EagerMap: A Task Mapping Algorithm to Improve Communication and Load Balancing in Clusters of Multicore Systems

第二条 2019 年 3 月
Doi: 10.1145 / 3309711

4 位作者，包括:

Matthias Diener
University of Illinois, Urbana-Champaign
61 出版物 400 引文 61 出版物 361 引文
SEE PROFILE

Laércio Lima Pilla
Laboratoire de Recherche en Informatique
SEE PROFILE

Philippe Olivier Alexandre. Navaux
Universidade Federal do Rio Grande do Sul
362 篇出版物 1,538 引文
SEE PROFILE

Eagermap:一种改进多核系统集群通信和负载均衡的任务映射算法

爱德华多 ·克鲁兹，马蒂亚斯 迪纳，拉里西奥 利马 ·皮拉，菲利普 纳沃克斯

Eagermap:一种任务映射算法，用于改进多核系统集群中的通信和负载平衡

1，Matthias Diener2，Laercio l. Pilla3，and
菲利普 o a 纳沃克斯 4

巴拉那联邦研究所(巴西，帕拉那维)
伊利诺伊大学香槟分校{尚佩恩，美国
3Univ.GrenobleAlpes，Inria，法国国家科学研究中心，格勒诺布尔
格勒诺布尔
4.南里奥格兰德州信息学研究所
巴西阿雷格里港

摘要

任务之间的通信和负载不平衡已经被认为是并行应用程序性能和能量利用率的主要挑战。改善通信的一个常见方法是增加其局部性，也就是说，减少数据传输的距离，优先考虑使用更快、更电子化的本地互连而不是远程互连。关于负载不平衡，核心应该执行类似数量的工作。在此上下文中需要解决的一个重要问题是如何确定任务到集群节点和核心的优化映射，从而提高总体局部性和负载平衡。本文提出了一种基于贪婪启发式算法的 EagerMap 算法来确定任务映射，该算法将应用程序通信模式与硬件层次结构进行匹配，同时考虑了任务负载。与以前的算法相比，EagerMap 速度更快，比例更好，并支持更多类型的计算机系统，同时保持相同或更好的任务映射质量。因此，EagerMap 是在各种现代并行体系结构上进行任务映射的有趣选择。

引言

优化并行应用程序的执行已经成为近年来的一个重要研究课题[5,19]。现代的并行系统由许多相互连接的集群节点组成，这些节点本身构成了一个具有深度内存阶层的并行共享内存机器。在这样的体系结构中，通信和负载不平衡对应用程序性能的影响比计算[39,18]更大。减少通信影响的一个常见方法是通过增加本地互连的使用和减少远程连接的使用来提高通信的本地性。负载不平衡可以

通过使用专用的负载平衡算法进行改进。这些优化可以在使用消息传递库(如 MPI[12])的并行应用程序中执行，也可以在使用共享内存范例(如 OpenMP 或 UPC[2,1])的应用程序中执行。

在这两种类型的并行应用程序中，并行应用程序的任务之间的通信性能都受到层次结构[43]的影响。通过共享高速缓存存储器或芯片内部互连的通信比处理器之间的通信快，因为芯片间互连较慢[35,14]。同样，机器内部的通信比集群或网格中节点之间的通信要快。网络速度也会变化，增加了通信延迟和带宽的差异。集群和网格带来了额外的挑战，因为每个节点都可以有一个不同的计算机配置。强烈通信的任务应该映射到层次结构中紧密相连的 PUs。为了改善负载平衡，我们需要分析并行应用程序的每个任务的负载，并以所有内核的负载相似的方式将它们映射到内核。在这种情况下，任务到处理单元(PUs)的映射对于并行应用程序的性能起着关键作用[44]。

在将任务映射到建筑结构的过程中需要四个主要步骤。第一步是检测任务之间的通信。在消息传递环境中，需要监视任务之间发送的消息。第二步是检测层次结构的拓扑，这也高度依赖于体系结构的类型。在共享内存架构中，可以使用 hwloc[11]之类的工具，而大多数集群环境都有供应商专用工具来管理拓扑。第三步是使用映射算法，结合通信、负载和拓扑信息，生成任务到 PUs 的映射。Nal 步骤是使用确定的映射执行应用程序，或者将任务迁移到它们分配的 PUs。

本文重点研究了第三步:映射算法。任务映射 ping 问题可以设计如下[14]。考虑两个图，一个表示并行应用程序，另一个表示并行体系结构。在应用程序图中，顶点表示任务，边表示任务之间的通信量。在体系结构图中，顶点表示机器组件，包括 cpu、缓存存储器、NUMA 节点、网络路由器、交换机和其他按层次结构组织的组件，而边表示链路的带宽和延迟。任务映射问题包括将应用图中的任务映射到体系结构图中的 cpu，使总通信开销最小，核心负载均匀。

确定最优映射的复杂性是 np 难[9]。由于任务和 cpu 的数量很大，为应用程序确定最优映射是不可行的。因此，启发式被用来计算最优映射的近似值。然而，现有的映射算法仍然存在较高的执行时间，因为它们是针对静态映射开发的，而且大多基于对图的复杂分析。这就降低了它们的适用性，尤其是对于在线地图，因为它们的开销可能会影响性能。

在本文中，我们提出了一种用于生成任务映射的电子算法 EagerMap。通过对通信任务进行强化分组，粗化了应用图。这种粗化遵循体系结构层次结构的拓扑。基于对应用程序行为的观察，我们提出了一种 ecient 贪婪算法

生成每一组的策略。它实现了高精度，比其他最先进的方法更快。粗化之后，我们将组图映射到体系结构图。

鹰图最初是在[15]中提出的。本文提出的主要扩展和改进措施如下。

原来的算法只支持对称树拓扑，这限制了它对共享内存机器的适用性。我们将其扩展到支持集群和网格，其中连接集群/网格的网络拓扑是任意的，每个计算节点的内部拓扑是对称树。

我们还设计了一个并行版本的 EagerMap，允许它更快地计算映射。

我们向 EagerMap 添加了负载平衡支持，它可以处理 oversub-edned 场景(比处理单元更多的线程/进程)。

相关工作

以往的研究在考虑通信的情况下对任务映射的影响进行了评估，结果表明任务映射可以使用多种硬件资源。在共享内存环境中，基于通信的任务映射减少了执行时间、缓存丢失和互连传输。在集群和网格环境中，将任务映射到同一个计算节点可以减少网络传输和执行时间[10,24]。通信成本也可以在虚拟环境中最小化[40]，证明它对云计算的重要性。

为了优化通信和负载均衡，提出了几种映射算法。大多数传统算法都是基于图划分的，例如 Scotch[38]。该算法基于分治思想，将进程子集递归地分配给进程子集。该算法将一组未处理的单元划分为两个不相交的子集，并调用图双划分算法将与单元相关的任务子集划分为两个子集。当双分割递归执行时，集合大小会减小，直到所有集合只有一个元素。Scotch 的一个缺点是，拓扑描述需要关于 cpu 之间通信延迟的信息，这些值在 nal 映射中有很高的意义。这对苏格兰威士忌来说是一个挑战，因为要确定其参数的准确值是很困难的。另一方面，EagerMap 不依赖于这些信息，这使得它更容易使用，并且可以产生更可靠的结果。其他遵循 Scotch 相同基本思想的工作有 Zoltan[20]，METIS[33,31]和 Chaco 在[22]中描述的工作也使用图划分对任务进行分组，但它使用贪婪技术将分组映射到拓扑结构。其中一些算法已经被并行化，例如 PT-Scotch[13]和 Par-METIS[32]。

Treematch[26]中使用了树表示法，这可以导致体系结构中更优化的算法以这种方式表示。然而，TreeMatch 中用于分组任务的算法具有指数级的复杂性，因为它为每个内存级别生成所有可能的任务组

等级。这可能导致不合理的映射计算时间取决于任务的数量。Treematch 的另一个局限性是，它不能计算大多数集群或网格的映射，因为它们的拓扑可能不用树表示。Treematch 还被移植到 Charm++environment[27]中，其中包含了对负载平衡的支持。[4]还为 Charm++提出了一个负载平衡器，重点是集成的 OpenMP 运行时。

Mpipip[12]是一个针对基于 mpi 的映射的框架和优化映射。Mpip 最初将每个任务映射到一个随机的 PU。在每次迭代中，mpipip 选择交换 cpu 的任务对，以尽可能地降低通信开销。通信开销取决于选择的参数，可能与 MPI 进程之间传输的消息数量或数据量有关。这个过程重复多次，提高了每次迭代映射的质量，并且可以一直继续下去，直到没有任何收获。Mpippp 的一个主要缺点是它过于依赖初始随机映射，这对映射质量没有任何保证。由于这个原因，mpip 可能会生成错误的映射。其他基于在每个迭代重新定义初始映射的概念的工作是[8]和[30]。

我们以前的工作[14]使用 Edmonds 的图匹配算法[21]来计算映射，它解决了完全加权图的最大权完美匹配来生成映射。该方案的思想是多次调用图匹配算法来生成应该映射在一起的线程组。每次调用匹配算法时，它都会生成具有高度通信的线程对。因为它总是生成线程对，所以解决方案仅限于任务和 pu 的数量为 2 的环境。另一个问题是，所提出的技术必须针对特定数量的线程和体系结构进行编码，而这些线程和体系结构不能自动工作。

