分 类 号 _____ 学 号 M201372807

学校代码 10487 密 级 _____

华中科技大学

硕士学位论文

基于任务复制和插入的分布式任务调度算法研究

学位申请人： 潘志舟

学 科 专 业： 计算机技术

指 导 教 师： 何 琨 副教授

答 辩 日 期： 2015 年 5 月 26 日

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Engineering

Task-Duplication and Insertion Based Scheduling Algorithm for Heterogeneous Computing Environments

Candidate : Zhizhou Pan

Major : Computer Technology

Supervisor : Associate Prof. Kun He

Huazhong University of Science & Technology

Wuhan, Hubei, 430074, P. R. China

May, 2014

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到，本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保密□ ，在_____年解密后适用本授权书。

本论文属于

不保密□ 。

（请在以上方框内打“ √ ”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘　要

具有约束关系任务的最优化调度在并行和分布式计算领域扮演着重要的角色。它研究如何将具有约束关系的多个任务分配到不同的处理机上执行，以最小化调度的长度(即Makespan)。分布式多任务调度问题是NP完全的，除非P=NP，否则在多项式时间内不能得到精确解。近几十年来，研究此问题的学者不断在探索怎样以更低的时间复杂度和更少的资源开销来近似求解该调度问题。

基于任务复制的启发构造策略普遍被认为是一个好的求解方向，在基于任务复制的模式下，同时结合任务间有空闲可插入的思想，重点针对异构环境设计了基于任务复制及插入的聚簇和归并算法（Task-Duplication and Insertion based Clustering and Merging Algorithm (TDICMA)）。算法主要包括初始分簇，任务复制，任务归并，任务插入等阶段。基于任务复制的分布式任务调度算法存在链式反应现象（chain reaction phenomenon），本轮无效的任务复制可能在下一轮就变成是有效的了。传统的基于任务复制的任务调度算法因为没有考虑到链式反应现象，导致算法结果不够理想。而TDICMA充分考虑到了连锁链式反应。

最后，实现开发了分布式任务调度算法的综合比较系统，可以自动随机生成大量测试用例，通过这个系统详细分析了有向无环图(Directed Acyclic Graph, DAG)中各部分属性对算法结果和性能的影响。基于此系统可以与国内外典型的分布式调度算法进行比较，实验重点比较了DCPD，HEFT，TANH算法，实验结果表明，TDICMA不仅具有很好的时间复杂度，同时也大大减少了调度长度。DAG参数的分析比较说明，TDICMA的其他性能如算法稳定性、适用性等方面也明显优于实验过程中比较的算法。

关键词：有向无环图，任务复制，异构环境，分布式计算 ，最优化调度

Abstract

Optimal scheduling of parallel tasks with precedence relationships in heterogeneous environments plays a significant role in the area of parallel and distributed computation. This problem is NP-hard problem and can't get deterministic solution at polynomial time unless P= NP. In recent twenty years, researchers have tried to solve this NP-hard problem with higher quality and lower complexity.

Task duplication-based (TDB) scheme is a well-known strategy, and the insertion strategy can also be considered based on the TDB scheme. We redesigned several key steps for the TDB process including the parameter calculation, task duplication, task merging and task insertion, and propose a Task-Duplication and Insertion based Clustering and Merging Algorithm (TDICMA). There exists a phenomenon of chain reaction during the process of the task duplication, i.e. a duplication that has been evaluated to be ineffective may become effective again after other duplication has been exerted. Traditional TDB algorithm has not considered this chain reaction, while TDICMA utilizes this chain reaction to get better performance.

In addition, a system for comparing distributed tasks scheduling algorithms is developed, which can automatically generate representative test cases. Through this system, we analyze the effects of DAG parameters in detail by comparing with typical algorithms published in the literature. The experiments with DCPD, HEFT and TANH show that TDICMA has improved the results of previous researches in the aspect of the quality of the schedules at the same time without increasing the computational complexity.

