通信检测机制不用并行的API，不需要随着应用程序或运行环境的改变而更新。我们的通信检测处于缓存行的粒度，有时对相同缓存行不同偏移的内存访问也会算进通信量中，称为空间通信错误。

3）线程映射：  
 需要提供一个检测通信模式是否变化的机制，再需要计算出新的映射。整个过程分为：aging算法，通信过滤器，mapping算法，对称映射过滤器。先对通信矩阵应用aging算法检测通信模式的变化；再开发一个通信过滤器，确定通信行为是否有效变化且是否需要重新计算映射。由于存储结构的对称性，不同的线程映射要被等同考虑。为避免不必要的映射，开发一个对称映射过滤器。开始时获取硬件通信矩阵，每次执行映射算法时再清除它，将噪声降到最小。

Aging算法：通过这个公式，将之前一次aging算法得到的矩阵值与这次硬件监测到的矩阵值作一定比例的线性加和，得到这轮通信矩阵的元素值。公式中AF取值为-0.1~+0.1.

通信过滤器：确定通信模式是同构的还是异构的，计算方差与均值的商，得到的结果高于阈值H-threshold，则为异构的即通信模式改变。将线程通信多的分成一个子线程组，每次新的通信矩阵生成时，执行算法为每个线程构建一个伙伴向量，比较之前与新的伙伴向量，不同的线程数越多，就达到了ID-threshold，意味着需要调用新的映射。但是低的阈值设置会有更高的精确度，高的阈值设置会有更小的额外开销，所以需要在程序执行过程中动态平衡。

映射算法：带权完全图的最大边匹配问题。两两配对寻找出通信量总和最大的配对方式。若只有两个核心共享一个cache，直接进行映射，让通信高的一对儿线程共享一个cache。若超过两个核共享一个cache或者有不止一种缓存架构，需要进行多次配对再映射的递归过程。若共享缓存的核数不是2的倍数，在图上构建顶点匹配的时候只找出x组通信最大的线程对儿，共2x个线程（2x<n<2(x+1））. 接着转化为之前的映射模式。

对称映射过滤器：根据系统的存储架构，画出一个树形结构。找出属于同一核心中的线程，他们之间映射方式的改变是对称的，所以不需要调整。这样计算每个核心之间的距离，从而估算出映射方式改变后，与原先的映射方式有多少不用，若大于零，允许进行重新映射，若等于零，不映射。

## 3.陈志文. 多核系统并发哈希表研究[D]. 湖南大学, 2018.

线程与处理器内核的关联：

NUMA架构中，使用操作系统的调度程序将应用线程分配给处理器内核，由调度程序考虑各方面的因素（系统状态和不同的策略目标）来匹配应用线程和相应的物理核心。将线程从一个核心迁移到另一核心会导致NUMA共享内存架构出现问题，他会断开线程与其本地内存分配之间的关联。也就是说，线程可能启动时在节点N1上分配内存，但当该线程后来迁移到N2的核心上时，之前在N1上保存的数据变成了远程数据，内存访问时间大幅增加。

所以可以通过调整，设定线程和处理器核心的关联（通过API或修改操作系统的数据结构），特定核心或核心集可与应用线程相关联。执行这样的处理器关联可确保内存分配对有需要的线程保持局部性，但也存在缺点：如果本可以使用更好的资源管理方式，处理器关联将会限制调度程序的选择，并产生资源争用的现象，从而对处理器性能造成不利的影响。

## 4.Intra-node Memory Safe GPU Co-Scheduling节点内考虑内存安全问题的GPU联合调度

### Reano C, Silla F, Nikolopoulos D, et al. Intra-node Memory Safe GPU Co-Scheduling[J]. IEEE Transactions on Parallel & Distributed Systems, 2017, PP(99):1-1.

1. GPU的利用率不高往往因为调度器的不适用。作者的schedGPU调度器，优化了GPU节点内的协同调度，以便应用程序考虑内存限制问题下可以安全使用GPU资源。采用一种共享缓存的调度方法。

考虑一个GPU多个CPU的服务器，面临两个挑战：1.原始，调度器调度多个应用程序，会让多个请求GPU资源的作业顺序执行。而且在执行之前进行调度，故无法适应一些运行时的要求。因无法确保随时满足一些内存要求（是否有足够的内存空间处理下面的应用程序），故调度器顺序执行。2.负载调度器一般为同一个GPU调度多个应用程序，但这样会导致潜在的作业终止问题（如果GPU内存不够，就会返回内存溢出错误）。因此传统没有安全的作业控制。

SchedGPU四个点：大多数调度是节点间粗粒度，这个是节点内细粒度；传统在执行之前调度，这个在运行时以毫秒级时间规模调度；这个考虑了内存的需求问题；大多数调度器是集中式的，这个是分布式的，在节点内进行，并应用到大规模节点上。

2.实现方法：C-S模型和共享内存模型。都避免内存的溢出，但有不同架构（C-S是在服务器端集中式控制的，shedmemory是分布式管理的）。

2.1 C-S：应用程序向守护进程请求资源，服务器控制应用是否能执行，或者阻塞，或者返回error. CUDA应用显式调用客户端library中的函数，预分配内存。客户端转发UNIX域socket给服务器建立连接。服务器端通过一个NVIDIA管理库（NVML）控制全局内存，并使用NVML访问物理设备。

2.2 Shared memory：上述方法的缺点是集中式服务器会出现非预期的终止，导致整体框架失败。因此提出交替的分布式方法，使客户端也对全局GPU内存负责。共享内存的架构通过Bost Interprocess Library管理。

通过四种方式保持方法的鲁棒性：1.checkpoint每个应用在结束时保存一份共享内存结构的备份，这样就在每个应用程序结束时构成了一个检查点，以便于故障恢复。2.综合的检查和恢复，新的客户端应用开始，库检查共享内存结构是否崩溃，崩溃了从备份恢复，不能恢复就重新初始化该结构。3.完整性检查，若某用户程序由于阻塞或等待而终止，检查分享给其进程的内存是否仍存活，若不存活，释放之前的内存等待新用户。4.止损，应用程序终止了，共享内存结构没阻塞，且该框架可被其他用户透明地使用。

为避免非一致性问题，同步的内存访问需要一定结构来保护。两种方法：其一是使用进程间互斥锁，确保只有一个客户端更新共享缓存，另外设置一个condition，用于收发内存释放的消息。但是这种方法有abandonment的限制，如果持有锁的进程终结了，其他进程会陷入无限等待。可以使用lock-free锁，但是只能用于进程内通信，不适用于schedGPU拥有进程间通信。

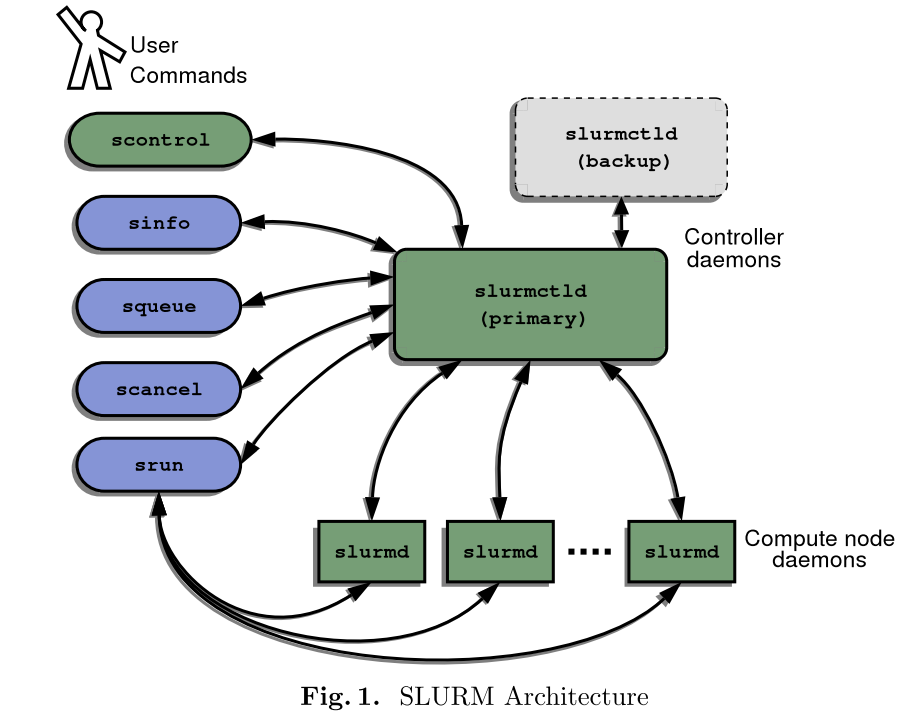
故第二种使用文件锁file locks，可被用户安全使用。但这种不能使用condition，因此靠用户协议使用系统信号来传递进程间通信的消息。系统信号收发内存释放的消息。

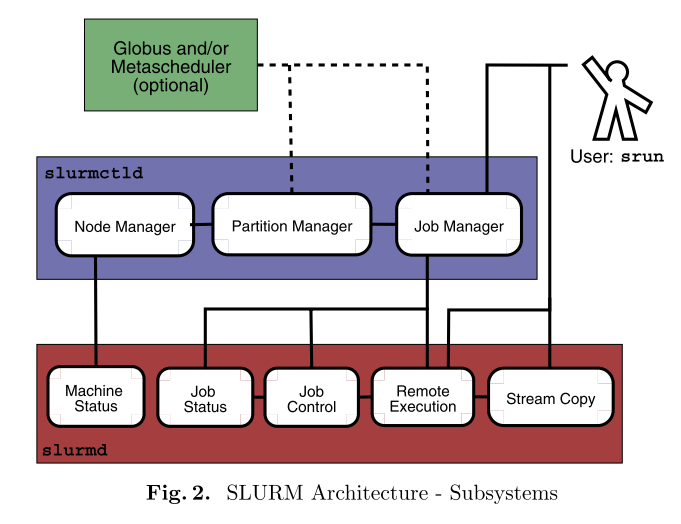
## 5.SLURM: Simple Linux Utility for Resource Management

### Yoo A B, Jette M A, Grondona M. SLURM: Simple Linux Utility for Resource Management[J]. Lecture Notes in Computer Science, 2002, 2862(2862):44-60.

集群资源管理系统Simple Linux Utility Resource Management. 一个简单的集群管理系统，可扩展到上千核，灵活、可容错、可通过很少量的工作移植到其他不同大小或架构的集群上。SLURM提供了一个简单、鲁棒、高度可扩展的作业执行环境。

作为集群资源管理器，SLURM有三个关键功能。首先，它在一段时间内为用户分配对资源的独占和/或非独占访问权限，以便他们可以执行工作。其次，它提供了一个框架，用于启动，执行和监视已分配节点集的工作。最后，它通过管理待处理工作的队列来仲裁对资源的冲突请求。用户和系统管理员使用简单命令与SLURM交互。





守护进程slurmd：一个多线程的守护进程，可运行在所有计算节点上。为其他用户启动作业，因此必须以root权限运行。执行五项主要任务：  
 1.机器和作业状态服务：响应控制器对机器和作业状态信息的请求，并将状态改变（例如，slurmd启动）的异步报告发送到控制器。

2.remote execution：按照slurmctld进程和srun/scancel命令的要求，进行启动，监测和清除的工作。启动一个进程可能要执行prolog程序，设置进程限制条件，设定真实有效的用户id，建立环境变量，设置工作目录，分配互联资源，设置核心文件路径，管理金承祖等等。而终结一个进程可能包括终止进程组的所有资源并执行epilog程序。

3.stream copy service流复制服务：允许处理远程业务的stderr, stdout, stdin. 作业输入可通过文件，srun进程或者/dev/null来重定向。作业输出可保存至本地文件或发送会srun命令。所有作业输出可以忽略stdout或stderr的位置而被本地化缓存，避免了局部任务阻塞。

4.job control作业控制：允许与远端执行环境进行异步交互。给任意本地管理进程集传送信号和显式的作业终止请求。

守护进程Slurmctld：作为控制器，保存很多slurm的状态信息。多线程，对不同数据结构的独立的读写锁。Slurmctld启动时，读取SLURM配置文件（也可读取由之前的slurmctld执行生成的checkpoint文件中的额外状态信息）。全部的控制器状态信息被定期写入磁盘并立即将增量更改入磁盘以容错。Slurmctld可在主机或待机模式下运行，不需要root权限。主要三个组件：

1. Node Manager节点管理：监测集群中每个节点的状态。定期地轮询slurmd的状态，异步接受状态更改通知。在节点可用前，保证节点处于规定的配置。
2. Partition Manager分区管理：将节点分成不重叠的集合，称为分区。每个分区可关联不同的作业限制和访问控制。分区管理根据分区和节点的状态和布局，为作业分配节点。发起作业请求，用scontrol改变节点和分区配置。
3. 作业管理：接受用户请求，将作业放入一个优先级排序的队列。每当作业有状态改变的时候，作业管理被唤醒。每个分区具有最高优先级的作业都是被尽可能地分配资源。一旦任何分区出现分配失败，更低优先级的作业就不会被该分区考虑了。完成一个调度周期后，作业管理调度线程进入休眠态。一旦作业被分配资源，作业管理器传送必要的状态信息到该资源节点，允许其开始执行。一个作业使用的所有节点都结束了工作，作业管理器进行清除工作并开始新的循环。

命令行工具：用户使用SLURM的接口，使用户可以远程执行和作业控制，还可以动态更改系统的配置。这些命令均使用API以直接适用于复杂的应用。

Scancel：取消作业/作业步骤，给作业节点上任意进程发送信号。

Scontrol：执行特权命令，许多只能由特权用户执行。

Sinfo：显示分区和节点信息的总结，提供过滤和输出格式选项。

Squeue：显示运行/等待作业队列，提供各种过滤、排序和输出格式选项。

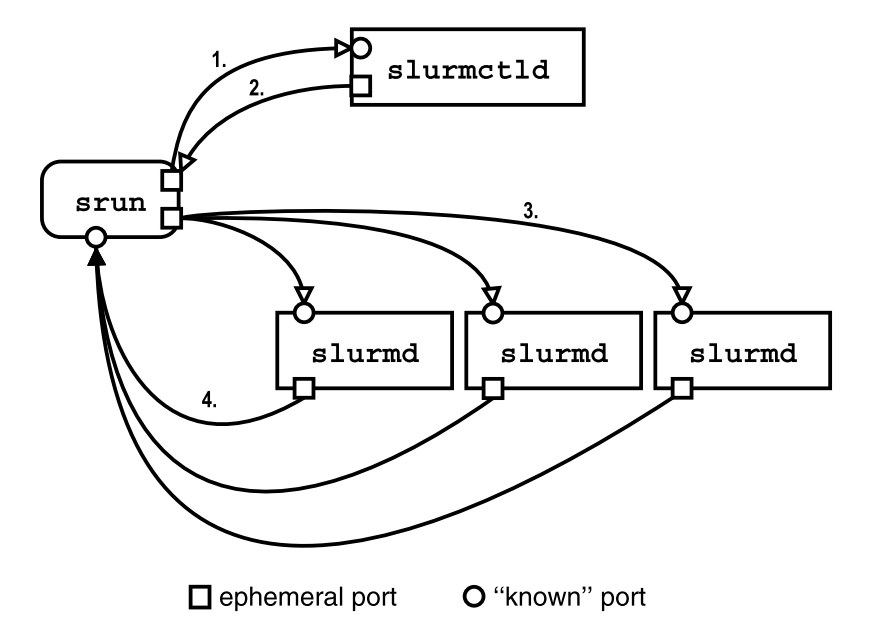
Srun：分配资源，将作业提交到SLURM队列，启动并行任务/作业步骤。每组并行执行任务都有一个相关的srun，启动它；如果srun持续存在，则管理它。可以提交作业进行批处理，这种情况下，作业提交后srun终止。还可以提交作业进行交互式执行，srun继续运行，管理正在运行的作业：srun与远程slurmd协商连接以启动作业，获取stdout和stderr，转发stdin，响应来自用户的信号。用户还可以指示srun分配一组资源并产生对这些资源具有访问权的shell。

Plug-ins：SLURM通用插件机制。SLURM插件是动态链接的代码对象，运行时由SLURM库显式加载。插件提供API，该API连接到诸如身份验证，互连，任务调度等任务上。定义一个函数集。例如，身份验证插件必须具有以下功能：slurm auth activate创建凭证，slurm auth verify验证凭据以批准或拒绝身份，slurm auth get uid获取用户ID，还必须定义使用的数据结构，插件类型，插件版本号。

通信层：使用插件机制可轻松使用其他通信层，SLURM配置文件允许识别每个节点的主机名以及用于通信的名称。SLURM可在不用socket和以太网情况下管理1000个节点，但还需要具有更高可扩展性的机制如STORM.

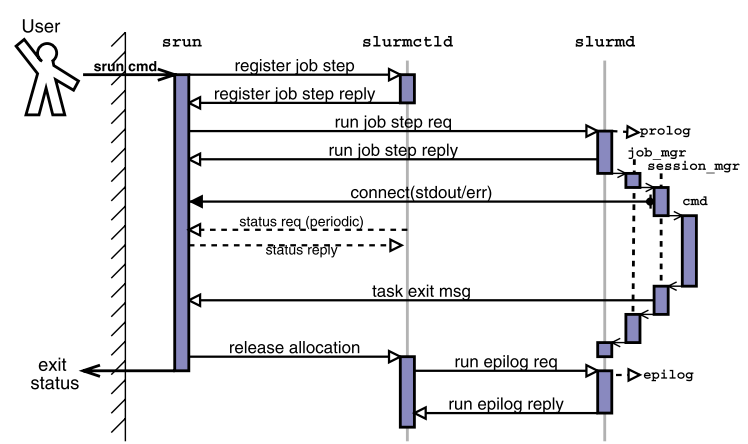
安全：集群的任何用户都可以提交并行作业来执行和取消自己的作业。任何用户都可以查看SLURM配置和状态信息。只有特权用户才能修改SLURM配置，取消任何作业或执行其他受限活动。SLURM中的特权用户包括用户root和SlurmUser. 如果其他人需要修改SLURM配置的权限，可使用set-uid程序向特定用户授予特定权限。

作业启动（初始化）：用户在SLURM下有三种模式，最简单的是interactive交互模式，stdout,stderr在用户终端实时显示，stdin和信号从终端透明地传递给远端作业。Batch模式，作业排队直到满足请求的资源，SLURM以submitting用户运行作业。Allocate模式，作业分配给用户，用户通过脚本或由srun生成的子shell手动运行作业步骤。

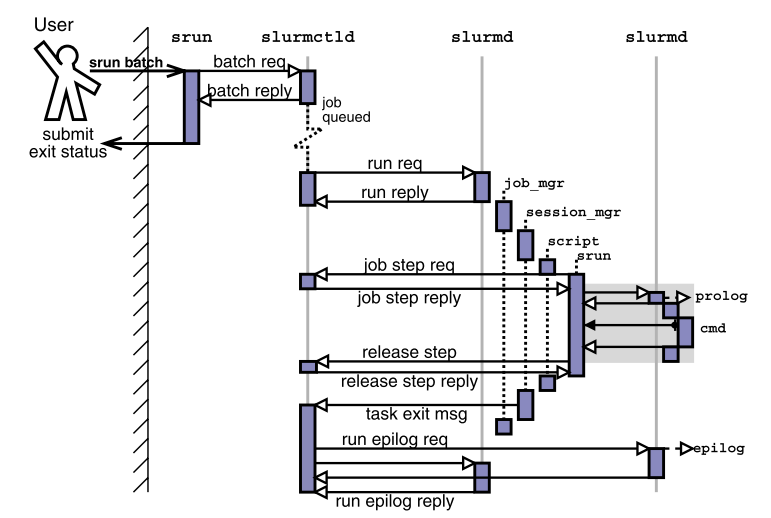


Srun给slurmctld发送资源请求；slurmcld处理响应，返回分配的节点列表和作业凭证；srun为job step中的每个任务打开一个监听端口，然后向slurmd发送一个运行job step的请求；最后slurmd初始化job step并连接回srun做stdout/err.