贪婪等级映射

Eagermap 接收三个输入:一个通信矩阵，包含每对任务之间的通信量、每个任务的负载以及体系结构层次的描述。输出 PU 执行各项任务的情况。为了表示体系结构层次结构，我们使用了一个树，其中顶点表示对象，比如 cpu 和缓存存储器，边表示它们之间的链接。在分析并行应用程序通信模式的基础上，采用电子贪婪策略进行任务分组。图 1 描述了几个并行基准测试套件[3,28,36,7]的不同的通信模式，我们使用第 4.2 节描述的方法获得了这些模式。我们在应用程序的通信行为中观察到三个基本特征，这些特征在电子映射策略中需要考虑:

通信行为有两种类型:结构化通信和非结构化通信。在具有结构化通信的应用程序中，每个任务与一组任务进行了更多的通信，例如将这些子组映射到层次结构中附近的 cpu 可以提高性能。在图 1 中，除 Vips 以外的所有应用程序都显示了结构化的通信模式。我们的 map-ping 算法用于处理结构化通信模式 be-

因为在使用非结构化通信的应用程序中，可能没有能够提高性能的任务映射。

在具有结构化通信模式的应用程序中，与并行应用程序中的任务总数相比，具有强内部通信的子群的规模通常较小。例如，在 CG-MPI(图 1b)的交流模式中，共有 64 个任务，其中 8 个任务子群进行了激烈的交流。每个子组之间的通信量远远高于不同子组之间的通信量。

在这一部分，我们详细地描述了 EagerMap，给出了一个实例，并讨论了它的复杂性。我们首先描述了该算法只考虑通信，然后将其扩展到兼顾通信和负载。

3.1EagerMap 算法的描述

该算法需要两个以前初始化过的变量:nLevels 和 execElInLevel。Nlevels 是 hier-archy 体系结构的共享层次的数量加上 2。此外，还需要创建一个级别来表示应用程序任务(级别 0)，另一个级别来表示处理单元(级别 1)。

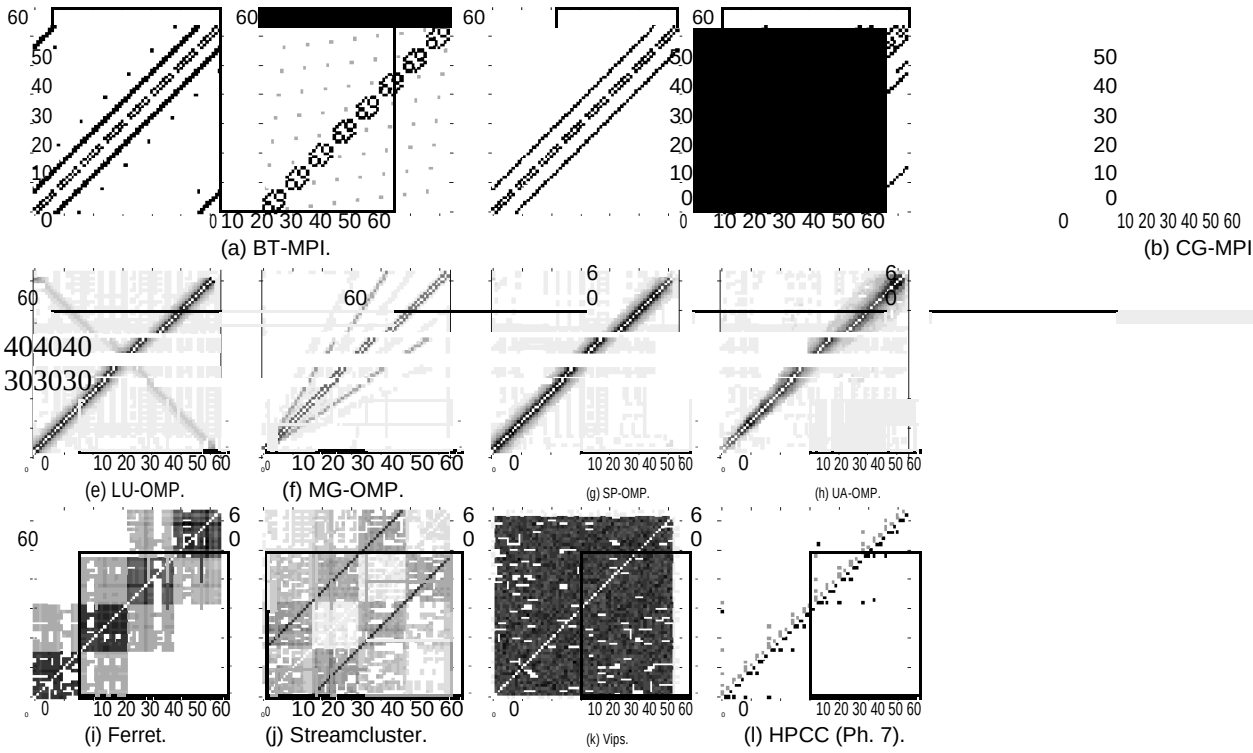


图 1:由 64 个任务组成的并行应用程序的示例通信矩阵。轴表示任务 id。单元格显示任务之间的通信量。颜色较深的细胞代表更多的交流。通信矩阵的生成如第 4.2 节所述。

算法 1:MapAlgorithm:EagerMap 的顶级算法。

```
输入:commMatrixInit[][]，nTasks
输出:map[]
Localdata: nElements，i，nGroups，rootGroup，
          commMatrix [][]，groups []，previousGroups []
Globaldata: nLevels，execElInLevel []，hardware topologyroot
开始
2     for i  0 ; i<nTasks ; i          i+1 do
3       groups[i].id      i;
4       groups[i].nElements       0;
结束
6     nElements      nTasks;
7     commMatrix       commMatrixInit;
8     for i  1 ; i<nLevels ; i         i+1 do
9       previousGroups      groups;
        /* GenerateGroupsForLevel is implemented in Algorithm 2.                    */
10      [nGroups, groups]      GenerateGroupsForLevel(commMatrix, nElements, i,
          前 viousgroups，execElInLevel[i];
如果我的 nLevels-1 然后
          /* RecreateMatrix is implemented in Algorithm 4.                          */
12        commMatrix   RecreateMatrix(commMatrix, groups, nGroups);
结束
元素 n 组;
结束
元素 nElements;
1; i n elements; i i + 1 do
Rootgroup.elements [ i ] groups [ i ] ;
结束
      /*MapGroupsToTopology 在算法 5.*/中实现
Mapgroupstotopology (archTopologyRoot，rootGroup，map) ;
返回图;
结束
```

Execelinlevel 是一个具有 nLevels 位置的向量。没有使用 execElInLevel[0]。Execelinlevel[1]包含处理单元的数量。对于位置 i，比如 1inLevels，该值是各个体系结构层次结构级别上的硬件对象数量。硬件对象包括核心、缓存、处理器和 NUMA 节点等。例如，execElInLevel[2]可以是核心数量，execElInLevel[3]是上一级缓存的数量，execElInLevel[nLevels1]是 NUMA 节点的数量。由于体系结构层次结构的私有级别对于我们的映射策略并不重要，因此在准备 nlevel 和 execElInLevel 时，我们只考虑共享级别。

3.1.1 顶级算法

顶层映射算法如算法 1 所示。它计算体系结构层次结构中每个级别的 map-ping 。 Groups 变量表示正在处理的级别的元素组。 Previousgroupsvari-able 表示前一级别的元素组。首先，它用应用程序任务初始化组(第 2 行中的循环)。然后，它遍历体系结构层次结构上的所有级别，如第 8 行。在为一个级别生成任务组(第 10 行)(在 3.1.2 节中解释)之后，算法生成一个新的通信矩阵(第 12 行，在 3.1.3 节中讨论)。这个步骤是必要的，因为我们将每组任务视为映射的基本元素

算法 2:GenerateGroupsF 或 level:为体系结构层次结构的一个级别生成组。

输入:commMatrix[][]，nElements，level，previousGroups[]，avlGroups
输出:nGroups，groups[]
本地数据:chosen[]，elPerGroup，leftover，gi，inggroup，i，newGroup

```
开始
2     nGroups       min(nElements, avlGroups);
3     elPerGroup        nElements / nGroups;
4     leftover          nElements % nGroups;
5     for i  0 ; i<nElements ; i  i+1 do
6        chosen[i]       0;
结束
8     gi      0;
 0; i n elements; i i + inggroup do
10inggroup elPerGroup;
11 如果剩下 0 那么
12          inGroup       inGroup + 1;
剩下的-1;
结束
            /* GenerateGroup is implemented in Algorithm 3.                          */
15          newGroup       GenerateGroup(commMatrix, nElements, inGroup, chosen,
              previousGroups);
16          newGroup.nElements       inGroup;
17          newGroup.id       gi;
18          groups[gi]       newGroup;
19          gi       gi + 1;
结束
返回[n 组，组];
结束
```