Keywords: Directed acyclic graph, task duplication, heterogeneous environments, distributed computation, optimal scheduling

目 录

1 绪论

分布式计算环境中需要解决一个大问题时，当计算量超过一定规模，单机处理有瓶颈的时候，往往采取的策略是不断的分解-合并来求解问题，即将这些大问题分割成很多小问题，由单机分别独自来处理这些小问题，各个单机通过网络传输各自的执行结果，然后将各自的计算结果合并，共同解决这个大问题，这在并行算法，集群网格计算等领域有重大的实际应用价值。

分布式系统环境中的应用程序需要 CPU，内存空间，网络带宽等一些资源，如何高效合理的分配这些资源给应用程序是分布式系统环境中重点需要考虑的问题,这里将分布式系统中的应用程序集统一抽象成任务集，资源集合抽象成处理机器集，则求解的问题抽象成如何把这些具有相关性关系的任务分配到处理能力不同的机器上进行处理，使得尽快处理完所有任务。该问题是 NP 完全的[1]，研究者们用启发搜索的方式近似求解该问题，在目前所有启发搜索的方案中，主要有以下几种启发方式，基于优先级列表，基于复制，基于分簇的方式。

在实际的分布式计算环境中，根据任务的约束关系，传输时间，机器的执行时间等属性是否可以事先得到，将解决此类调度问题的算法分成两种，动态任务调度和静态任务调度。动态调度算法在算法过程中获取 DAG 图的属性内容，机器集的计算时间等数据信息，这些信息经常变化，还得考虑数据延迟，数据误差等因素，动态调度算法更加接近于实际应用环境，不仅以最小化调度长度为目标，还以算法的稳定性，容错性，完备性等为算法指标。静态调度算法是研究动态调度算法的基础，本文研究的算法属于静态调度算法，DAG 的属性内容，机器集的计算时间等数据信息都是事先已知的。

1.1 研究背景及意义

新时期下，随着“互联网+”模式的提出，各种多终端多传感器应运而生，极大推动了移动互联网和物联网的兴起，数据量也呈现爆炸式增长，更是催生了云计算等

新兴领域的发展，另外，在气候变化模拟，图像处理，DNA 匹配，密码加密和解密等领域也涉及到巨大规模数据的计算。

虽然现在单机的性能在不断提高，甚至有更多的超级计算机的出现，但是考虑到成本代价等因素，传统的单机执行模式所带来的瓶颈和局限越来越大。让人欣慰的是，现在网络传输技术发展迅速，单机之间的传输非常方便，各个机器之间都是互联的，这给分布式环境下的任务计算模型(分解-合并)提供了物理基础。

分布式任务最优化调度问题有重大的理论和技术研究价值，该问题广泛应用于实际领域，如在基因工程中进行 DNA 匹配，图形图像识别，气候预测，地质勘探等。

1.2 国内外研究现状

一直以来，算法领域当中 NP 难问题的求解具有很大挑战性，一般的求解思路包近似算法，概率算法，并行算法，以及智能算法等角度。分布式任务调度问题属于 NP 难问题，近几十年来，研究此问题的学者纷纷从各种角度对问题进行近似求解。很多学者按照算法思想和不同的实际背景对求解该问题的算法类型进行了归纳和分类[2]。

根据 DAG 图属性内容是否可以事先得到，将求解调度问题的算法分为静态调度算法[2,4,5,6,7,8]和动态调度算法[3]。根据分布式计算的实际环境，可以分成求解同构环境[9,10,11,12,13]和异构环境[18,19,20,21,22]的算法；从启发式策略角度来看，包括基于优先级列表的调度[22,23,27,32,39],基于聚簇的调度[10,12,14,28,38]和基于任务复制的调度[15,16,24,37,40].

基于优先级列表和基于聚簇是完全不同的两种启发策略,前者主要是通过合理构造任务的优先级来对任务进行调度，而后者以簇为单位，不断的将任务加入到不同的簇中，这两种策略通常不能融合在一起使用,但这两种策略都可以融合基于复制的策略。

1.2.1 基于优先级列表的调度

基于优先级列表的调度首先根据任务的优先级产生任务的先后执行顺序,然后按照列表中的顺序分别将任务分配在处理机上执行.这类算法的研究主要区分在如何定

义任务的优先级和分配方式上[27,35,36]。

K.shin 等人[17]提出了三种定义任务优先级的方式：S-level，T-level，B-Level。S-level 是指在不考虑传输延迟的情况下从当前节点到出口节点的最长路径(只考虑任务的处理时间)。B-level 是指考虑传输的延迟的情况下从当前节点到出口节点的最长路径。T-level 是指在考虑传输延迟的情况下从入口节点到当前节点的最长路径.稍微复杂的做法是同时考虑上述三种方式下的优先级别并分别给予不同的权重，还有一种定义任务优先级的是将处于关键路径的任务赋予高优先级别[40]，这种方式的提出是基于如果这些关键路径上的任务被延迟了，则有很大的可能会推迟整个应用程序运作。

