面向异构计算的高性能、低复杂度任务调度

Haluk Topcuoglu，*IEEE 会员*； Salim Hariri，*IEEE 计算机协会会员*；Min-You Wu，*IEEE 高级会员*

摘要-高效的应用调度对于在异构计算环境中实现高性能至关重要。应用调度问题已被证明在一般情况下以及在几种受限情况下是 NP-完全的。由于该问题的重要性，人们对其进行了广泛的研究，并在文献中提出了各种算法，这些算法主要针对具有同构处理器的系统。虽然文献中也有一些针对异构处理器的算法，但它们通常需要很高的调度成本，而且可能无法以较低的成本提供高质量的调度。在本文中，我们提出了两种适用于有界异构处理器数量的新型调度算法，其目标是同时满足高性能和快速调度时间的要求，分别称为异构最早完成时间（HEFT）算法和处理器上的关键路径（CPOP）算 法 。HEFT 算法在每一步选择向上等级值最高的任务，并将所选任务分配给处理器，通过基于插入的方法使其最早完成时间最小化。另 一 方 面 ，CPOP 算法使用向上和向下排名值的总和来确定任务的优先级。另一个不同之处在于处理器选择阶段，该阶段将关键任务调度到能使关键任务总执行时间最小化的处理器上。为了与相关工作进行稳健且无偏见的比较，我们设计了一个参数图生成器，用于生成具有各种特征的加权有向无环图。基于随机生成的图和一些实际应用的图进行的比较研究表明，我们的调度算法在调度的质量和成本方面都大大超过了以前的方法，这主要体现在调度长度比、加速、最佳结果频率和平均调度时间指标上。

索引词条-DAG 调度、任务图、异构系统、列表调度、映射。

1 导言

D通过高速网络互联的多组资源提供了一种新的计算平台，即异构计算系统，它可以支持执行计算密集型并行和分布式应用。异构计算系统需要为执行应用程序提供编译时和运行时支持。在可用资源上有效调度应用程序的任务是异构计算系统的重要功能之一。实现高绩效的关键因素。

一般的任务调度问题包括将应用程序的任务分配给合适处理器的问题，以及在每个资源上对任务执行进行排序的问题。当一个应用程序的特征（包括任务的执行时间、任务间通信的数据量和任务依赖性）是先验已知的时候，就可以用*静态*模型来表示。

在静态任务调度问题的一般形式中，应用由有向无环图表示

- *H.托普库奥卢现就职于土耳其伊斯坦布尔 81040 号 Goztepe Kampusu 马尔马拉大学计算机工程系。*
  电子邮件：haluk@eng.marmara.edu.tr.
- *S.哈里里现供职于亚利桑那大学电气与计算机工程系，地址：Tucson, AZ 85721-0104。*
  电子邮件：hariri@ece.arizona.edu.
- *M.-Y. Wu 是新墨西哥大学电气与计算机工程系的一员。Wu 现供职于新墨西哥大学电子与计算机工程系，地址：University of New Mexico, Albuquerque, NM 871S1-1S56。*
  电子邮件：wu@eece.unm.edu.

手稿于 2000 年 8 月 28 日收到；2001 年 7 月 12 日修订；9 月 6 日接受。2001.
如需获取本文的转载信息，请发送电子邮件至：tpds@computer.org，并注明 IEEECS Log Number 11278S。

(DAG) 中，节点代表应用任务，边代表任务间的数据依赖关系。每个节点标签表示任务的计算成本（预计计算时间），每个边标签表示任 务 间的通信成本（预计通信时间）。该问题的目标函数是将任务映射到处理器上，并对其执行进行排序，从而满足任务优先级要求，并获得最短的总体完成时间。任务调度问题在一般情况下是 NP-完备的 [1]，在一些受限情况下也是如此 [2]，例如将一个或两个时间单位的任务调度到两个处理器上，以及将单位时间任务调度到任意数量的处理器上。

由于任务调度问题对性能至关重要，人们对其进行了广泛研究，并在文献中提出了各种启发式方法 [3]、[4] 、[5] 、[6] 、[7] 、[8] 、[9] 、[10] 、[11] 、[13] 、[12] 、[16]、[17]、[18]、[20] 、[22] 、[23] 、[27] 、[30] 。这些启发式算法分为多种类型（如列表调度算法、聚类算法、基于重复的算法、引导式随机搜索方法），主要用于具有同构处理器的系统。

在列表调度算法[3]、[4]、[6]、[7]、[18]、[22] 中，通过为每个任务分配优先级来构建一个有序的任务列表。按照优先级顺序选择任务，然后将每个选定的任务调度到能使预定成本函数最小化的处理器上。这类算法能提供高质量的调度，其性能在一定程度上可与其他类型的算法媲美。

1045-9219/02/$17.00 © 2002 IEEE

降低调度时间 [21]、[26]。聚类算法 [3]、[12]、[19]、[25] 一般适用于处理器数量无界的情况，因此可能无法直接应用。聚类算法需要第二阶段（调度模块）将算法生成的 任务簇合并到一定数量的处理器上，并对 每个处理器内的任务执行进行排序[24]。同样，基于任务复制的启发式方法也不实用，因为它们的时间复杂度非常高。例如，BTDH 算法[30]和 DSH 算法[18]的时间复杂度均为 $O(n^4)$；CPFD 算法[9]的复杂度为 $O(e \times n^2)$，用于在一组同构处理器上调度连接有 e 条边的 n 个任务。

遗传算法[5]、[8]、[11]、[13]、[17]、[31]（GA）是针对任务调度问题研究最广泛的引导式随机搜索技术。虽然它们能提供高质量的调度，但其执行时间明显高于其他替代方法。研究表明，基于遗传算法的解决方案对第二最佳解决方案的改进不超过 10%，而且基于遗传算法的方法需要大约一分钟才能产生一个解决方案，而其他启发式方法只需要几秒钟的执行时间[31]。此外，要为基于 GA 的解决方案中使用的控制参数集找到最佳值，还需要进行大量测试。

一些研究小组也对异构系统的任务调度问题进行了研究 [6]、[7]、[8]、[10]、[11]、[13]、[14]。这些算法可能需要分配一组控制参数，其中一些算法面临着调度成本过高的问题 [6]、[8]、[11]、[13]。这种逐级调度技术在任何时候都只考虑当前级别中的任务（即准备就绪的任务子集），由于没有考虑所有准备就绪的任务，因此可能效果不佳。此外，[14] 中的研究提出了一种动态重置器，它需要给定 DAG 的初始调度，然后使用算法的三种变体提高其性能，这不在本文的讨论范围内。

在本文中，我们提出了两种新的静态调度方法针对一定数量完全连接的异构处理器的算法：异构最早完成时间（HEFT）算法和处理器关键路径（CPOP）算法。虽然异构系统的静态调度是离线的，但为了提供实用的解决方案，算法的调度时间（或运行时间）是关键的约束条件。因此，这些算法背后的动机是以较低的成本（即较低的调度时间）提供高质量的调度（或具有较好调度长度的输出）。HEFT 算法在每一步选择*向上等级*最高的任务（定义见第 4.1 节）。然后将所选任务分配给采用插入式方法使其最早完成时间最小化的处理器。任务的向上等级是指从任务到退出任务的关键路径（即最长路径）的长度，包括任 务 的计算成本。CPOP 算法在每一步都会选择具有最高（向上等级 + 向下等级）值的任务。算法

该算法的目标是将所有关键任务（即 DAG 关键路径上的任务）调度到单个处理器上，从而最大限度地减少关键任务的总执行时间。如果所选任务为非关键任务，则处理器选择阶段将以最早执行时间为基础，采用基于插入的调度方式，如 HEFT 算法。

作为这项研究工作的一部分，我们设计了一个参数图生成器，用于生成加权有向无环图，以研究调度算法的性能。本文的比较研究基于随机生成的任务图和实际应用的任务图，包括高斯消除算法[3]、[28]、FFT 算法[29]、[30]和[19]中给出的分子动态代码。对比研究表明，我们的算法在性能指标（调度长度比、加速、效率和产生最佳结果的出现次数）和成本指标（提供输出调度的调度时间）方面都大大超过了以前的方法。

本文的其余部分安排如下：在下一节，我们将定义研究问题和相关术语。在第 3 节中，我们介绍了任务调度算法的分类以及异构系统调度方面的相关工作。第 4 节介绍我们的调度算法（HEFT 和 CPOP 算法）。第 5 节介绍了我们的算法与相关工作的比较研究，该研究基于随机生成的任务图和几个实际应用的 任务图。在第 6 节中，我们介绍了 HEFT 算法的几个扩展。第 7 节是对所做研究的总结和未来工作计划。

2 任务调度问题

调度系统模型由应用程序、目标计算环境和调度性能标准组成。*应用程序*由有向无环图 $J = (V, E)$ 表示，其中 V 是 n 个任务的集合，E 是任务间 e 条边的集合。(每条边 $(i, j) \in E$ 表示优先级约束，即任务 n_i 应在任务 n_j 开始之前完成执行。*数据*是一个 $n \times n$ 的通信数据矩阵，其中*数据*$_{i,h}$ 是需要从任务 n_i 传输到任务 n_h 的数据量。

在给定的任务图中，没有父任务的任务称为*入口任务*，没有子任务的任务称为*出口任务*。某些任务调度算法可能需要单入口和单出口任务图。如果有多个退出（进入）任务，它们会以零成本边连接到零成本*伪退出（进入）任务*，这不会影响调度。

我们假设目标计算环境由一组 q 个异构处理器 Q 组成，这些处理器以完全连接的拓扑结构相连，在这种拓扑结构中，假定所有处理器之间的通信都是无竞争执行的。在我们的模型中，还假设