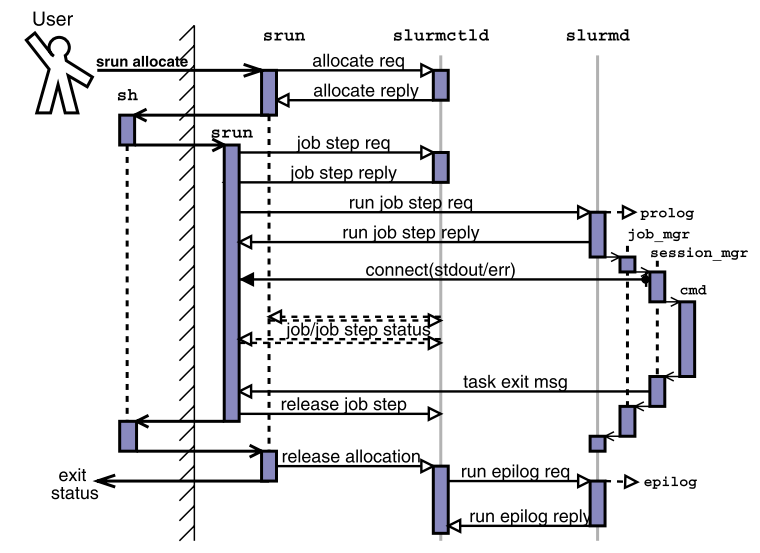
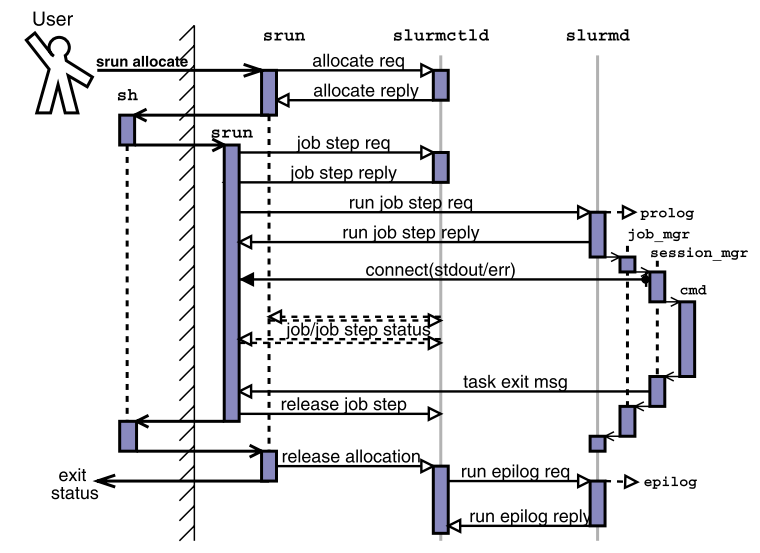
Interactive模式初始化：用户唤醒srun的交互模式。执行完命令行操作后，srun向slurmctld发送请求资源和作业步骤的初始化，srun等待答复，但是不一定会立即收到，srun可能会阻塞直到资源可用。确认资源可用，slurmctld回复作业凭证、节点列表。Srun给每个slurmd发送节点信息请求初始化作业，获得回复。每个slurmd唤醒一个作业线程处理请求，也因此为每个请求的作业欢迎一个任务线程。线程连回srun打开的端口，执行stdout,stderr，srun保持主机和端口的连接。一旦stdout/stderr成功连接，任务线程去初始化用户在节点上的执行、环境、当前工作目录和互连资源。用户进程退出，任务线程记录退出状态，给srun发送退出消息。所有本地进程终结，作业线程退出，srun等待所有任务退出或者在第一个任务退出后定期清除剩余进程。全部任务结束，srun给slurmctld发送释放节点消息，最后退出。Slurmctld收到通知后，向每个相关节点的slurmd上发送请求，运行epilog程序，运行成功，过程结束。



Batch模式初始化：批处理作业提交后，srun给slurmctld发送批处理请求，包含输入输出位置、当前工作目录、环境和节点数量。Slurmctld为请求按优先级排队。一旦资源可用且作业优先级足够高，slurmctld为作业分配资源并连接分配的第一个节点。这里作业要么是srun的另一个调用，要么是具有多个srun调用的作业脚本。远端节点的slurmd响应运行请求，初始化作业线程、任务线程和用户脚本。在脚本内执行的srun检测它能访问到的分配和初始化。Job step完成，脚本中的srun通知slurmctld并终结，脚本继续执行并初始化更多的作业。作业脚本完成后，其余过程跟交互模式一样。



Allocate分配模式初始化：用户希望自己分配作业并在该分配下以交互方式运行job step. srun向slurmctld发送分配请求，如果资源可用，响应分配的节点列表、作业ID等，srun进程为用户终端生成一个shell，可以访问分配，等待shell在作业完成时退出。



## 6.多核平台上线程映射策略的性能评估和分析：英特尔架构上SPEC OMP应用的案例研究

### Mazouz A, Touati S A A, Barthou D. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of SPEC OMP applications on intel architectures[C]// International Conference on High PERFORMANCE Computing and Simulation. IEEE, 2011:273 - 279.

线程亲和度加速了程序执行时间，对不同的线程映射/放置策略做了完整的性能实验研究。四个不同线程映射的应用程序和五个基于共享缓存的敏感应用程序，三种众核机器：工作站、服务器、高性能机。在ccNUMA架构的机器上，不论使用何种测试策略，使用一些线程亲和度的修改，性能都好于操作系统的线程放置，意味着目前LINUX系统调度策略即便考虑到了负载平衡和使用率，就性能而言不是最好。在小型的Core2和Nehalem机器，线程绑定的优势没那么大，但性能稳定性更好。

**1.实验设置：**

测试机有：（1）8核桌面工作站Core2，单节点SMP机器，两个socket，每个处理器4个核，一组核分别共享一个数据和指令L2Cache，每个核有一个独立L1Cache. （2）8核服务器Nehalem，一个SMP单节点，两个片，每片四核，每四核共享一个L3Cache，每个核一个私有L2Cache和分离的L1指令数据Cache. （3）96核高性能集群，ccNUMA架构，四个计算节点，每个节点4个socket，每个socket6个核，一个socket共享3MBL2Cache和16MBL3Cache，每个核私有L1Cache.

使用不同版本gcc, icc编译器共三种测试，icc效果最好。

实验期间每台机器只有一个实验用户工作，重置所有shell环境变量，使用不同线程数执行（4,6,8on core2/nealem; 16,96 on ccNUMA），用均值和中位数作为指标，对两大类线程映射规则测试（应用敏感，应用不敏感）。

**2.映射规则：**

应用敏感型：线程到核映射独立于不同的应用程序。（1）不使用亲和度affinity，即让OS决定线程的放置，这种策略允许实际执行时的线程迁移。（2）随机策略，每次重复的运行对应于一种随机的线程亲和度，单次执行时线程不迁移。（3）compact icc，icc编译器使用这种方式使OpenMP线程尽可能靠近，这种方式对在线程之间有高数据重用的应用程序有益，因为可以很容易访问socket中的共享缓存。（4）scatter icc，是OpenMP线程尽可能跨越整个socket，这种方式对线程内部有很高数据局部性的应用程序有益，因为他们可以从大的私有缓存中获取数据，避免与其他线程共享。

应用不敏感型：这类线程放置策略依赖于每个应用程序的亲和度图。图，V是线程，是每对线程的增益函数。例如，如果我们依赖于线程之间的数据重用计算亲和度，增益函数代表线程之间对相同内存cache行的共同访问数。当图构建好之后，亲和度图要被分解成不相连的子集，称作分区。分区之间没交集且并集是所有线程，每个子集包含的线程将被放置到共享相同cache层的相邻核中。图中所有子集增益函数值之和要最大。这个分解问题是NP完全问题，所以要使用启发式的算法。当k=2即只需要找到线程对儿时，问题可以转化为一个图中的最大权值匹配问题。所以不同的线程放置策略对应于不同的图k分区问题：（1）GP，使用k分区方法，只把线程放置到socket上，即每个socket上的线程一组。（2）LP compact，在应用多项式方法得到线程对儿后，以紧凑方式分配线程对。（3）LP scatter，分散方式分配线程对。（4）LPGP，第一步计算最优的线程对后，第二步执行图的k分区方法，自下而上的层次策略，线程先被配对然后放置到共享L2/L3cache的核上，接着线程对根据其亲和性被放置到不同的socket中。（5）GPLP，自上而下的分层策略。

**3.实验结果分析**

（1）core2和nehalem机器上，改变线程亲和度导致边缘性的加速或减速，故OS自己的线程放置调度或许不是个差的方法。但是性能差异很大。（2）在ccNUMA机器上，观察到所有线程放置策略都有加速，证明使用Linux线程调度器不是好的选择。（3）使用全部96个线程时，加速比是最大的，原因是操作系统的线程调度为负载平衡提供了更高的优先级，即倾向于调度线程到空闲的可用内核，尽管增加了内存访问成本，但可以保持所有核心处于忙碌状态。所以若出现一些内核远程访问内存节点，数据无法保证，则会观察到很差的性能。（4）从性能角度应用敏感和应用不敏感策略差别不大，故与简单容易的icc compact或scatter方法相比，其他方法变得不划算。一种可能原因是程序执行期间自定义方法不允许线程迁移。这点有利于研究基于线程迁移的其他亲和度解决方案。

## 7.1.Maximum Matching and a Polyhedron With O,1-Vertices

### Edmonds J. Maximum matching and a polyhedron with 0[J]. Journal of Research of the National Bureau of Standards, 1965, 69(1 and 2):125-130.

## 7.2.Multilevelk-way Partitioning Scheme for Irregular Graphs

### Karypis G, Kumar V. Multilevelk-way Partitioning Scheme for Irregular Graphs[J]. Siam Review, 1998, 41(2):35.

重点是将匹配问题与连续线性不等式理论联系起来。证明了一个定理：凸集双重随机矩阵（convex set of doubly stochastic matrices）的极值点是其置换矩阵。该定理基于二分图的匹配问题。不论组合学是否被完全理解，可以使用线性编程的对偶性推导出的对伴随一类组合结构的凸多面体的描述，独立于相关算法的特定性质。该论文显示，应用线性编程解决组合学问题时，相关不等式的数量不重要，组合结构更为重要。

图划分，多级划分问题，图的谱划分方法，并行稀疏矩阵方法。

图的多级k划分方法MLRB，基本思想是：图先粗化成顶点较少的小图，最小化的图直接划分成k块，再在划分后的图基础上细化，恢复成原图。使用直接法划分成k块（而不是递归二分法连续计算）的优点是：完整的图只需要粗化一次，时间复杂度由O(|E| logk)降到O(|E|). 但直接计算出好的k划分比好的二分法更困难，这也是为何多数k划分需要通过递归二分法。在多级设计的上下文中，只需要对最粗略的图进行一个粗略的划分，但在细化阶段需要更复杂的考虑。

图的k划分问题一般表述为给定长度n（图的节点数）的向量P，对图的每个顶点v，P[v]是1到k的实数，表示顶点所在的块号。给定划分P，由于划分而造成的减少边的数目称作edge-cut.

**1.粗化：**

（1）图的边塌陷的思想。图G0是原始图，在大多数粗化模型中，图Gi的一组顶点的集合合并成图Gi+1的一个顶点。为使更粗略的图的还原度更高，图Gi+1的顶点v的权值是图Gi合成顶点集的点的权值之和。为了保留图的顶点间的连接关系，Gi+1点v与其他点的连接边是Gi合成点集的所有点外连边的整合。这样做保留了图的特性：1.粗略图划分的edge-cut与精细图划分的edge-cut相同；2.如果粗略图划分平衡，则细化后的精细图也划分平衡。

匹配过程matching指图是一群边的集合，这些边两两没有共同顶点。因此，构造图Gi的下一级粗略图Gi+1就是寻找图Gi的匹配matching，将顶点变为多节点，未匹配的顶点直接拷贝到Gi+1中，匹配量尽可能大。若在任两条边不竞争同一节点的条件下，不能再加入任何其他边，则称为图的最大匹配maximal matching. 若匹配的计算方法不同，最大匹配也可能不同。

最粗略的图Gm仅具有少量顶点，或连续地较粗糙的图比前一阶段仅减少很少量的点时，粗化阶段结束。例如选择粗略图的顶点数为ck，确定一个常数c可以使粗略图划分的每一块大小相同。

（2）三种选择最大匹配的方法：

RM(Random matching)：以随机顺序访问图的顶点。访问到顶点u时，若该点还没被匹配，随机选择他的一个相邻顶点v作为匹配对象，将(u, v)进行记录；若没有这样的点v，则顶点u保留未匹配状态。算法时间复杂度O(|E|).

HEM(Heavy Edge Matching)：RM方法很好，但目标如果是最小化edge-cut，就不能用随机法。给定粗略程度一定的图Gi，更粗略的图Gi+1由匹配Mi生成。定义A是边的集合，W(A)是A中所有边的权值之和。HEM的总体思想是，因此总的粗略图的边权值经匹配过程而减少，因此选择一个匹配Mi使具有最大边权值。由于更粗略的图具有更小的边权值，也因此具有更小的edge-cut. 计算重边匹配使用一种类似于随机匹配的随机计算方法，还是以随机顺序访问顶点，但在匹配时，选择有效的相邻顶点中权值最大的。注意这种方法不能保证整个图获得最大的权值，但在实验中获得了很好的效果。该方法时间复杂度是O(|E|).

一种改进的HEM方法：降低了粗略图的平均度average degree. 以随机顺序访问图中顶点，对顶点v，H是相邻顶点中边权值最大的点集（顶点可以有多个），对每个H中的点u，是点u到点v其他相邻顶点的权值和。在改进的HEM方法中，点v选择H中值最大的点u进行匹配。时间复杂度O(|E|).

**2.初始化划分段：**

为图Gm=(Vm, Em)计算一个k划分。由于在粗化阶段设置了粗略图边和顶点权值到精细图的映射，图Gm具有充足的信息，以实现划分平衡和edge-cut最小。

一种初始化划分的方法是持续对图进行粗化直至只有k个顶点，这k个点作为最原始的划分分区。但有问题：1.多数情况下，图已经缩减得很小了，使得接下来继续粗化代价很大；2.即使能粗化到只有k个点，每个点上的权值可能也大不相同，使得划分不平衡。

可以使用多级递归二分算法。

**3.细化阶段：**

划分后的图Gm一步步细化为Gm-1,…, G1. 由于图Gi+1的每个点v包含的是图Gi的不同的顶点子集。由图Gi+1的分区Pi+1变为图Gi的分区Pi，通过简单地将点v的精细顶点子集中所有顶点放入点v的分区中。注意即使粗一层的图的划分有局部最小问题，较细一层的图呈现的划分也不会局部最小，因为更精细的图会有更多的自由度来优化划分，减少edge-cut. 故仍然可能通过启发式局部细化来改进划分。

3.1

很多局部细化算法基于KL分区算法及其变形的思想。KL算法通过递增地交换一对儿分区中的节点，来减少edge-cut，直到划分方法达到局部最小。

一种KL算法的变形是计算出将某个顶点v移到其他分区的edge-cut增益，为两个分区的这些顶点，根据他们的增益，设置两个优先级队列。开始时未上锁，每个顶点都可以移动到另外的分区。算法迭代地从两个优先级队列之一，选择增益最大的节点v，将其移动到另外一个分区去。当一个顶点移动后，该顶点上锁，并更新相邻顶点的增益。每个节点都移动后，算法记录此时的edge-cut. 不允许上锁节点移动是因为防止抖动。当没有别的顶点未上锁时，算法停止。检索整个移动过程，寻找到edge-cut规模最小的那个时间节点，将那个节点之后的顶点移动操作退回。以上过程完成的结果就作为下一阶段的初始划分。

KL算法非常有效，因为在连续细化的每一级粗略图中，初始化分都是一个好的划分。

3.2

上述KL算法是图划分为两块时的方法，然而图的k划分要复杂的多，某个顶点可以被移动到其他任何分区中去，需组合地增加优化空间。

其中一种改进是设置k(k-1)个优先级队列，每个队列对应一种移动方式。算法的每步，从这些队列中找到获得增益最高的移动，执行能够保留或者提升划分平衡性的移动。移动过后，更新所有队列。这种方法时间复杂度达到O(k\*m)，远远高于2划分，所以只适合k较小的情况。所以多级递归多分算法与多级递归二分算法执行时间相近，尽管前者粗化阶段耗时很少。

3.3

一种简化k路划分KL的算法，使得时间复杂度独立于分区数量。

首先是一些定义和概念。对于图Gi=(Vi, Ei)，和它的划分向量Pi. 定义顶点的邻居概念neighborhood N(v)是指与点v相邻的顶点所在的分区的集合。如果v是一个分区内部顶点，N(v)是空集。N(v)的值可以很大。所以在细化过程中，点v可以移动到任一N(v)的分区中。对图中每个顶点v，都计算其移动到邻居分区的增益。另外，对某一分区b，计算，表示顶点v到邻居分区b中所有相邻点的权值之和，称为外度external degree of v to partition b. 也计算ID[v]，表示点v到与其处在同一分区的相邻顶点的权值之和，称为内度internal degree of v. 基于此，点v移动到分区b去的增益为.

还得考虑分区平衡问题，只有满足以下平衡条件，才进行顶点移动。Wi是个k元向量，Wi[a]是图Gi中分区a的权值。设定阈值为： 点v从分区a移动到分区b的条件是：. 条件1确保顶点的移动不会导致权值超过阈值，常数C的不同区分出细化时图平衡容忍度的不同。条件2对负载平衡不那么重要，但保证了一个分区内的顶点不会太少。

3.4

**贪心细化算法GR**. KL算法的前瞻性lookahead非常重要，很可能一簇顶点跨越分区边界移动后，提升了局部edge-cut，但是当整个顶点簇跨越边界移动后，全局edge-cut降低了。但是在多级机制的上下文中，lookahead不那么重要，因为在粗化阶段这些点簇全部粗化为了一个点。因此，在较粗级别的图中，移动一个顶点意味着原始图移动了一簇顶点。

如果顶点只会在获得良好增益的情况下移动，那么优先级队列就没用了。因此一些KL算法的变形只简单地随机访问处于分区边界的顶点，然后在多级环境中，执行那些看起来会获得很大增益的移动。GR算法基于此组成一系列迭代过程，每次迭代，所有点检查看自己是否能被移动。

GR的过程为，设点v在分区a中，如果点v的N(v)是空集，则不移动点v；否则，找到N(v)的子集N1(v)，里面包含的分区b是移动v后不影响图的平衡的可移动分区。现在，如果满足以下条件之一，v可以被移动至其中一个分区b. 1).是其中最大的；2).. 由上可看出，GR算法移动点v的目的是在不影响平衡条件下减少edge-cut. 如果移动无法改变edge-cut，那么可选择增强平衡性但不增加edge-cut的移动。移动点v后，算法更新之前计算的关于v内度和外度。实验说明，GR算法一般只需要迭代4-8次

3.5

**全局KL细化算法（GKLR）**。GR缺乏跳出局部最优的能力。GKLR加入一些限制的爬山功能，并且类似于KL，具有优先级队列。算法使用一个全局队列根据顶点的增益存储他们。

扫描所有顶点，将外度大于等于内度的点加入到队列中。例如v是这样的点，b是点v外度最大的一个分区，将v插入队列时其增益值为. 接着算法选择队列中增益最大的点，然后在满足平衡条件下将其移动到分区b. 注意移动有可能导致edge-cut增加，因为不论增益是正是负，算法都进行移动，直到移动x次后，全局edge-cut不减少了后停止。

注意GKLR每一步移动定点的选择与传统方法略有不同。GKLR在所有可能的移动中选择一个增益最大的，但是取决于分区的权值，这个移动可能不会发生，取而代之的是顶点v移动到另一个分区a，但增益相对较小。但是这时候队列中如果存在另一个点u，他的最大移动增益大于v移动到分区a的增益，且点u的移动能被准许，那么算法会选择移动点u而不是点v. 因此在每步，算法不是简单选择增益最高的点。因为单优先级队列只包含外度大于等于内度的点，因此其爬山能力不如普遍的KL算法。

Karypis G, Kumar V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs[M]. Society for Industrial and Applied Mathematics, 1998.

## 8. Enhancing operating system support for multicore processors by using hardware performance monitoring

### Azimi R, Tam D K, Soares L, et al. Enhancing operating system support for multicore processors by using hardware performance monitoring[J]. Acm Sigops Operating Systems Review, 2009, 43(2):56-65.

***引用9***

**论文中的第三部分实验研究：提升局部性**

**一、**多核系统中的通信时延因两个核的物理位置关系的不同而差别很大。操作系统的调度没有考虑到非均匀共享的延迟。但是如果操作系统能检测线程的共享模式并因此调度线程，那么线程就可以主要在相同的片上进行通信。而自动检测共享模式不容易，需要特殊的硬件支持或侵入式检测。作者提出方法，**使用POWER5 PMU在线确定线程共享的时延。**

1.首先使用一些硬件监测方式（stall breakdown model）

（副：可使用PMU，作者使用PMU中的两个特征：stall breakdown, data addressing sampling.

