# 线程级动态映射方案

## 引言

随着高性能计算平台的不断发展和并行应用程序规模的逐步扩大，如何匹配计算任务和计算资源已经成为高性能计算中的一大难题。映射优化成为解决此问题的主要方法。因此，如果我们能够为以大型CFD程序为例的这种复杂并行应用开发出一种映射机制，那么我们就可以显着提高程序性能，甚至降低能耗。

进一步地，为了提升优化效果，除了可以在进程间使用MPI并行优化外，还可以使用更细粒度的OpenMP多线程并行优化。但是在OpenMP编程模型中，线程之间通过共享内存进行数据交换和共享，这使得若程序在NUMA架构计算机中运行，会产生通信开销不均匀的问题。如图1所示，倘若两个需要通信的线程分布在共享同一个高速缓存的CPU核心中，则他们之间只需要读写该共享缓存上的数据就可以完成通信，且访存操作对缓存命中率的影响不大；但如果两个需要通信的线程位于不同处理器上，则其中一个线程要读取另一个线程所在处理器的局存中的数据，即两个线程通过片上互连通信，这样的开销是巨大的。因此，进一步在线程这一级研究映射优化可以有效改善线程间通信的不均匀性，也为研究针对大型应用程序的总体映射优化增加了层次性和弹性。



图1 NUMA架构多线程共享内存通信效率不平衡

传统映射优化方法大多是静态的，尽管这种方式利用众多性能指标把各种可能的映射方案都比较得很全面，而且对特定的应用程序优化效果显著。但静态映射方法只能满足核数较少的情况，并且需要预先多次运行程序，因此总体上讲静态方法的通用性，可扩展性和可移植性较差。因此基于多线程并行应用程序，如果我们设计一种运行时的动态映射优化方案，能灵活适应程序的计算操作与访存操作在运行时动态变化的特点，则既能消除节点内部线程间通信的不平衡，又能使程序计算任务在运行时很好地匹配系统的计算资源和存储资源。

因此我们针对大型流体机械并行应用程序（rotor35）的OpenMP程序，在较细粒度的线程级设计并实现了针对众核计算机架构的CPU节点上线程到CPU核心的映射机制。在本项目“面向E级计算机的大型流体机械并行计算软件系统及示范”中，本研究涉及完成的任务目标是 “面向E级计算机系统的分层弹性映射机制”。

## 2. 主要创新

### 2.1动态线程到核映射机制

#### 2.1.1框架描述

我们设计并实现了一种动态的线程到核的映射机制，其基本流程和基本模块如图2和图3所示。



图2 动态线程映射机制流程图

图3 动态线程映射机制模块图

映射机制的执行过程是：

程序开始运行时，映射机制持续检测线程的创建。一旦程序运行于多线程环境，映射机制持续执行通信量检测，即下面流程中的步骤a；同时以一定时间间隔周期性地重复依次执行分组计算和线程的绑定，即下面流程描述中的步骤b和步骤c. 如此运行，直至检测到所有线程退出，程序终止或程序退出多线程运行环境。步骤a-c的具体内容描述如下。

a. 程序在多线程运行时的环境下，检测并统计线程间通信量。使用基于缺页错误的通信检测方法，在运行时持续地检测每个线程产生的缺页错误信息，记录一段时间内不同线程在相同内存页上产生的缺页错误。并维护一个大小为线程总数的通信量矩阵（其中，表示程序运行的线程总数，矩阵元素(,) ()表示线程和线程的通信量数值），持续性地将一段时间内在相同页面产生缺页错误的不同线程之间计为一次通信过程，以此更新通信量矩阵的值。