计算可能与通信重叠。此外，给定应用的任务执行假定为非抢占式。W 是一个 $n \times q$ 的计算成本矩阵，其中每个 $m_{i,j}$ 给出了在处理器 p_j 上完成任务 n_i 的估计执行时间。在调度之前，任务会被标上平均执行成本。任务 n 的平均执行成本$_i$ 定义为

$$\overline{m_i} = \sum_{j=1}^{q} m_{i,j}/q。 \quad (1)$$

处理器之间的数据传输速率存储在大小为 $q \times q$ 的矩阵 B 中。处理器的通信启动成本在 q 维向量 L 中给出。从任务 n_i （安排在 p_m 上）向任务 n_h（安排在 p_n 上）传输数据的边缘 i 的通信成本 h 定义为

$$ci_{,h} = L_m + \frac{数据_{i,h}}{B_{m,n}}. \quad (2)$$

当 n_i 和 n_h 被调度到同一处理器上时，$c_{i,h}$ 将变为零，因为我们假设处理器内的通信成本与处理器间的通信成本相比可以忽略不计。在调度之前，平均通信成本用于标记边。边 (i, h) 的平均通信成本定义为

$$\overline{ci_{,h}} = \overline{L} + \frac{数据_{i,h}}{\overline{B}}, \quad (3)$$

其中，$\overline{B}$ 是域内处理器之间的平均传输速率，L 是平均通信启动时间。

在介绍目标函数之前，有必要定义了 EST 和 EFT 属性，它们来自给定的部分时间表。$E2T(n_i, p_j)$ 和 $EFT(n_i, p_j)$ 分别是任务 n_i 在处理器 p_j 上的*最早*执行开始时间和*最早*执行结束时间。对于入口任务 n_{entry}、

$$E2T(n_{entry}, p_j) = 0。 \quad (4)$$

对于图中的其他任务，从入口任务开始递归计算 EFT 和 EST 值，分别如 (5) 和 (6) 所示。要计算任务 n_i 的 EFT，必须已调度了 n_i 的所有直接前置任务。

$E2T(n_i, p_j) = \max_{m \in pred(n_i)} \{anail[j], 最大$

任务 n_m 在处理器 p_j 上调度后，n_m 在处理器 p_j 上的最早开始时间和最早完成时间分别等于任务 n_m 的实际开始时间 $A2T(n_m)$ 和实际完成时间 $AFT(n_m)$。图中的所有任务都排定后，计划长度（即总完成时间）将是退出任务 n_{exit} 的实际完成时间。如果有多个退出任务，且没有采用插入伪退出任务的惯例，则计划长度（也称为 *makepan*）定义为

$$mahespan = max\{AFT(n_{exit})\}. \quad (7)$$

任务调度问题的*目标函数*是确定将给定应用程序的任务分配给处理器，使其调度长度最小。

3 相关工作

静态任务调度算法可分为两大类（见图 1），即启发式算法和引导式算法。

基于随机搜索的算法。前者可进一步分为三类：列表调度启发式算法、聚类启发式算法和任务重复启发式算法。

列表调度启发式。列表调度启发式根据优先级维护给定图中所有任务的列表。它有两个阶段：*任务优先级*（或*任务选择*）阶段，用于选择优先级最高的就绪任务；*处理器选择*阶段，用于选择合适的*处理器*。

最小化预定成本函数（可以是执行开始时间）的处理器。其中一些例子如下

如修正关键路径（MCP）[3]、动态水平调度[6]、映射启发式（MH）[7]、插入调度启发式[18]、最早时间优先（ETF）[22]和动态关键路径（DCP）[4]算法。大多数列表调度算法都是针对一定数量的完全连接的同构处理器。列表调度启发式算法通常更实用，与其他算法相比，能以更短的调度时间提供更好的性能结果。

聚类启发法。这类算法将给定图形中的任务映射到数量不限的聚类中。在每一步中，被选中进行聚类的任务可以是任何任务，不一定是准备就绪的任务。每次迭代都会通过合并一些簇来完善之前的聚类。如果两个任务被分配到同一个群组，它们将在同一个处理器上执行。聚类启发式

生成最终时间表需要额外的步骤： a

$(AFT(n_m)+ c_{m,i})$、 (5)

集群合并步骤，用于合并集群，使剩余的集群数量与处理器数量相等；集群映射步骤，用于在可用处理器上映射集群；以及任务排序步骤，用于

$$EFT(n_i, p_j) = m_{i,j} + E2T(n_i, p_j)、 \quad (6)$$

其中，$pred(n_i)$ 是任务 n_i 的直接前置任务集，$anail\ j$ 是处理器 p_j 准备好执行任务的最早时间。如果 n_h 是处理器 p_j 上最后分配的任务，那么 $anail\ j$ 是处理器 p_j 完成任务 n_h 的执行时间，当我们采用非插入式调度策略时，它已准备好执行另一个任务。$E2T$ 等式中的内*最大*块返回*就绪时间*，即 n_i 所需的所有数据到达处理器 p_j 的时间。

对每个处理器内的映射任务进行排序 [24]。这一组中的一些例子有：主导序列聚类（DSC）[12]、线性聚类方法[19]、定向移动[3]和聚类与调度系统（CASS）[25]。

任务重复启发式算法。基于重复的调度算法背后的理念是通过对任务图中的部分任务进行冗余映射来调度任务图，从而减少进程间通信开销[9], [18], [27], [30]。基于重复的算法因重复任务的选择策略而异。基于重复的算法

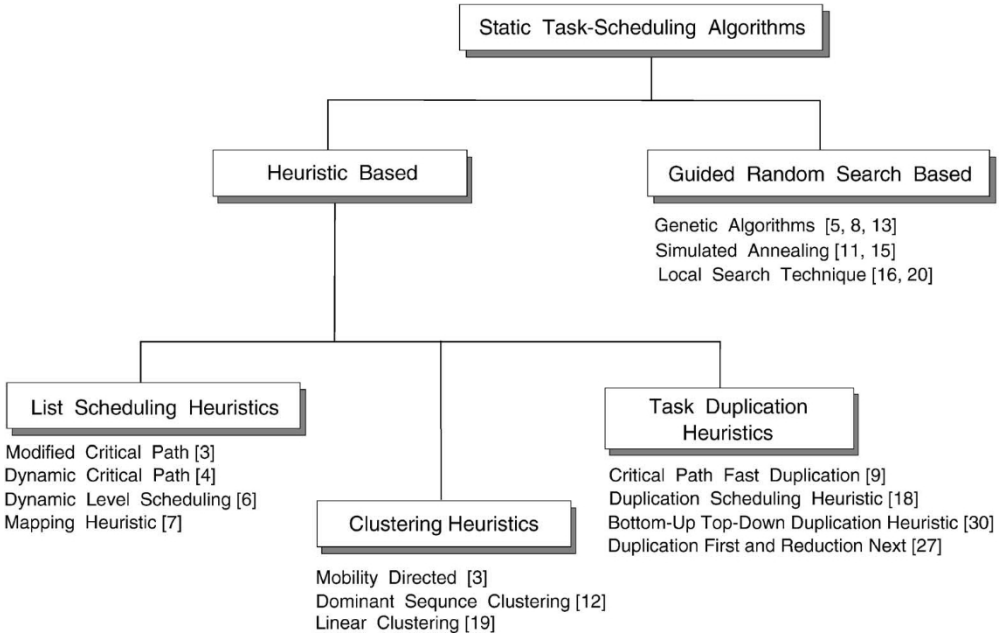


图 1.静态任务调度算法的分类。

这组算法通常适用于数量不受限制的相同处理器，其复杂度值远远高于其他组的算法。引导式随机搜索技术。引导式随机搜索技术（或随机搜索技术）使用随机选择来引导自己通过问题空间，这与随机搜索方法中的随机行走不同。这些技术将从以往搜索结果中获得的知识与一些随机化特征相结合，从而生成新的结果。遗传算法（GA）[5]、[8]、[11]、[13]、[17] 是目前最流行、应用最广泛的技术，适用于多种任务调度问题。GA 能生成高质量的输出调度，但其调度时间通常比基于启发式的技术要长得多 [31]。此外，遗传算法中的几个控制参数也应适当确定。用于任务图调度的最优控制参数集可能无法为另一个任务图提供最佳结果。除遗传算法外，还有模拟退火法 [11]、[15] 和局部搜索法 [16]、[20] 是这一组中的其他方法。

3.1 异构环境的任务调度启发式方法

本节将介绍已报道的支持异构处理器的任务调度启发式算法，即动态水平调度算法[6]、水平化最小时间算法[10]和映射启发式算法[7]。

动态调度（DLS）算法。在每个步骤中，算法会选择使*动态水平*值最大的（就绪节点、可用处理器）对，*动态水平*值等于 $DL(n_i, p_j) = ranh^s(n_i) - E2T(n_i, p_j)$。任务的计算成本是该任务在处理器上计算成本的中值。在该算法中，向上排名计算不考虑通信成本。对于异构环境，一个

任务在所有处理器上的中位执行时间与任务在当前处理器上的执行时间之差。一般 DLS 算法的时间复杂度为 $O(n^3 \times q)$，其中 n 是任务数，q 是处理器数。

映射启发式（MH）。在该算法中，计算任务在处理器上的计算成本是由任务要执行的指令数除以处理器速度计算得出的。不过，在调度前设定任务的计算成本和边的通信成本时，假定处理元素相似（即同质处理器）；而在调度过程中，异质元素就会出现。