1.通常很难直接解释/翻译性能计数器的值，因为统计的都是具有许多交互组件的复杂系统的低级别微架构事件。由于要考虑很多要素，因此测量每个独立事件对处理器性能的影响是艰巨的non-trival. 例如，L2缓存有1%未命中率的重要性取决于每个L2缓存未命中直接（导致执行指令的延迟）或间接（使其他执行流水线结构饱和或其他）产生的后果penalty. 这些后果在无序处理器中更难测量，因为为了隐藏内存延迟，许多指令同步执行。  
 作者在无序流水线处理器中建立了一个简化CPI breakdown模型，处理器的周期由completion周期和stall周期组成，completion周期是至少有一条指令在执行流水线的末尾，因此该指令可以retire；而stall周期则是任何指令都不在执行流水线末尾，因此无一可以retire. 作者利用PMU特征，在IBM POWER处理器上，stall周期主要由cache miss，分支预测错误，TLB miss等特定原因引起。根据实验观察，TLB地址转换数据结构的问题是程序特定段stall周期产生的原因。

Stall breakdown模型可能会不精确，但专注于stall周期足以识别出处理器潜在的性能瓶颈。实验表明，使操作系统focus减缓引起stall的重要问题能实质上在应用级提升性能。）

来决定跨处理器的通信在引起stall周期方面是否是关键的。可从实验结果得出stall中大约有7%由远端缓存访问失效引起。

2.第二阶段，使用data address sampling特征，追踪线程的共享模式。

（副：data address sampling.

该技术可应用于很多现代微架构处理器。使用特定寄存器data address register(DAR)进行data address sampling. DAR由PMU自动更新，每当访存指令（load, store）中有确定的用户事件（数据cache miss, TLB miss）发生时更新。

硬件data address sampling对分析多核性能很有用，因为核间的交互尤其通过数据间的共享。若PMU能识别资源中提取的样本数据地址，就可以表征多线程应用在多线程多核上运行的具体的数据共享和大体的竞争模式。

进一步，如果能获得应用程序cache访问模式的准确信息，就能测量片上共享缓存的竞争强度和它的模式。）

3.第三阶段，线程根据数据共享度高的放置在同一集群上，进行映射。

4.最后操作系统调度线程迁移。

**二、在线检测共享模式**：使用数据地址抽样data address sampling，检测远端cache失效的地址，为每个线程构建数据结构shMap. ShMap是8位向量，系统中共256个计数器以防空间超载，每个计数器对应一个虚拟地址空间，其值反映了检测到的某个内存区域region对某个线程的共享量。Region的大小取决于缓存一致性的单元大小。但是区域的数量通常远大于shMap向量中条目的数量，因此使用一个简单哈希函数将区域映射到条目中，条目仅在对应的线程访问远端区域的缓存时自加，而在同一片上共享数据的线程不更改shMap中的值。

作者想依赖于PMU提供的地址。然而在大多数PMU，这一特征不直接可用，故只能使用间接方法，以合理的精确度捕获远端cache访问的地址。连续记录L1 cache失效的地址，由于data address sample只在性能计数器溢出时进行，因此最近一次L1数据cache失效很可能是请求了远端cache访问并造成相应的性能计数器溢出。为了处理大量数据，仅记录N次远程访问中的一次。N值自定义，取决于1.PMU测得远程访问的频率；2.运行开销。

若shMap设置较小，哈希冲突率会很高。使用空间采样spatial sampling降低冲突率，消除混叠效应。不监测完整的虚拟地址空间，公平地随机选择一些区域作为一个小样本集，但区域必须至少有一个远程访问才有资格选中。

以上方法的基本思想是：一旦检测到缓存行子集中的高级别共享，就能清楚证明共享的实际强度很高，足以证明集群的合理性。

## 9. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors

### Tam D, Azimi R, Stumm M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors[C]// ACM Sigops/eurosys European Conference on Computer Systems. ACM, 2007:47-58.

基于共享模式感知的调度

在大规模系统上，应用SMP-CMP-SMT架构，数据共享开销差异大。作者用PMU检测共享模式，粒度细到cache行，并且开销小。

## 10 [多核编程模型运行时环境的自适应性研究](http://d.wanfangdata.com.cn/Periodical/xajtdxxb201106023)

### 曹仰杰, 杨海兵, 钱德沛,等. 多核编程模型运行时环境的自适应性研究[J]. 西安交通大学学报, 2011, 45(6):130-134.

有OpenMP, TBB和Cilk等众多并行编程模型，但仍存在弊端：

1. 资源竞争加剧。这些多核编程模型运行时环境都包含针对内部线程的负载均衡策略，但调度策略仅限于单一应用程序内部工作线程的负载均衡。很大程度上忽视了系统整体计算资源的协调分配，导致系统整体吞吐率降低。实验显示，作者研究时多核编程模型的运行时环境造成了较大的资源竞争，应用程序运行时间增加。
2. 系统性能优化困难。支持多核编程模型的各类运行时环境相互独立，彼此之间不能通信。多种模型混合使用时，系统整体性能难以优化，且操作系统缺乏针对不同多核编程模型运行时环境的优化方法。
3. 可扩展性较差。运行时环境需要在程序运行时指定系统包含的核数。显式指定核数的方式制约系统的可扩展性和可移植性。

提出了ACSM自适应协同调度模型：

去除运行时环境对需要指定处理器核数的限制，模型根据负载情况自动分配核资源。引入动态反馈机制，以及在不同运行时环境中进行交互控制。模型将系统任务划分为受控任务和非受控任务。对非受控任务，进行统一的简单控制管理策略，初始时分配好，运行时不管，任务结束后资源回收。对受控任务，设计如下控制算法。

自适应work-stealing算法（A-WS），在cilk运行时环境中验证。初始A-WS为每个工作者创建对应的工作线程。运行时周期性地动态指导资源分配。各worker分四个状态：工作working，窃取stealing，特殊窃取mugging，休眠sleeping. 窃取状态时，本地工作栈/队列为空，依据相应窃取协议窃取其他工作线程的负载。协议是：每个周期末，A-WS根据工作线程的平均利用率，采用启发式策略，反馈下一周期资源需求量。

## 11.Enhancing OpenMP and Its Implementation for Programming Multicore Systems

### Chapman B M, Huang L. Enhancing OpenMP and Its Implementation for Programming Multicore Systems[C]// Parallel Computing: Architectures, Algorithms & Applications, Parco, Forschungszentrum Jülich & Rwth Aachen University, Germany, September. DBLP, 2008:3-18.

## 12. [An OpenMP runtime profiler/configuration tool for dynamic optimization of the number of threads](http://xueshu.baidu.com/s?wd=paperuri%3A%280a92d38722260c3d186dbef744c66c6a%29&filter=sc_long_sign&tn=SE_xueshusource_2kduw22v&sc_vurl=http%3A%2F%2Fieeexplore.ieee.org%2Fdocument%2F7522136%2F&ie=utf-8&sc_us=8046453152040699716)

### Dancheva T, Gusev M, Zdravevski V, et al. An OpenMP runtime profiler/configuration tool for dynamic optimization of the number of threads[C]// International Convention on Information and Communication Technology, Electronics and Microelectronics. IEEE, 2016:192-197.

设计openmp运行时工具去预测最优线程数。主要为证明两个假设：1.存在一个或多个数据集中的属性的子集，他们构成的随机森林生成的预测，在任何测试用例中，其性能不会比通过分配一定数量的核给openmp的num\_threads变量所得的性能低。2.存在一个或多个数据集中的属性的子集，不管线程数增多还是减少，其随机森林预测工具的性能相比默认的openmp模型都有提升。

首先使用开源跨平台库SIGAR API获取各种属性参数（num\_threads, cpu使用情况，ram使用情况及页错误，虚存使用情况，进程运行情况）构成数据集。Pthreads实现线程级并行，选取benchmark，指定线程数范围，默认运行50次。利用产生的运行时间性能找到最佳线程数并把相应的运行时状态写入磁盘。Openmp在执行并行块儿之前需要调用omp\_set\_num\_threads，然后唤醒封装函数，使用随机森林（random forest）预测线程数并作为参数返回。

实验尽可能模拟多的运行时状态，方便由数据集构建决策树，配置参数使得性能更好。通过有原始数据集重复生成额外数据集进行数据集扩展。多次重复实验，随机森林对数据集进行一般化处理，因为每种属性的考虑都是独立的。当第一次调用set\_num\_threads时，创建森林。初始化配置之后，使用森林预测，结果是收集到的运行时间值。从总数据集中抽出四个相互独立，总体不同的子集，共同构成用于预测的数据集。在两种环境中执行（无压力环境：没有额外负载；有压力环境：仿真时使用stress程序包POSIX，是系统同时多线程执行这些程序）。

## 13.太湖之光和天河二号缓存架构和通信问题

1. 太湖之光：一个芯片4个核心组，每个核心组8\*8=64个计算核心，整个芯片共260个核心。40个机柜\*4超级节点\*32节点板\*4节点卡\*2节点\*1processor（4cluster+4\*8GB内存）；

一个CPE Cluster由一个MPE+64个计算核心+1MC内存控制器组成，共享32KB L1指令+数据cache，和256KB L2cache. 每个CPE有一个16KB L1指令cache和64KB高速暂存。

MPE&CPE基于RISC指令集，64-bit，SIMD单指令多数据流。

互连网络分三层：连接超级节点的中心交换网络+超级节点内部256节点全互连网络+底层资源共享网络

## 14.线程+数据映射方法kMAF提高内存亲和性

### Diener M, Cruz E, Alves M, et al. Kernel-Based Thread and Data Mapping for Improved Memory Affinity[J]. IEEE Transactions on Parallel & Distributed Systems, 2016, 27(9):2653-2666.

1. 背景：带有多级cache和内存控制器内存层次结构的复杂和多样性，出现了NUMA架构。在NUMA架构下，有两种提升内存亲和度的方法：1）映射线程：共享数据映射给共享cache的核，优化cache访问和通信开销。2）映射访存最多的内存页到内存控制器，不引入额外负载。映射时若考虑局部性和负载平衡两方面问题，会获得更好的程序性能。
2. 数据映射背景：

a. OS-Based mapping. 分为first-touch, interleave, next-touch. First-touch内存页分配给第一个访问该页的NUMA节点，以后不再迁page，这是Linux默认的策略。有时first-touch会造成overloading。比如当由某一个节点初始化大量数据的时候。

故在这种情况下，更好的是将pages均匀分散到各节点中。Interleave就是这样一种分散式的方法，同样在Linux中可通过numact1工具修改。

Next-touch方法，page将被迁移到下次访问它的节点中，但由于一个页可能被多个节点访问，故可能造成过量迁移。

许多OS级优化方法着手于细化data mapping. 而最新的Linux系统版本又引入了平衡技术，使用并行程序的page faults检测访存，执行抽样的next-touch策略。每当有page fault，迁移page，但这种balancing技术不保留访存历史，故可能造成大量迁移，balancing也不执行线程映射。

另一方法在AMD架构CPU上，检测访存性能并保留历史记录，限制不必要的迁移，此外还支持page的复制。还有研究动态地将pages从过载的节点迁移到负载低的节点。

b. compile-based and runtime-based mapping

在用户空间（编译器，运行时环境）执行映射，只知道单应用程序的信息，但如果系统同时执行多个应用程序，他们的映射决策就会冲突，这就限制了方法的适应性。一些方法需要源代码注释annotation，并仅限具体的并行编程库。Some认为远端NUMA节点的访存对最优性能不好，所以引入一些OpenMP指导语句执行数据的分布，通过手动的方式选择最优的分布策略，但对不同的硬件架构性能会不同。Some使用编译器插桩代码预测并行循环的访存性能，并使用预测结果在循环之前迁移pages. 或者编译器插入instrumentation代码，识别共享memory的访问模式，用以指导迁移。

有的Library能使NUMA感知内存分配情况（libnuma，MAi），有了这些库，数据结构可根据编程者的specification（特定的NUMA节点，交叉策略）进行分配。但这种技术把映射的压力给到编程者，且可能需要对每种不同的架构重写代码。MAi库的一种改进是可选择地使用源代码预处理器确定数组的数据映射策略。Some使用访存追踪技术，这可能对获得最大增益有好处，但不一定有普遍性。

c.Hardware-based mapping

从硬件计数器中静态生成机制指导映射。受限于具体的架构。Some用PMU追踪内存地址，应用在Intel Itanium机器上。Profile机制受限于高开销，只能在每个应用开始时使用，故不能动态控制。Some使用的方法受限于具有TLB（Translation Lookaside Buffers）的机器，仅涵盖一少部分系统。

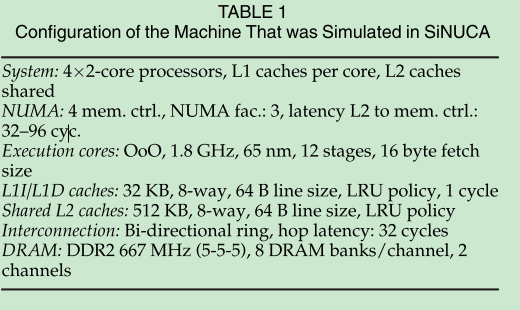
2) 线程映射背景：

在shared-memory架构下，通常检测不同线程的通信情况实现映射，一般是implicit. Some测量了不同映射方式下每个周期的指令数，执行取得最高IPC值的映射。当线程数过多时，干脆执行1000次随机映射，选择效果最好的。

一些机制使用硬件计数器实现非直接静态通信收集。Some使用计数器提供由远端cache决定的内存地址的请求。但是这种对通信模式的检测不完整。还有机制使用并行程序的页错误检测通信，通过追踪产生page faults的线程以及出现错误的pages，依据于此执行映射。

1. 线程和数据映射的影响（实验分析）

借助微架构模拟器microarchitecture simulator，使用SiNUCA（精确到cycle的性能验证模拟器，支持Intel x86架构）。模拟机的参数为：



4个processor均有一个内存控制器，共组成4个节点。通过模拟器中的McPAT建模工具估算能耗。选择OpenMP基准测试程序SP benchmark，因为它对数据和线程映射具有高感知度。使用Pin工具生成访存的追踪。设置了四种映射配置configuration，映射大概分为local和remote两种模式，分别组合线程+数据映射和他们两种模式。线程映射的local&remote很好理解，根据线程对共享数据访问规定他们的远近；数据映射的local指每个内存页都被映射到访问他们最多的节点上，同理remote内存页映射到访问最少的节点上。所以四种映射配置表示为TrDr, TrDl, TlDr, TlDl.

实验以最差的TrDr作为基准，TrDl性能提升31%，好于TlDr的15%。原因是local的数据映射优化了对主存数据的访问，减少了线程到他们要访问的数据的距离，因为这些访问通常都是片间的，属于高延迟操作。而local的线程映射提高了共享cache的使用，减少L2cache空间的争用，也提高了cache命中率和失效率，但由于片上访问cache的操作算是低延迟操作，因此性能提升没那么大。

TlDl的性能提升有62%，高于TrDl+TlDr之和，由于线程映射，它减少了cache失效率和cache-to-cache间的数据传输；也由于数据映射减少了主存平均延迟和互连竞争。而local线程映射还优化了经local数据映射后的pages.

除了性能上的提升，还有能效优化。首先是减少了程序运行时间，静态能耗（leakage）按比例减少。其次更少的cache miss和互连traffic也会减少动态能耗。对各部分的能效优化中，L1cache失效的减少，互连竞争的减少和主存访问次数的减少是动态能耗降低的主要因素，因为大部分访问都变成了L2cache的访问。整体的能耗减少幅度略小于运行时间减少幅度。

以上的TlDl属于Oracle-based映射，但他没考虑检测和执行映射的困难性，Kmaf执行的是有效的在线映射。

1. 方法：automatic的线程+数据映射。使用page faults作为评价标准，引入extra faults增加精确性。分析之后计算合适的映射过程。
2. 整体概述：a.检测访存性能。追踪page fault，并在执行过程中引入额外的低延迟page fault，提升检测精确度。b.存储和性能分析。把产生page faults的内存地址和线程ID以页和子页的粒度存入两个table中，分别用于数据和线程映射。Kmaf还为线程映射提供一个共享矩阵。c.执行线程映射。设计映射算法阶段性评估共享矩阵。d.执行数据映射。每次页错误时分析下访问page的性能，决定是否需要迁移页。
3. 检测访存性能：由于访存一般直接由硬件执行，不通知OS，因此收集访存信息具有挑战性。多数系统中，只有出现page fault，才通知OS，因此我们观察page faults.

每当页表中有虚拟地址到物理地址的映射信息缺失，或信息失效，CPU为OS产生一个page fault的信号，包含缺失页的虚拟地址信息。之后由OS确定造成fault的线程ID. 错误地址是完整的地址，因此检测时可以应用不同的粒度（cache line大小，page大小）

使用Extra page fault提升检测精度。正常情况下，每个page只发生一次page fault，OS通过给页表中插入一个translation解决页错误后，对该页的顺序访问不会再产生更多faults. 但在多线程并行初次访问一个页时，一个单独页可能发生multiple page faults. 因此统计时实际上是少算了的。故为了克服这种限制，提升检测的精确度，在执行过程中，kMAF在执行时插入extra page faults，这些页错误可插入在所有存储数据内存区域（不包含代码段）。周期性遍历并行程序所在页表，更新其中的条目使得下次访存产生页错误。多数系统中，页表都包含一个校验位present指示条目是否有效，更新操作清除该校验位即可（因为这样的改动不损失页表中已有的信息，因此不会产生太大开销的操作）。kMAF可根据程序内存消耗情况，衡量extra page faults数量。

1. 存储并分析检测到的性能情况：

经过检测，收集到产生页错误的完整的物理地址和线程ID. 以页为粒度处理数据映射，以更细的粒度处理线程映射。

数据映射的页粒度。存储的是页分配给NUMA节点的信息，保存一个vector向量NV，其中的元素存储着节点n中页p的页错误次数。产生的每个页错误，对其中的物理地址做如下处理：将物理地址右移页大小对数位，得到的index用于访问存储NV向量的NUMA表，产生页错误的线程所在的节点其对应的自增。

线程映射采用cache line粒度。需要另外构建一个共享检测机制。将内存分裂成可配置大小的内存块，默认256B，利用一个哈希表sharing table实现共享检测，表内存储最近访问不同内存块的线程ID. . Hash函数使用linux系统默认的。对每个内存块，保存一个block vector（BV），向量含有最近访问过block的线程列表。列表有两部分MRU+LRU，当线程访问了某个块，它的ID插入进MRU中，把之前访问该块的一个线程的ID移进LRU中，并把LRU中最早的一个线程ID移除。将前后访问这个同一内存块的两个线程表示在共享矩阵中的元素自增。

1. 线程映射：

共享缓存系统中，线程数不多的情况下，线程映射问题是：给定通信性能和硬件架构的条件下，寻找线程到处理单元的映射方式，最大化局部性locality. 之前大多数方法需要基于图的性能描述。

作者选择EagerMap算法，很适合在线映射机制，具有快速执行时间，并实现了与Tree-Match和Scoth相同的映射效果。属于贪心映射策略，具有高稳定性，使得结果上看减少了很多不必要的线程迁移。

把EagerMap放在kMAF的内核中，硬件拓扑由内核中的有效信息直接生成，不需要借助辅助工具。周期性运行kMAF检测和EagerMap映射，算法输出每个线程可执行在的计算单元，之后迁移线程。

1. 数据映射：

与线程映射的周期性评估不同，数据映射在每次页错误时都要评估，原因有二：并行应用有数百万个pages，同时对每个页做数据映射不切实际；在页错误时分析访存性能并执行大量页面迁移，不需要额外的上下文切换。

初次对页的访问，kMAF算法使用first-touch方法，之后的顺序访问中，采用如下方法：计算错误页面的独占性exlusivity，从NUMA向量（描述了一个页面是否被/最多次地被一个单节点访问）中计算，exclusivity用于选择应用locality-based映射还是balance-based映射，最后根据映射算法结果迁移页面。

页面的独占性表示为，按公式计算，N是计算节点总数，p是页面，是页面p的向量，是页面p对节点i的值，是页面p向量中的最大值（反映对于页p来说产生错误最多的节点）。如果页面有较高exclusivity值，则它较多地被某个单节点访问到，故越适合使用优化局部性locality-based的映射。设置两个阈值和。独占性值高于，使用局部性映射；低于，使用平衡性映射；处于二者之间，不做映射。

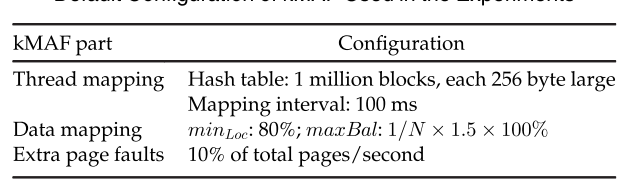
局部性映射策略：先设置一个过滤器，减小页面迁移的数量，只保留产生页错误较多的情况。公式为，max和max2分别表示页p的NV向量中最大和第二大的值。迁移时肯定是将页面p移动到值最大的节点上。即选择的迁移节点为.