在下一个层次上。

Groups 变量隐式生成一个组树。级别 0 代表任务。级别 1 表示任务组。级别 2 表示一组任务。换句话说，在每个级别上，通过粗化前一级别生成一个新的应用程序图。在第 8 行 nish 中的循环之后，groups 变量表示层次结构级别 nLevels1 并包含 nElements 元素。我们设置了 rootGroup 来指向这些最高级别的元素(第 17 行中的循环)。最后，算法将表示任务的树 rootGroup 映射到表示体系结构拓扑的树 archTopologyRoot。本程序在第 3.1.4 节中进行了说明。

3.1.2 为架构 Hier-archy 级生成组

在算法 2 中描述的 GenerateGroupsForLevel 算法处理给定体系结构层次结构级别的所有组的创建。它期望上至前一个处理级别的层次结构级别已经组合在前 viousgroups 中。组的最大数量是该级别的硬件对象的数量。属于每个组的元素的选择由 GenerateGroup 执行。

在算法 3 中，GenerateGroup 算法将表示大量相互通信的元素分组。对于分组，算法采用贪婪策略如下。第 2 行中循环的每次迭代都向组添加一个元素。元素表示的添加元素

算法 3:GenerateGroup:生成一组通信的元素。

输入:commMatrix[][]，totalElements，groupElements，selected[]，previousGroups[]

输出:组

Localdata: i，j，k，w，wMax，winners []，winner

开始

```
2     for i  0 ; i<groupElements ; i  i+1 do
3         wMax        -1;
4         for j            0 ; j<totalElements ; j  j+1 do
```

如果选择[j]0 然后

```
6                 w   0;
7                 for k     0 ; k<i ; k  k+1 do
8                     w      w + commMatrix[j][winners[k]];
```

结束

如果我们 wMax 那么

Wmax w;

赢家 j

结束

结束

结束

被选中的[获胜者]

胜利者[我]胜利者

Group. elements [i] previousGroups [winner] ;

结束

返回组

结束

相对于组中已有的元素，winner 变量提供了最大的通信量。选择的变量用于避免多次选择同一个元素。Generategroup 可以在第 4 行的循环中并行化，其中每个线程将并行计算它的本地优胜者，我们将在第 3.5 节中解释。

3.1.3 为架构层次的下一级计算通信矩阵

算法 4 中描述的 RecreateMatrix 将重新生成用于体系结构层次结构的下一级别的通信矩阵。新的通信矩阵具有 n 群的阶。它包含组之间的通信量。它是通过计算不同群体元素之间的交流量而得出的。

3.1.4 将组树映射到架构拓扑树

将组树映射到体系结构拓扑树的算法是 MapGroupsToT 操作系统，详见算法 5。它在体系结构层次结构的级别上执行递归。当递归停止条件达到体系结构拓扑的最低级别即处理单元(PU)(第 2 行)时发生。如果停止条件不是 fullllled，则算法已经知道每个级别的最大组数永远不会超过该

级别的硬件对象数，正如 GenerateGroupsF 或 level 中所解释的那样。因此，如果递归中体系结构层次结构的级别是共享的(第 6 行)，则算法只分配下一级别的一个硬件对象

算法 4:RecreateMatrix:计算下一级别的通信矩阵。

输入:commMatrix，groups[]，nGroups

输出:newCommMatrix[][]

本地数据:i，j，k，z，w

开始

```
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[i].nElements; k  k+1 do
6                   for z  0 ; z<groups[j].nElements; z  z+1 do
7                     w   w + commMatrix[ groups[i].elements[k].id ][
                      groups[j].elements[z].id ];
```

结束

结束

```
10        newCommMatrix[i][j]       w;
11        newCommMatrix[j][i]       w;
```

结束

结束

返回 newCommMatrix;

结束

算法 5:MapGroupsT/oT 操作系统:将组树映射到硬件拓扑树。

输入:hardwarobj，group，map[]

本地数据:i

开始

如果 hardwarobj.typeProcessingUnit，那么

```
3       for i  0 ; i<group.nElements ; i          i+1 do
4         map[ group.elements[i].id ]       hardwareObj.id;
```

结束

1 then

7 for i 0; i group.nmr elements; i i + 1 do

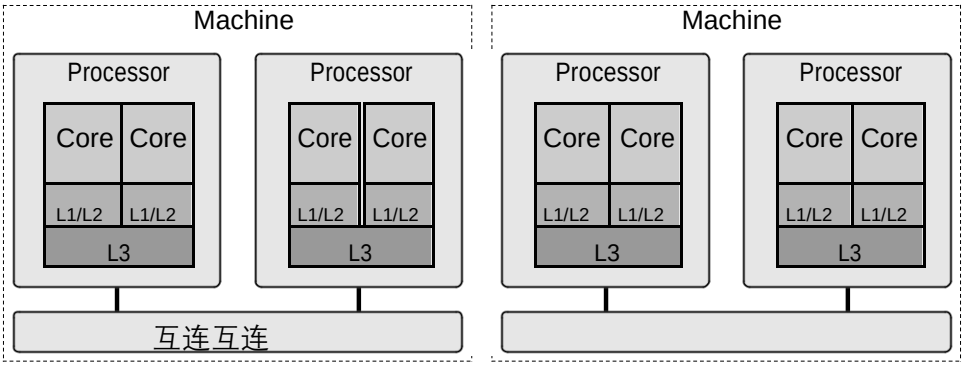
Mapgroupstotopology (hardwarobj.linked [i]，group.elements [i]，map) ;

结束

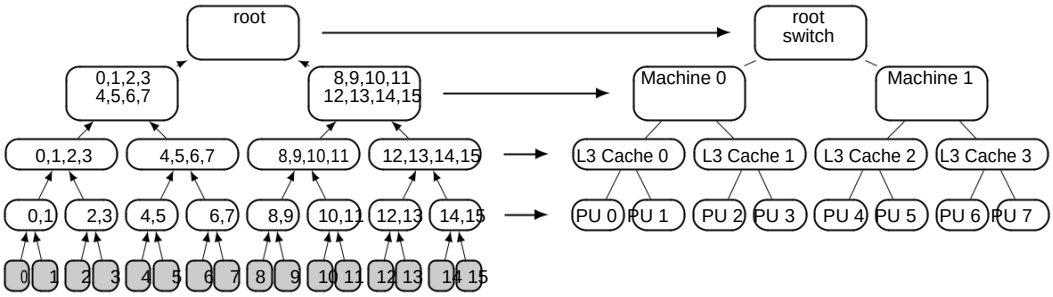
还有 10 个

Mapgroupstotopology (hardwarobj.linked [0]，group，map) ;

结束



(a)示例中使用的拓扑。



(b) Task group tree. (c) Architecture hierarchy tree.

图 2:在一个具有 8 个 cpu、2 个 cpu 共享的 L3 缓存、每台机器 2 个处理器和 2 台机器的体系结构中映射 16 个任务。

并递归地调用自身以获得下面级别的每个元素。
否则，该级别是私有的，不考虑用于映射(第行)。

3.2 多层次结构中的映射示例

为了更好地理解我们的映射算法是如何工作的，我们提供了一个例子，在一个由 2 台机器组成的集群中映射 16 个任务，每台机器由 8 个核心组成，l3 缓存由 2 个核心和每台机器的 2 个处理器共享。图 2a 示出了机器的层次结构，需要在图 2c 所示的拓扑树中进行转换。拓扑树可以使用 hwloc[11]之类的工具自动生成，也可以由程序员手动生成。私有级别，如 L1 和 L2 缓存，以及处理器，与第一级别无关。全局变量 nLevels 的值为 4，因为层次结构中有 4 个级别。因此，execElInLevel 有 4 个位置。位置 0 表示任务，未使用。位置 1 表示 cpu(本例中为核心)，其值为 8。位置 2 表示 L3 缓存，其值为 4。最后，位置 3 表示机器，它的值是 2。

对于算法 1(MapAlgorithm)，第 8 行中的循环执行 3 次。在第一次迭代中，它生成在每个核心中执行的任务组。由于任务的数量是 16 个，

核心的数量是 8 个，它生成 8 组，每组 2 个任务。通信矩阵的使用在第一

图 10

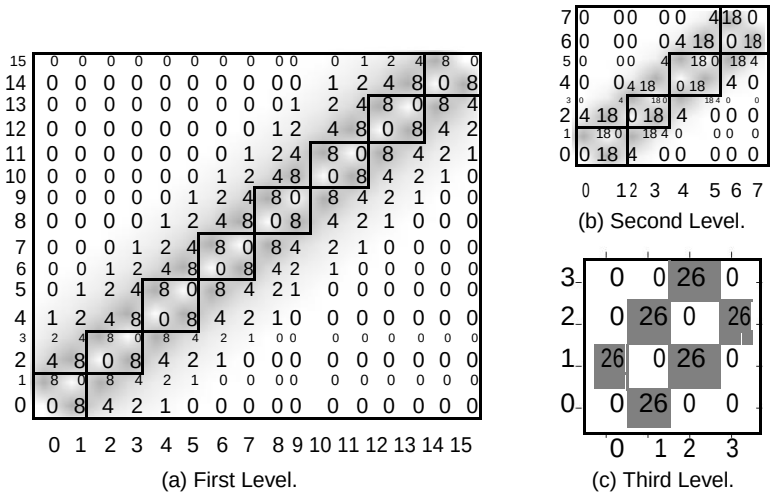


图 3:示例中使用的通信矩阵。在算法生成的任务组周围绘制一个框。

i	winners		winner	wmax	w_0	w_1	w_2	w_3	w_4	$w_{5\ 15}$
0			0	0	0	0	0	0	0	0
1	0		1	8	-	8	4	2	1	0
2	0	1	-	-	-	-	-	-	-	-