在任务的优先级别定义好后,基于列表的任务分配方式通常有基于插入[6,12,13]和基于非插入[29,31,32,34]。在基于插入的模式下，如果发现处理机上的相邻任务之间有足够的空闲时间可供另外一个任务插入，则可以选择插入相应的任务从而使得整个调度长度减少，如当前处理器上相邻的两个任务 *i* 和任务 *j*,任务 *j* 需要所有前驱任务完成时才能开始，此时 *j* 可能需要等待一段时间，则这段等待时间就是在该处理机上的空闲时间。

1.2.2 基于聚簇的调度

基于聚簇的调度是指首先将任务分簇,然后将同一个簇上的任务分配给同一个机器.利用这种方式可以通过合并不同的簇上的任务来减少传输延迟,从而提高任务调度的效率.在关于聚簇的调度方案中，一种是处理的机器是有限数目的，还有一种是处理机器数目是无限的情况，在两种不同的情景下面得到的聚簇调度效果是显著化的[14,25,26,28]。

1.2.3 基于任务复制的调度

基于任务复制的调度(Task-Duplication-Based, TDB)是指在有必要的时候将任务分别复制在不同的处理机上执行，从而不需要等待前驱从其他处理机上传输数据过来所需要的传输延迟。因此同一个任务可能有多份拷贝在不同的机器上执行.基于任务复

制的策略通过复制任务到处理机上消耗了机器集的计算代价，但是节省了任务在不同机器间需要传输的通信开销，各种基于 TDB 的算法[24,30,33,37,40]的关键主要是寻找最合适的需要复制的任务，使得调度长度减少。

1.3 研究内容

本课题的研究内容主要是在分布式计算环境下，形式化任务调度问题，定义相关优化目标，使得能够利用网络互连的多台机器解决在普通单机模式下不能够解决的复杂计算问题。

本文将分布式的任务调度问题归约成如何高效的分配任务集上的任务在处理机器上使得任务集上的任务都尽可能早的完成，以最小化调度长度（makespan）为优化目标。

1.4 组织结构

第二章，问题描述和定义。首先给出分布式系统环境下任务调度的问题描述和形式化定义。

第三章，相关求解算法介绍。首先根据分布式系统实际环境介绍了同构调度算法和异构调度算法，在根据算法特点分布介绍了基于优先级列表(HEFT,DCPD)，基于聚簇(MJD,DCP)，基于任务复制(TDS,TANH,DCPD)的典型算法求解。

第四章，介绍了基于任务复制及插入的聚簇和归并求解算法(TDICMA)，主要描述了问题求解的基本流程，包括参数计算，生成初始聚簇，任务复制，任务归并，选取迭代次数，任务复制插入等。然后根据一个实例来具体阐释 TDICMA 的各个算法流程，最后一部分给出 TDICMA 算法的时间复杂度分析。

第五章，实验部分。实现了一个分布式任务调度算法求解的比较系统，能够产生大量实验用例，可以分析分布式系统环境中不同属性参数对调度结果的影响。首先对各个算法的总的调度结果做了统计分析，然后分析比较了各个参数对算法的影响，最

后还分析比较了链式反应现象，主要比较了 HEFT，DCPD，TANH 算法。

第六章，总结与展望。总结了本文的研究工作，并对今后研究的改进方向做了展望。

2 问题描述和定义

2.1 问题形式化描述

分布式计算当中最核心的问题就是任务调度问题，这里将机器集上的任务调度问题抽象描述成在有向无环图上分配任务到最合适的机器，使得最小化调度长度。

在分布式系统环境中，包括同构和异构的情况，所谓同构是指不同的机器对同一个任务有一样的处理时间，否则就属于异构的情况. 同构调度问题是解决异构调度问题的基础。

这里的任务约束是指满足如下约束条件:

1 任务约束。执行过程是原子的，一旦任务放在机器上执行的时候，就不可中断。

2 链路约束。对于有优先关系的任务，对于当前任务 *i*，只有所有直接前驱任务的数据都到达了，当前任务 *i* 才可以开始执行。

3 资源约束。同一台机器上的同一个时间段只能执行单个任务，不可并行执行，不能重叠。

2.2 问题形式化定义

一般用加权的有向无回路图（*Directed Acyclic Graph,DAG*）*G = (V, E, P, T, C),*来形式化定义这个问题.

*V={1,2,...,n},*表示任务集合，*n*是任务的数目.

*E={(i, j): i, j ∈V; i→j},*表示约束边关系集合，说明任务*j*依赖于任务*i*.

P={1, 2, ..., m}，可用的处理机的集合，*m*表示可用的处理机的数目.