平衡性映射策略：在访存模式稳定后，kMAF通过只执行平衡性映射限制迁移的总数。选择进行平衡性映射的公式为：，如果条件满足，算法执行Inerleave数据映射（背景部分介绍过）。平衡性映射中，页p要迁移到的节点按如下公式计算：.（页面p的地址右移页大小的对数位，再对节点总数取模）。这种映射优点有二：映射的节点是根据页面的地址直接计算的，不需要存储或遍历程序的全局状态；减少了不必要的页面迁移。

为了实施映射，kMAF需要知道页面的虚拟地址和目标节点。先检查页面是否已在节点上，若不在，使用Linux内核的unmap\_and\_move()函数执行实际的迁移。不需要保存虚拟地址，因为页错误时虚拟地址也是有效的。

1. kMAF实现：

将它作为一个linux内核module实现，不需要reboot重新安装。创建一个内核线程实现extra page faults，其他内核线程执行线程映射。其他参数配置如下：



1. 开销分析：

存储开销。需要为NUMA表，sharing表和共享矩阵分配内存。NUMA表大小为N\*P\*1byte（节点数\*页面数\*1字节）。共享表（哈希表）存储共享性能，每个block放2个线程ID，每个ID大小2字节，如果默认要1million个block，至少需要约4MB. 共享矩阵每个元素占4字节，矩阵需要线程数T^2个元素，若应用有1024线程，共大约需要4MB.

时间开销。主要是引入extra page faults，解决页错误，计算映射，迁移。计算映射的时间复杂度是，引入额外页错误随内存利用情况线性增加。解决页错误的复杂度恒定。

## 15 pin工具使用学习笔记

### Pin官网示例

*1.BBL——basic blocks. 把pin对程序的一次instrument称为一个trace，pin将一个完整的trace划分为多个基本块BBL，一个BBL是一段单入口单出口的指令序列。*

*Routine，在pintool工具中，当新代码要生成时，由pin调用pintool寄存器的instrumentation回调routine. Instrumentation回调routine表示instrumentation组件，它会检查要生成的代码，调查代码的静态属性，决定是否以及在哪里调用分析函数。*

*在Image instrumentation模式下，由于Image instrumentation依赖于symbol信息来决定routine的边界，因此必须在PIN\_Init前调用PIN\_InitSymbols.*

*Instrumentation范围的大小关系：Image>section>routine>>instruction.*

*2.对于一个pin的instrumentation程序proccount.cpp为例：代码实现统计程序的procedure的调用次数及信息，和每个procedure中指令的个数，执行后输出文件proccount.out*

*//*

*// This tool counts the number of times a routine is executed and*

*// the number of instructions executed in a routine*

*#include <fstream> #include <iomanip> #include <iostream> #include <string.h> #include "pin.H"*

*ofstream outFile;*

*// Holds instruction count for a single procedure*

*typedef struct RtnCount*

*{*

*string \_name;*

*string \_image;*

*ADDRINT \_address;*

*RTN \_rtn;*

*UINT64 \_rtnCount;*

*UINT64 \_icount;*

*struct RtnCount \* \_next;*

*} RTN\_COUNT;*

*// Linked list of instruction counts for each routine*

*//定义全局变量RtnList*

*RTN\_COUNT \* RtnList = 0;*

*// This function is called before every instruction is executed*

*VOID docount(UINT64 \* counter)*

*{*

*(\*counter)++;*

*}*

*//字符型常量函数，查找给定字符path中最后一次出现字符’/’的位置，若存在，返回该字符下一位置的指针；若不存在，返回path首地址。用于获取image名。*

*const char \* StripPath(const char \* path)*

*{*

*const char \* file = strrchr(path,'/');*

*if (file)*

*return file+1;*

*else*

*return path;*

*}*

*// Pin calls this function every time a new rtn is executed*

*//*

*VOID Routine(RTN rtn, VOID \*v)*

*{*

*// Allocate a counter for this routine*

*//每次调用函数Routine，内存分配一个结构体指针变量rc.*

*RTN\_COUNT \* rc = new RTN\_COUNT;*

*// The RTN goes away when the image is unloaded, so save it now*

*// because we need it in the fini*

*//初始化rc的值*

*rc->\_name =* [*RTN\_Name*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#gaf9d9689a59240ecc29c80d584f268c72)*(rtn);*

*rc->\_image = StripPath(*[*IMG\_Name*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__IMG__BASIC__API.html#ga50e1d947e0949aa33f0d35804968d560)*(*[*SEC\_Img*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__SEC__BASIC__API.html#gac57c906aeabfa47c07300e6a70d70c05)*(*[*RTN\_Sec*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#ga4883c839d424578a9f6983d05e6a9a12)*(rtn))).c\_str());*

*rc->\_address =* [*RTN\_Address*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#ga83a81fae96c9faabe0f1c90a0d7e865f)*(rtn);*

*rc->\_icount = 0;*

*rc->\_rtnCount = 0;*

*// Add to list of routines*

*//采用头插法将rc插入到链表RtnList里*

*rc->\_next = RtnList;*

*RtnList = rc;*

*//打开给定的routine*

[*RTN\_Open*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#gaf8714086f8aebc9feacccc8cd02dc561)*(rtn);*

*// Insert a call at the entry point of a routine to increment the call count 统计routine个数*

[*RTN\_InsertCall*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#ga76bde295a78d1232fd6ff98a5ff011cf)*(rtn,* [*IPOINT\_BEFORE*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INST__ARGS.html#gga707ea08e31f44f4a81e2a7766123bad7a7c7cbebb7a62a40e9f803b1db2e6ce20)*, (AFUNPTR)docount,* [*IARG\_PTR*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INST__ARGS.html#gga089c27ca15e9ff139dd3a3f8a6f8451da7a11763427dff3afb81caab97b02881b)*, &(rc->\_rtnCount), IARG\_END);*

*// For each instruction of the routine*

*for (INS ins =* [*RTN\_InsHead*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#gab1bd91206939b88057664c46ef8eac86)*(rtn);* [*INS\_Valid*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INS__BASIC__API__GEN__IA32.html#ga3a8b61fffa9ae4ad9f899b21ce37397c)*(ins); ins =* [*INS\_Next*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INS__BASIC__API__GEN__IA32.html#ga227ce58a739b1573125c11071ecb48de)*(ins))*

*{*

*// Insert a call to docount to increment the instruction counter for this rtn 统计指令的个数*

[*INS\_InsertCall*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INS__INST__API.html#ga74a956a0acde197043d04f4adcde4626)*(ins,* [*IPOINT\_BEFORE*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INST__ARGS.html#gga707ea08e31f44f4a81e2a7766123bad7a7c7cbebb7a62a40e9f803b1db2e6ce20)*, (AFUNPTR)docount,* [*IARG\_PTR*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__INST__ARGS.html#gga089c27ca15e9ff139dd3a3f8a6f8451da7a11763427dff3afb81caab97b02881b)*, &(rc->\_icount), IARG\_END);*

*}*

[*RTN\_Close*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#ga37272253377eb65c9eb5ff47ec4f1e6b)*(rtn);*

*}*

*// This function is called when the application exits*

*// It prints the name and count for each procedure*

*VOID Fini(INT32 code, VOID \*v)*

*{*

*outFile << setw(23) << "Procedure" << " "*

*<< setw(15) << "Image" << " "*

*<< setw(18) << "Address" << " "*

*<< setw(12) << "Calls" << " "*

*<< setw(12) << "Instructions" << endl;*

*for (RTN\_COUNT \* rc = RtnList; rc; rc = rc->\_next)*

*{*

*if (rc->\_icount > 0)*

*outFile << setw(23) << rc->\_name << " "*

*<< setw(15) << rc->\_image << " "*

*<< setw(18) << hex << rc->\_address << dec <<" "*

*<< setw(12) << rc->\_rtnCount << " "*

*<< setw(12) << rc->\_icount << endl;*

*}*

*}*

*/\* ===================================================================== \*/*

*/\* Print Help Message \*/*

*/\* ===================================================================== \*/*

*INT32 Usage()*

*{*

*cerr << "This Pintool counts the number of times a routine is executed" << endl;*

*cerr << "and the number of instructions executed in a routine" << endl;*

*cerr << endl << KNOB\_BASE::StringKnobSummary() << endl;*

*return -1;*

*}*

*/\* ===================================================================== \*/*

*/\* Main \*/*

*/\* ===================================================================== \*/*

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

*{*

*// Initialize symbol table code, needed for rtn instrumentation*

[*PIN\_InitSymbols*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__PIN__CONTROL.html#ga8cf4aca0b0bdbc7fc0ae965883d8e3c2)*();*

*outFile.open("proccount.out");*

*// Initialize pin*

*if (*[*PIN\_Init*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__PIN__CONTROL.html#ga783d3bd40c3fb2ca51b1f9af31a70c9c)*(argc, argv)) return Usage();*

*// Register Routine to be called to instrument rtn*

[*RTN\_AddInstrumentFunction*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__RTN__BASIC__API.html#gad10b9862dbe2f9bef8f7978492c35d01)*(Routine, 0);*

*// Register Fini to be called when the application exits*

[*PIN\_AddFiniFunction*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__PIN__CONTROL.html#gac167b8d28d255e53a0ac1f6e9edcd3ec)*(Fini, 0);*

*// Start the program, never returns*

[*PIN\_StartProgram*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__PIN__CONTROL.html#gaded401aeb030a76ee3396137b06ad808)*();*

*return 0;*

*}*

*3.TEB线程环境块，存放着进程中所有线程的各种信息。使用函数PIN\_SafeCopy()从原内存区域复制指定数量的字节到目标内存区域（该函数保证调用过程的安全，即使源或目标区域不可访问）。使用PIN­­\_SafeCopy使pintool可以读写应用程序中某些field的值，例如在windows系统中，运行一个tool的分析代码，就由Pin取代了特定TEB区域的内容，tool若直接访问这个区域，就会看到更新后的值，而不是原来的值。推荐在每次tool读写应用程序的内存时，使用这个API函数。*

*4.instrument线程执行的应用程序：使用ThreadStart()和ThreadFini()回调函数，即使这两个函数在有锁环境下执行，仍然可能与其他pin的分析routine产生资源竞争。所以需要额外使用PIN\_GetLock()防止竞争。*

*1）设置系列锁，对输出文件的访问上锁：*

*// lock serializes access to the output file.*

*FILE \* out;*

*PIN\_LOCK pinLock;*

*2）每次当有线程创建时，调用一个routine，利用文件类型指针变量out，在获得写文件锁的时候，给输出文件打印线程ID，随后清除缓存区的out，释放锁。*

*VOID ThreadStart(*[*THREADID*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__PIN__THREAD__API.html#ga645289be59039349ad77ad2fa7b0e2f3) *threadid,* [*CONTEXT*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__CONTEXT__API.html#ga73f8f88949aaecf53a6d23f56399c676) *\*ctxt, INT32 flags, VOID \*v)*

*{*

[*PIN\_GetLock*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__LOCK.html#ga9569245cd781216f8f1a54d3a0962ddf)*(&pinLock, threadid+1);*

*fprintf(out, "thread begin %d\n",threadid);*

*fflush(out);*

[*PIN\_ReleaseLock*](https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__LOCK.html#ga00a837236be573c0c548191e0846df1d)*(&pinLock);*

*}*

*3）同理当线程销毁时，调用routine，向文件out中打印线程ID和代码code.*

*再为线程malloc内存时，调用routine，向文件中打印线程ID和分配的大小size.*

*4）instrumentation过程：定义rtn变量，寻找image中名为malloc的操作，若rtn有效，执行instrumentation插入调用，获取线程malloc信息。在最后执行一个fini的routine.*

### 有关Pin的一篇论文: Building Customized Program Analysis Tools with Dynamic Instrumentation[C]// ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005. Luk C K.

#### 摘要和背景介绍

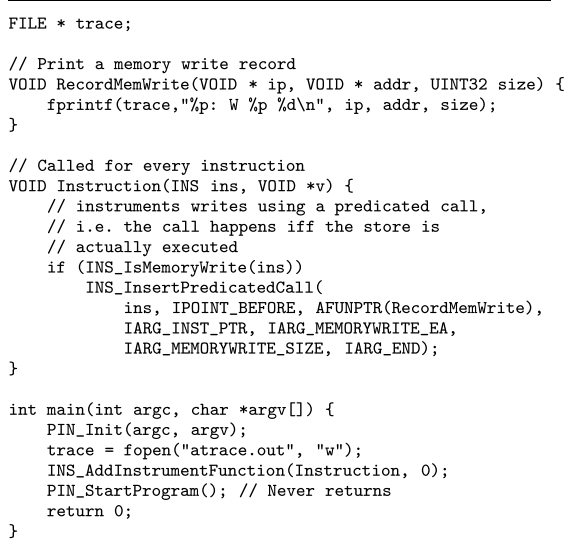
1. Pin提供的剖分工具pintools使用pin提供的丰富的API，以C/C++编写。Pin遵循ATOM模型，允许编写者在指令级分析程序，无需详细了解底层指令集。API被设计尽可能独立于机器架构，使pintool源在不同架构之间兼容，且pintool可以必要时访问体系结构信息。Pin进行剖分过程是透明的transparent. 使用动态编译dynamic compilation检测可执行文件。

2. 可在很多阶段执行instrumentation：在源码中，在编译时，在后续链接时，在运行时。Pin工具是一个在运行时对Linux程序进行二进制instrumentation的软件系统。Pin的设计强调易用性，可移植性，透明性，有效性，鲁棒性。

#### 用pin进行instrumentation

Pin API可观察到进程的所有架构状态（寄存器，内存，控制流）。用户为程序进程添加precedures，用户自己写一些剖分例程，自己决定在哪里调用分析例程，例程的参数是架构状态或常量。Pin还可以让剖分例程覆盖掉程序的寄存器和内存，从而提供改变程序性能的有限能力。

下图例子是用户自己创建的一个pintool，打印了程序运行时所有写内存的地址和大小，称为一个trace. 主函数初始化pin，记录一个叫做instruction的步骤，并告诉pin程序开始执行。动态编译器JIT在每当有一个新的指令进入代码cache时调用instruction步骤，给解码的指令传递一个句柄。若该指令写内存了，pintool就在指令前插入一个调用RecordMemWrite，传递指令指针、内存操作的有效地址、写内存的字节数。INS\_InsertPredicatedCall确保RecordMemWrite仅当内存指令断定为true时被调用。



1）相同的pintool源代码可以适用于所有体系结构，用户无需了解指令捆绑、编址模式等等信息。

2）Pin是基于调用（call-based）的模型，用户通过添加或删除中间语言的语句插入剖分过程。用户无需理解指令集特性和学习中间语言，就能很好地观察到架构状态。

3）插入的剖分代码可能会覆盖掉临时的寄存器和条件语句，但是pin可以有效保存并恢复调用剖分前后的运行状态，因此消减这种副作用且不改变程序的性能。

4）无论指令集是什么，pin都能写有效的、不依赖架构的剖分代码。还可使用结合内联函数、寄存器重分配等优化策略使基于调用的pin剖分过程很高效。

#### 两个pin应用的实例

1.Opcodemix

Opcodemix检测程序执行的操作码的动态组合，可以以一个基本块、一个例程或者一个image作为基本单位。在每个trace基本块之前插入一个计数器。程序终止后遍历所有计数器，从基本快的起始地址中可以确定出该块所属的函数和该块的指令组合。Opcodemix的输出依赖于指令集，但实现是通用的。

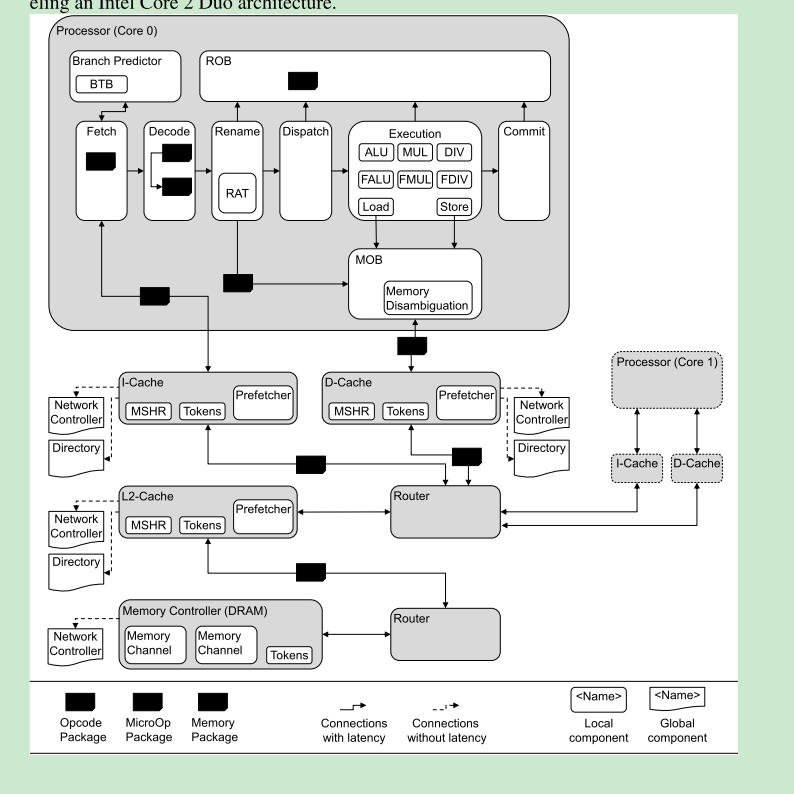
2.Pinpoint

Pinpoints包的目的是将寻找模拟程序的域的繁琐过程自动化，同时验证这些域是否具有代表性，然后为这些域生成trace. 大型商用程序执行过程过于耗时，有大量资源需求，在模拟器上很难执行。用SimPoint解决，使用段分析寻找可用于模拟的代表性程序域；使用pin收集SimPoint的剖分（PinPoints）和指令trace. PinPoints已被用于收集各种程序（单线程+多线程）的指令trace.

## 16.SiNUCA

用C++开发，面向对象，生成模块化组件。

以下列场景为例：用SiNUCA架构（主要组件和互连）对一个英特尔酷睿2（双核）系列架构建模。



### 一、模拟器的组件

1.图中Memory package封装了所有有关内存的操作，Opcode package封装了模拟器进行trace以及trace前后的指令，MicroOp Package封装了Opcode package解码后的微操作。

2.每个内存部件都有一个token，内存间的通信（例如一个cache和内存控制器）都要在通信前向目标包所在的内存单元请求Token，当包传递的过程需要超过一个脉冲的时间时，token的存在可以避免发生死锁。

3.Processor部件负责处理Opcode package的内容，由fetch提取，decode解码，rename重命名，dispatch分派任务，execute执行，commit提交的步骤组成。

4.Cache Memory缓存单元，负责对指令和数据cache建模，每个cache组件实现一个MSHR，维护tag数组，以减少内存使用和trace大小。MSHR(Miss-Status Handling Register)，一个buffer，存放cache失效信息，相当于cache的管理器。若cache命中，访存请求等待被处理并给请求者发送消息，cache失效则复制请求发送给上一级内存组件。

5.预提取器Prefetcher，模拟器将缓存处理的内存包发送给预提取器，方便对以后需要的内存地址做提取和预测。实现多种预提取策略，并且只要cache的MSHR还有空间，就给cache提供生成好的访问请求。（目前，在90年代很多方法已经实现了预提取器）

6.内存控制器，有多个channel组成，channel由多个bank组成。所有访存请求中发生的所有级别的cache miss都会被发送至内存控制器，内存控制器在这些访存请求中做调度。支持NUMA结构。内存信道channel，包含一个或多个内存bank，bank具有自己的读写缓冲buffer.