图 4:在我们的例子中生成第一个组时，GenerateGroup 算法的示例。

迭代如图 3a 所示。我们演示了如何在图 4 中生成第一组(生成组，算法 3)。当 i 为 0 时，赢家向量为空，任务 0 被选为组中的第一个。当 i 为 1 时，在赢家向量中，与任务交流最多的任务是任务 1。因此，任务 1 被选为获胜者并进入组。然后，GenerateGroup 算法返回一个由任务[0,1]组成的组，隐式存储在 group.elements 向量中。

GenerategroupsforLevel(算法 2)重复同一个过程，直到形成该级别的所有组。然后，图 3b 中的下一级通信矩阵由 reatematrix(算法 4)计算出来。Mapalgorithm 第 8 行中循环的第二次迭代行为类似，但是从前一次迭代中选择哪组任务共享每个 L3 缓存。因为组的数量是 8，并且有 4 个 L3 缓存，所以它生成了 4 个新组，每个组中有 2 个来自前一级的元素。

在 MapAlgorithm 第 8 行循环的第三次迭代中，它选择共享每台机器的任务组。使用了图 3c 中所示的通信矩阵。由于在前一次迭代中生成的组数为 4，并且有 2 台计算机，因此它生成了 2 个新的组，每个组包含 2 个元素。要完成组树，rootGroup 变量指向这两个组。然后，在 MapAlgorithm 的第 19 行中，图 2b 中所示的任务组树结束。

Mapgroupstot 操作系统(算法 5)将任务组顶点映射到 ver-

结构层次结构的顶点。映射从根节点开始。任务[07]映射到机器 0。任务[03]映射到 L3 缓存 0。任务[01]映射到 PU0(核心 0)。递归继续，直到所有的任务组都映射到一个 PU(核心)。在这个示例中，任务 x 和 x+1(其中 x 是偶数)将映射到核心 x2。

3.3EagerMap 的复杂性

为了分析鹰映射的复杂性，我们首先引入了方程中使用的变量。E 是要在体系结构层次结构的当前级别(任务或任务组)中映射的元素数量，它对于每个级别是不同的。G 是每个组的元素数。P 是处理单元的数量。N 是要映射的任务数量。L 是体系结构层次结构上的级别数。为了便于计算复杂度，我们考虑了 pn。

Generategroup 的复杂性是:

$$\sum_{i=1}^{\frac{G}{X}}\sum_{j=1}^{\frac{E}{X}} i = \sum_{=1}^{\frac{G}{X_i}} E\ i\ E\ G^2 \tag{1}$$

The complexity of GenerateGroupsF orLevel is shown in Equation 2. The number of groups is $\frac{E}{G}$. Also, E is an upper bound for G.

$$\sum_{i=1}^{E=G} \text{GenerateGroup} \sum_{=1}^{E=G} X_i\ E\ G^2 \quad \frac{E}{G}\ E\ G^2\ E^3 \tag{2}$$

The complexity of RecreateMatrix is $O(E^2)$. The complexity of Mapsatot 操作系统与执行深度首次搜索(o(v+)相同

其中 v 是顶点数，c 是边数。自从

我们的群隐式地形成了一棵树，我们知道 cv1。这个树的最后一层有 n 个顶点，倒数第二层有 p 个顶点。其他级别的顶点数是前一级别的数目，除以分片器的数目。共享者的数量大于 1，因为我们只跟踪共享级别。因此，vn+p+2po(n)。由此，映射群的运算复杂度为 o(n)。

顶级算法 MapAlgorithm 的复杂性取决于所有以前的算法，如公式 3 所示。为了计算复杂度，我们必须考虑到 e 的值在体系结构层次结构的每个层次上都发生了变化。

$$\sum_{i=0}^{L} \text{GenerateGroupsF orLevel} + \text{RecreateMatrix} + \text{mapst group / oT opology} \tag{3}$$

L

$$= \sum_{=0} O\,Ei^{3} + O\,Ei^{2} + O(N) \qquad (4)$$

我们可以将公式 4 重写为公式 5，考虑到算法迭代 l+1 次(l 级的架构拓扑和一个级别的

图 12

算法 6:LbGenerateGroupsF 或 level:为体系结构层次结构的一个级别生成组。

输入:commMatrix[][]，nElements，level，previousGroups[]，avlGroups
输出:nGroups，groups[]
Localdata: chosen []，gi，i，newGroup，totalLoad
开始

```
2     totalLoad        0;
3     for i  0 ; i<nElements ; i  i+1 do
4        chosen[i]       0;
5        totalLoad          totalLoad + previousGroups[i].load;
```

结束

```
7     gi      0;
8     for i      0 ; i<nElements ; i  i+newGroup.nElements do
9        loadThreshold       totalLoad = avlGroups;
         /* LbGenerateGroup is implemented in Algorithm 7.                    */
10       newGroup       LbGenerateGroup(commMatrix, nElements, chosen,
           previousGroups, loadThreshold, i);
11       newGroup.id       gi;
```

Totalload totalLoad-newGroup.load;
Avlgroups avlGroups-1;
群体[群体]newGroup;

```
15       gi   gi + 1;
```

结束
返回[n 组，组];
结束

这意味着，在最坏的情况下，拓扑将是一棵有 p 片的二叉树，顶点的最大数目等于 2p1。

$$\text{"}\ O_{i=1} \quad \frac{P}{2^{i}\ \ 1}\ {}^{3}! + O \quad \frac{P}{2^{i}\,1}\ \ 2!\#$$

X

$$+ O(N^3) + O(N^2) + O(N) \qquad (5)$$

$$2\ O(P^{3}) + O(P^{2}) + \qquad O(N^3) + O(N^2) \quad + O(N) \qquad (6)$$

哦

利用方程 6 中所示的简化公式，证明了 EagerMap 的多项式复杂度为 o(N3)。

3.4 考虑任务量

在并行应用程序中，任务占用 CPU 不同的时间是很常见的，因此对计算能力有不同的要求。我们将任务完成执行任务所需的 CPU 能力称为任务负载。为了在映射任务时考虑任务负载，从而平衡负载，前面提出的算法需要进行轻微的修改。大多数改进都是在通用的 groupsf 或者

level(算法 2)和通用的 GenerateGroup(算法 3)中完成的。在这些算法中，元素的总数与

图 13

算法 7:LbGenerateGroup:生成一组通信的元素。

输入:commMatrix[][]，totalElements，selected[]，
previousGroups[]，loadThreshold，done

输出:组

Localdata: i，j，w，wMax，winers [] ，winner

```
开始
2     group.load      0;
3     group.nElements   0;
4     for i  0 ; done<totalElements AND group.load<loadThreshold; i  i+1 do
5        wMax       -1;
6        for j     0 ; j<totalElements ; j  j+1 do
如果选择[j]0 然后
8                    w   0;
K 0; k i; k k + 1 do
10                      w      w + commMatrix[j][winners[k]];
结束
如果我们 wMax 那么
Wmax w;
赢家 j
结束
结束
结束
被选中的[获胜者]
胜利者[我]胜利者
Group. elements [ i ] previousGroups [ winner ] ;
21          done      done + 1;
22          group.nElements       group.nElements + 1;
23          group.load       group.load + previousGroups[winner].load;
结束
返回组
结束
```

任务，并且他们试图保持每个组的任务数量尽可能接近nElementsnGroups。为了考虑任务负载，我们修改了元素的概念来表示任务负载。通过这种方式，算法使用任务加载来代替任务。并且该算法使用总任务负载而不是总元素数。其思想是对在相同组中通信的任务进行分组，但是以负载而不是任务数量来限制组的大小。

考虑负载的修改算法可以在 algorithm6(LbGenerateGroupsForLevel)和7(LbGenerateGroup)中找到。关键思想是在循环的每次迭代中重新计算组的最大负载(loadThreshold)(算法 6 中的第 9 行)。我们这样做是为了更均衡地平衡下一个组的负载，因为由于任务之间的不同负载，组的实际负载很少完全是 totalLoadavlGroups。如果不这样做，最后一组的负载通常很低。对于算法 7，主要的差别是组的大小受到平均负载

loadThreshold 的限制，在第 4 行，而在非负载平衡版本的算法中，组的大小被限制为每组的平均任务数。

3.5 平行鹰图

Generategroup 函数是 Ea-gerMap 关键路径中唯一可以并行化而不改变串行版本行为的函数。

图 14

算法 8:parallelGenerateGroup:生成一组通信的元素。

输入:commMatrix[][]，totalElements，groupElements，selected[]，previousGroups[]

输出:组

Localdata: i，j，wMax，nt，winners []，winner，threadResult []