这种算法使用*静态*向上等级来分配优先级。(作者还尝试在等级值中加入通信延迟）。在该算法中，处理器的任务*就绪时间*是指处理器完成上一个分配任务并准备执行新任务的时间。MH 算法不会将任务安排到两个已安排任务之间的空闲时隙。在 n *个*任务和 q *个*处理器的情况下，当考虑到竞争时，时间复杂度等于 $O(n^2 \times q^3)$；否则，时间复杂度等于 $O\ n^2\ q$。

平均最小时间（LMT）算法。这是一种两阶段算法。第一阶段使用*级别*属性对可并行执行的任务进行分组。第二阶段将每个任务分配给最快的可用处理器。在同一级别中，计算成本最高的任务优先级最高。每个任务被分配给一个处理器，该处理器应使该任务的计算成本和与前一级任务的总通信成本之和最小。对于全连接图，当有 n *个*任务和 q *个*处理器时，时间复杂度为 $O(n^2 \times q)$。[2]

(×)

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task.
3. Sort the tasks in a scheduling list by nonincreasing order of $rank_u$ values.
4. **while** there are unscheduled tasks in the list **do**
5. Select the first task, n_i, from the list for scheduling.
6. **for** each processor p_k in the processor-set ($p_k \in Q$) **do**
7. Compute $EFT(n_i, p_k)$ value using the *insertion-based scheduling* policy.
8. Assign task n_i to the processor p_j that minimizes EFT of task n_i.
9. **endwhile**

图 2.HEFT 算法

4 任务调度算法

在介绍 HEFT 和 CPOP 算法的细节之前，我们先介绍用于设置任务优先级的图属性。

4.1 HEFT 和 CPOP 算法使用的图属性

在我们的算法中，任务是按其调度优先级排序的，而调度优先级则基于向上和向下排序。任务 n 的*向上排序*递归定义为

$$ranh_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)}(\overline{C_{i,j}} + RANH_u(N_j)), \quad (8)$$

其中，$succ(n_i)$ 是任务 n_i 的直接后继者集合，$c_{i,j}$ 是边 (i, j) 的平均通信成本，m_i *是任务* n_i 的平均计算成本。由于秩是通过从退出任务开始向上遍历任务图来递归计算的，因此称为*向上秩*。对于退出任务 n_{exit}，向上秩值等于

$$ranh_u(n_{exit}) = \overline{w_{exit}} \quad (9)$$

基本上，$ranh_u(n_i)$ 是指从任务 n_i 到退出任务的关键路径长度，其中包括任务 n_i 的计算成本。文献中有一些算法仅使用计算成本计算秩值，称为*静态向上秩*，$ranh^s_u$。

同样，任务 n_i 的*向下秩*递归定义为

$$ranh_d(n_i) = \max_{n_j \in pred(n_i)}\{ranh_d(n_j) + \overline{w_j} + \overline{c_{j,i}}\}, \quad (10)$$

其中，$pred(n_i)$ 是任务 n_i 的直接前置任务集。向下秩的计算方法是从任务图的入口任务开始向下递归遍历任务图。对于入口任务 n_{entry}，向下秩值等于零。基本上，$ranh_d(n_i)$ 是入口任务到任务 n_i 的最长距离，不包括任务本身的计算成本。

任务优先级阶段。该阶段要求根据平均计算成本和平均通信成本，用向上的等级值（$ranh_u$）设定每个任务的优先级。任务列表是通过按 $ranh_u$ 的递减顺序对任务进行排序生成的。打破平局是随机进行的。可以采用其他策略来打破平局，例如选择紧随其后的任务具有更高的向上等级。由于这些替代策略会增加时间复杂度，我们倾向于采用随机选择策略。可以很容易地看出，$ranh_u$ 值的递减顺序提供了任务的拓扑顺序，这是一种线性顺序，保留了优先级约束。

处理器选择阶段。对于大多数任务调度算法来说，处理器的最早可用时间 p_j 是指 p_j 完成其最后一项分配任务的时间。然而，HEFT 算法有一个基于插入的策略，它考虑了在处理器上两个已排定任务之间的最早空闲时隙插入任务的可能性。空闲时隙的长度，即在同一处理器上连续调度的两个任务的执行开始时间和完成时间之差，应至少能够满足待调度任务的计算成本。此外，在该空闲时隙上进行调度应保留优先级约束。

在 HEFT 算法中，在处理器 p_j 上搜索任务 n_i 的合适空闲时隙的起始时间等于 n_i *在* p_j 上的*就绪时间*，即 n_i 的所有输入数据（由 n_i 的前置任务发送）到达处理器 p_j 的时间。

4.2 异构-最早-完成-时间（HEFT）算法

HEFT 算法（图 2）是一种应用调度算法，适用于数量有限的异构处理器。该算法分为两个主要阶段：*任务优先*

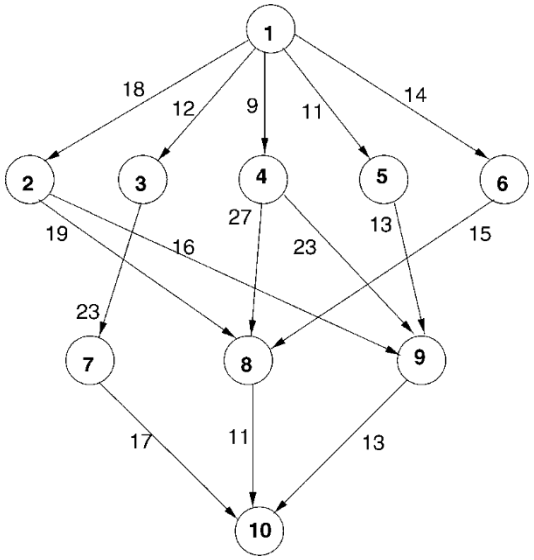
级排序阶段，用于计算所有任务的优先级；*处理器选择阶段*，用于按照优先级顺序选择任务，并将每个选定的任务调度到其 "最佳 "处理器上，从而最大限度地减少任务的完成时间。

对于密集图，当边的数量与 $O(n^2)$ （n 为任务数）成正比时，时间复杂度约为 $O(pn^2)$。

图 4a 展示了 HEFT 算法在图 3 样本 DAG 中得到的时间表。表 1 第一列给出了给定任务图的向上秩值。相对于 HEFT 算法，任务的调度顺序为 $\{n_1, n_3, n_4, n_2, n_5, n_6, n_9, n_7, n_8, n_{10}\}$，n，n，n。

4.3 处理器关键路径 (CPOP) 算法

虽然我们的第二种算法，也就是图 5 所示的 CPOP 算法，具有任务优先级和处理器优先级，但它并不像 CPOP 算法那么简单。



Computation Costs

Task	P1	P2	P3
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

图 3.包含 10 个任务的示例任务图。

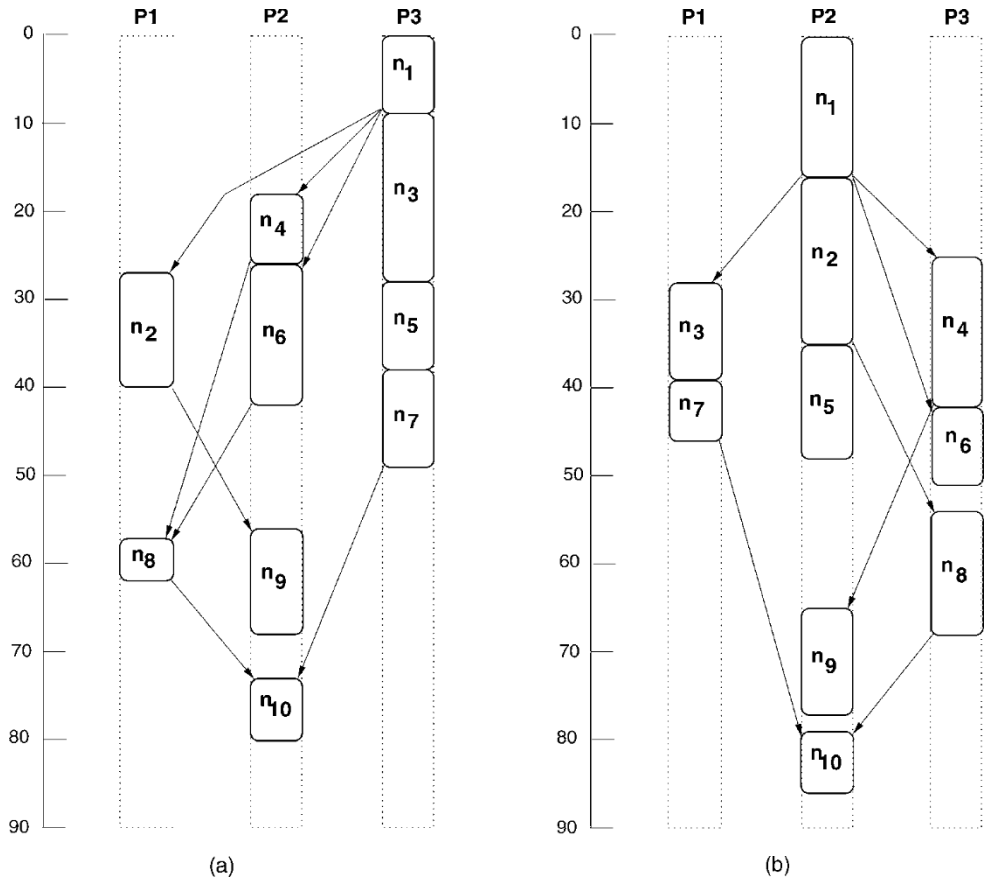


图 4.使用 HEFT 和 CPOP 算法对图 3 中的任务图进行调度。(a) HEFT 算法（计划长度 = 80）。(b) CPOP 算法（计划长度 = 86）。