7.模拟器实现一个NoC（片上多处理器互连网络）的路由router，使用包中的路由信息自动传输包。网络控制器Network Controller在模拟的架构下产生一个通信图，网络中所有要传输的包要包含一个指针指向路由表中的路由信息。路由信息描述经过的中间组件的路径。

8.Directory模块模拟了MOESI缓存一致性协议，对所有内存操作适用。为cache行维护锁lock，控制同步；改变cache行的状态（例如在写操作后或广播失效信息时，更改cache行的状态）；写回以及告知cache命中或失效信息。在读请求时，directory为对应的内存地址创建读锁，此时当前内容只读。写操作时，为cache行创建写锁，其他cache不再能够对相同内存地址的cache行做其他操作。

7、8两部分都是模拟器中的虚拟模块，这部分建模产生查找表lookup table，对他们的访问不产生延迟；但是其他模块都是实际模块，在这些部分中建立连接会产生延迟。

二、

## 17. Yu Z, Xiong W, Eeckhout L, et al. MIA: Metric Importance Analysis for Big Data Workload Characterization[J]. IEEE Transactions on Parallel & Distributed Systems, 2018, PP(99):1371-1384.

### 摘要

使用ensemble learning，提出一个表征工作负载的方法MIA. 实验证明MIA具有可应用性，对三组benchmark进行评测，发现由MIA算法得到的重要指标，对他们进行调整会使程序获得更高的性能提升。

### 背景

大数据应用广泛，跑大数据负载需要复杂的大规模计算集群，和融合了大数据框架的MapReduce/Hadop/Hive等。很需要表征大规模集群的性能，加以优化，但具有挑战，挑战主要源自系统的大规模分布式特性和多层软件栈。

系统通常由数千服务器组成，了解这样的系统，需要层次化表征节点级、处理器级的cache，分支预测，TLB等、系统级的网络互连和存储设备。

另一方面，大数据系统软件不仅包含大量应用级作业，而且由大量多层次的系统软件栈组成（分布式文件系统，虚拟机监视器hypervisor，操作系统，大数据分析框架例如Hadoop）。这些软件的表征也要多级执行（作业，任务…）。此外，大数据系统中软硬件紧密结合，因此需要涉及很多度量。

构建指标与性能之间的关系f很困难。一种是使用机器学习构建f的经验数据模型，但是构建过程开销小，又精确的数据模型很难。很关键的是知道什么指标比较重要，即获取关键指标。这个过程需要深入了解大数据系统的软硬件。

### 作者工作简介

设计方法MIA，使用集成学习，量化大数据系统的指标体现在系统级性能（DPS每秒数据处理）和节点级性能（IPC每周期指令）时的重要性。尽管之前已有工作对大数据系统进行表征，但都依赖研究者的直觉进行考虑。

进一步提出MKP（MIA-based Kiviot Plot），对筛选出来的n个性能指标按影响程度降序排列。通过这种方法，不同benchmark的性能相似性/非相似性可通过比较相应的Kiviot图观察出来。

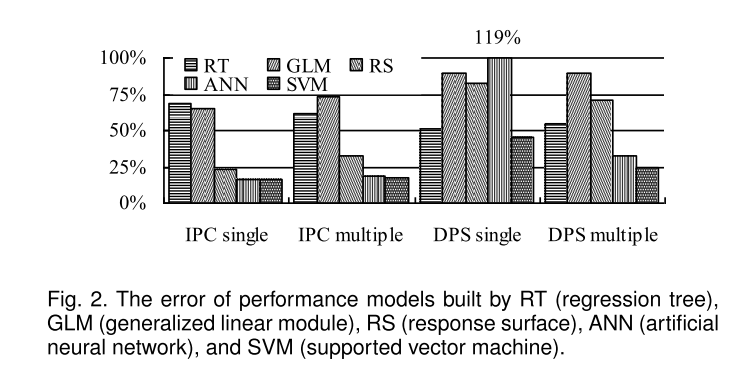
还提出benchmark相似性矩阵。最终实验测试MIA, MKP, BSM等能有效直观地表征复杂大数据系统的性能。发现对于指标IPC来说，失效率是重要的因素；对指标DPS，临时读写操作的数量是最重要的。执行MIA时必须使用多层面的指标，因为某一层的指标可能会对另一层的性能产生影响。

验证MIA的实用性，利用MIA的表征结果来指导性能优化。同时调整两个不同方面的参数，发现更重要的参数调整后效果确实更好。

### 高级建模工作的限制

为全面理解大数据工作负载的性能，需要对成百上千的指标进行考虑。表征工作负载的特点需要方法来为指标构建性能模型作为函数。许多方法如回归树，机器学习技术等可以构建性能模型。如果有大量训练数据，模型会很精确，但是收集大量训练样例不容易。

下图是106个训练样本时，几种方法的误差。IPC对应节点级，DPS对应系统级，图中IPC/DPS single代表使用的输入指标分别来自相应的单一级别，multiple表示结合了两个及标的指标。所有情况下平均误差均超过了19%，所以当前环境下数据集量少，很不准确。故使用更先进的方法集成学习。

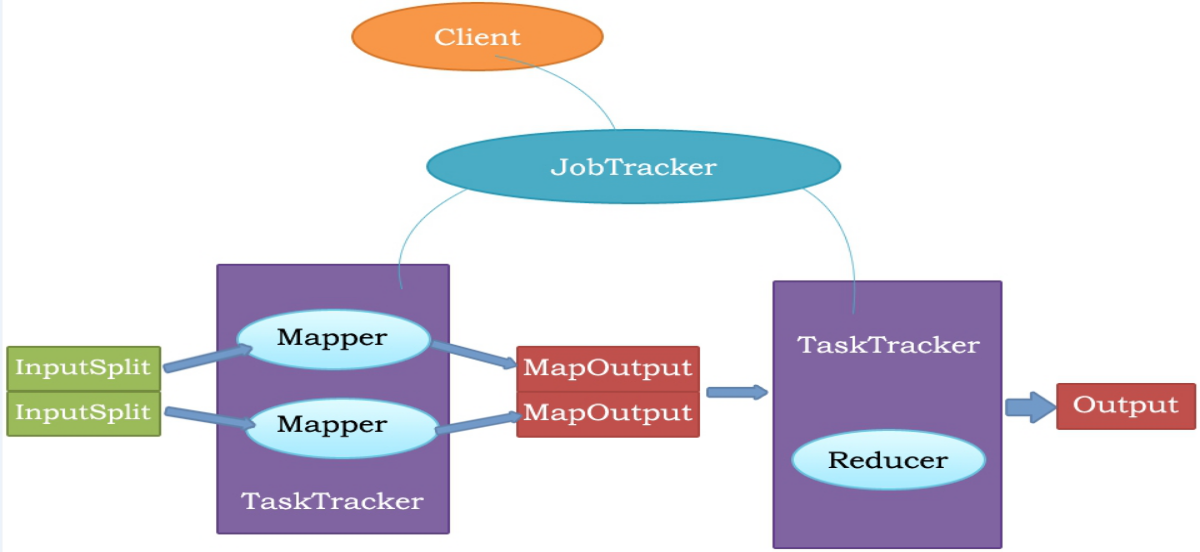


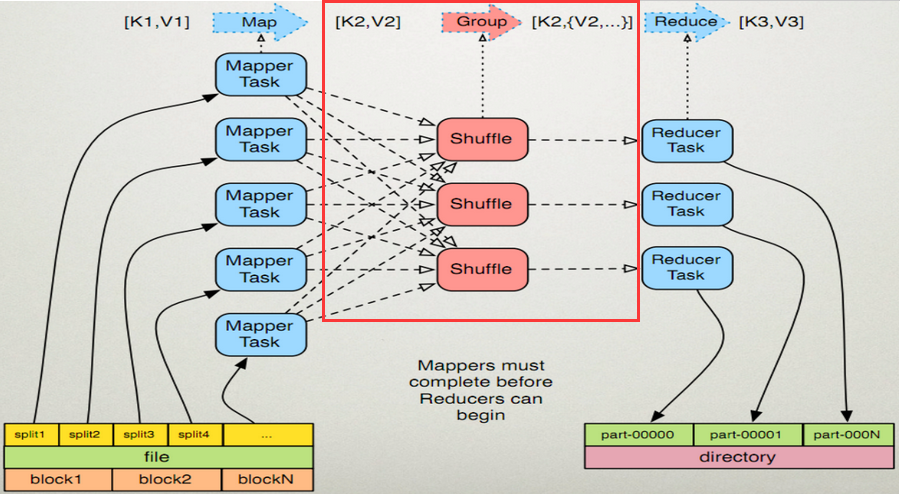
### MapReduce & Hadoop

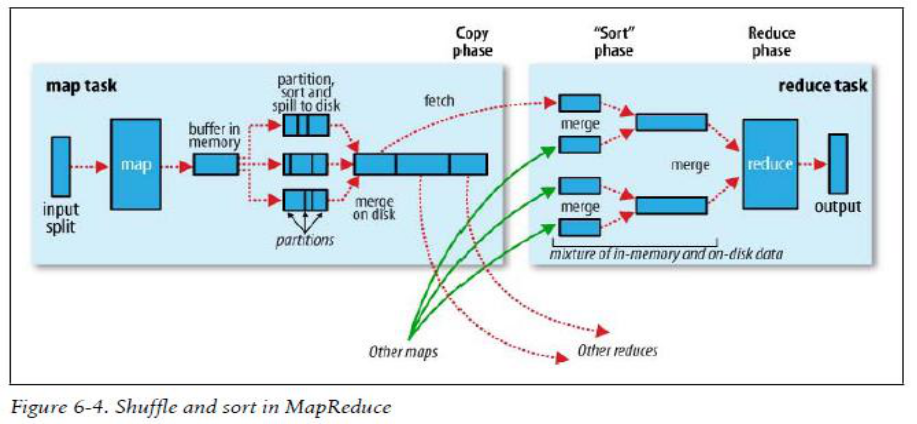
大规模编程模型/框架，用于处理并生成大量key/value形式的数据集，用户只需要写map/reduce函数。HDFS（Hadoop分布式文件系统）将大量数据集分成小的block. 执行Hadoop程序分为map和reduce两个阶段。Map阶段，每个map任务读取一个small block并处理之，输出中间文件复制给reduce节点；reduce阶段提取map输出文件中的key/value数据，执行sort, merge, process等操作生成最终输出。

**Mapreduce运行原理：**

1. Map处理：1）读取HDFS文件，每一行解析成一个键值对key/value. 2）每个键值对调用一次map()，处理后转换为一些新的key/value键值对，输出。3）对输出的键值对进行分区（使用分区是因为根据业务需要产生多个输出文件；多个分区->多个reduce任务在运行->提高整体job的运行效率）。4）不同分区中的数据进行排序sort，分组（例如键值对<h,1><u,1><h,1><m,1>分组后变为<h,{1,1}><u,{1}><m,{1}>）。5）对分组后数据进行规约combiner（每个map可能产生大量输出，combiner作用是在map端对输出先做一次合并，减少传输到reduce端的数据量。只用于reduce输入key/value与输出key/value类型完全一致，且不影响最终结果的场景）。
2. Reduce处理：1）shuffle：多个map任务输出，按照分区通过网络copy到不同的reduce节点上。2）每个reduce节点对多个map函数输出进行合并、排序。调用reduce函数，输入分组后的数据，实现自己的业务逻辑，输出产生新的<k,v>. 3）reduce输出的<k,v>写入HDFS中。







**更直观的流程表述：**

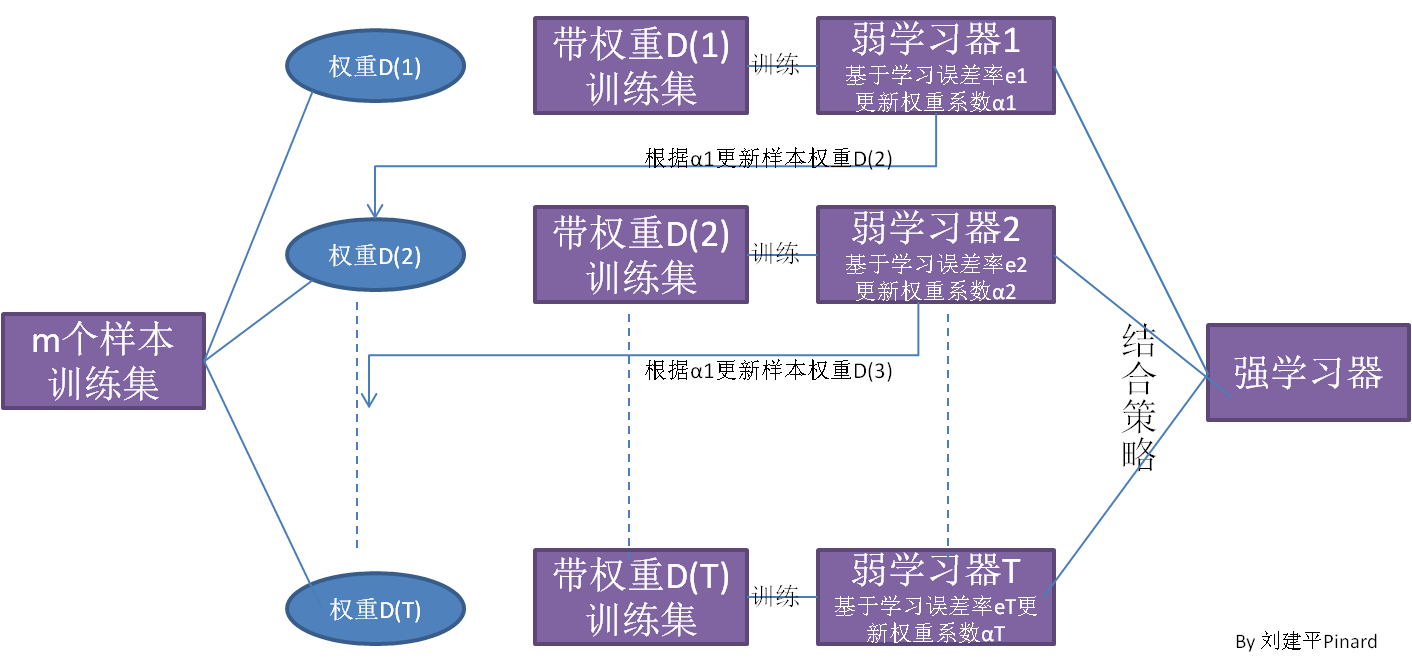
a. 首先在map端，map任务结束后，输出先存放在缓存中，若存放超出了缓存阈值，后台线程把内容写入到本地磁盘中的指定目录下新建的溢出写文件中。Map输出一开始缓存在内存中，溢出后才写入磁盘而不是HDFS，通过缓存减少磁盘I/O的开销。因此设计map函数时应尽量减少内存使用，为shuffle过程预留更多内存。b. 写磁盘前，进行partition、sort和combine等操作。等最后记录写完，将全部溢出文件合并为一个分区并排序的文件。写磁盘时采用压缩方式，将map输出结果进行压缩，有效减少网络开销。c. 磁盘中的数据送入reduce中。Reducer通过Http方式得到输出文件的分区。一个reduce端可能从n个map结果中获取数据，但每个map执行速度不完全相同。有map运行结束，TaskTracker就会得到消息，进而将消息汇报给JobTracker，然后reducer就会得到JobTracker的信息。Reducer定时获取该信息，默认有5个数据复制线程从map端复制数据。d. 复制来的数据同样先写reduce端缓存，溢出后再写磁盘，同样会进行排序分区结合等过程，最后一次合并的结果作为reduce的输入。e. 当reducer的输入文件确定后，shuffle阶段才算结束，之后执行reduce操作，最后把结果存到HDFS上。

### 集成学习ensemble learning

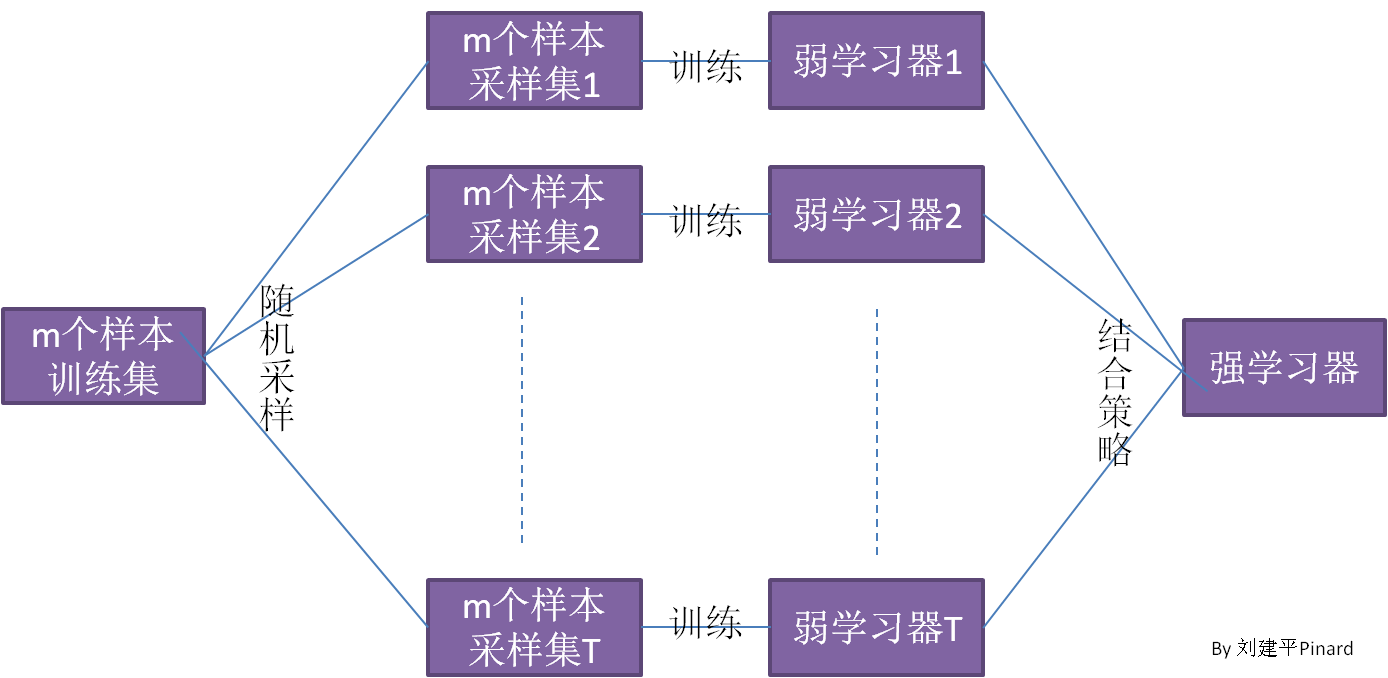
**1.简要介绍：**

训练多个个体学习器，通过一定结合策略，形成一个强学习器。个体学习器可以是同质的，也可以异质，但同质的应用最广泛。同质个体学习器按照学习器之间是否存在依赖关系分为两类：1.个体学习器间存在强依赖关系，一系列个体学习器基本都需要串行生成（代表算法 boosting）；2.个体学习器间不存在强依赖关系，可以并行生成（代表算法 bagging，随机森林算法）

1）Boosting：如下图所示，从训练集中用初始权重训练出弱学习器1，根据该弱学习器的误差率表现，更新训练样本的权重，基于调整权重后的训练集训练出弱学习器2，如此重复进行，直到总弱学习器数为T（预先设定好），通过集合策略进行整合，得到最终强学习器。



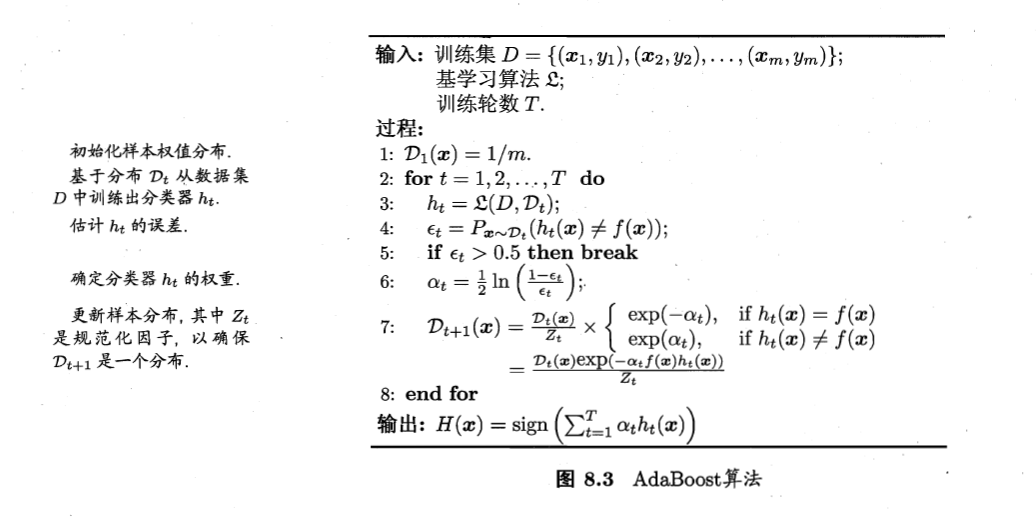
2）Bagging：训练集通过T次随机采样得到，分别独立训练出T个弱学习器，再集合策略得到最终的强学习器。随机采样一般是自助采样法（Bootstrap sampling），对于m个样本的训练集，每次先采集一个样本到训练集，再把样本放回，下次采样样本仍有可能被采集到。