开始

```
2       for i  0 ; i<groupElements ; i  i+1 do
3           wMax   -1;
# pragma omp parallel
                    ThreadData: id，j，
                    w，k
5                   id      getThreadID();
# pragma omp master
7                   nt        getNumberOfThreads();
8                   threadResult[id].wMax          -1;
# pragma omp for
0; j totalElements; j j + 1 do
如果选择[j]0 然后
12                                w   0;
K 0; k i; k k + 1 do
W + commMatrix [ j ][ winners [ k ] ;
结束
如果 wthreadResult[id].wMax
Threadresult [ id ] . wmaxw;
获胜者 j;
结束
结束
结束
结束
Wmax-1;
为 j0;jnt;jj+1do
如果 threadResult[j].wMax，则
Wmax threadResult [ j ] . wMax;
获胜者 threadResult[j].获胜者;
结束
结束
被选中的[获胜者]
胜利者[我]胜利者
Group. elements [ i ] previousGroups [ winner ] ;
```

结束
返回组
结束

图 15

还可以并行化 RecreateMatrix 和 MapGroupsToT 操作系统。不管怎样，并行化它们的开销超过了收益。

并行版本的 EagerMap 包括并行化算法 3 第 4 行的循环。此循环负责查找将属于某个组的下一个任务。并行算法的思想是，我们可以并行搜索下一个任务，其中每个线程计算总任务的一个子集，并确定其本地任务，最大限度地实现组通信。

算法 8 给出了并行算法。我们用类似于 OpenMP 标准的符号表示这个并行化。在并行块的开头，第 4 行中，我们初始化包含最大化组通信的 lo-cal 任务的 vectorthreadResult。在第 10 行的 for 循环中并行搜索本地任务。然后，从每个线程找到的本地任务中选择最大化通讯的任务，如第 24 行。虽然没有临界区域，但由于平行区域的持续时间较短，我们不认为会出现线性加速。任务数量越多，加速度越快。

对与第一层次结构级别相关的 GenerateGroup 的函数调用是最昂贵的过程，因为它们处理任务的分组。因此，对于第一个层次结构级别，我们使用并行版本。对于后续级别，GenerateGroup 的成本大幅下降，因此最好使用顺序版本。

3.6Non-TreeTopologies 跑步地图

本节中介绍的以前的算法仅在对称树拓扑中工作。这使得它们仅适用于共享内存机器，或者所有机器具有相同拓扑结构的集群。在本节中，我们将解释如何扩展 EagerMap 以适用于任何类型的集群或网格。其思想是预先计算哪些任务将在集群的每台机器上运行，然后为每台机器分别调用前面提出的算法。我们认为每台机器内部的通信比不同机器之间的通信要快得多。

执行这种预计算的算法与其他算法非常相似，并且也基于贪婪分组策略。算法做的第一件事就是计算每台机器将执行的任务数量。为此，它计算了所有机器中可用的 cpu 总数，并以这样一种方式分配任务，即每台机器每个 PU 具有最接近的可能任务数。例如，如果任务的数量等于 cpu 的总数，那么每台机器每个 PU 就有一个任务。然后，它生成一组任务，每台机器将以与算法 3 类似的贪婪方式运行这些任务，在算法 3 中，将要添加到一个组中的下一个任务是最大限度地与组中已有元素进行通信的任务。

如前所述，nal 步骤是为每个组分别调用 EagerMap。通过这种方式，我们可以处理任何类型的集群/网格，而不用考虑拓扑结构或计算机是否具有不同的结构。在这个步骤中，可以用与 3.4 节中解释的非常相似的方式来考虑负载平衡。将在每个组上运行的任务的划分可以并行化，如第 3.5 节所述。

图 16

Eagermap 评价方法研究

在这一节中，我们将展示如何评估我们的建议。我们讨论了在我们的评估中使用的基准和体系结构，如何获得通讯矩阵，以及其他用于比较的映射策略。

4.1 基准

对于映射算法的评估，我们使用了来自 NAS 并行基准测试(NPB)[3]的 MPI 实现、OpenMPNPB 实现[28]、高性能计算挑战(HPCC)基准测试[36]和 PARSEC 基准测试套件[7]的应用程序。Nas 基准测试采用输入大小 b，HPCC 采用输入矩阵 40002，PARSEC 采用本地输入大小。

大多数实验基于具有静态通信的应用程序。Hpcc 具有几个交流阶段，用于表明电子在线算法，如 EagerMap，对于具有动态行为的应用是重要的。所有应用程序都被设置为创建相当于机器的 cpu 数量的线程数。

4.2 通信矩阵的生成

对于基于 MPI 的基准测试，我们使用 eztrace 框架[41]跟踪任务发送的所有 MPI 消息，并根据任务之间发送的消息数量构建通信矩阵。我们还使用字节数生成通信矩阵，尽管每个单元格的值是不同的，但总体模式是相同的。对于对每个步骤都有不同通信模式的 HPCC 基准，我们还使用 eztrace 静态地生成矩阵，因为通信模式在执行之间不会发生变化。对于使用内存访问进行隐式通信的共享内存基准测试，我们基于 Pin 二进制分析工具[34]构建了一个内存跟踪程序，类似于 This 工具跟踪任务的所有内存访问。通过比较不同任务对相同内存地址的访问次数，建立通信矩阵，并在每次匹配时增加通信矩阵。

4.3 生成任务负载

为了度量任务负载，我们计算每个线程执行的指令数量，这比传统的 CPU 时间[17]更能代表负载。因此，当每个线程执行的结构数量不同时，任务之间的负载就会不平衡。

4.4 硬件架构

我们在共享内存方面的实验中使用的硬件架构由 4 个 IntelXeonX7550 处理器组成，总共 64 个 cpu。在这个结构中，任务之间的交流有三种可能性。在同一核心上运行的任务可以通过快速的 L1 或 L2 缓存进行通信，并具有最高的通信性能。在不同核心上运行的任务

虽然必须通过较慢的 L3 缓存进行通信，但是仍然可以从芯片内部的快速互连中受益。当任务通过物理处理器进行通信时，需要使用缓慢的芯片间互连。

在集群实验中，我们使用 OpenMPI1.6.5 和两种类型的 machines:(a)2IntelXeonE5530，共有 16 个 CPU;(b)2IntelXeonCPUE5-2640v2，共有 32 个 CPU。我们只使用了机器中一半的 PUs，因为由于 SMT 和 OpenMPI 的不良支持，它比使用所有的 PUs 提供了更好的性能。我们用这些机器评估了 6 个不同的集群:

484 台 a 型机器，总共 32 台任务

2 台 16 台 b 型机器，共计 32 台任务

4164 台 b 型机器，总共 64 台任务

8168 台 b 型机器，共计 128 台任务

32 台 a 型和 1 台 b 型机器，32 台任务，64 台 a 型机器，2 台 b 型机器，64 台任务

为了评估负载感知的 EagerMap，我们使用了带骑士角架构的 XeonPhi 协同处理器模型 3120P。我们使用这种体系结构是因为，由于有大量的核心(57 个核心，每个 4 路 SMT)，负载平衡在其性能中起着关键作用。

4.5 比较

我们将 EagerMap 与(i)一个随机映射进行比较，后者是 30 个不同随机映射的平均结果;(ii)TreeMatch[25,26];(iii)Scotch[38];和(iv)mpipippp[12]。大多数性能结果都被规范化为默认操作系统映射的性能，即 Linux 完全公平排程器文档[29]，它专注于保持任务和内核之间的负载平衡。

结果

我们评估了算法的性能和质量，以及映射应用程序任务时所获得的性能改进。

5.1 映射算法的性能

图 5 显示了所有基准点的四个映射算法的执行时间。对于 128 个任务，EagerMap 比 Scotch 快 10 倍，比 TreeMatch 快 1000 倍，比 mpippp 快 100,000 倍。Treematch 具有指数级的复杂性，因此需要更长的执行时间。由于这个原因，TreeMatch 和 EagerMap 之间的执行时间差异与任务数量一起增加。Mpippp 比其他机制慢得多，因为它需要执行多次迭代来重新构造初始的随机映射。当任务数量超过 128 个时，它不会提高执行效率。

5.2 地图的质量

计算映射的质量决定了可以实现的效益。质量是通过实现的局部化程度来衡量的。我们使用

图 18

公式 7 计算质量，这是在 TreeMatch 的源代码中提供的。在这个公式中，n 是任务的数量，m[i][j]是任务 i 和 j 之间的通信量，map[x]是被映射的处理单元(PU)，lat[a][b]是层次结构中 PU 的延迟。我们使用 LMbench[37]计算延迟。

$$\text{MappingQuality} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{M[i][j]}{lat[map[i]][map[j]]} \qquad (7)$$

图 6 给出了图 1 所示的通信矩阵的映射质量结果。正如预期的那样，具有更多结构化通信模式的应用程序表现出了最大的改进。Cg-mpi、LU-MPI 和 HPCC(第 7 阶段)的改进效果最好，因为它们的通信可以很容易地通过映射相邻任务来优化。