b. 根据步骤a中维护的通信量矩阵，计算线程的分组。使用我们提出并设计实现的LHMapping算法，对所有线程进行分组，分组结果将决定哪个线程映射到哪个CPU核心中。算法输入表示线程间通信量的矩阵*A*，以及程序运行所在平台的硬件对象拓扑树（Hwloc工具提取得到），从拓扑树的最底层开始，逐层进行分组，每一层的分组都采用了贪心策略，根据通信量矩阵值尽量将互相之间具有较大通信量的线程分在同一组中。算法输出用*map*数据结构表示的分组结果。组数及组内元素数量由系统硬件架构决定。

c. 根据线程分组结果，执行映射。线程分组结果表示出了线程与CPU核心的一一对应关系。因此直接依次调用Linux内核函数sched\_setaffinity()为每个线程设置其对应的CPU亲和度，暂时将线程绑定到对应的CPU核心上。在之后的一段时间内，系统不会对程序运行中的任何线程进行迁移。直至下一次分组计算的结果产生，才再一次重新执行线程的绑定。

#### 2.1.2具体细节

1. **线程间通信量检测与统计**

Linux操作系统内核提供了Kprobe函数组，可以实时检测出程序运行时系统中的缺页中断。动态映射机制需要记录检测到的每一次发生缺页错误的线程ID和内存页的物理地址。

此外，通信量检测检测机制会引入额外的缺页错误。我们周期性地更改程序所在进程页表的某一个页表项，若页表项中present位为1，则将其改为0，使该页表项对应的内存页失效。这样如果下次有线程需要访问这一部分的数据，就会在这个页面产生额缺页中断。引入的额外缺页错误尽管使系统在程序运行时增加了一部分处理中断的开销，但由于产生了更多不同线程访问相同内存数据的情况，因此整体上增加了线程间通信量统计的精确度。

在统计时，我们定义并初始化一个哈希表。哈希表项对应一个将物理地址空间按照一定粒度进行划分后的内存块，用除留余数法作为哈希函数。每个哈希表项为一个队列，队列里每一项记录程序运行时该内存块上发生缺页错误的不同线程ID. 队列大小可以是一个变量，它可以控制通信量统计的精度。在我们的机制中设置队列大小为4.每当一个新的线程ID入队，就将该新线程与仍在队列中的其他线程计为发生一次通信事件。我们用这种方式持续记录，随时更新通信量矩阵。

通信检测机制检测出的缺页错误，包含了两种情况。其一是操作系统本身产生的缺页；其二是检测机制额外引入的缺页错误。前者的中断恢复过程由操作系统本身的中断处理程序完成；后者的由检测机制自行完成。自定义的中断处理程序操作是：把发生缺页错误的内存页的数据给需要读取它的线程，并恢复该页的页表项。中断处理完毕后，系统恢复，程序得以继续执行，而统计通信量的过程可以与程序本身并行执行。

1. **通信量矩阵的处理**

为了提升整体映射优化效果，我们需要在每一次进行分组计算后，对矩阵*A*做一个处理。即将矩阵中的所有元素值*a*(*i*,*j*) (*i*乘以一个系数，我们称之为削减系数. 由于通信量矩阵的统计过程贯穿线程的整个生命周期，因此这样处理减少了较早的线程间通信量数值对后续通信量矩阵数据的影响，进而减少了对分组计算精度的影响。

为了方便计算，我们实现的映射机制暂，避免了更多的额外开销。即令

1. **周期性控制**

我们的动态线程到核映射机制使用周期性方法依次执行分组计算模块和映射执行模块。因此相邻两次分组计算和映射执行过程中间设置一个时间间隔，记为*interval*. 为了在尽可能提升优化效果同时避免过多额外开销，*interval*的取值必须有一个合理的范围。暂设初始的. 对相邻两次分组计算，若后一次的分组结果与前一次相比结果一致或仅有很小偏差，则说明程序的运行特性和线程通信模式没有发生明显变化，则可以考虑将*interval*增加50*ms*.反之说明线程间通信模式变化很大，需要及时调整，故使*interval*减少50*ms*. 在程序多线程的运行时，始终保持*interval*大于50*ms*且小于1*s*.