*T*为 $n \times m$ 表示的二维矩阵，对于 $i \in V$ 且 $p \in P$，$T(i, p)$ 就表示任务*i*在*p*处理机上所需要的计算代价.

*C*表示 $n \times n$ 的二维矩阵，对于 $i \in V$，$j \in V$ 且 $(i, j) \in E$，那么 $C(i, j)$ 就表示当任何*i*和任务*j*分配在不同机器上时将前驱的数据传输给后继任务时所需要的传输延迟、

$PRED(i)=\{j:(j,i)\in E\}$ 表示i的前驱任务集

$SUCC(i)=\{j:(i,j)\in E\}$ 表示i的后继任务集

我们的求解的目标是如何将任务集V上的任务分配到机器集合P，满足边约束关系E，使得在所有任务都被执行后，任务集上的任务都尽可能快的被完成，使得任务的实际完成时间与任务的开始时间的差尽量小，即最小化调度长度（makespan）

2.3 本章小结

本章首先对分布式系统环境下任务调度问题进行形式化描述，将其抽象描述成了一个数学问题，同时强调了同构情况和异构情况的不同特点。另外给出问题的数学定义，规划出了求解的目标。

3 文献综述

3.1 同构调度算法分析

同构系统中任务调度问题是解决异构情况下任务调度问题的基础。它最早也是起源于分布式内存机器中任务的优化调度问题，并在并行计算，集群，工作流项目中得到进一步研究。

分布式系统同构环境下，处理器集合里面所有的机器对任务集的处理能力相同，则任务 i 在处理器集合 P 的计算时间 $T(i,p_1)=T(i,p_2)=.....=T(i,p_m)$，因此这里将任务 i 在处理机器上的执行时间简化成 $T(i)$.

3.1.1 TDS 算法

同构系统环境下的 TDS（*Task Duplication based Scheduling*)算法[4]跟异构 TDS，TANH 算法是一脉相承的，都是采用了基于聚簇和基于任务复制的策略，TDS 算法首先由 *DAG* 图的拓扑序列计算每个任务的最早可能开始时间 *est* 和最早可能结束时间 *ect*，以及关键任务前驱 *fpred*，另外还定义了任务的最晚可能开始时间 *last* 和最晚可能结束时间 *lact*。具体相关定义如下：

$$est(i)=\min_{j\in PRED(i)}\{\max_{k\in PRED(i),k\neq j}\{ect(j),ect(k)+C(k,i)\}\} \quad \text{(式 3.1)}$$

$$ect(i)=est(i)+T(i) \quad \text{(式 3.2)}$$

$$fpred(i)=\{j\mid ect(j)+C(j,i)\geq ect(k)+C(k,i)\}_{j\in PRED(i),k\in PRED(i)} \quad \text{(式 3.3)}$$

$$lact(i)=\min_{j\in SUCC(i)}\{^{last(j),i=fpred(j)}_{last(j)-C(j,i),i\neq fpred(j)} \quad \text{(式 3.4)}$$

$$last(i)=lact(i)-T(i) \quad \text{(式 3.5)}$$

式 3.1 的主要想法是基于任务 i 会受到所有前驱任务的影响，所以应该选择一个合适的前驱和任务 i 放在同一台机器上使得任务 i 的所有前驱任务对其的影响最小。

式 3.3 说明了在任务 *i* 的所有前驱任务中，我们定义数据到达 *i* 最晚的那个前驱为关键前驱，即对任务 *i* 造成最大限制的任务为关键前驱。式 3.4 是按照 DAG 图的逆拓扑序列来求得，定义这个参数主要的思想是对于当前处理的每一个任务，可能都会影响到所有后继任务的执行，那么我们的目标就是在不影响后继任务尽可能早执行的同时，当前任务最晚可以什么时候结束，式中假设我们是将我们的关键前驱任务和当前任务放在同一个处理器上的。

最后我们计算每个任务到出口节点的最长路径（忽略传输延迟），定义为 *level* 如下：

$$level(i)=\max_{j\in SUCC(i)}\{level(j)+T(j)\} \qquad \text{(式 3.6)}$$

接下来算法流程如下：

算法 3.1：同构 TDS

Input：DAG 图，任务在机器集上的执行时间

Output：任务在机器集上的调度结果

```
1   将 level 的值进行升序排序，然后加入到处理队列中来
2   While 队列不为空
3       取出当前任务 i，将其分配到新的空闲机器上
4       While i 不是 入口节点
5           j=fpred(i)
6           If j 被分配过
7               if last(i)-lact(j)>=C(j,i) 成立
8                   j=i 的任一个未被分配过的前驱
9               else
10                  在 i 的前驱中找一个未分配任务 k，使得 k 满足 ect(j)+C(j,i)=ect(k)+C(k,i)
11                  j=k
12          Endif line 6
13          将 j 加入到这台机器上
```

```
14                i=j
15            Endwhile line 4
16      Endwhile line 2
```