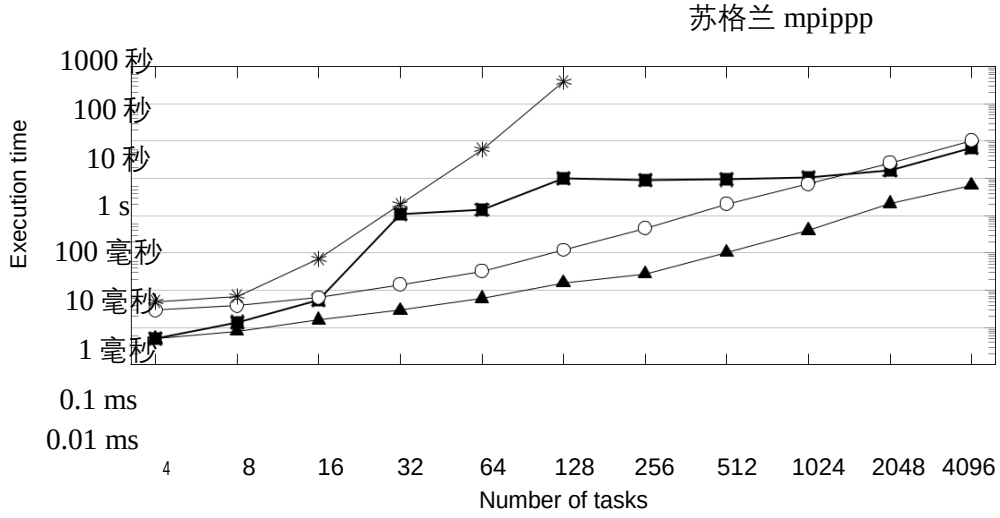


图 5:映射算法的执行时间(毫秒)，用于映射任务的数量。

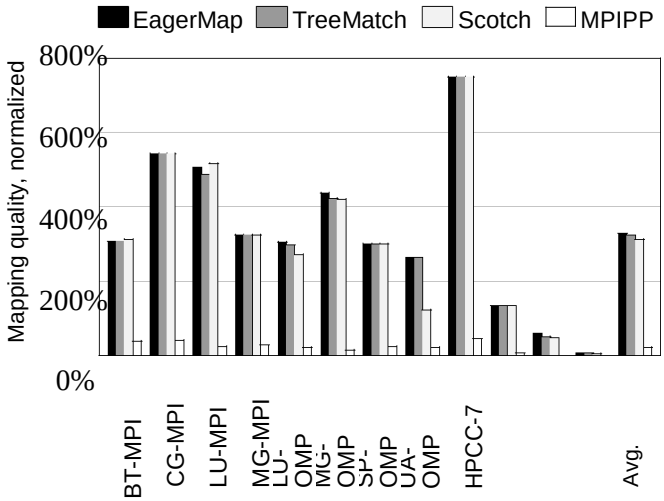


图 6:映射质量的比较，归一化为随机映射-ping。价值观越高越好。

图 19

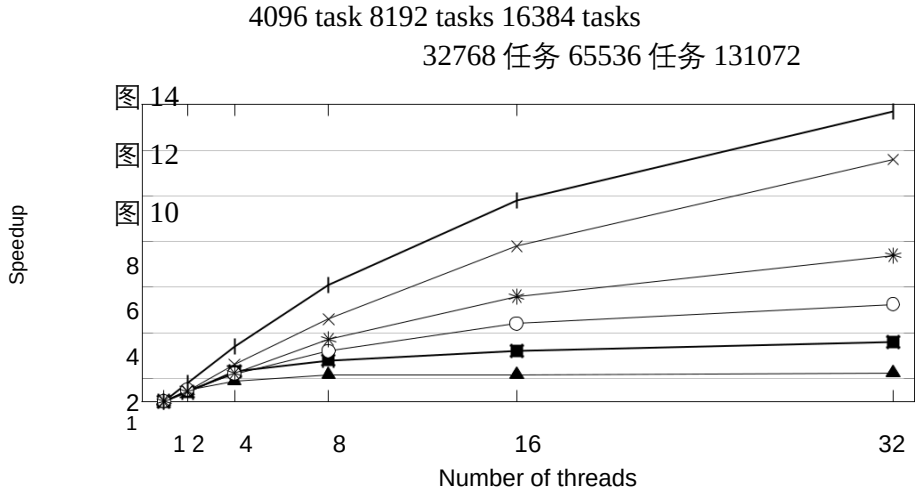


图 7:EagerMap 的并行版本的加速，使用不同数量的任务被映射，改变执行线程的数量。

在 BT-MPI 和 MG-MPI 中，远近任务之间进行通信，给映射算法带来了更多的挑战。原因是如果相邻的任务被映射到一起，远程任务之间的通信不会改善。同样地，将遥远的任务映射在一起并不能改善相邻任务之间的交流。在具有较少结构化模式的应用程序(如 Streamcluster)中，由于通信量较大的任务与通信量较小的任务之间的通信比率较低，因此预计改进程度较低。Vips 是唯一一个具有非结构化通信的应用程序，因此任务映射都不能改善其通信。

由于 mpippp 是建立在初始随机映射的基础上的，所以用 mpippp 得到的质量与随机映射相似。虽然 mpipip 可以处理很少的任务，但是当任务数量增加时，排列的可能性呈指数增加。这使得它更多地使用和新的组合来改进最初的解决方案。和 Scotch 在所有应用程序中都得到了类似的结果。这个结果表明，由于我们在第 3 节中观察到的通信模式的特点，EagerMap 能够取得和更复杂的算法一样好的结果。

5.3 并行版本 EagerMap 的加速

图 7 中可以找到任务数目映射时 EagerMap 并行版本的加速比。我们分析了相对于任务数的加速比，因为它是执行时间中最重要的参数。任务树结构对加速比的影响很小，因为并行算法仅应用于层次结构的第一级(通常只应用于第一级)，如第 3.5 节所解释的。正如我们在 3.5 节中所解释的，我们不期望线性加速，因为并行阶段的持续时间很短。并行阶段在映射计算期间多次重新启动，在并行化过程中增加了开销。另一方面，当需要映射的任务数量增加时，开销的影响会减小，这在 gure 中可以看到。这种情况发生的原因是，随着任务的增加，任务的持续时间

图 20

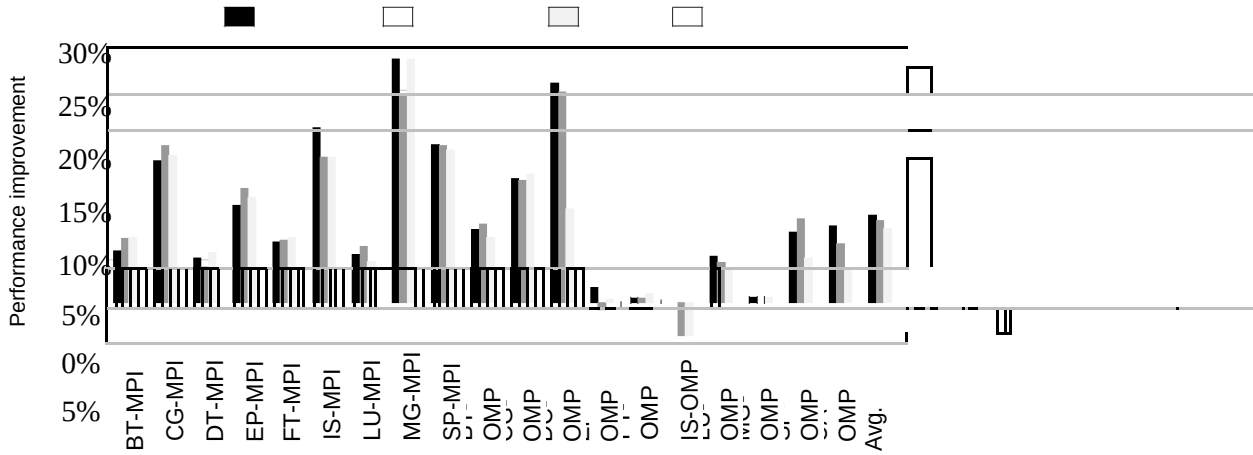


图 8:与共享内存环境中 NAS-MPI 和 NAS-OMP 基准的操作系统映射相比，性能有所提高。

平行相位增加。对于 131072 个任务，使用 32 个线程的加速比为 13.7，对于更多的任务，加速比应该更高。

需要注意的是，当任务数量很大时，应该鼓励使用 EagerMap 的并行版本。这是因为执行时间随着任务数量的增加而快速增长。例如，对于 32768 个任务，单线程执行需要 43 分钟。对于 131072 个任务，单线程执行时间增加到 25 小时。使用 32 线程的并行版本，执行时间分别减少到 5.8 分钟和 1.8 小时。

5.4 共享内存环境下的性能改进

我们使用 IntelXeonX7550 处理器上的算法获得的映射来执行应用程序。Mpi 和 OpenMPNAS 基准测试的执行时间结果如图 8 所示。对于这些应用，由于它们的通信模式是稳定的，我们静态地计算了映射。作为在线映射的一个案例研究，我们使用 HPCC 基准标记，如图 9 所示，因为它包含 16 个阶段，具有不同的通信行为。对于每个阶段，我们调用映射算法并相应地迁移任务。执行时间结果对应于 30 次执行的平均值，并且被规范化为操作系统原始策略的执行时间。所有算法的信度区间都小于 1%。