与 HEFT 算法中的选择阶段一样，它使用不同的属性来设置任务优先级，并使用不同的策略来确定每个选定任务的 "最佳 "处理器。

任务优先排序阶段。在这一阶段，向上排列 $(ranh_u)$ 和向下秩 $(ranh_d)$ 值，计算所有任务的平均计算成本和平均通信成本（步骤 1-3）。CPOP 算法使用临界

的路径。该路径的长度、
$|CP|$，是路径上任务的计算成本和路径上任务间通信成本的总和。图的关键路径上的计算成本之和基本上就是任务调度算法生成的计划长度的 *下限。*

每项任务的优先级由求和分配的上行和下行等级。关键路径长度为

表 1
图 3 任务图中 HEFT 和 CPOP 算法使用的属性值

n_i	$rank_u(n_i)$	$rank_d(n_i)$	$rank_u(n_i)+rank_d(n_i)$
n_1	108.000	0.000	108.000
n_2	77.000	31.000	108.000
n_3	80.000	25.000	105.000
n_4	80.000	22.000	102.000
n_5	69.000	24.000	93.000
n_6	63.333	27.000	90.333
n_7	42.667	62.333	105.000
n_8	35.667	66.667	102.334
n_9	44.333	63.667	108.000
n_{10}	14.667	93.333	108.000

等于入口任务的优先级（步骤 5）。最初，入口任务是选定的任务，并标记为关键路径任务。选择优先级值最高的直接后继任务（选定任务的直接后继任务），并将其标记为关键路径任务。为了打破平局，会选择优先级最高的第一个直接后继任务。

我们维护一个优先队列（键值为 $ranh_u + ranh_d$），以包含任何给定时刻的所有*就绪*任务。使用二进制堆实现优先级队列，插入和删除任务的时间复杂度为 $0(log\ n)$，检索具有最高优先级的任务的时间复杂度为 $0(1)$。每一步都会从优先级队列中选择具有最高 $ranh_u\ ranh_d$ 值的任务。

处理器选择阶段。关键路径处理器、p_{CP}，则关键路径上任务的累计计算成本最小（步骤 13）。

如果任务在关键路径上，则将其调度到关键路径处理器上；否则，将其分配到能最小化任务最早执行完成时间的处理器上。这两种情况都考虑了基于插入的调度策略。CPOP 算法的时间复杂度等于 $0(e \times p)$。图 4b 显示了 CPOP 算法针对图 3 得出的调度，该图的调度长度为 86。根据表 1 中的值，图 3 中的关键路径为 $\{n_1, n, n_{29}, n_{10}\}$。如果所有关键路径任务都安排在 P l、P 2 或 P 3 上，则路径长度将分别为 66、54 或 63。P 2 被选为关键路径处理器。根据 CPOP 算法，任务的调度顺序为 $\{n_1, n_2, n, n_{37}, n, n, n_{459}, n_6, n_8, n_{10}\}$。

5 实验结果与讨论

在本节中，我们将对我们的算法和第 3.1 节中的相关工作进行比较评估。为此，我们将两组图作为测试算法的工作量：随机生成的应用图和代表现实世界中一些数字问题的图。首先，我们介绍了用于性能评估的指标，随后的两节介绍了实验结果。

5.1 比较指标

这些算法的比较基于以下四个指标：

- 调度长度比 (SLR)。图上调度算法的主要性能指标是其输出调度的调度长度（*makespan*）。由于使用了大量具有不同属性的任务图，因此有必要将调度长度归一化为一个下 限 、

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ of tasks by traversing graph upward, starting from the exit task.
3. Compute $rank_d$ of tasks by traversing graph downward, starting from the entry task.
4. Compute $priority(n_i) = rank_d(n_i) + rank_u(n_i)$ for each task n_i in the graph.
5. $|CP| = priority(n_{entry})$, where n_{entry} is the *entry* task.
6. $SET_{CP} = \{n_{entry}\}$, where SET_{CP} is the set of tasks on the critical path.
7. $n_k \leftarrow n_{entry}$.
8. **while** n_k is not the exit task **do**
9. Select n_j where $((n_j \in succ(n_k))$ **and** $(priority(n_j) == |CP|))$.
10. $SET_{CP} = SET_{CP} \bigcup \{n_j\}$.
11. $n_k \leftarrow n_j$.
12. **endwhile**
13. Select the critical-path processor (p_{CP}) which minimizes $\sum_{n_i \in SET_{CP}} w_{i,j}, \quad \forall p_j \in Q$.
14. Initialize the priority queue with the entry task.
15. **while** there is an unscheduled task in the priority queue **do**
16. Select the highest priority task n_i from priority queue.
17. **if** $n_i \in SET_{CP}$ **then**
18. Assign the task n_i on p_{CP}.
19. **else**
20. Assign the task n_i to the processor p_j which minimizes the $EFT(n_i, p_j)$.
21. Update the priority-queue with the successors of n_i, if they become ready tasks.
22. **endwhile**

图 5.CPOP 算法。

这就是所谓的时间表长度比（SLR）。图上算法的 SLR 值定义为

$$SLR = \frac{\text{沼泽}}{\sum_{n_i \in CP_{MIN}} \text{分} \min_{p_j \in Q}\{mi_j\}}. \qquad (11)$$

钟

分母是 CP_{MIN} 上各任务的最小计算成本之和（对于未计划的 DAG，如果将每个节点 n_i 的计算成本设为最小值，则关键路径将基于最小计算成本，用 CP_{MIN} 表示）。图的 SLR（使用任何算法）都不能小于 1，因为分母就是下限。任务调度算法中，SLR 最低的算法就是性能最好的算法。我们在实验中使用了多个任务图的平均 SLR 值。

- 加速。给定图形的加速值是用顺序执行时间（即图形中任务的累计计算成本）除以并行执行时间（即输出时间表的有效期）计算得出的。顺序执行时间的计算方法是，将所有任务分配给单个处理器，使计算成本的累计值最小。

$$\text{加速} = \frac{\min_{pj \in Q}\{\sum_{ni \in V} mi,j\}}{\text{沼泽}}. \qquad (12)$$

如果计算成本之和最大化，则加速度会更高，但调度算法的排名也会 相同。效率，即加速值与所用处理器数量之比，是第 5.3 节中给出的实际问题应用图的另一个比较指标。

- 更高质量计划的出现次数。与其他算法相比，每种算法产生的计划质量更好、更差和相同的次数分别计为

实验。

- 算法的运行时间。算法的运行时间（或调度时间）是指获得输出调度的执行时间

的平均成本。这一指标基本上给出了每种算法的平均成本。在给出可比 SLR 值的算法中，运行时间最短的算法是最实用的。通过检查所有可能的任务-处理器配对来最小化 SLR，可能会与最小化运行时间发生冲突。

5.2 随机生成的应用图谱

最后，它根据时间表计算性能指标。

5.2.1 随机图形生成器

我们的随机图形生成器需要以下输入

参数来构建加权 DAG。

- 图中的任务数 (n)。
- 图的形状参数，α。我们假设 DAG 的高度（深度）是从均匀分布中随机生成的，其均值等于 $\sqrt{n}/\alpha$ 的最小积分）。

  每个层的宽度从均值等于 $\alpha \sqrt{n}$ 的均匀分布中随机选择。）选择 $\alpha >> 1.0$，可以生成密集图（并行度高的较短图）；如果 $\alpha << 1.0$，则会生成并行度低的较长图。
- 节点的出度（*out degree*）。
- 通信与计算比（*CCR*）。它是平均通信成本与平均计算成本的比值。如果一个 DAG 的 CCR 值非常低，它就可以被视为计算密集型应用。
- 处理器计算成本的百分比范围 β。它基本上是处理器速度的异质性因子。百分比值越高，表明任务在不同处理器上的计算成本差异越大；百分比值越低，表明任务在系统中任何给定处理器上的预期执行时间几乎相等。图中每个任务 n_i 的平均计算成本，即 $\overline{m_i}$，从范围为 $[0, 2 \times m_{DAG}]$ 的均匀分布中随机选取，其中 m_{DAG} 是给定图的平均计算成本，在算法中随机设置。然后，系统中每个处理器 p_j 上每个任务 n_i 的计算成本在以下范围内随机设定：

$$\overline{m_i} \times \left(1 - \frac{\beta}{2}\right) \le m_{i,j} \le \overline{m_i} \times \left(1 + \frac{\beta}{2}\right). \qquad (13)$$

在研究中，我们首先考虑了随机生成的应用图。我们使用随机图生成器来生成加权应用 DAG，这些应用 DAG 具有不同的特征，这些特征取决于下面给出的几个输入参数。我们基于仿真的框架允许为随机图生成器使用的参数分配数值集。该框架首先执行随机图生成器程序以构建应用程序 DAG，然 后 执行调度算法以生成输出调

度，最后执行随机图生成器程序、

在每个实验中，这些参数的值为从下面给出的相应集合中分配。一个参数应在一次实验中按其集合中给出的所有值分配，如果这些值有任何变化，应在论文中明确写出。请注意，out_degree 集合中的最后一个值是为实验生成全连接图的图中节点数。

- $2ET_V$ = {20、40、60、80、100}、
- $2ET_{CCR}$ = {0.1，0.5，1.0，5.0，10.0}、
- $2ET_2$ = {0.5，1.0，2.0}、
- $2ET_{out\ 度}$ = {1, 2, 3, 4, 5, n}、
- $2ET_\beta$ = {0.1, 0.25, 0.5, 0.75, 1.0}.