3）结合策略：1.平均法。对弱学习器的输出求平均得到最终预测输出。2.投票法。T个弱学习器对样本x的预测结果中，数量最多的类别为最终的分类类别，若平票，随机选一个类别。可以对投票规则稍作复杂的调整。3.学习法。如stacking，将弱学习器的结果作为输入，再训练一个中间学习器，得到最终结果。

**2.算法详细：**

1）Adaboost：算法流程如下图所示



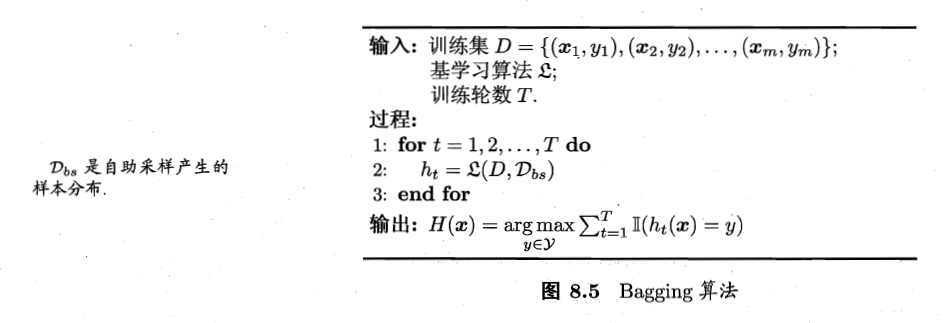
Boosting算法要求基学习器（弱学习器）能对特定的数据分布进行学习，这可通过“重赋权法”实施，在训练过程的每一轮中，根据样本分布为每个训练样本重新赋予一个权重。对于无法接受带权样本的基学习算法，通过“重采样法”，每一轮根据样本分布重新采样。

Boosting要求每一轮学习前，检查当前生成的基学习器是否满足条件（如：学习率比随机猜测要好），一旦不满足，抛弃学习器，学习过程停止。一旦这样做，有可能达不到要求的T轮学习，因此学习性能不佳；采用“重采样法”可获得“重启动”机会，避免训练过早停止。

Boosting主要关注降低偏差，因此能基于generalization性能相当弱的学习器，构建很强的集成。

2）bagging 并行式集成学习方法代表

使用自助采样法，采样出T个含m个训练样本的采样集，基于每个采样集分别训练出一个基学习器，再将这些基学习器进行结合。结合时bagging通常对分类任务采用投票法，对回归任务采用平均法（每个基学习器相同权重）。



由于采样过程实际只用到了约63.2%的样本，剩下的样本可用作验证集来对泛化性能做“包外估计”。

Bagging主要关注降低方差，因此在不剪枝决策树、神经网络等易受样本扰动的学习器上效用更为明显。

3）随机森林RF bagging的扩展变体

以决策树为基学习器，进一步在决策树训练过程中引入随机属性选择。（传统决策树选择划分属性时在当前结点的属性集合中选择一个最优属性；RF中，对基决策树的每个结点，先从属性集合中随机选择一个包含k个属性的子集，再从子集中选择一个最优作为划分）。

这样做在样本扰动基础上增加了属性扰动，进一步增加了基学习器的差异性。RF方法收敛性与bagging相似，且收敛到更低的泛化误差，训练效率优于bagging.

### MIA方法

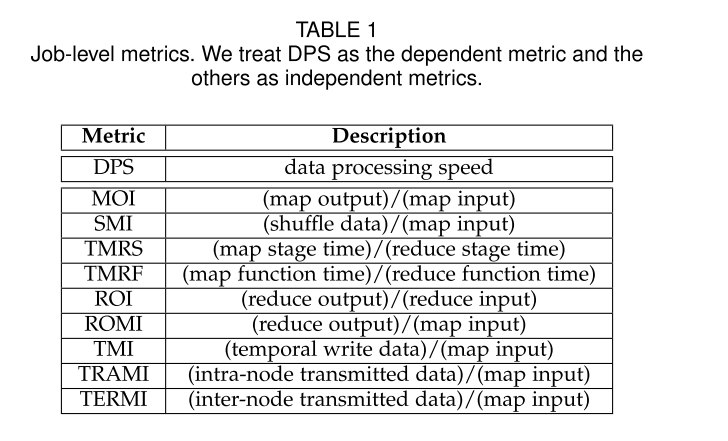
**1. MIA工作流：**

profiling，存储，建模，应用。1）剖分Hadoop工作负载的多级性能特征。2）存储剖分后的结果，存进数据库。设计多个table存储多级指标：tbl\_micro对应处理器级（L1数据cache失效率，分支预测错误率等），tbl\_sys对应系统级。3）MIA建模利用集成学习算法stochastic gradient boosted regression tree(SGBRT)，使用数据库中存储的作为训练数据，创建性能模型。4）定量分析各指标在某些方面（IPC/DPS）的重要性。5）两个应用实例：MKP可视化负载的性能+比较两个负载的相似性、指标的重要性值作为向量中的元素，实质上构造了一个矩阵BSM，矩阵实际上总结了两个benchmark的相似性。

最核心的就是建模过程，这个方法决定了MIA结果的精度和得到结果的开销。需要在精度和开销之间平衡，且很多训练样本会导致over-fitting. 之前讨论过传统机器学习方法精度不足。考虑使用SGBRT既有精度也有低开销，robust to over-fitting.

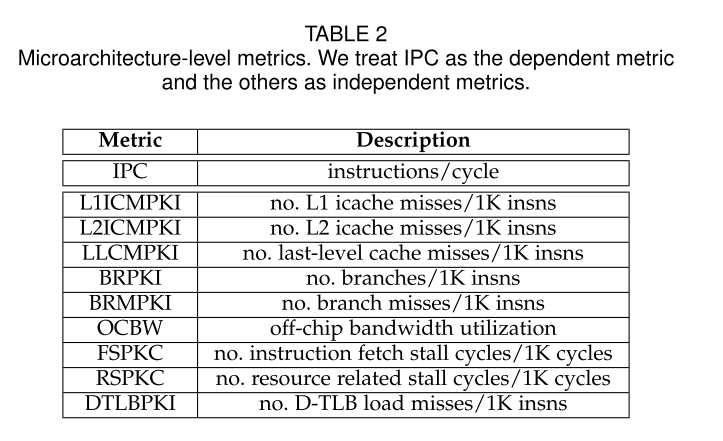
**2.选择的指标：**

1）作业级指标。



这些指标为作业级的输入输出数据量、处理速度、通信等方面提供了广阔的视图。1. DPS衡量程序处理数据的速度，=数据量/程序执行时间，它是系统性能的最直观视图，值越高越好。2.在MapReduce的map阶段，定义MOI，map输出数据大小和输入数据大小的比例。3.SMI是map操作中，参与shuffle处理阶段的数据所占比例。4.TMRS是map和reduce处理阶段执行时间的比例。5.TMRF，执行map函数和reduce函数的时间比例。6.ROI是reduce阶段输出数据和输入数据大小的比例。7.ROMI是reduce阶段输出数据和map阶段输入数据的大小比例。8.TMI是临时写入本地文件系统的数据和map输入数据量之比。9.TRAMI是节点内进程之间传输的数据量和map输入数据量之比。10.TERMI是节点间传输的数据量和map输入数据量之比。

2）微架构级microarchitecture指标.



这些微架构级指标共同为表示单节点性能提供了良好视图。每个节点上使用硬件性能计数器收集指标，对所有节点取平均。1.IPC每个周期执行的指令数，是评价节点性能的可靠指标。2.L1ICMPKI+L2ICMPKI分别量化每千条指令L1和L2级指令cache的失效数，是指令追踪和代码局部性的度量。3.LLCMPKI是每千条指令最后一级cache的失效数，是数据追踪和数据局部性的度量。没有考虑L1和L2数据cache，因为他们失效的大多数延迟可以被无序执行隐藏。4.BRPKI每千条指令的分支数，BRMPKI每千条指令的分支预测失效数。5.OCBW，对每个核测量的片外带宽利用率，单位每秒字节数，公式(64 × 2.4 ×) × llcm/clkuh. 6.IFSKC+RSKC停顿周期数。7.DTLBPKI，每千条指令DTLBmiss数。

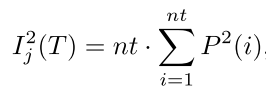
**3.算法确定重要的指标MIA**

MIA开始时，确定一些依赖性很强的指标，比如微架构级的IPC和作业级的DPS. 接着对独立的指标变量按照其对依赖性变量的重要度排序。使用集成学习方法SGBRT（随机梯度boost回归树）完成这个排序过程。SGBRT组合多个独立的回归树产生的预测，生成最终预测。

如何利用SGBRT建立性能模型？

1.对每组程序输入对（program-input pair），收集IPC、DPS以及其他所有表中描述的指标。

2.分别建立N（程序输入对的总数）个DPS和IPC向量，每个向量表示第i组输入对的DPS、IPC值以及其他指标对在第i组程序输入对下的值。把各组输入对整合起来，构成两个矩阵DPS+IPC.

3.把这两个矩阵作为集成学习器的输入，训练出一个DPS模型和一个IPC模型。构建好IPC和DPS的模型后，开始对独立的指标进行排序。排序方法如下：对一个单回归树T，度量指标xj的重要性，基于回归树中选择指标xj对树进行分割的次数。该重要性度量用以下公式表示：，nt是指标xj被用于分割树的次数，P2(i)是第i次用指标xj对树分割后获得的平方增益，P(i)是依赖性指标（IPC/DPS）的相对变化率。如果指标xj在M个树中都被用来分割，则总重要性度量是他们的简单算术平均。

4.最后对每个计算出结果的独立性指标进行标准化，使他们值的总和是100%.

在表征节点级指标时，是否还要考虑系统级指标？同理表征作业级时，是否还要考虑节点级指标？发现将节点级和作业级指标都作为独立指标考虑时，在节点级建立的性能模型精确度更高，说明了各级之间的指标有很强的相互作用性。

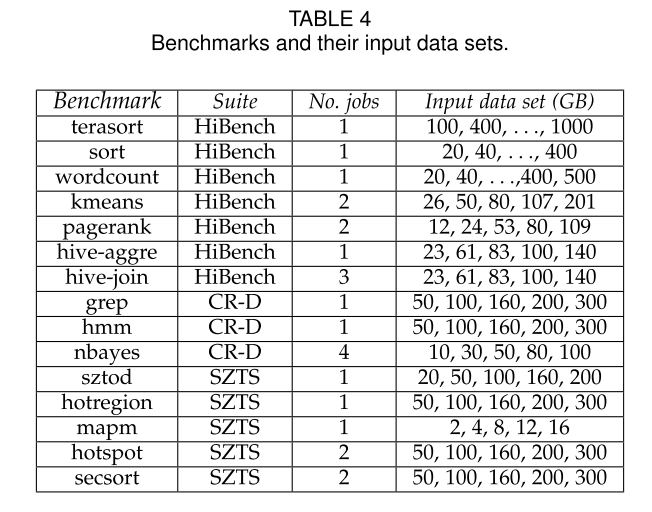
### 实验与结果

**实验设置**

搭建一个hadoop集群，1个控制节点，8个数据节点。节点配备带宽1GB/s的以太网卡，两个以太网交换机互连。A交换机用于全局时钟同步，B用于Hadoop路由通信。每节点2TB磁盘，16GB内存，2个Xeon E5620的8核处理器。

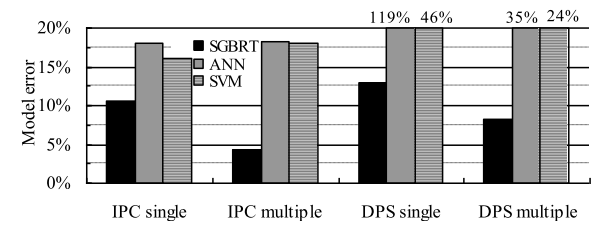
使用oprofile v0.98测量微架构级的指标。使用sysstat v9.06收集系统级指标。每5s读取一次hadoop框架的日志文件，观察作业特征。每5s读取目录/proc/net/dev获取网络上的运行特征。

使用benchmark如下：



**实验结果**

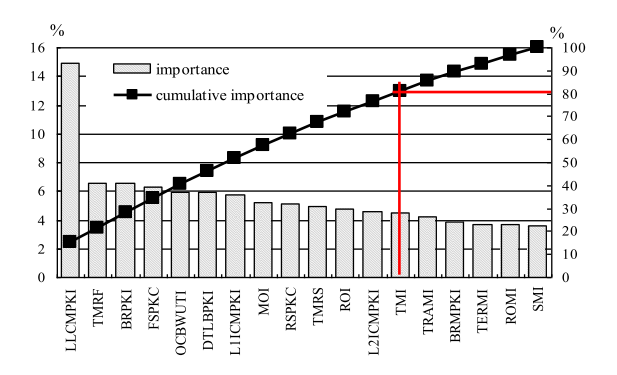
1.首先验证机器学习算法SGBRT的精确度，与ANN、SVM等比较。



在构建某个依赖指标的性能模型时，同时考虑两个层级的独立指标的影响，在SGBRT算法下获得更低的错误率。四个SGBRT值分别是10.6%、4.3%、13%、8.2%.

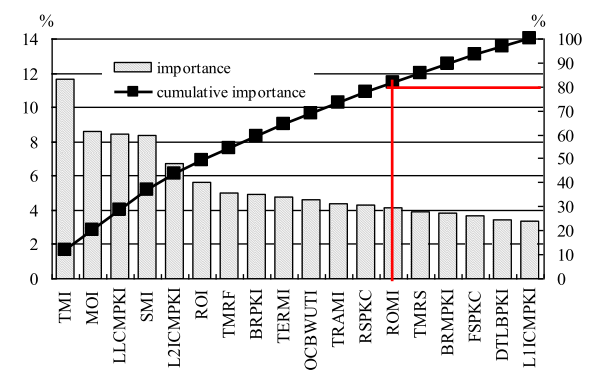
2.验证MIA的排序结果。

1）对于IPC的重要性指标：



构建性能模型时，同时考虑两层的独立指标。从结果上看，对IPC来说，最重要的指标是LLC-MPKI（最后一级cache失效数），而第二重要的指标竟然是TMRF（map函数和reduce函数的执行时间比例），证明作业级指标与体系结构级的性能有很强的相关性。

2）对DPS的重要性指标：



从图中看出TMI是影响DPS最重要的指标。发现LLC-MPKI同时重要地影响着DPS，表明最高一级的cache失效率不但显著影响单节点的IPC，还反过来影响节点整体的DPS吞吐量。值得注意的是当机器集群的大小改变时，指标的重要性是有可能改变或者不变的。例如一个负载消耗了一个16服务器集群的所有内存带宽，假如改成32个服务器，内存带宽仍然会饱和；而在其他情况下，改变集群大小相对的指标重要性可能会改变。

## 18. An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing

### 背景

把应用程序的虚拟拓扑（通信模式）映射到机器物理拓扑上，使得程序执行时间优化。处理器并行架构设计的趋势是核越来越多，要弥合目标机器级和应用程序级的性能差异，实现扩展。重点是应用程序数据的组织，访问和移动方式。局部性问题存在于应用程序层、节点层、网络和存储单元。进线程传输数据速度取决于这些单元的亲和度以及相对位置。

### 总体描述

**可扩展性key：数据局部性**

大方向是，并行机器每个节点核数在增加。超大规模计算机会有数十万到数百万节点，每个节点集成数百到数万个核心。未来计算机将有更深的层次结构。很多问题的解决办法是提升并行应用程序的数据局部性（数据被体系结构的多个硬件处理单元放置，访问和移动的方式）。

提升数据局部性，与并行程序在软件处理实体（进程，线程，任务）之中调度工作负载的性能有关联。这些软件处理实体在程序运行时访问和交换数据，但不一定是以常规的方式，因此有优化的空间。例如交换或共享大量数据的实体放置在物理上彼此靠近的硬件处理单元上。

许多超大规模系统的能量消耗于数据移动（局部处理器核内存之间+全局之间），故必须提供机制或API管理数据局部性，有时能减少数据访问时延。

**并行编程模型中的数据局部性管理**

提升并行程序的可扩展性是一定要考虑数据局部性的，无关选择的范式。因此可以优化、改变当前存在的编程范式进行局部性管理。

现存的编程模型：MPI（消息传递接口）依靠一个平面网络（flat network）的分布式内存模型；OpenMP使用全局地址空间；PGAS（分区全局地址空间）语言使用与MPI类似的平面，分区地址空间。

结果底层硬件和编程模型间的差异加大。为弥合这个差异，首先应考虑编程范式实现的优化，其次是范式的定义和应用需要更具体。更准确地，应用程序和底层硬件之间的交互应当明确。提升程序的数据局部性取决于上下文和使用的编程范式，有着不同的效果。如在MPI程序下，优化会降低通信成本；对于多线程应用程序，预期结果是应用线程间有更好的数据共享。更好地利用底层硬件：网络，内存层次结构，处理单元。

**虚拟拓扑（通信模式）**

一种提高数据局部性的方法是使处理单元专属于特定软件处理实体，这种方法下，需要确定程序虚拟拓扑到目标硬件架构的匹配matching. 虚拟拓扑表达了软件处理实体（进程、线程、处理单元等）之间的依赖关系。例如显式通信模型中（MPI），依赖关系表示为进程之间交换的信息。隐式通信模型（OpenMP）依赖关系表示为线程对公共存储位置的访问。

虚拟拓扑也称为程序的通信模式，可静态可动态。静态指处理单元数量在程序运行过程中保持不变+软件处理实体的依赖关系在连续程序执行中保持不变。否则二者之一改变定性为动态。基于OpenMP的程序具有动态模式，因为进入或退出每个新的并行部分时会创建/销毁新的线程。虚拟拓扑（通信模式）是并行应用程序的key特征。

**问题描述**

实现映射，将应用程序图中的顶点s映射到机器拓扑图中的顶点t.

每个具体的映射需要一个评价指标，表示映射后的优化效果。指标分为两大类：dilation, congestion.

Dilation是指A映射到H后邻居间的成对距离之和/最大值是H中顶点x, y之间距离的最小值，dilation指标可定义为. 若映射算法中包含路由函数，意味着路由不总是沿着最短路径，则dilation值会增大。Dilation指标的值与网络中的动态能耗强相关。

另一种指标是congestion，统计有多少通信对使用某一链路。以这种指标评价的映射必须包含路由函数。用表示从u到v经过边e的可能性，则定义某个边e的指标congestion为. 接着可以用不同的方法生成包含所有网络链路的全局congestion度量，最常见的是最大值.

映射优化算法的目标就是最小化这两类指标。

### 拓扑映射算法

映射问题是NP难的，目前使用的求解器可以针对小规模的输入实例解决二次分配问题，但对于大规模计算网络不实用。真实映射场景下，理想的架构很少，因此特殊的理想化的映射很少。考虑映射为图内任意点A到任意点H的映射，过去很多研究成果研究出一些启发式算法，可分为几类：

1.贪婪算法的变体 2.图划分算法 3.基于图相似度的方法 4.基于子图同构的方法。

使用一些技术计算映射，计算是三步的过程：

1.收集目标应用程序的虚拟拓扑（通信模式）；

2.收集目标底层架构的物理拓扑；

3.用算法计算两种拓扑之间的匹配。

该综述只讨论实现映射的放置策略的各种技术

### 放置策略

1.资源绑定：直接绑定processing elements到专用的硬件processing units. 在linux/GNU操作系统中，使用命令numact1/taskset可以实现。但没有可用的绑定方式用于大范围的机器架构和系统。

硬件局部性工具Hwloc[1]部分解决了这个问题，它提供一个通用、不依赖操作系统的接口，把底层存储层次和处理器的布局进行显示、抽象，管理处理单元的放置。强制绑定比不强制绑定好，依靠操作系统调度会带来更多cache失效，对性能有害；不强制绑定，运行性能不稳定，难以预测。绑定之后，实验性能标准差更低，实现系统降噪和程序稳定运行。

绑定技术依靠命令，独立于应用程序，但不保证可执行性（系统有差异，命令不一样），运行时改变绑定方式难。

[1] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, “Hwloc: a generic framework for managing hardware affinities in HPC applications,” in *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, (Pisa, Italia), IEEE Computer Society Press, 2010.