1. **映射的执行**

我们通过CPU亲和性设置实现将计算好的线程与CPU核心一一对应的关系实现在系统中。具体映射执行模块的步骤如下。

1. 输入分组计算结果*map*，它是一个拓扑树；初始化变量*i*=0.
2. 访问*map*的最底层第*i*个叶子节点，它保存了线程ID与具体的CPU核心ID的对应信息。选取编号为*tid[i]*的线程，其对应的CPU核心编号为*cpu[i]*.
3. 调用Linux内核定义的宏void CPU\_ZERO(cpu\_set\_t *\*mask*)，清除系统或用户之前设置的CPU位掩码。
4. 调用Linux内核定义的宏void CPU\_SET(int *cpu[i]*, cpu\_set\_t *\*mask*)，将编号是*cpu[i]*的CPU核心加入CPU集中，*mask*关联掩码第*cpu[i]*位被设置为1.
5. 可以选择调用Linux系统多线程接口库中的函数int pthread\_setaffinity\_np (pthread\_t *tid[i]*, size\_t *cpusetsize*, const cpu\_set\_t *\*mask*)，把线程*tid[i]*限制在步骤3和4中设置的*mask*所代表的CPU集中运行。或者直接调用进程的CPU亲和性设置函数int sched\_setaffinity(pid\_t *pid*, size\_t *cpusetsize*, cpu\_set\_t *\*mask*)进行相同的设置，这里*pid*使用线程的编号*tid[i]*. 至此完成对线程*tid[i]*的核绑定。
6. 令*i* = *i* + 1. 重复步骤2-5，直至所有线程完成绑定，映射执行完成。
7. **映射机制的实现**

经过分析我们最终将所设计的映射优化机制实现为Linux操作系统的内核模块，并装载入内核。只要在加载过线程动态映射优化的内核模块的操作系统中运行多线程应用程序，系统就会执行我们设计的映射优化机制。

由于我们的研究使用了不同的平台进行性能测试，因此在程序移植时需要保证多个平台的操作系统具有相同内核版本。本研究在实验时使用内核3.10.0版本。

在超级用户模式下，首先执行 “make” 命令编译，得到一个 “.ko” 目标文件；再执行 “insmod” 命令，将内核模块装载入内核中。

### 2.2映射分组算法LHMapping

由于已有映射计算的开源包Scotch Mapping Library所提供的算法复杂度较高，不适合本文提出的动态映射机制采用。因此我们提出并实现了LHMapping算法（Loop-based Hierarchical Mapping Algorithm），它是一种可以保持原有映射精度且计算开销更小的算法。LHMapping算法沿用了Scotch Mapping Library中根据体系结构拓扑树进行多层分组的计算方法。整体上在分组时采用贪心策略。算法输入通信量矩阵的数据和系统硬件信对象的拓扑树信息。输出一个表示分组结果的多维数组。