3.1.2 MJD 算法

MJD 算法[12]是基于聚簇和任务复制的混合启发调度算法，该算法引入了粒度的概念，首先，算法定义粒度如下：

$$g_1(i) = \min\{T(j) \mid (j,i) \in E\} / \max\{C(j,i) \mid (j,i) \in E\} \quad \text{(式 3.7)}$$

$$g_2(i) = \min\{T(j) \mid (i,j) \in E\} / \max(C(i,j) \mid (i,j) \in E\} \quad \text{(式 3.8)}$$

$$g(i) = \min(g_1(i), g_2(i)\} \quad \text{(式 3.9)}$$

$$g(G) = \min\{g(i) \mid i \in V\} \quad \text{(式 3.10)}$$

式 3.7 和式 3.8 分别定义节点 *i* 在入边方向和出边方向的粒度，然后式 3.9 定义节点 *i* 的粒度是入边方向的粒度和出边方向的粒度的较小者；式 3.10 定义图 G 的粒度就是所有节点粒度中的最小值。

该算法不仅给出了理论下限，而且对其给出了严谨的分析和证明，算法给出的下限是这样的：若存在 0<ε<1，满足 $g(G) \geq (1-\varepsilon)/\varepsilon$，则 MJD 算法能够保证最后得到的调度长度小于最优调度的(1+ε)倍。

MJD 算法首先求解每个任务的最早开始时间 *est(v,C)*和对应的簇 *C(v)*,其中注意到这里 *v* 的所有祖先节点的最早开始时间和对应的簇都是已经计算完成的了，而且计算的顺序是满足图的拓扑排序的，可以发现，对于 *C(v)*集合里不是 *v* 的祖先，则删除这些任务不会提高 *est(v,C)*的值，因此可以判定 *C(v)*簇只有 *v* 和 *v* 的祖先任务。

让 *C(v)*是包含 *v* 和 *v* 的部分祖先的集合，定义如下边集合：

$$Cross(u,w) = \{(u,w) \mid u \in C(v), w \notin C(v), u \sim w \sim v\} \quad \text{(式 3.11)}$$

其中 *u~w~v* 表示 *u* 到 *v* 经过 *w* 可达。则计算 *est(v,C)*的时候有两种情况：

情况 1：不考虑 $Cross(u,w)$，则 *est(v,C)*满足如下值，其中 *GREEDY _ SCHEDULE* 的结果是单机情况下任务调度的最优调度长度。

$$est(v,C) \geq GREEDY\_SCHEDULE(C(v)-\{v\}) \quad \text{(式 3.12)}$$

情况 2：考虑 $Cross(u,w)$，则我们可以知道在集合 $Cross(u,w)$ 中，对边 (u,w) 我们定义数据准备的时间 $drt(u,w)$)如下：

$$drt(u,w) = est(u) + T(u) + C(u,w) \quad \text{(式 3.13)}$$

任务 *v* 可以开始调度的时间是受到 $drt(u,w)$ 的限制的，只有当 (u,w) 上的数据都准备好了任务 *v* 才可以开始执行。因此我们如下定义 $est(v,C)$ 的值：

$$est(v,C) \geq \max\{^{GREEDY\_SCHEDULE(C(v)-\{v\})}_{\max\limits_{(u,w)\in Cross(u,w)}\{drt(u,w)\}}\} \quad \text{(式 3.14)}$$

则任务 *v* 的最早开始时间定义如下：

$$est(v) \geq \min_{c\in C}\{est(v,c)\} \quad \text{(式 3.15)}$$

随着 *C(v)*的扩大， $GREEDY\_SCHEDULE(C(v)-\{v\})$ 是递增的，因此当满足如下条件：

$$GREEDY\_SCHEDULE(C(v)-\{v\}) \geq drt(Cross(u,w)) \quad \text{(式 3.16)}$$

则继续加入任务到 C(v)当中没有意义的。因此 MJD 构造 *C(v)*的方法是：初始的时候令 *C(v)={v}*，然后在不满足式 3.16 的时候不断加入 *v* 的直接前驱任务，否则构造结束。