一般来说，具有更多结构化通信模式的应用程序具有更高的性能改进。例如，与 OS 映射相比，CG-MPI 的性能显著提高，因为它的通信模式显示了很高的映射潜力，如第 5.2 节所讨论的。平均而言，EagerMap 的性能提高了 10.3%，TreeMatch、Scotch 和 mpippp 的性能分别提高了 9.6%、8.7%和 8.4%。Hpcc 基准测试的结果(图 9)表明，与相关工作相比，echermap 的改进更大。此外，我们的低开销

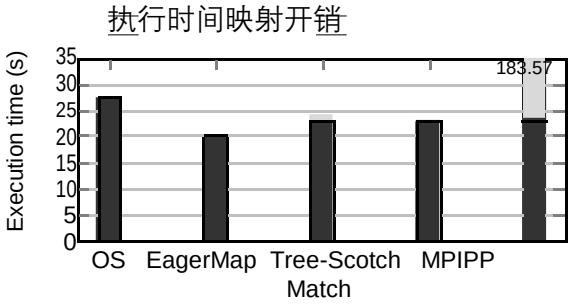


图 9:使用在线映射的 HPCC 执行时间。

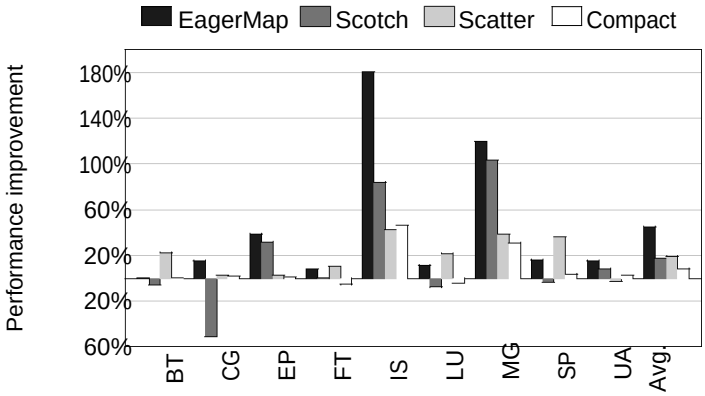


图 10:与操作系统 map-ping 相比，来自于 XeonPhi 中 NASOpenMP 基准的负载感知 EagerMap 的性能改进。

算法对应用程序性能的影响不亚于 TreeMatch 和 mpippp。

5.5 负载感知算法的性能改进

在 XeonPhi 中，负载感知型 EagerMap 的性能改进如图 10 所示。平均而言，EagerMap 的性能提高了 44.9%，而 Scotch、Scatter 和 Compact 的性能平均分别提高了 17.8%、19.3%和 9.6%。分散和紧凑不执行通信和负载的任何分析，只遵循预先设计的规则。这表明，EagerMap 通过实际考虑应用程序的通信和负载方面，可以提供更好的性能改进。另一方面，Scotch 还考虑到映射的负载和通信。这表明 EagerMap 可以比 Scotch 更好地利用这些信息。

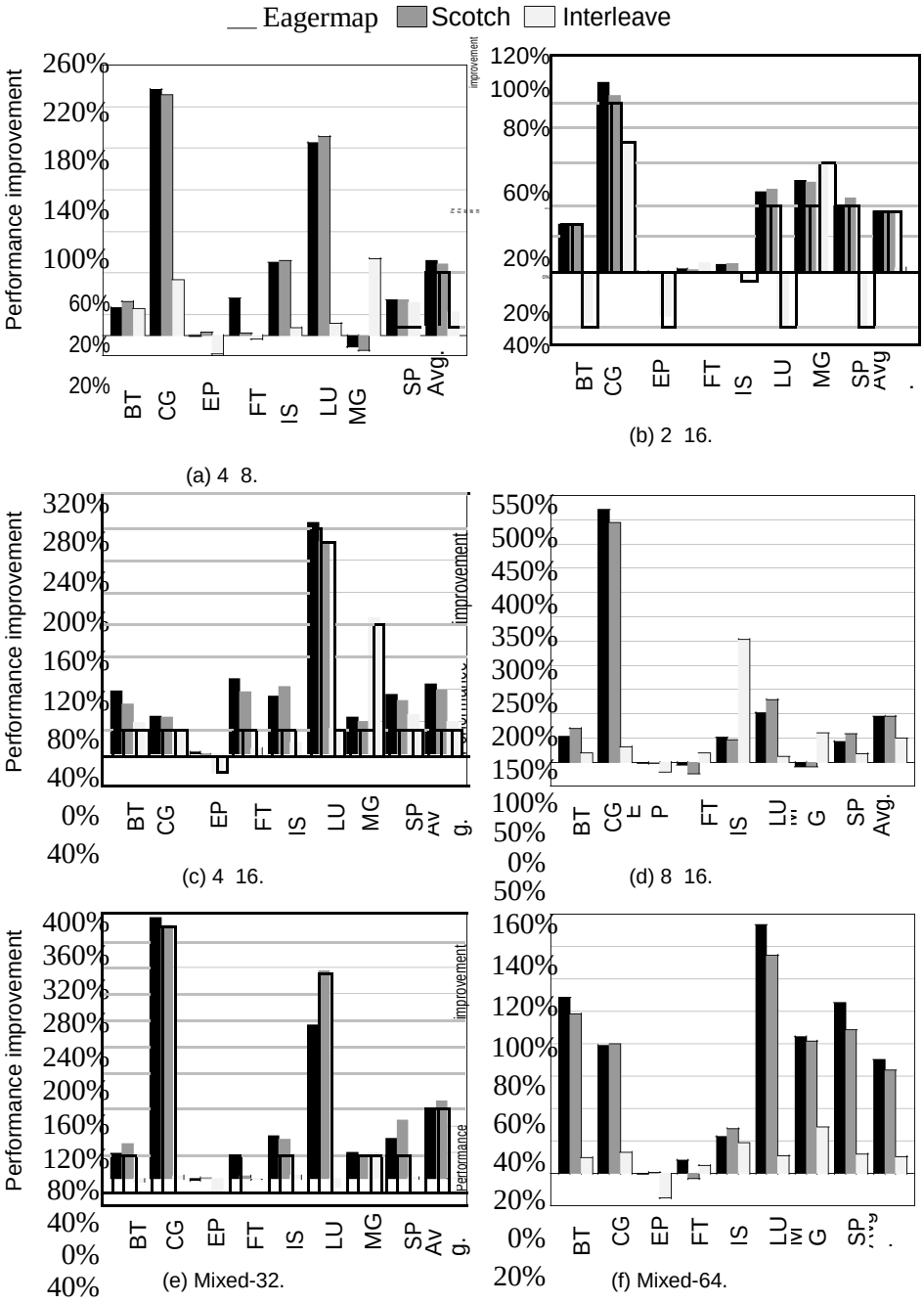


图 11:与集群中 NAS-MPI 基准的操作系统映射 ping 相比的性能改进。

5.6 集群中的性能改进

从 MPINAS 基准的应用程序的集群实验的结果可以在图 11 中找到。结果被归一化为随机映射。由于 OpenMPI 不支持节点之间的任务迁移，因此只静态地执行映射。

图 23

Eagermap 实现了与 Scotch 类似的性能结果。然而，如第 5.1 节所示，EagerMap 计算映射所需的时间大约是 Scotch 的 10 倍。在 32 个任务的实验中，在 48、216 和 Mixed-32 集群中，应用 CG 取得了最好的结果，Mixed-32 集群的性能提高了 394%。在 416 个任务中，有 64 个任务，LU 是改进最好的应用程序，改进率为 286%。除了 416 和 Mixed-64(应用程序使用二次任务数)之外，BT 和 SP 在所有集群中使用的核都少于总核，Scotch 提供了更好的结果。在 416 和 Mixed-64 中，EagerMap 为 bet-ter。这表明 EagerMap 更善于优化通信，而 Scotch 更善于平衡机器之间的负载。对于 MG，Interleave 策略取得了良好的结果，因为巧合的是，它的映射非常接近 MG 的最佳映射。

结论

并行体系结构中任务到 cpu 的映射关系对通信性能和负载均衡的影响。通信性能可以通过映射任务来提高，这些任务可以通信到附近的内存阶层或者集群或网格中的同一个节点，利用更快的内部连接。通过将任务映射为负载均匀分布在 cpu 之间的方式，可以改善负载平衡。映射算法选择哪个 PU 将执行每个任务，并在这些类型的映射中发挥关键作用。当映射算法同时考虑这两个指标时，会出现更多的挑战，因为这些指标可能导致相互矛盾的决策。

在本文中，我们提出了 EagerMap，它最初只在共享内存架构中工作，并且只考虑了通信模式。我们增加了对负载平衡、集群映射和网格的支持，并提出了一个并行版本来加速其执行。在分析并行应用程序通信模式的基础上，采用了一种更为简单的方法来选择应该映射到一起的任务。我们用大量具有不同通信特性的基准进行了实验。结果表明，EagerMap 计算的任务映射比现有技术更好，开销大大降低，缩放性更好，这使得 EagerMap 更适合在线映射。