算法全部过程用伪代码表示如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm 1: *Top-Level* Algorithm** | | | |
| **Input:** *CommMatrixInit[][]*, *nThreads* | | | |
| **Output:** *map[]* | | | |
| **Local Variables:** *nElements*, *i*, *nGroups*, *rootGroup []*, *CommMatrix[][]*, *groups[]*, *pregroups[]* | | | |
| **Global Variables:** *HardwareTopoRoot*, *nlevels*, *nObjects[]* | | | |
| **begin** | | | |
|  | **for** *i* = 0 ; *i* < *nThreads* ; *i* = *i* + 1 **d**o | | |
|  |  | groups[i].nElements = 0; | |
|  | *nElements* = *nThreads*; | | |
|  | *CommMatrix* = *CommMatrixInit*; | | |
|  | **for** *i* = 1 ; *i* <= *nlevels* ; *i* = *i* + 1 **do** | | |
|  |  | *pregroups* = *groups*; | |
|  |  | *[nGroups*, *groups[i]]* = *GenerateOneLevel*(*CommMatrix*, *nElements*, | |
|  |  | *nObjects[i]*, *i*, *pregroups*); | |
|  |  | **if** *i* < *nlevels* **then** | |
|  |  |  | *CommMatrix* = *UpdateMatrix*(*CommMatrix*, *groups*, *nGroups*); |
|  |  | *nElements* = *nGroups*; | |
|  | *rootGroup.nElements* = *nElements*;  **for** *i* = 1 ; *i* < *nElements* ; *i* = *i* + 1 **do** | | |
|  |  | *rootGroup.element[i]* = *groups[nlevels][i]*; | |
|  | *GroupsToTopology*(*HardwareTopoRoot*, *rootGroup*, *map*); | | |
|  | **return** map; | | |
| **end** | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm 2: *GenerateOneLevel*** | | | | |
| **Input:** *CommMatrix[][]*, *nElements*, *nobject*, *level*, *pregroups[]* | | | | |
| **Output:** *nGroups*, *groups[level]* | | | | |
| **Local Variables:** *chosen[]*, *elpergroup*, *nel*, *leftover*, *newgroup*, *i*, *j* | | | | |
| **begin** | | | |
|  | **if** *nElements* > *nobject* **then** | | |
|  |  | *nGroups* = *nobject*; | |
|  | **else** | | |
|  |  | *nGroups* = *nElements*; | |
|  | *elpergroup* = *nElements*/*nGroups*; | | |
|  | *leftover* = *nElements* % *nGroups*; | | |
|  | **for** *i* = 0 ; *i* < *nElements*; *i* = *i* + 1 **do** | | |
|  |  | *chosen[i]* = 0; | |
|  | *j* = 0; | | |
|  | **for** *i* = 0 ; *i* < *nElements* ; *i* = *i* + *nel* **do** | | |
|  |  | *nel* = *elpergroup*; | |
|  |  | **if** *leftover* > 0 **then** | |
|  |  |  | *nel* = *nel* + 1; |
|  |  |  | *leftover* = *leftover* − 1; |
|  |  | *newgroup* = *GenerateGroup*(*CommMatrix*, *nElements*, *nel*, *chosen*, *pregroups*); | |
|  |  | *newgroup.elements* = *nel*; | |
|  |  | *newgroup.id* = *j*; | |
|  |  | *groups[level][j]* = *newgroup*; | |
|  |  | *j* = *j* + 1； | |
|  | **return** *[nGroups*, *groups[level]]*; | | |
| **end** | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 3: *GenerateGroup*** | | | | | | |
| **Input:** *CommMatrix[][]*, *nElements*, *nel*, *chosen[]*, *pregroups[]* | | | | | | |
| **Output:** *newgroup* | | | | | | |