这些组合产生了 2250 种不同的 DAG 类型。由于每种 DAG 类型都会随机生成 25 个 DAG，因此实验中使用的 DAG 总数约为 56*K*。分配多个输入参数，并从一个大集合中选择每个参数，会导致 DAG 的数量增加。

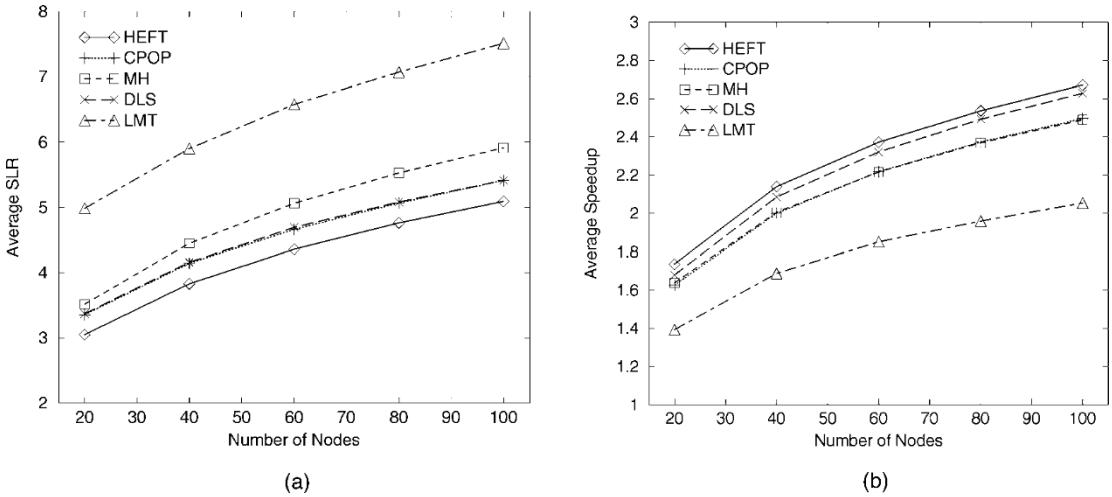


图 6：(a) 平均 SLR 和 (b) 与图大小有关的平均提速。

生成具有不同特征的多样化 DAG。基于不同 DAG 的实验可避免偏向于某种特定的调度算法。

5.2.2 *业绩成果*

根据不同的图特征对算法的性能进行了比较。第一组实验比较了各种图大小下算法的性能和成本（见图 6 和图 7）。基于 SLR 的 算法性能排名为 {HEFT、CPOP、DLS、MH、LMT}。(需要注意的是，本文中的每个排名都是从给定的比较指标中最好的算法开始，以最差的算法结束）。在所有生成图上，HEFT 的平均 SLR 值比 CPOP 算法好 7%，比 DLS 算法好 8%，比 MH 算法好 16%，比 LMT 算法好 52% 。算法的 平均提速排名为 {HEFT、DLS、（CPOP=MH）、LMT}（图 6b）。根据这些实验，无论图的大小如何，HEFT 算法在 SLR 和加速方面都优于其他算法。CPOP 算法在平均 SLR 方面优于相关研究；对于各种图的大小，CPOP 算法的加速值均低于 DLS 算法。在平均运行时间方面（见图 7），HEFT 算法最快，DLS 算法最慢。平均而言，HEFT 算法比 CPOP 算法快 10%，比 MH 算法快 32%，比 DLS 算法快 84%，比 CPOP 算法快 20%。LMT 算法增加了 48%。

下一个实验与图形结构有关。当 *2*（图的形状参数）等于 0.5 时，即生成的图深度较大，并行程度较低，结果表明，HEFT 算法的性能比 CPOP 算法好 8%，比 MH 算法好 5%。

12%，DLS 算法为 16%，LMT 算法为 40%。当 *2 等于* 1.0 时，HEFT 算法的平均 SLR 值比 CPOP 算法好 7%，比 MH 算法好 14%，比 DLS 算法好 7%，比 LMT 算法好 34%。当 *2* 等于 2.0 时，HEFT 算法比 CPOP 算法好

6 MH算法降低了15%，DLS算法降低了8%，LMT算法降低了31%。在所有三种不同的图结构中，HEFT 算法的性能最佳。

在另一项实验中，比较了不同 CCR 值下算法生成的计划质量。当 $CCR \leq 1.0$ 时，算法的性能排名为 {HEFT、DLS、MH、CPOP、LMT}。当
当 $CCR > 1.0$ 时，性能排名变为 {HEFT、CPOP、DLS、MH、LMT}。与 CCR 值较低的图形相比，CPOP 算法能为 CCR 值较高的图形提供更好的结果。在平均通信成本大于平均计算成本的图形中，将关键路径集中在最快的处理器上可获得更好的日程质量。

最后，在使用的 56250 个 DAG 中，统计了实验中每种调度算法与其他算法相比产生更好、更差或相同调度长度的次数。表 2 中的每个单元格显示了左侧算法与顶部算法的比较结果。综合 "一栏显示了左侧算法比所有其他算法综合表现更好、相同或更差的图的百分比。算法排名、

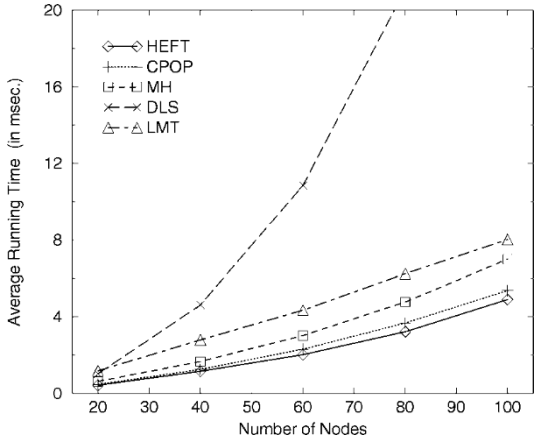


图 7.算法的平均运行时间与图的大小有关。

表 2
调度算法的配对比较

		HEFT	CPOP	DLS	MH	LMT	Combined
HEFT	better		45181	42709	49730	56059	86%
	equal	*	215	802	689	2	1%
	worse		10854	12739	5831	189	13%
CPOP	better	10854		24774	34689	53922	55%
	equal	215	*	108	76	3	< 1%
	worse	45181		31368	21485	2325	45%
DLS	better	12739	31368		44056	55873	64%
	equal	802	108	*	2170	1	1%
	worse	42709	24774		10024	376	35%
MH	better	5831	21485	10024		55342	41%
	equal	689	76	2170	*	6	1%
	worse	49730	34689	44056		902	58%
LMT	better	189	2325	376	902		2%
	equal	2	3	1	6	*	< 1%
	worse	56059	53922	55873	55342		98%

根据最佳结果的出现次数，排序为{HEFT, *DLS, CPOP, MH, LMT*}。然而，按平均可比价格计算的排名是{HEFT、*CPOP*、*DLS*、*MH*、*LMT*}。虽然就最佳结果出现的次数而言，DLS 算法优于 CPOP 算法，但 CPOP 算法的平均 SLR 值略高于 DLS 算法。

5.3 真实世界问题的应用图表

除了随机生成的任务图，我们还考虑了三个实际问题的应用图：高斯消除算法[3]、[28]、快速傅立叶变换[29]、[30]和[19]中给出的分子动力学代码。

5.3.1 高斯消除

图 8a 给出了高斯消除算法的顺序程序 [3]，[28]。图 8b 给出了 $m = 5$（m 是矩阵的维数）这种特殊情况下的算法数据流图。每个 $T_{h,h}$ 代表一个枢轴列操作，每个 $T_{h,j}$ 代表一个更新操作。在图 8b 中，关键路径为 $T\ T\ T\ T\ T\ T_{1,11,22,22,33,33,44,44,5}$， T T这是任务数最多的路径。

在高斯消除应用的实验中，使用了相同的 CCR 和范围百分比值（见第 5.2 节）。由于应用图的结构是已知的，因此我们不需要其他参数，如任务数、out_degree 和形状参数。我们使用一个新参数，即矩阵大小（m）来代替 n (图中的任务数）。
高斯消除图等于$\frac{m^2+m-2}{2}$。

图 9a 给出了当处理器数量等于 5 时，不同矩阵大小（从 5 到 20）的算法平均 SLR 值，增量为 1。本实验中最小的图有 14 个任务， 最大的图有 209 个任务。在所有算法中，HEFT 和 DLS 算法的性能最好。矩阵大小的增加会导致更多任务不在关键路径上，从而导致每种算法的时间跨度增加。

为了进行效率比较，我们在实验中使用的处理器数量从 2 到 16 不等，以 2 的幂次递增；CCR 和范围百分比参数具有相同的值集。图 9b 给出了矩阵大小为 50 时高斯消除图的效率比较。HEFT 和 DLS 算法的效率优于其他算法。当处理器数量增加到 8 个以上时，HEFT 算法的效率优于 DLS 算法。由于矩阵大小是固定的，因此处理器数量的增加会减少每种算法的时间跨度。作为实验的一部分，我们比较了算法在不同处理器数量下的运行时间（保持矩阵大小固定）。结果表明，尽管 DLS 算法的性能与 HEFT 算法不相上下，但它是其 中 最慢的算法。例如，当 16 个处理器的矩阵大小为 50 时，DLS 算法调度给定图形所需的时间是 HEFT 算法的 16.2 倍。综合考虑性能和成本结果，HEFT 算法是其中最高效、最实用的算法。