2.rank reordering[2][3]. 每个处理单元都有标识码（数字），用这些数字进行数据交换和同步操作。Rank reordering技术修改这些标识码以映射程序的虚拟拓扑。这个方法仍然可能出现不良决策下的副作用side-effect（如cache失效等）。所以可以结合rank reordering和资源绑定。首先程序启动时进行强制绑定，不应用调度策略，避免调度的副作用；在程序执行时根据匹配算法的结果进行重排序。

[2]B. Brandfass, T. Alrutz, and T. Gerhold, “Rank reordering for MPI communication optimization,” Computer & Fluids, vol. 80, pp. 372–380, 2013.

[3]G.Mercier and E. Jeannot, “Improving MPI applications performance on multicore clusters with rank reordering,” in EuroMPI, vol. 6960 of Lecture Notes in Computer Science, (Santorini, Greece), pp. 39–49, Springer, 2011.

3.

## 19. BlackBox线程分配预测和调度算法Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors[J]. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(12):2513-2525.

BlackBox线程调度方法，解决TSBSched方法需要应用程序源代码关键信息和改变源代码的限制。

### UltraSPARC T2处理器

多线程处理器，8个核心连接到一个crossbar上共享一个L2Cache，每个核8个hardware contexts，共64应用线程可同时并行。核上的hardware contexts分为两组（即两个硬件管道），每组4个。线程依据其分布情况共享三级资源。1.核间共享：处理器上所有线程共享该级别资源：L2cache，片上互连网络（crossbar），内存控制器，片外接口（I/O）。

2.核内共享：同一核内部的线程共享：L1指令&数据Cache，指令&数据TLB，load/store unit，浮点&图形unit，加密处理unit.

3.管道内共享：执行在同一个pipeline上的线程共享：取指令unit，整数执行unit.

### BlackBox调度算法

系统级方法，目的是估计不同线程分配方式的性能，并决定采用性能最好的分配方式。

首先，在没有详细的应用程序所需硬件数据的条件下，使用最少的处理器信息，确定一系列线程分配（profiling thread assignments），用于构建线程干扰模型。依据这一组线程分配的实测性能，估计应用程序的任意线程分配的性能。算法分析执行线程之间的干扰造成的性能减损，但不分析应用程序的执行时间信息和线程独立执行造成的性能减损，因此可被看作一个黑盒。

实现两个目标：

1. 不需要再详细了解应用程序的硬件要求。
2. 独立于体系结构。算法只需知道不同级别共享资源的层次和硬件context的数量。

**算法流程**

1. 程序profiling. 分析要实验的应用程序，输出Basetime table&Slowdown table. 测量同时运行的线程间的干扰，主要是两方面：
2. 对共享资源的竞争。作者对UltraSPARC T2处理器的资源共享级别进行特征化。在负载选择阶段，考虑各级别线程之间的干扰很重要；当负载一旦选择后，运行在核N上的线程与没运行在核N上的线程之间的干扰就可忽略了。因此分析范围减小到执行在同一核上的线程间。

缩小了考虑范围，使得作者可以采用暴力探索的方式（执行一个核中所有可能的线程分配方式）。选定目标线程，执行所有可能结合的负载，每执行一种负载代表一次实验。每次实验测量目标线程的性能数据，因为该数据体现了目标线程由于资源竞争产生的性能损失，因此数据记录入slowdown table.

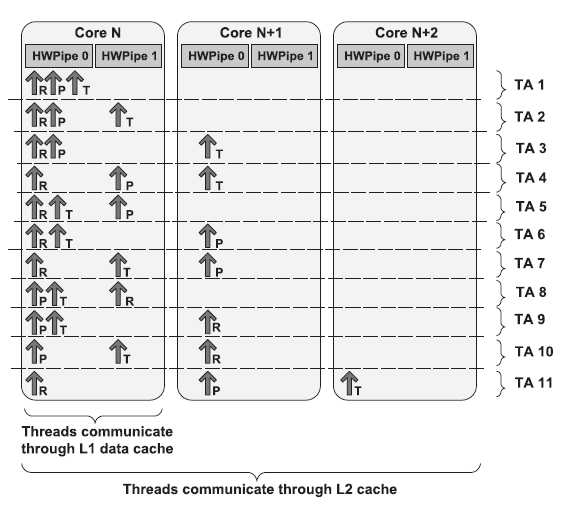
下面是IPFwd程序运行在UltraSPARC T2处理器的一个实例。此应用程序由三个线程组成（R接受，P处理，T传输）

执行一个目标应用程序，同时执行一个压力程序，两个程序都执行相同的网络数据包处理，然而实验只监测目标程序的性能。因此压力程序时共享处理器资源中的干扰，会影响线程分配的性能。接着量化性能的损失，假定目标线程为Rtg，组合Rtg与压力程序线程Rst, Pst, Tst运行在同一核内的所有可能情况。因此slowdown table有很多条目。实验测试线程Rtg时，目标程序的其他线程Ptg, Ttg孤立地在其他核上执行，保证了干扰只来自于压力程序。

1. 由于数据和指令共享的获利。多个线程共享数据和指令，他们会受益于同一处理器内核上的协同调度。

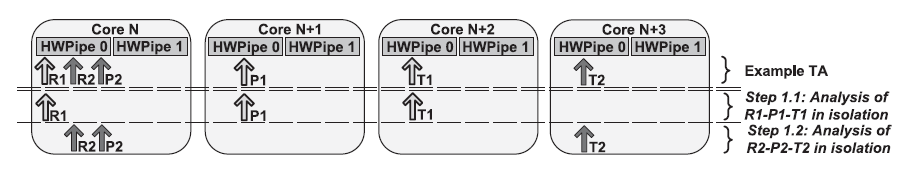
执行单个应用程序，设置所有可能的线程分配，测量每次线程分配的性能（程序运行时间），测量数据存入Base Time Table.

针对IPFwd应用程序，当连续不断的线程运行在同一核内时，通过L1cache通信；运行在不同核上时，通过L2cache通信，这样会造成额外的L1cache失效和延迟。



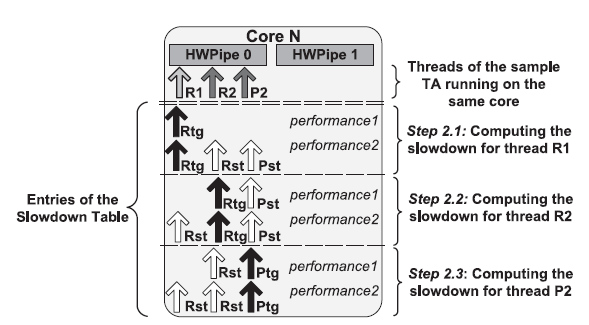
据统计此实验有11种不同的分配情况，如图所示。TA11中各线程在不同核上，所以值的任何更新将通过全局共享的L2高速缓存进行。TA7中R，T线程在同一核上，数据更新在L1cache中本地进行，不需要使其他核中的cache行失效，因此在本例中TA7/TA11应用程序线程R，T之间数据和指令共享可以提供性能收益。

1. 预测性能，基于存储在Base time table和Slowdown table中的数据预测线程分配性能，输出不同线程分配的列表。整个预测过程分三阶段：分析每个程序/程序实例单独运行时的性能；



假设有两个IPFwd程序同时运行。

1. 第一步，分别分析两个程序单独运行时的性能，每个程序的负载运行时伴随一个base\_performance，该值从Base Time Table中读取。如图所示，线程R1，P1，T1等同于情况TA11，故从Base Time Table中读取TA11情况的运行时间；程序2的线程R2，P2，T2等同于情况TA3，同理从表中读取TA3的运行时间。
2. 第二步，对硬件资源竞争情况建模。考虑不同程序或不同程序实例的线程之间的硬件资源竞争问题。用来量化slowdown的信息保存在Slowdown Table中。



如上图所示，此情况是R1，R2，P2线程共享一个核一个pipeline内的资源。

逐次分析，当程序1的线程R1运行时核内有程序2的线程R2，P2干扰，对应Slowdown Table里的Rtg-Pst-Tst. 首先读表里Rtg单独运行在一个核而其他相同程序的线程运行在不同核时的性能，记录为performance1，对应上图线程R1单独运行在一个核上。再读表里当线程Rtg与线程Rst和Pst共享一个管道时程序1的性能，记录为performance2，代表上图线程R1与R2和P2竞争。由performance1和performance2计算slowdown，，

程序2的线程R2运行有其他线程干扰，计算slowdown(R2)，同理计算slowdown(P2)，方法类似于计算slowdown(R1). 此模型中，运行在其他核上的线程不考虑slowdown.

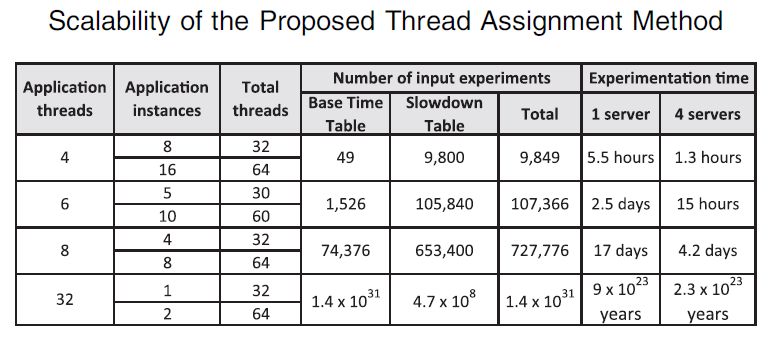
1. 第三步计算给定线程分配的最终预测性能。首先读取两个程序的base\_performance，其次计算每个程序的slowdown. 程序总的减损. 程序总的性能是base\_performance与slowdown之比.

某种线程分配的总性能是各个程序在该分配下性能的和。对每种线程分配都如此计算，得到总的性能预测列表。

1. 选择，预测后的N个性能最好的分配方法。尽管BlackBox预测高精度。但是可能错误预测实际具有最高性能的线程分配。因此在选择阶段测量多个预测最佳分配的实际性能。选择最终具有最好测量结果的线程分配方式。

**可扩展性**

为了收集Base Time Table和Slowdown Table的数据，需要运行每种线程分配和每个应用线程运行在单核上的配置。但当时网络并行应用程序所需线程数很少（数据包处理很短，拆分成多个线程容易引入通信开销，复杂网络程序通常包括三个线程：接受-处理-发送），所以依赖两个表的实验是可行的。程序剖分只用做一次，所以预测过程可以全自动。下表是不同线程数在UltraSPARC T2处理器上运行时所需进行的剖分次数的可扩展性分析



可看出单服务器上运行实验耗时很长，可以使用多个服务器同时实验。而且实验时间不随应用程序实例增加而变化，只随线程数增加变化。所以只要组成单个程序线程数不多，该方法还是可行的。

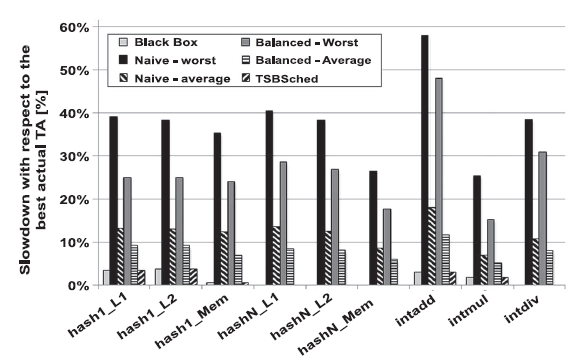
### 实验环境和实验结果

两个SPARC Enterprise T5220服务器\*（一个UltraSPARC T2处理器+64GB内存）。实验中性能瓶颈是处理数据包的速度，由所选线程分配的性能决定。

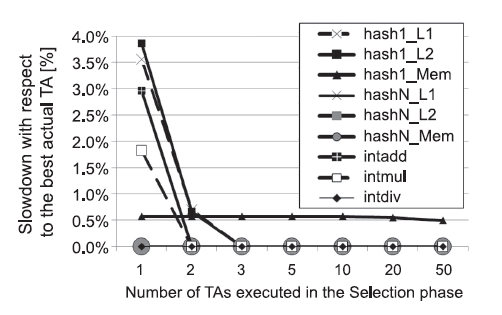
网络系统使用一些轻量级的运行时环境减少开销，例如Oracle开发的Netra DPS，该系统不提供虚拟内存和运行时调度，无上下文切换。运行任务的分配在编译时静态完成。系统不提供中断控制和守护进程，任务在一个硬件环境下不中断一直运行。

由于这个轻量级系统环境，测试程序选择Netra DPS分布式系统提供的IP转发程序IPFwd.

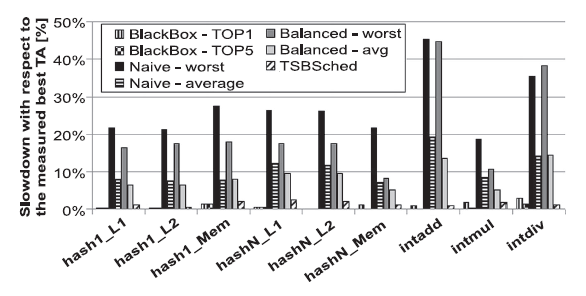
实验将BlackBox方法分别与简单调度、最新的操作系统负载平衡策略、TSBSched方法进行比较，分为6、24线程两组测试，结果为



X轴是不同测试程序，Y轴是相对于实际最优线程分配策略，几种方法给出的分配策略产生的性能差。可看出BlackBox方法整体性能不错，与TSBSched方法五五开。上图实验中BlackBox策略没有采用算法介绍的选择阶段。下图是选择阶段选取前N个性能最好的线程分配时，N的值对实验结果的影响。



从结果看采用选择段在9组测试程序中有4组进一步提升了性能。接着作者做了24线程的实验，但是此时全部线程分配方法太多了，因此随机选1000个进行预测。



### 存在问题

1. BlackBox方法可扩展性差，只适用于线程数较少的情况，一旦线程数增多，需要随机选择线程分配，造成精度损失。
2. 作者的实验基于UltraSPARC T2处理器，使用的是网络收发包的并行应用程序，可移植性不好，其扩展实验环境和应用场景的可能性还有待商榷。

## 20. 共享内存通信检测Diener M , Cruz E H M , Alves M A Z , et al. Communication in Shared Memory: Concepts, Definitions, and Efficient Detection[C]// Euromicro International Conference on Parallel. IEEE, 2016.

### 通信定义

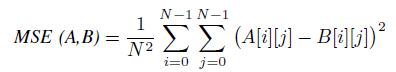
1. 显隐式通信。显式通信有send/recv函数，编程API有MPI，Charm++等；隐式通信通过内存访问共享变量，编程API有OpenMP，Pthreads等。一些基于MPI的扩展实现了通过共享内存的通信，Nemesis扩展：申请分配一个共享内存段用于通信，传输MPI函数调用，通过访问共享内存段在网络层进行通信。因此显隐式通信都能通过提升访存性能来优化。
2. 真假通信事件。在隐式通信中，不是所有对共享数据的内存访问都是真实通信过程。假通信分为以下三类：
3. Spatial false. 两个线程访问同一cache行，但不是同一个字中的数据，因此不代表数据传输过程。但是由于缓存一致性协议、失效、cache line传输，体系结构对待这种情况与对待真正的通信情况相同。因此实际以cache行为粒度进行通信统计时把这种情况也算在内。
4. Temporal false. 两个线程前后不同时访问同一内存地址，但cache行中的数据发生了变化。这种情况的出现频率取决于cache的大小和配置，也为通信检测造成了困难，需要考虑时间局部性来减少这种情况的影响。
5. Logical false. 某运行时程序同时需要大量寄存器，所需数量超过了硬件提供的数量，编译器需要将寄存器溢出到内存中，通信中的线程稍后会reload相同的值。这种行为不属于数据交换，理应视为通信错误，然而情况同1），会影响系统结构，所以统计时也算作通信。

综上，通信事件定义为两次对内存中同一cache line的访问由不同线程完成且中间该缓存行没有被替换。

1. 内存访问读写操作。写操作时，根据一致性协议会执行一些cache行失效，会产生更多片上互连的竞争，因此代价比读操作更高。然而读操作数量更多，也由于同步问题和较多的数据依赖更容易使流水线中断。由于二者都能影响实际性能，所以描述通信时把二者同等对待。
2. 通信方向和通信矩阵。

显式通信由于有收发双方是一定有通信方向的，而隐式通信分三种情况：读读、写读、写写。读读和写写都是无向的，因此把共享内存通信视为无向。根据通信量构建无向图、矩阵、归一化矩阵。

1. 通信模式比较。



### 简化的通信定义

1. 上述对通信的定义用于检测通信性能，精确度高，但有缺点：
2. 通信基于cache大小和cache层次配置，不同配置就会有不同通信性能，所以对应用程序的通信性能进行分析不完全有用。
3. 精确地分析通信性能需要在完整的cache模拟器上对程序进行分析，有很高开销。或者能直接访问到cache中的内容，但现代体系结构不支持。
4. 以cache line为粒度的通信分析需保存大量数据，具有很高存储开销。可提高分析粒度减小开销。

因此提出简化的通信定义。移除对cache层次的要求，以某种粒度考虑全部内存地址的访问。

1. 通信分析算法。每次访存过程，执行此function. 以内存块为粒度，每个内存块维护个队列（队长为2）。

Input:

//输入本次访存的地址address、执行线程tid、检测粒度g（内存块大小）

address: memory address that was accessed;

tid: thread ID of the thread that performed the access;

g: granularity of detection, number bits to shift

Output: updates communication events

// memory block of the address, contains a queue of up to 2 threads:

1 block = address >> g; //计算块号，block变量中包含一个队列

// number of threads that accessed the block; can be 0, 1 or 2:

2 nthreads = block.size(); //统计队列中元素个数

3 if nthreads == 0 then // no previous access

4 block.push back(tid); //队列里没有元素，将该访存tid加入队列

5 if nthreads == 1 && block.front() != tid then // 1 previous access

6 communication event(block.front(), tid);

//若tid与队列中线程不同，记一次线程间通信

7 block.push back(tid); //这次访存压入队列

8 if nthreads == 2 then // 2 previous accesses

9 t1 = block.front();

10 t2 = block.back();

11 if t1 != tid && t2 != tid then

//本次tid与队列中两个都不同，分别统计线程通信

12 communication event(t1, tid);

13 communication event(t2, tid);

14 block.pop front(); //弹出队首元素

15 block.push back(tid); //本次访问tid压入队列

16 else if t1 == tid then

17 communication event(t2, tid); //本次与上一次tid相同 计一次通信

18 else if t2 == tid then

19 communication event(t1, tid);

//本次与上上次tid相同 计一次通信 弹出队首元素 新线程进入队列

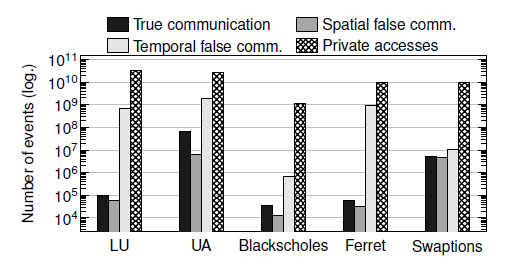
20 block.pop front();

21 block.push back(tid);

### 实验

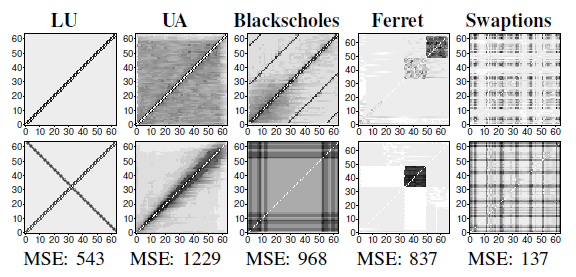
分别使用cache模拟器和实体机（Intel Nehalem微架构处理器，三层cache结构）进行实验，模拟器进行精确定义通信实验，实体机进行简化定义的通信实验。选择OpenMP实现的NAS benchmark中的LU, UA程序和pthreads实现的PARSEC benchmark的三个程序。