| **Local Variables:** *i*, *j*, *m*, , | | | | | | |
| **1** | **begin** | | | | | |
| **2** |  | **for** *i* = 0 ; *i* < *nel*; *i* = *i* + 1 **do** | | | | |
| **3** |  |  | = −1; | | | |
| **4** |  |  | **for** *j* = 0; *j* < *nElements*; *j* = *j* + 1 **do** | | | |
| **5** |  |  |  | **if** *chosen[j]* = 0 **then** | | |
| **6** |  |  |  |  | = 0; | |
| **7** |  |  |  |  | **for** *k* = 0; *k* < *i*; *k* = *k* + 1 **do** | |
| **8** |  |  |  |  |  | = + *CommMatrix[j][newgroup.element[k]]*; |
| **9** |  |  |  |  | **if** > **then** | |
| **10** |  |  |  |  |  | = ; |
| **11** |  |  |  |  |  | *m* = *j*; |
| **12** |  |  | *chosen[m]* = 1; | | | |
| **13** |  |  | *newgroup.element[i]* = *pregroups[m]*; | | | |
| **14** |  | **return** *newgroup*; | | | | |
| **15** | **end** | | | | | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 4: *UpdateMatrix*** | | | | | | | |
| **Input:** *CommMatrix[][]*, *groups[level]*, *nGroups* | | | | | | | |
| **Output:** *newCommMatrix[][]* | | | | | | | |
| **Local Variables:** *i*, *j*, *k*, *t*, *W* | | | | | | | |
| **1** | **begin** | | | | | | |
| **2** |  | **for** *i* = 0; *i* < *nGroups* − 1; *i* = *i* + 1 **do** | | | | | |
| **3** |  |  | **for** *j* = *i* + 1; *j* < *nGroups*; *j* = *j* + 1 **do** | | | | |
| **4** |  |  |  | | *W* = 0; | | |
| **5** |  |  |  | | **for** *k* = 0; *k* < *groups[level][i]*; *k* = *k* + 1 **do** | | |
| **6** |  |  |  | |  | **for** *t* = 0; *t* < *groups[level][j]*; *t* = *t* + 1 **do** | |
| **7** |  |  |  | |  |  | *W* = *W* + |
|  |  | |  |  | *CommMatrix[groups[level][i].element[k].id]* |
|  |  | |  |  | *[groups[level][j].element[t].id]*; |
| **8** |  |  |  | | *newCommMatrix[i][j]* = *W*; | | |
| **9** |  |  |  | *newCommMatrix[j][i]* = *W*; | | | |
| **10** |  | **return** *newCommMatrix*; | | | | | |
| **11** | **end** | | | | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithm 5: *GroupsToTopology*** | | | | | | |
| **Input:** *HardwareRoot*, *RootGroup*, *map* | | | | | | |
| **Local Variable:** *i* | | | | | | |
| **1** | **begin** | | | | | |
| **2** |  | **if** *HardwareRoot.type* = *CPUCore* **then** | | | | |
| **3** |  |  | **for** *i* = 0; *i* < *rootGroup.nElements*; *i* = *i* + 1 **do** | | | |
| **4** |  |  |  | | | *map[rootGroup.element[i].id]* = *HardwareRoot.id*; |
| **5** |  | **else if** *HardwareRoot.shares* > 1 **then** | | | | |
| **6** |  |  | **for** *i* = 0; *i* < *rootGroup.nElements*; *i* = *i* + 1 **do** | | | |
| **7** |  |  |  | | *GroupsToTopology(HardwareRoot.linked[i]*, *RootGroup.element[i]*, *map*); | |
| **8** |  | **else** | | | | |
| **9** |  |  | | *GroupsToTopology*(*HardwareRoot.linked[0]*, *RootGroup*, *map*); | | |
| **10** | **end** | | | | | |