5.3.2 *快速傅立叶变换*

图 10 给出了递归一维 FFT 算法 [29]、[30] 及其任务图（当有四个数据点时）。图中，A 是一个大小为 m *的*数组，保存多项式的系数，数组 Y 是算法的输出。算法由两部分组成：递归调用（第 3-4 行）和蝴蝶操作（第 6-7 行）。图 10b 中的任务图可以分为两部分--虚线上方的任务是递归调用任务，虚线下方的任务是蝶式运算任务。对于大小为 m *的*输入向量，有 $2m-1$ 个递归调用任务和 $m \log_2 m$ *个*蝶式运算任 务 。(我们假设对于某个整数 h，$m = 2^h$ ）。在 FFT 任务图中，从起始任务到任意出口任务的每条路径都是临界路径，因为任意层任务的计算成本都相等，而且两个连续层之间所有边的通信成本都相等。

× - × =

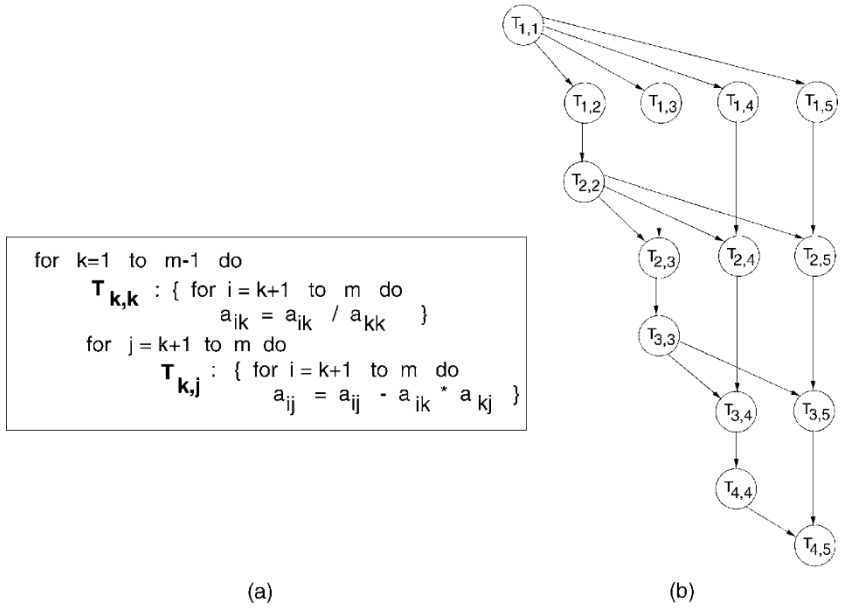


图 8. (a) 高斯消除算法，(b) 大小为 5 的矩阵的任务图。

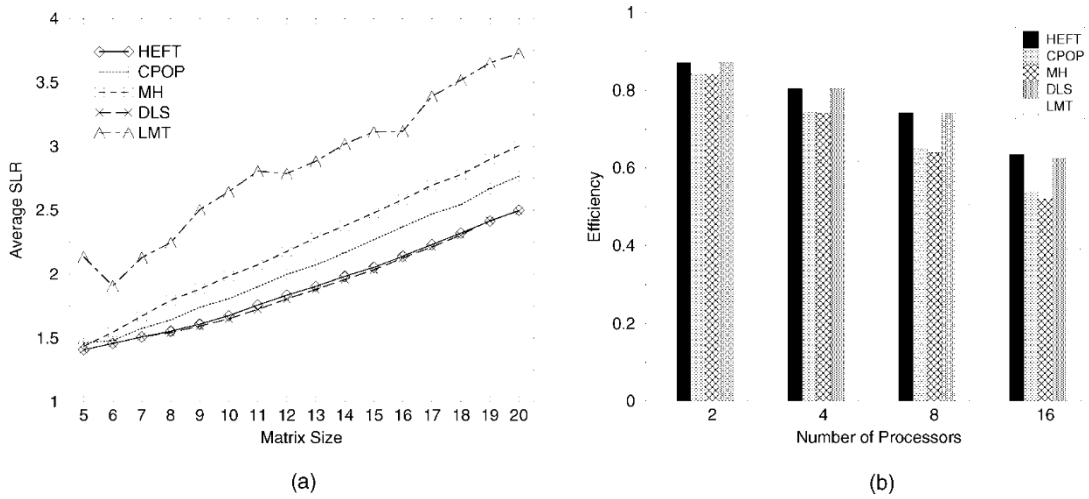


图 9. (a) 高斯消除图的平均 SLR 和 (b) 效率比较。

在与 FFT 相关的实验中，与高斯消除应用一样，只使用了第 5.2 节中给出的参数中的 CCR 和范围百分比参数。图 11a 显示了不同输入点大小的 FFT 图的平均 SLR 值。可以看出，HEFT 算法在大多数情况下都优于其他算法。图 11b 显示了在 64 个数据点的图形中， 每种算法在不同处理器数量下的效率值。在所有情况下，HEFT 和 DLS 算法都给出了最高效的时间表。

当调度算法的运行时间为

根据数据点数量和所用处理器数量对 FFT 图形进行比较

(见图 12），可以看出 DLS 算法是成本最高的算法。请注意，图 12a 中的处理器数量为 6 个，图 12b 中的输入点数量为 64 个。

5.3.3 分子动力学代码

图 13 是 [19] 中给出的修改后分子动力学代码的任务图。该应用是我们性能评估的一部分，因为它具有不规则的任务图。由于应用中的任务数是固定的，图的结构也是已知的，因此我们在实验中只使用了 CCR 和范围百分比参数值（见第 5.2 节）。图 14a 显示了当处理器数量等于 6 时，五种不同 CCR 值的算法性能。而