作为未来的工作，我们打算研究 EagerMap，以允许它以分布式方式在集群中执行。

Eagermap 是通过 GPL 授权的，可以在。

感谢

这项研究得到了欧盟 H2020 项目的资助，以及 mcti/rnp-brazil 在 HPC4E 项目下的赠款协议 689772。它也得到了英特尔的支持。

参考资料

安巴尔，o.Serres，e.Kayraklioglu，A.-H。巴达维和 t.El-Ghazawi。深度并行体系结构中层次局部性的开发。关于架构和代码优化的 ACM 事务(TACO)，13(2):1{25,2016。

阿齐米、塔姆、苏亚雷斯和斯图姆。通过硬件性能监控增强多核处理器的操作系统支持。操作系统评论，43(2):56{65，apr2009。

贝利，e.Barszcz，j.t.巴顿，d.s.Browning，r.l.Carter，弗雷德里克森，t.a.拉辛斯基，r.s.Schreiber，h.d.Simon，v.Venkatakrishnan，s.k.Weeratunga。Nas 并行基准测试。国际超级计算机应用杂志，5(3):66{73,1991。

贝克，梅农，怀特，迪纳和甘蓝。基于综合运行时方法的多级负载平衡。在 ieee/acm2018 年集群、云和网格计算(CCGrid)国际研讨会上。

巴拉德，e.卡森，j.Demmel，m.Hoemmen，n.Knight，and 施瓦茨。数值线性代数的通信下界和最优算法。23(May):1{155,2014.

N.Barrow-Williams，c.Fensch 和 s.Moore。Splash-2 和 Parsec 的通信特性。在 IEEE 工作负荷角色塑造国际研讨会上，第 86 页{97,2009。

C.Bienia，s.Kumar，j.p.Singh，k.Li.基准套件:角色塑造和架构含义。在并行结构和编译技术国际会议上，第 72 页{81,2008。

和 p.g.Kropf。一种快速的分布式映射算法。在矢量和并行处理联合国际会议(CON-PAR90{VAPPIV)中，第 405 页{416,1990。

作者:s.Bokhari。关于映射问题。美国电气和电子工程师学会计算机汇刊，C-30(3):207{214,1981。

布兰德法斯，t.阿鲁兹和 t.格霍尔德。Mpi 通信优化的秩重排。计算机与流体，80(7 月):372{380,2013。

布洛克蒂斯，j.Clet-Ortega，s.moreau，n.Furmento，b.Goglin，梅谢尔、s.蒂博特和 r.纳米斯特。Hwloc:一个用于管理 HPC 应用程序中硬件的通用框架。在 EuromicroConfer-enceonParallel，DistributedandNetwork-basedProcessing(PDP)，pages180{186,2010。

陈，陈，黄，罗伯特，库恩。Mpippp:一种自动导向的 SMP 集群和多集群并行处理放置工具集。在 acm/ieee 国际高级会议上

性能计算，网络，存储和分析(SC)，第 353 页{360,2006。

谢瓦利埃和佩莱格里尼。Pt-scotch:一种 e 类平行图排序工具。并行计算，34(6-8):318{331,2008 年 7 月。

克鲁兹，迪纳，阿尔维斯，纳沃克斯。利用缓存协议实现共享内存应用的动态线程映射。平行与分布式计算杂志，74(3):2215{2228，mar2014。

克鲁兹，迪纳，皮拉，纳沃克斯。一种基于通信的任务映射的 e-cient 算法。并行、分布式和基于网络的处理(PDP)国际会议，第 207 页{214,2015。

克鲁兹，迪纳，皮拉，纳沃克斯。分层多核体系结构中的硬件辅助线程和数据映射。AcmTrans.Archit.代码优化，13(3):28:1{28:28，september2016。

克鲁兹、迪纳、塞尔帕、纳沃克斯、皮拉和高仁。多核系统中利用线程映射改进通信和负载均衡。在欧微并行、分布式和基于网络的处理(PDP)国际会议上，第 93 页{100,2018。

W j 达利。图形处理器计算到 Exascale 及更远。技术报告，nVidia，2010。

德维西，卡亚，b.乌卡，和 v.催化剂。快速高质量的拓扑感知任务映射。在 IEEE 国际并行和分布式处理研讨会(IPDPS)，第 197 页{206,2015。

迪瓦恩，e.g.博曼，r.t.希菲，r.h.比塞林，和 u.v.催化剂。面向科学计算的并行超图划分。在 IEEE 国际并行与分布式处理研讨会(IPDPS)中，第 124 页{133,2006。

埃德蒙兹。最大匹配与 0,1 顶点多面体。国家标准局研究日志{b.数学与数学物理，69B(1 和 2):125,1965。

和 a.Noe。基于网格和环面结构的过程映射算法。2015 年第 23 届欧洲微观组织并行、分布式和基于网络的进程国际会议，第 236 页{243,2015 年 3 月。

和 r.Lelandy。Chaco 用户指南 2.0 版。技术报告，Sandia 国家实验室，1995 年。

伊藤、后藤和小野。多核并行环境下区域分解算法中核到子域的自动优化映射。80:88{93，jul2013.

和 g.Mercier。Mpi 进程在层次 NUMA 体系结构上的近似最优布局。在欧元平行处理，第 199 页{210,2010。

和 f.Tessier。多核集群中的进程布局:算法问题和实用技术。并行和分布式系统汇刊，25(4):993{1002，apr2014。

和 f.Tessier。Charm+中的拓扑与严格意义上的高仿真分布式负载均衡。在高性能计算通信优化研讨会(com-HPC)，第 63 页{72,2016。

金，m.弗鲁姆金，j.Yan。Naspar 并行基准的 OpenMP 实现及其性能。1999 年 10 月美国宇航局技术报告。

[29]m.t.琼斯。在 linux2.6 内部完全公平的 sched-
乌勒。
2009.[在线;2018 年 6 月访问]。

约万诺维奇和 s.马里奇。高度并行计算系统中动态任务调度的启发式算法。未来计算机系统，17(6):721{732，apr2001。

G.Karypis 和 v.Kumar。非结构化图分割与稀疏矩阵排序系统，版本 2.0。技术报告，明尼苏达大学，计算机科学系，1995。

G.Karypis 和 v.Kumar。不规则图的并行多层 k-路分割方案。在 acm/ieee 超级计算会议上，第 1{21,1996。

G.Karypis 和 v.Kumar。一种快速高质量的不规则图分割多级方案。Siamj.Sci.20(1):359{392，dec1998.

C.-k.陆，r.Cohn，r.Muth，h.Patil，a.Klauser，g.Lowney，s.Wal-lace，v.j.Reddi，和 k.Hazelwood。引脚:使用动态仪表构建定制的程序分析工具。在 ACMSIGPLANConfer-enceonProgrammingLanguageDesignandImplementation(PLDI) 中，第 190 页{200,2005。

罗、李、丁。缓存中的线程数据共享:理论与度量。103{115，NewYork，NY，USA，2017.在 ACMSIGPLANPPoPP 中，第 103 页。美国计算机协会。

P.Luszczek，j.j.Dongarra，d.Koester，r.Rabenseifer，b.Lucas，j.Kep-ner，j.Mccalpin，d.Bailey，d.Takahashi，j.Jack，andr.Rabenseifner.高性能计算挑战基准套件简介。技术报告，2005 年。

和 c.Staelin。便携式性能分析工具。在 USENIX 年度技术会议(ATC)，第 23 页{38,1996。

佩莱格里尼。流程图与结构图的对偶递归双分割静态映射。在 ScalableHigh-PerformanceComputingConference(SHPCC) ，pages486{493,1994。

作者:j.Shalf，s.Dosanjh，j.Morrison。百万兆等级运算技术挑战。在高性能计算计算科学中，第 1 页{25,2010。

格林斯基，r.Reutiman 和 a.Chandra。Starling:使用分散的自觉迁移最小化虚拟计算平台中的通信开销。在国际并行处理会议(ICPP)中，第 228 页{237,2010 年 9 月。

和 j.Dongarra。Eztrace:性能分析的通用框架。集群、云和网格计算(CCGrid)国际研讨会，第 618 页{619,2011。

D.Unat ， a.Dubey ， t.HoeE. ， j.Shalf ， m.Abraham ， m.Bianco ，b.l.Chamberlain ， r.Cledat ， h.c.Edwards ， h.Finkel ， k.Fuerlinger ，f.Han-nig ， e.Jeannot ， a.Kamil ， j.Keasler ， p.h.j.Kelly ， v.Leung ，h.Ltaief，n.Maruyama，c.j.Newburn，m.Pericas.高性能计算机系统数据本地化的趋势。并行和分布式系统汇刊，28(10):3007{3020,2017 年 10 月。

王，台德，马尔斯，唐，戴维森，马尔斯。基于硬件资源整体视图的线程映射性能分析。在 IEEE 系统与软件性能分析国际研讨会(ISPASS)，2012。

翟志刚、沈天行、何建国。大规模并行应用中通信轨迹的快速获取。并行和分布式系统汇刊(TPDS)，22(11):1862{1870,2011。