假设L是分组的层数，E表示某一层待分组元素的总数，n表示并行程序中待分组的线程总数。算法的时间复杂度可以表示为

因此，LHMapping算法的时间复杂度是.

在分组过程中，只有保存分组结果的拓扑树和通信量矩阵占用存储空间，因此算法的空间复杂度为.

## 3. 实验测试

为测试上述动态线程映射方法的有效性，目前在80核的多核集群上，对单流道轴流压气机转子数值模拟程序的多线程并行版本进行了实验。程序分别创建6, 12, 24, 36个线程，表示程序在80核集群上至少跨1, 2, 3, 4核运行。分别使用加载实现动态线程映射的内核模块前和加载后的linux系统进行实验，比较使用映射机制前后程序的运行时间，结果如图4所示，与映射优化前的程序运行时间相比，进行映射优化后，程序性能提升幅度不大，原因是轴流压气机转子数值模拟程序通信过程较为平均，但在线程数较多的36线程，仍有明显的总体运行时间的减少，运行效率提高约5.38%.  


图4 不同线程数的rotor35-omp程序在执行映射机制前后运行时间的对比

接着，测试了LHMapping算法本身的性能。使用算法运行时间和算法指导映射后的精确度两项指标评估LHMapping算法本身的性能。并将LHMapping算法与Scotch Mapping Library分组算法进行对比。先运行一次目标应用程序并同时进行线程间通信量统计，得到应用程序总体的通信量矩阵。随后将通信量矩阵和体系结构硬件对象的拓扑信息输入两个算法，分别运行两种映射分组算法得到分组结果，并记录运行时间。



图5 LHMapping算法和Scotch算法的计算时间对比结果

在全部线程数规格的程序中，LHMapping算法的运行时间都比Scotch短2-4倍。这说明我们提出的LHMapping算法具有更快的分组计算效率。且根据图中的曲线，在线程数不多的情况下，两种算法的运行时间都随着线程数的增加呈现接近指数型增长的趋势。这符合我们时间复杂度分析的结果。

此外，为了评估LHMapping算法的精度，证明算法的有效性，我们引入映射精确度的指标，称为*MappingAccuracy*. 它是TreeMatch中提出的一种评价映射精度的指标。其公式为：

其中，*n*是线程的数量，*Comm* [*i*][*j*]是线程*i*和线程*j*之间的通信量值，而*latency*[*map*[*i*]][*map*[*j*]]表示线程*i*映射到的核心与线程*j*映射到的核心之间的数据传输时延。*latency*的值可以通过Imbench中提供的程序计算得到。理论上，映射精确度的值越高，映射的效果越好。我们选择对64线程SP-OMP程序，根据两种映射算法的分组结果，分别计算他们的映射精确度*MappingAccuracy*. 其结果如表5‑2所示。

表 5‑2 分组算法的映射精确度对比结果

|  |  |
| --- | --- |
| 分组算法 | MappingAccuracy |
| LHMapping |  |
| Scotch |  |

由表中数据可知，LHMapping算法分组的映射精确度略低于Scotch Mapping Library中的算法；但二者处于同一数量级，相差不大，因此对映射结果没有实质性的影响。经过对比评估，LHMapping算法整体上具有比Scotch算法更好的性能。

随后又在C80平台上，对NAS Parallel Benchmark基准测试程序集的OpenMP实现版本的程序，进行了映射机制的优化效果测试。映射机制中使用我们提出并实现的LHMapping算法，与使用Scotch映射算法和OpenMP提供的Compact映射模式进行对比，比较执行映射前后程序运行时间和Cache失效率的性能增益情况，如图6所示。



图6 映射优化机制在C80上对NPB-OMP程序优化效果的对比

整体上看，使用LHMapping算法的映射优化机制对SP, BT, CG, LU, MG等程序具有明显的优化效果，对程序运行时间的加速最高达到约25.4%，Cache misses的最高减少约23.5%.

此外，使用LHMapping算法的映射优化机制对于通信量不太大，通信与计算比较平衡的FT，LU，MG等程序的优化效果更好。对于完全没有处理器间通信行为的计算密集型程序EP，映射机制没有优化效果，反而由于引入了额外开销使程序性能变差。对于全交换通信程序IS，映射机制几乎没有优化效果，事实上如果对此类应用程序针对通信不平衡问题进行映射优化，均没有很好的优化效果。

## 4. 结论

本研究在OpenMP多线程并行一级，为解决线程间数据访问与共享不平衡问题，设计并实现了融合线程间通信量统计、周期性线程分组计算、线程的CPU亲和度设置等的线程到核的映射优化机制。并为进一步提升映射优化的效果,提出了基于循环层次化分组的映射分组算法LHMapping。映射优化机制整体上改善了线程间通信不平衡的问题，减少了由于数据分布不均等访存问题引起的Cache失效，并有效缩短了并行程序的运行时间。

上述研究为后续利用E级计算机不同层次的计算资源，实现节点之间低交互、节点内部高效传输的通信机制提供了设计思路。未来我们将尝试开发多线程动态映射机制的可扩展性，并从多个维度对应用程序总体性能进行优化。