调度算法的效率比较如下

在图 14b 中，处理器的数量是变化的

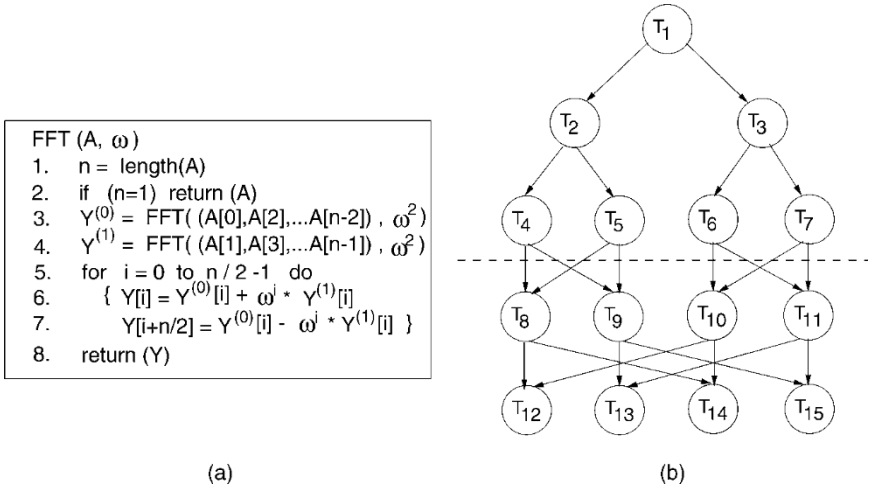


图 10：（a）FFT 算法，（b）生成的四点 FFT DAG。

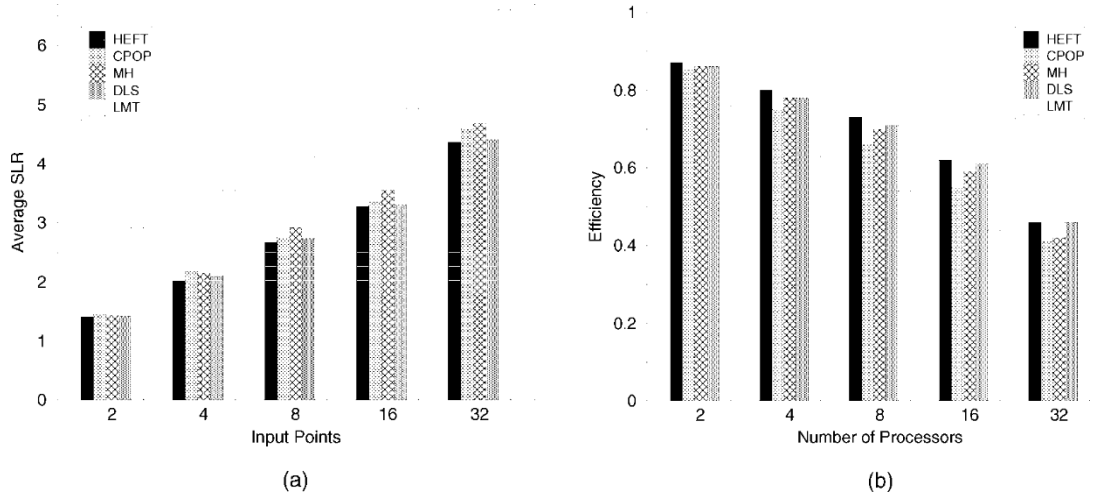


图 11.(a) 平均 SLR 和 (b) FFT 图形的效率比较。

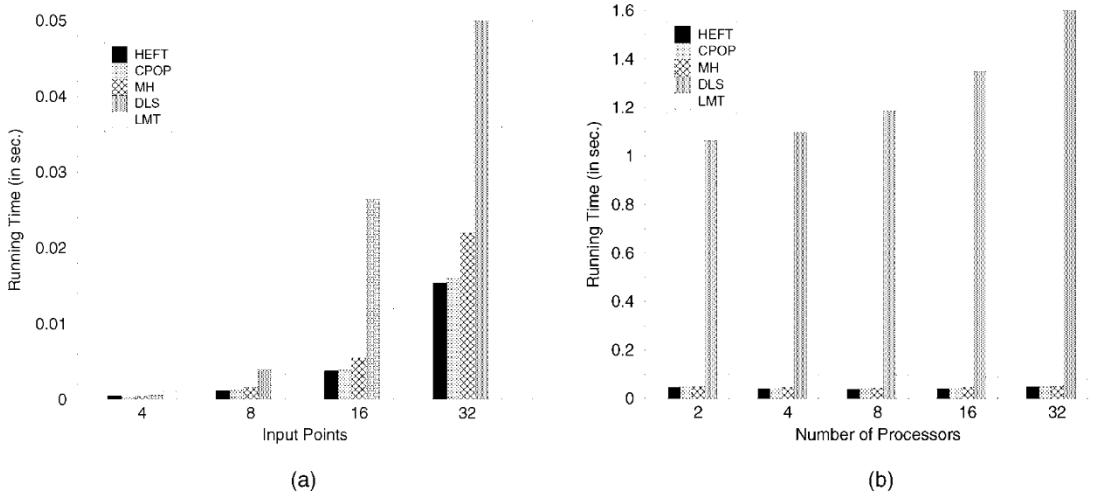


图 12.FFT 图调度算法的运行时间。

由于图 13 中任何一级都最多有 7 个任务，因此实验中的处理器数量最多为 7 个。

处理器。还发现 DLS 和 LMT

算法的运行时间几乎是其他三种算法（HEFT、CPOP 和 MH）的三倍。综合上述结果，HEFT 算法是该应用中最实用、最高效的算法。

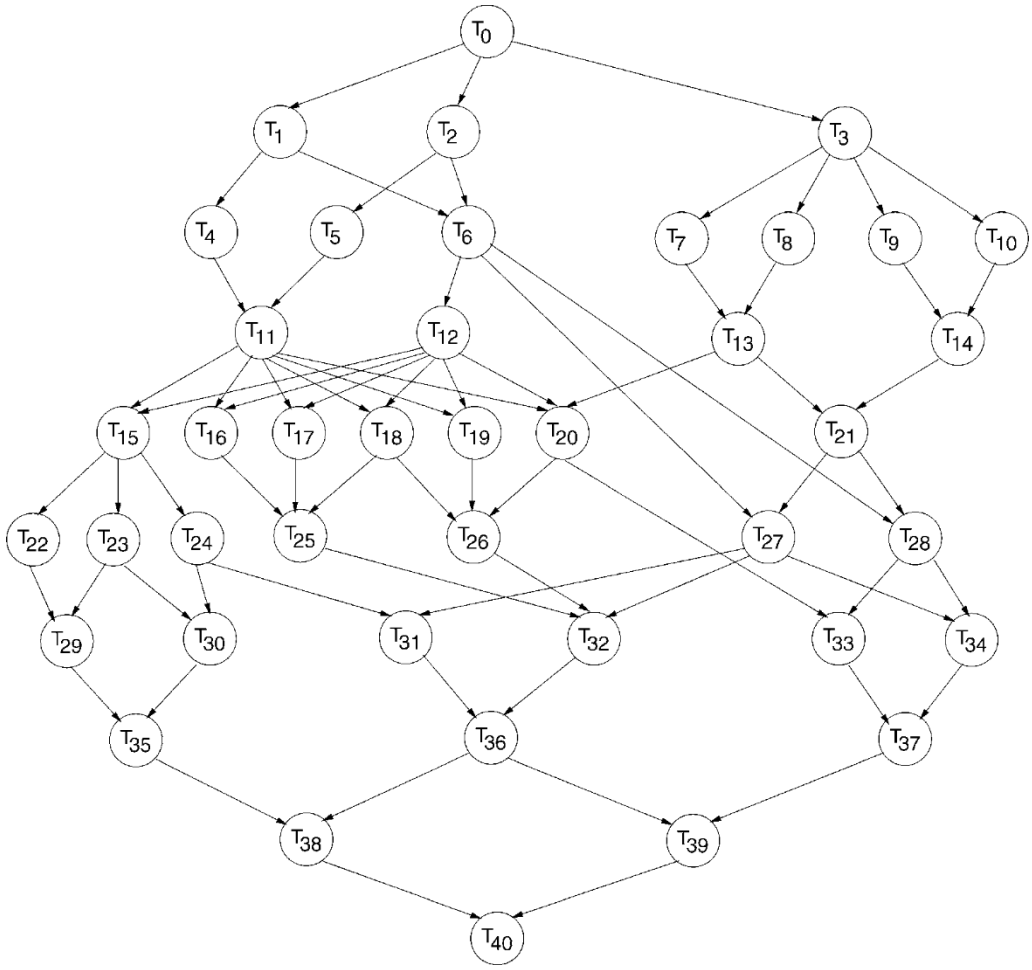


图 13.分子动力学代码的任务图[19]。

6 各阶段的备选政策 HEFT 算法

我们为 HEFT 算法的任务优先级排序阶段提出了三种替代策略（如 A1、A2 和 A3 所示），在实验中，优先级排序以向上等级为基础。在 A1 中，优先级值等于向上和向下等级之和。在 A2 中，符号的右半部分给出了 n_i 的直接前置任务的最新执行完成时间，该任务已经排定。A3 与 A2 类似，只是考虑了通信成本。在实验中，我们发现原始优先级策略的结果比这些替代策略好 6%。

- A1: $priority(n_i) = ranh_u(n_i) + ranh_d(n_i)$、
- AZ：$优先级(n_i) = ranh_u(n_i) + \max_{n_j \in pred(n_i)} AFT(n_j)$
- A3:

$$优先级(n_i) = ranh_u(n_i) + \max_{n_j \in pred(n_i)} \{AFT(n_j) + c_{j,i}\}$$

。

通过扩展，我们可以对 HEFT 算法的处理器选择阶段进行如下修改：对于每个选定的任务，其直接子任务中的一个会根据下面给出的三种策略之一被标记为临界子任务。如果临界子任务的其他直接前置任务已经排定，则将所选任务及其临界子任务排定在同一处理器上，使临界子任务的最早完成时间最小化；否则，与 HEFT 算法一样，将所选任务排定在使其最早完成时间最小化的处理器上。三种关键子任务选择策略（B1、B2 和 B3）要么使用通信成本，要么使用向上等级，要么两者兼而有之。

- B1: $critical\text{-}child(n_i) = \max_{n_c \in succ(n_i)} c$ 、$_{i,c}$
- BZ: $critical\text{-}child(n_i) = \max_{n_c \in succ(n_i)} ranh_u(n_c)$、
- B3:

$$关键儿童\ n_i(\ ) = \max_{n_c \in succ(n_i)} \{ranh_u(n_c) + c_{i,c}\}。$$

另一种可提高性能的扩展方法是将直接子任务考虑在内。例如，对于图 15 所示的任务图，HEFT 算法生成的输出计划长度为 8；如果将任务 A 及其直接子任务（任务 B）安排在同一处理器上，使任务 B 的最早完成时间最小化，则计划长度将减至 7。考虑到这

对于小型 CCR 图，原始 HEFT 算法的性能优于这些替代算法。对于高 CCR 图形，通过在选择处理器时考虑关键子任务，可以观察到一些好处。当 $3.0 \leq CCR < 6.0$ 时，B1 策略略优于原始 HEFT 算法。如果 $CCR \geq 6.0$，B2 策略的性能比原始算法和其他交替算法高出 4%。

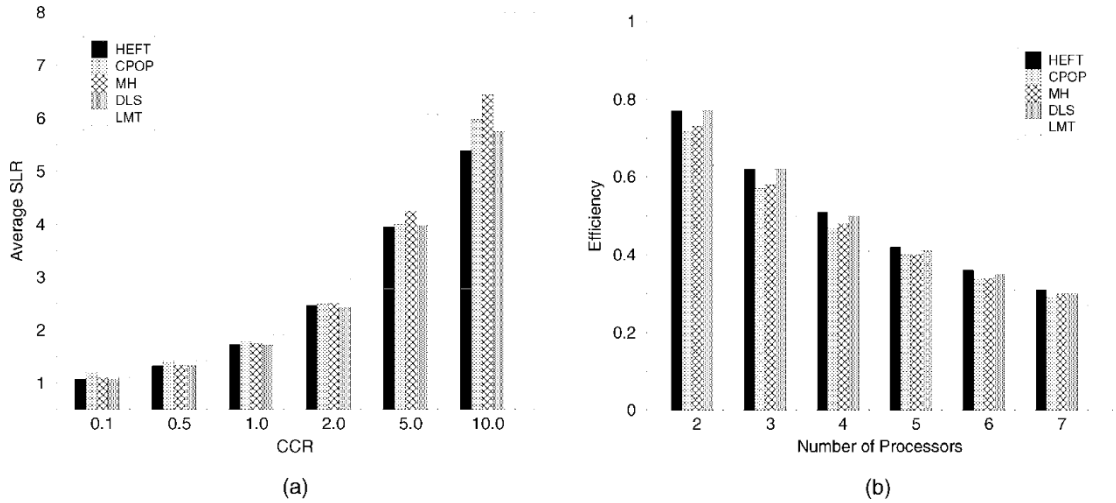


图 14. (a) 分子动力学代码任务图的平均 SLR 和 (b) 效率比较。

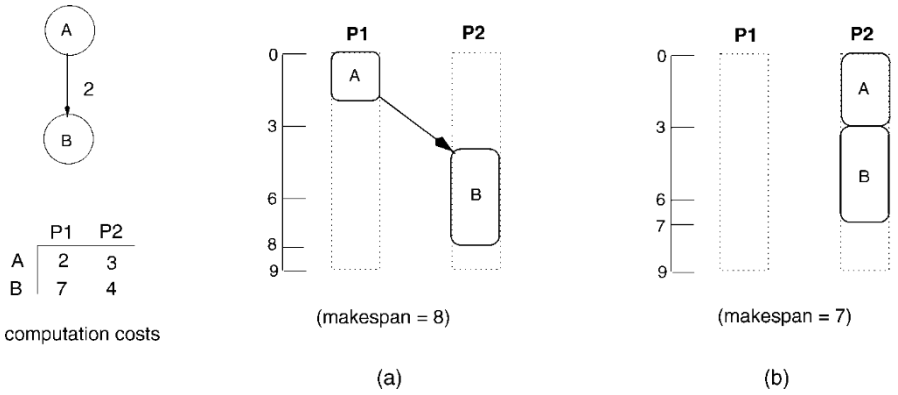


图 15.使用 (a) HEFT 算法和 (b) 其他方法对任务图进行调度。

7 结论

在本文中，我们提出了两种新算法，分别称为 HEFT 算法和 CPOP 算法，用于在异构处理器系统中调度应用图。基于使用大量（56K）随机生成的具有各种特征的应用图和几个实际问题（如高斯消除、FFT 和分子动力学代码）的应用图进行的实验研究，HEFT 算法在性能和成本指标（包括平均调度长度比、加速、最佳结果频率和平均运行时间）方面都明显优于其他算法。由于 HEFT 算法性能稳定、运行时间短，而且能在多种图结构中提供稳定的性能，因此它是异构系统中 DAG 调度问题的可行解决方案。根据性能评估研究，我们还观察到 CPOP 算法比现有算法具有更好的性能和更短的运行时间，或者与现有算法具有可比性。