1. 下图是各种通信定义在几个测试程序中的占比

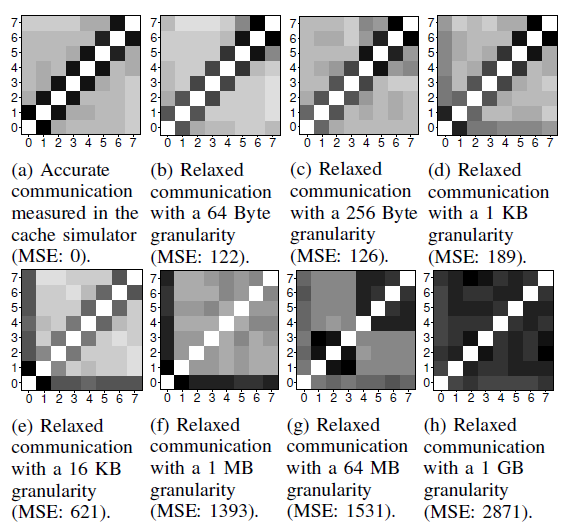


各种程序的通信情况差距很大，UA比LU的通信多很多，因此线程映射对这类程序效果很好。Temporal false通信占比一直很高，所以很需要过滤掉这一部分。

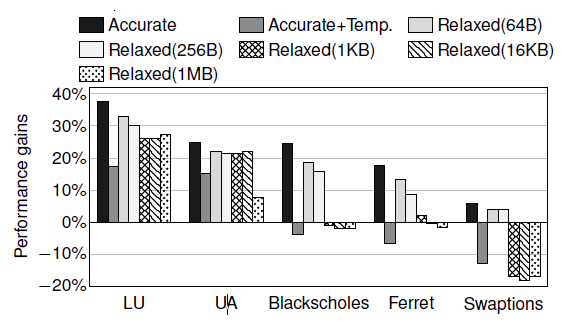
1. 使用精确通信检测得到的矩阵和包含temporal false communication得到的矩阵之间的比较，发现差距很大，因此需要过滤时间错误。



1. 对程序UA，精确通信检测和不同粒度的简化定义通信检测之间的差距比较如下。在粒度<1KB差距不是很大因此检测精度可以接受。



1. 不同水平的通信检测指导映射计算，带来的性能收益和损失情况如下图，简化定义通信检测的粒度较小时，映射结果可以接受。



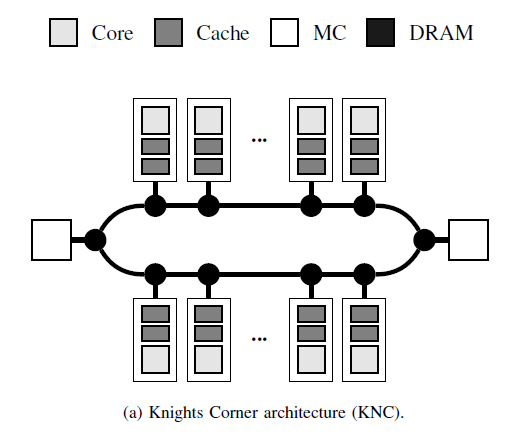
## 21. 众核架构映射评估Cruz E H M , Diener M , Serpa M S , et al. Improving Communication and Load Balancing with Thread Mapping in Manycore Systems[C]// 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). 2018.

对线程映射在两种众核架构（KNC，KNL）通信和负载平衡的影响做细致的分析。

### 架构介绍

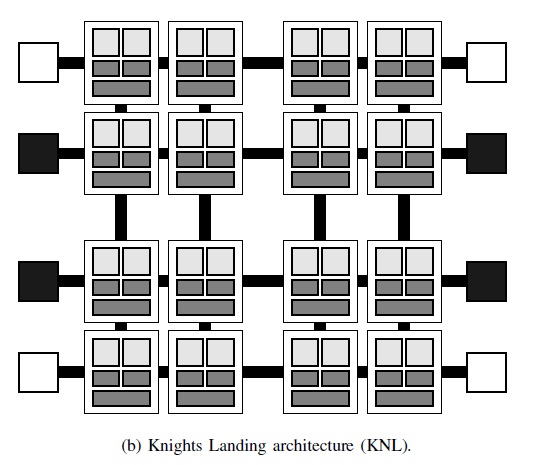
1. KNC

环形众核处理器，核心、cache内存、内存控制器MC、分布式标签目录和一个双向环。遵循基于MESI的缓存一致性协议。



若L2 cache失效，访问该失效地址对应的标签目录：若另一个L2 cache有相同的cache line，则从那个缓存转发数据；否则内存控制器从主存相应地址提取数据。有序流水，可同步4线程。

1. KNL



Tiles互连成网状结构，一个tile含2核，每核1L1cache，2核共享L2cache，分布式标签目录。除了内存控制器访问外部DDR4内存设备，系统还有MCDRAM内存，在cache模式下它可作为DDR4的cache工作；在flat模式下可作为独立内存。遵循MESIF缓存一致性协议，比KNC的协议更好。无序流水，可同步4线程。

若cache失效，检查失效缓存对应的tag directory，若其他缓存中有，从其他缓存中调取；若各缓存中都没有，可采取两种方法：1.MCDRAM工作在cache模式，把MCDRAM当成全局共享的L3cache，内存控制器先检查MCDRAM里有没有数据，若没有再从DDR内存中调取；2.MCDRAM是flat模式，直接检测缺失的内存地址在MCDRAM还是DDR里。Tag directory和内存控制器中的地址分布是可配置的，这种模式称为clustering模式。Clustering模式下常见的两种配置为：1.quadrant. 将tiles分为四个象限，地址由硬件在象限之间划分；2.sub-NUMA clustering, 每个内存控制器和MACRAM形成一个NUMA节点，操作系统负责选择节点存储页面。

### 通信和负载平衡对众核架构的影响

#### 1.通信

共享内存环境下，线程间通过访问相同内存地址进行通信。对于应用程序，写共享数据块相当于给其他线程发送消息，读共享数据块相当于类似于接受消息。为保证数据的持续性，cache一致性协议跟踪所有缓存行的状态。某核/线程要读的数据本地没有，一致性协议确定其他缓存中是否有相同缓存行的副本，若有，执行缓存到缓存的传输；否则访问主存。若某核要写共享数据块，一致性协议使其他缓存中的对应缓存行失效。

根据缓存一致性协议的行为，硬件只把不同线程对缓存中有的共享数据块的访存视为通信，因此并行程序的内存占用量和cache大小会影响硬件处理线程间通信。因为众核处理器缓存一般比多核处理器小，因此期望众核处理器具有不同的通信性能。

KNC和KNL架构都有同步多线程结构，即一个核内有多个线程。在KNC结构中有两种映射通信方式：1.线程映射到同一个核上但属于不同的虚拟核，他们之间在L1/L2 cache上的通信；2.线程映射到不同物理核上，使用片上互连通信。在KNL结构中，除了这两种，线程还可映射到同一个tile上的不同核，使用共享的L2cache进行通信。

使用生产者-消费者的测试程序进行实验，在KNC平台上，由于系统是环状的互连网络，所以使线程能共享缓存，会获得更高的通信性能，如果使测试程序的缓冲区大小设为L1缓存的大小，使线程映射到共享一个L1缓存，则具有最高的通信性能。但是当测试程序的缓冲区大小增加到不适应L1和L2的缓存时，考虑优化通信的映射无法提高性能。在KNL机器上，映射后使用共享缓存通信比不共享缓存加速的效果不如KNC显著，证明KNL的网状互连比KNC的环状互连更好地处理互连流量。

由上得出一般性结论：一般情况下，如果应用程序每个核心（线程）使用的内存占用量低于L2缓存的大小，基于通信的映射可以提高性能。

#### 2.负载平衡

实验验证众核平台下（KNC, KNL）映射对负载平衡的影响，用衡量负载的不平衡程度。在志强系列具有同步多线程处理器的平台中，运行在相同物理核不同虚拟核上的线程会竞争更多的资源，共享相同的流水和L1缓存。运行在不同物理核上的线程竞争较少资源（互连网络、标签目录、内存控制器）。因此在志强系列处理器中评估负载平衡，考虑运行在相同物理核中的线程。

负载最平衡的线程映射在KNC中微程序运行带来了209%的性能收益（该程序及使用内存也使用ALU，即不是访存密集型或计算密集型程序），表明KNC每个SMT线程同时最多仅可处理一个负载。KNL平台比KNC能更好地利用指令级并行，因为KNL支持无序流水。线程映射若能使不平衡降到最小，更多的功能单元就可被已在流水线中的指令使用。

若应用程序有更少的资源竞争，则映射为负载平衡带来的收益就更小。

### 并行程序中的通信和负载平衡

#### 1.通信与各并行程序

需要instrument应用程序，使用pin工具检测所有内存访问。考虑优化通信的映射，只能在NAS Parallel Benchmark程序集中的LU, MG这类程序中有效果，因为这类程序除了相邻线程会通信外，远端线程也有部分通信。而CG, EP, FT这类程序各个线程之间通信量差不多，所以基于通信的线程映射不能提高性能。

#### 2.负载平衡与各并行程序

需要确认应用程序的指令类型，以确定核心对功能单元的竞争数量。需要以CPU时间形式分析负载，且分析程序的指令。使用Linux测量各线程的CPU时间，用PAPI库测量线程执行每种指令的指令数。

通过实验观察，CPU时间不能正确表示线程的负载（大多数程序通过cpu时间测量出的不平衡很小，而指令的不平衡很大）。从OS视图，负载被均匀分不到各核心上，但实际上一些和对功能单元的竞争比其他核更高。不同程序之间，指令的负载不平衡规律不明显，但测量线程的负载时不应只考虑cpu时间，且必须考虑指令。

### 线程映射策略的评估

#### 1.不同线程策略

基准是Linux默认的调度。

Compact方式：连续的线程执行在内存结构上接近的虚拟核上。Compact模式通信最优。

Scatter方式：compact相反，这种映射方式可把负载分布到各个核上。

简单负载平衡方式：设计算法（每核单线程），输入表示机器核心的向量cores[]和表示线程负载的向量loads[]，核心数ncores和线程数nt：1.首先初始化各核信息；2.过程中遍历所有核，如果核j的剩余线程最多或核j剩余线程数与其他核相等但剩余可用负载最多，把核j视为当前核c. 把第i轮负载分配给当前核中的线程，输出映射数组，更新核信息。

#### 2.性能结果

在KNC平台上，1.compact模式整体没带来什么加速效果，除了CG, IS两种不平衡度不高的程序；LU程序性能反而下降，因为有两种截然相反的通信模式（相邻线程通信和远端线程通信）。2.scatter和简单负载平衡映射提供的性能提升差不多，因为这两种使负载更平衡。

在KNL平台上，有些程序如EP，虽然本身的负载和通信不平衡很少，但映射后也有很好的性能提升，因为操作系统本身的调度方式执行了大量不必要的线程迁移。

对多数应用程序来说，运行在KNC或KNL平台上，考虑负载平衡的映射带来的性能提升要比考虑通信的映射带来的提升更大，因为此种平台本身的SMT特性，对功能单元的竞争极高，同时内存层次结构简单（少共享的cache和单NUMA节点）。

## 22. CDSM通信检测Diener M , Cruz E H M , Navaux P O A , et al. Communication-aware process and thread mapping using online communication detection[J]. Parallel Computing, 2015, 43:43-63.

CDSM方法唯一需求是硬件和操作系统支持基于分页的虚拟内存技术。

### CDSM的思想

#### 通过page fault分析内存访问的性能

并行应用执行时，产生page fault后，CDSM记录产生页错误的进程ID，并建立与发生错误的内存物理地址的索引（即使处理的是相同物理地址的数据，进程不同，虚拟地址空间也不同。因此使用物理地址进行索引）。页错误可以覆盖全地址空间，因此对内存地址可以以不同粒度来划分，然后进行通信检测。

在CDSM中，作者将内存分为不同的memory block，计算memory block块号为. 对每个内存块，维护一个list，记录了4个进程ID的条目，表示当前内存块中最近四次页错误的进程ID记录. list中的两个不同进程之间产生页错误计一次通信事件。如果list满了，新来一个进程ID时，不断更新list，移除最久远的那次页错误记录。最后把list中的通信事件信息存储到一个大小为进程数\*进程数的二维通信矩阵中。

#### 提升检测精度，为每个内存页增加额外的页错误

正常情况下，每个进程只在第一次访问page的时候产生一次页错误。CDSM产生额外的页错误。CDSM**定期迭代并行应用程序的页表，修改页表条目**。更新页表条目的方法有：1.通用方法是清除页的present位，在大多数支持分页的架构下都适用。2.在某些架构（x86\_64）下，可**更新页表的reserved bits保留位**，这样会产生一个易于解决的页错误。

**在任何情况下，都需要在更新页表条目后，从快表TLB中删除该页表条目**，因为人为增加的额外页错误不代表页表中缺失信息，且CDSM自身可以快速解决错误，无需内核例程的干预。由于处理简便，extra page fault导致的额外开销被降低。

### CDSM的实现

定义额外数据结构：一个hash table记录进程ID和页错误物理地址；一个通信矩阵matrix.

1. **CDSM处理过程和要点：**
2. 更改操作系统关于页错误的中断控制：首先判断page fault是额外添加的还是本身系统产生的，若是extra page fault，CDSM自己处理；若不是extra，OS处理。
3. 记录page fault信息。根据访问的物理内存地址和自定义的块大小，计算块号。输入块号和进程ID数据到hash table中。
4. 若不是第一次访问该内存地址，且由不同线程完成，统计为通信，更新matrix条目。
5. 用一个内核线程实现生成extra page fault的过程。定期**更改某个进程的页表，清除掉其中某个其他进程已访问过的内存地址所在页的页表信息中的present位。**这样这个进程下次访问这个内存地址时就会产生page fault.
6. CDSM处理页错误：移除对页表的更新。接着CDSM直接返回应用程序，不调用OS页错误控制器。
7. **CDSM实现为x86\_64架构的内核模块。下面是实现的4个部分：**
8. 定义如上描述的数据结构。hash table的大小正比于物理内存大小，每个GB的主存约有2.3MB的内存用于创建的哈希表。matrix大小正比于进程/线程数，可以动态扩展，每个矩阵元素占4字节，共申请1MB的空间用于通信矩阵就肯定够用了。
9. 在**页错误的中断处理程序中**，**添加kprobe**更改掉linux内核默认的页错误的中断处理过程。在更改后的中断处理程序中，1）判断page fault是否自己处理；2）调度内核工作队列中的一个工作项，工作项执行统计page fault信息的工作；3）调度后程序继续执行。

**将页错误处理与通信统计分开执行**，原因是：1.应用程序需要在页错误处理后就执行，以减少额外开销，这时候可能还没统计通信。2.使得通信检测统计过程可以在程序运行时的非关键时间内执行（如**程序由于锁同步而停止的时间**）。

1. 增加额外的页错误，内核线程执行该过程。只允许添加extra page fault在**可进行通信的虚拟内存区域**（即内核线程只更改这个区域内的页表项）。在多进程程序中，Linux的虚拟内存子系统将可通信的区域标记为shared；在多线程并行程序和混合并行程序中，线程共享相同的虚拟地址空间，因此可通信区域更大。不管哪种并行程序，extra page fault只允许在数据段，而非代码段。

确定合适的虚拟内存区域后，**内核线程清除相应页表项的reseved位以及快表中的相应项**。为控制开销，内核线程控制extra page fault占应用程序每秒pages访问次数的百分比。

1. 额外开销由：1.引入extra page fault（额外的上下文切换、对页表的周期性访问和更改），可减少页错误处理的计算量降低开销。2.通信检测中访问哈希表和通信矩阵的时间，时间复杂度很固定。
2. **CDSM评价：**

实现在内核模块中，仅仅需要超级用户权限，并不需要重新编译内核、重启机器。1.执行内核级的映射要优于执行在用户空间的映射（运行时库、直接在应用程序上）。因为如果有多个应用程序在执行，用户级的映射只有系统状态的部分视图，因此映射决策可能与其他产生冲突。2.内核级的映射支持多种并行编程模型。若要支持使用用户级线程的编程模型（charm++），需要扩展CDSM，把用户级线程到内核级线程的对应通知给内核。若用户级库有映射机制，CDSM可以提供通信矩阵等信息。

### 映射计算

建立模型。两个无向图，一个通信图，一个层次结构图。通信图中顶点表示进程/线程，边表示通信量；层次结构图中，顶点表示不同级别的内存和处理单元，边表示内存、处理单元之间的互连。

计算映射使用Scotch mapping library[1]的双递归二分算法，6.0版本。（更短的算法执行时间，映射32个进程<1ms，时间复杂度为O(N^3)）若进程数更多（>1000），可使用其他算法如Treematch或Scotch算法的并行版本。

Scotch算法无缝支持执行的进程/线程数多于或少于体系结构可并行的进程/线程数。也不限制每核仅1进程/线程。

[1] F. Pellegrini, Static mapping by dual recursive bipartitioning of process and architecture graphs, in: Scalable High-Performance Computing Conference (SHPCC), 1994, pp. 486–493.

### CDSM的操作过程

1. **monitor程序执行。**monitor内核函数do\_fork(), do\_exit()，检测应用程序和进线程的开始和结束。程序开始，CDSM申请allocate哈希表、矩阵；install控制器检测page fault；创建内核线程等。程序结束，所有进程退出，CDSM删除控制器，释放占用内存，终止内核线程。
2. **增加page fault.** extra page fault比例控制在占应用程序每秒访问page的1%，这个比例保证了更多数量的page fault并未导致计算错误或开销增大。页错误线程每10ms执行一次遍历页表，达到限制或到页表尾部终止遍历；下次迭代，线程从前一次遍历结束的位置插入页表更改，创造extra page fault.
3. **周期性计算mapping.** 每次计算使用当前的matrix. 可以动态调整映射间隔，如果当前计算出的mapping与上一次没变化，将映射的时间间隔增加50ms；若与上一次不一样，映射时间间隔/2 . 控制时间间隔在50ms-2s之间，控制额外开销+良好地应对通信变化。
4. **进线程迁移。**使用内核函数**sched\_setaffinity()**迁移进/线程，重新设置分配的核。最后引入aging技术，避免不必要的迁移。

## 23. A Locality and Memory Congestion-aware Thread Mapping Method for Modern NUMA Systems

### 背景介绍

NUMA系统中，最大化局部性会降低程序的性能，因为这会增加内存控制器上的数据流量拥塞。

一些研究仅通过分析并行程序的线程间的空间通信性能考虑基于访存局部性和访存平衡的线程映射。

考虑分析线程间的时间通信性能，减少内存拥塞很有必要。内存拥塞仅发生在不同处理器上的多个线程同时访问同一个节点上的内存。

提出一个分析时间空间通信性能，考虑局部性和内存拥塞的映射方法。提出一个线程间隐式通信检测的方法，一个分析时空通信性能的加权数据聚类的方法，以及一个利用分析结果优化线程到核映射的方法。

### 映射方法（静态）

把两个不同线程对同一cache行的两次连续访问视为两个线程间的通信事件。

第一步，运行目标程序，使用基于二进制插桩的tracing工具获取按时间排序的通信事件。以64B大小的内存块粒度检测线程的访存。检测到的每次通信事件记录信息包含独立的通信事件ID，线程对的IDs，通信事件的时间戳（us为单位）。

第二步，通过加权k-means聚类方法识别经常同时进行通信的线程组。由于不同线程的大量并发通信意味着有更高的内存拥塞风险，因此使用通信时事件的数量作为聚类的权重。给定通信时间戳的集合和类集合，聚类准则函数是，其中k个聚类C1-Ck，每个聚类中表示类中时间戳的平均值，是t时刻的通信数量。聚类得到的是按时间分类的时间段，根据贝叶斯信息准则Bayesian Information Criterion确定最优的类的数量。

最后一步，计算映射，使用算法Decongested Thread Locality (DeT-Loc). 使用工具Hwloc获取拓扑信息（节点数，每个节点处理器核心数，每个核心物理实体数）。算法利用聚类结果生成通信事件的线程对构成的组，组内每个线程对所属的通信事件都在同一个聚类里，但是同一个线程对IDs可能在多个组中出现，因为他们可能在程序的多个时间点通信。首先，计算一个线程对的负载，. 接着计算一个组里的线程对的负载，. 映射策略依次选择具有最大的组和组里具有最大的线程对。把一个线程对的线程映射到同一个节点上，把同组中的不同线程映射到不同节+点上。不可避免地，想避免拥塞就会增加一些远端访存。