针对 HEFT 算法的任务优先化和处理器选择阶段，研究了几种可供选择的策略。针对处理器

选择阶段，该阶段试图最小化每个选定任务的关键子任务的最早完成时间。

未来的一项研究计划是分析研究算法的进度质量（即平均有效期值）与可用处理器数量之间的权衡。在可用处理器数量可能不足的情况下，这种扩展可能会对时间跨度的下降提出一些限制。我们计划扩展 HEFT 算法，以便根据处理器和网络负载的变化重新安排任务。虽然我们的算法假定网络是完全连接的，但我们也计划通过考虑链路争用，使这些算法适用于任意连接的网络。

参考资料

[1] M.R. Gary and D.S. Johnson, *Computers and Intractability：W.H. Freeman and Co.*W.H. Freeman and Co., 1979.
[2] J.D. Ullman, "NP-Complete Scheduling Problems," *J. Computer and Systems Sciences, vol.* 10, pp.
[3] M.Wu 和 D. Gajski，"Hypertool：的编程辅助工具消息传递系统》，*IEEE Trans.并行和分布式系统》*，第 1 卷，第 330-343 页，1990 年 7 月。
[4] Y.Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling：将任务图分配给多处理器的有效技术"，《*IEEE Trans.Parallel and Distributed Systems,* vol. 7, no.5, pp.

[5] E.S.H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans.*5, no. 2, pp.

[6] G.C. Sih 和 E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans.*4, no. 2, pp.

[7] H.El-Rewini 和 T.G. Lewis，"在任意目标机上调度并行程序任务"，《*并行和分布式计算*》，第 9 卷，第 138-153 页，1990 年。

[8] H.H. Singh 和 A. Youssef，"使用遗传算法映射和调度异类任务图"，《*异类计算研讨会论文集*》，第 86-97 页，1996 年。

[9] I.Ahmad and Y. Kwok, "A New Approach to Scheduling Parallel Programs Using Task *Duplication*," *Proc.并行处理*》，第 2 卷，第 47-51 页，1994 年。

[10] M.Iverson, F. Ozguner, and G. Follen, "Parallelizing Existing Applications in a Distributed *Heterogeneous* Environment," *Proc. Heterogeneous Computing Workshop,* pp.

[11] P.Shroff, D.W. Watson, N.S. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in *Heterogeneous Environments*," *Proc. Heterogeneous Computing Workshop*, pp.

[12] T.Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans.并行和分布式系统*》，第 5 卷，第 9 期，第 951-967 页，1994 年 9 月。

[13] L.Wang, H.J. Siegel, and V.P. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in *Heterogeneous Computing Environments,*" *Proc. Heterogeneous Computing Workshop,* 1996.

[14] M.Maheswaran 和 H.J. Siegel，"异构计算系统的动态匹配和调度算法"，《*异构计算研讨会论文集*》，第 57-69 页，1998 年。

[15] L.Tao, B. Narahari, and Y.C. Zhao, "Heuristics for Mapping Parallel Computations to *Heterogeneous* Parallel Architectures," *Proc*.

[16] M. Wu, W. Shu, and J. Gu, "Local Search for DAG Scheduling and Task Assignment," *Proc.Parallel Processing*, pp.

[17] R.C. Correa, A. Ferreria, and P. Rebreyend, "Integrating List Heuristics into Genetic Algorithms for Multiprocessor Scheduling，" *Proc.Eighth IEEE Symp.Parallel and Distributed Processing (SPDP '96)*，1996 年 10 月。

[18] B.Kruatrachue 和 T.G. Lewis，"Grain Size Determination for Parallel Processing"，*IEEE Software*，第 23-32 页，1988 年 1 月。

[19] S.J. Kim 和 J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. Int'l Conf.并行处理*》，第 2 卷，第 1-8 页，1988 年。

[20] Y.Kwok, I. Ahmad, and J. Gu, "FAST： A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," *Proc. Int'l Conf.Parallel Processing, vol.* 2, pp.

[21] Y.Kwok and I. Ahmad, "Benchmarking the Task Graph Scheduling Algorithms," *Proc.First Merged Int'l Parallel Pocessing Symp./Symp.Parallel and Distributed Processing Conf.*, pp.

[22] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communica- tion Costs," *SIAM J. Computing, vol.* 18, no. 2, pp.

[23] H.El-Rewini, H.H. Ali, and T. Lewis, "Task Scheduling in Multiprocessor Systems," *Computer*, pp.

[24] J.Liou and M.A. Palis, "A Comparison of General Approaches to Multiprocessor Scheduling," *Proc. Int'l Parallel Processing Symp.*, pp.

[25] J.Liou and M.A. Palis, "An Efficient Clustering Heuristic for Scheduling DAGs on Multiprocessors," *Proc. Symp.Parallel and Distributed Processing,* 1996.

[26] A. Radulescu, A.J.C. van Gemund, and H. Lin, "LLB: A Fast and Effective Scheduling Algorithm for Distributed-Memory Sys- tems," *Proc. Second Merged Int'l Parallel Processing Symp./Symp.Parallel and Distributed Processing Conf.,* 1999.

[27] G. Park, B. Shirazi, and J. Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multi-Processor Systems," *Proc. Int'l Conf.Parallel Processing*, pp.

[28] M.Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, "*Parallel* Gaussian Elimination on an MIMD Computer," *Parallel Computing, vol.* 6, pp.

[29] T.H. Cormen、C.E. Leiserson 和 R.L. Rivest，《*算法导论*》。麻省理工学院出版社，1990 年。

[30] Y.Chung and S. Ranka, "Applications and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors," *Proc. 计算》*，第 512-521 页，1992 年 11 月。

[31] T.T.Braun、H.J. Siegel、N. Beck、L.L. Boloni、M. Maheswaran、A.I. Reuther、J.P. Robertson、M.D. Theys、B. Yao、D. Hengsen 和 R.F. Freund，"异构计算系统上一类元任务的静态映射启发式比较研究"，*异构计算研讨会论文集*，第 15-29 页，1999 年。

Haluk Topcuoglu 分别于 1991 年和 1993 年获得土耳其伊斯坦布尔 Bogazici 大学计算机工程学士和硕士学位。1999 年，他获得纽约雪城大学计算机科学博士学位。他目前是土耳其马尔马拉大学计算机工程系助理教授。他的研究兴趣包括异构环境中的任务调度技术、集群计算、计算机科学和计算机技术。

他在计算机科学、并行和分布式编程、网络技术以及遗传算法等领域拥有丰富的经验。他是电气和电子工程师学会、电气和电子工程师学会计算机协会以及 ACM 的成员。

萨利姆-哈里里于 1982 年获得俄亥俄州立大学理学硕士学位，1986 年获得南加州大学计算机工程博士学位。他是亚利桑那大学电气与计算机工程系教授，也是先进远程系统中心（CAT）主任：下一代网络中心系统中心主任。他是《*集群计算》（Cluster Computing*）的主编：集群计算》*杂志的主编。*

网络、软件工具和应用。他目前的研究重点是高性能分布式计算、基于代理的主动和智能网络管理系统、高速网络的设计和分析，以及为高性能计算和通信系统及应用开发软件设计工具。他与他人合作撰写了 200 多篇期刊和会议研究论文，并著有《*高性能分布式计算》*一书：*网络、架构和编程》一书的作者，该书*将于 2002 年由 Prentice Hall 出版。他是电气和电子工程师学会计算机协会会员。

吴旻佑获得中国北京中央研究院研究生院硕士学位和美国加州圣克拉拉大学博士学位。他是新墨西哥大学电子与计算机工程系副教授。他曾在伊利诺伊大学厄巴纳-香槟分校、加州大学欧文分校、耶鲁大学、雪城大学、新墨西哥州立大学担任过各种职务。

他的研究兴趣包括并行和分布式系统、并行计算机编译器、编程工具、VLSI 设计和多媒体系统。他的研究兴趣包括并行和分布式系统、并行计算机编译器、编程工具、超大规模集成电路设计和多媒体系统。他在上述领域发表了 90 多篇期刊和会议论文，并编辑了两期关于并行操作系统的特刊。他是 IEEE 高级会员和 ACM 会员。他被列入*国际信息技术名人录和美国名人录*。

d 有关此主题或任何计算机主题的更多信息，请访问我们的数字图书馆 http://computer.org/publications/dlib。