最后，MJD 算法按照如下两个步骤不断迭代的方式来构造调度方案，最一开始，除了出口节点设为标记外，剩余节点任务都是未标记状态，然后不断迭代如下两个步骤，直至全部任务节点都是标记。

1 选择一个标记的任务 *v*，将 *C(v)*加入调度方案。

2 将 *v* 设置为未标记，将 $Cross(u,w)$ 中的所有源节点设为标记。

3.1.3 DCP 算法

DCP 算法由何琨，赵勇，黄文奇学者提出[34,35]，在 MJD 算法的基础了上做了几个方面的优化：

(1) MJD 利用任务 v 在对应的簇中的最早可能开始时间 $est(v,C)$ 来计算任务的最早可能开始时间，而 DCP 改进了这一点，直接根据任务 v 的最早可能时间 $est(v)$ 来计算。

(2) 不用判定 Eq.12，选取任务加入当前簇的条件不同，DCP 采取的策略是只要该任务不会使 $est(v)$ 增大，则就加入任务。这种方案充分考虑了任务复制的特点，以减少调度长度为目标，尽量将任务的祖先都加入进来，从而减少任务间的传输延迟。

(3) 在考虑计算 $est(v)$ 的时候，同时将任务和簇作为粒度，这样可以更快速的计算出该值。

DCP 算法首先以簇为粒度，根据跨簇任务的数据的到达时间为准进行贪心调度，然后选择复制任务的候选集，复制任务候选集的选取主要是以当前任务的最早可能开始时间 *est* 不在减少的关键任务或者是关键簇为参考标准。基于上述几个方面的优化，DCP 算法给出了比 MJD 更好的性能下界并也对其作出了证明。

3.2 异构调度算法分析

异构环境在实际分布式系统中更加常见，不同的处理器对相同的任务具有不同的处理能力，这对同构调度算法带来了更多的变化，因此有的学者对同构调度算法做改进使得适用于异构环境，也有的学者利用了新的启发性构造方法对异构任务调度进行求解。异构调度算法在工作流处理，图形图像渲染，并行任务计算，生物 DNA 匹配等领域有重大的实际应用意义。

3.2.1 异构 TDS 算法

异构 TDS[5]（Task Duplication based Scheduling）是基于同构 TDS 调度算法的。算法流程跟同构 TDS 类似，首先计算任务最早的可能开始时间，最早可能结束时间，喜爱机器的程度等参数，但是计算的方式不同，在异构情况下，因为每台机器的计算时间不同，因此异构 TDS 算法定义了在每个任务在每台处理器上的最早可能开始时间 $est(i,p)$,最早可能结束时间($ect(i,p)$),另外还按照这个 $ect(i,p)$ 对机器做了一个排序，定义为对机器的喜爱程度*(fproc(i,1)，fproc(i,2)…)*，另外还定义了任务在每台机器的最晚允许结束时间($lact(i,p)$)，最晚允许开始时间($last(i,p)$)，以及 *level* 值，这些和同构 TDS 算法的参数定义类似，只是将 *T(i)*替换成了 $T(i,p)$ 。其他相关参数具体定义如下：

$$est(i,p)=\max_{k\in PRED(i)}\{^{ect(k,fproc(k,1)),\ fproc(k,1)=p}_{ect(k,fproc(k,1))+C(k,j),\ fproc(k,1)\neq p} \quad \text{(式 3.17)}$$

$$ect(i,p)=est(i,p)+T(i,p) \quad \text{(式 3.18)}$$

$$ect(i,\ fproc(i,1))\le ect(i,\ fproc(i,2))....\le ect(i,\ fproc(i,m)) \quad \text{(式 3.19)}$$

$$fpred(i)=\{j\mid ect(j,\ fproc(j,1))+C(j,i)\ge ect(k,\ fproc(k,1))+C(k,i)\} \quad \text{(式 3.20)}$$

其中计算 fpred(i)的时候不管当前任务 *i* 和前驱任务最喜爱的机器是否相同，始终都计算了传输延迟。

下面是异构 TDS 算法生成初始聚簇的算法流程，具体如下：

算法 3.2：异构 TDS 生成初始聚簇

Input：DAG 图，任务在机器集上的执行时间表

Output：任务在机器集上的调度结果

1 将 level 的值进行升序排序，然后加入到处理队列中来

2 **While** 队列不为空

3 取出当前任务 i，按照 fproc(i,1),fproc(i,2),…,fproc(i,m)的顺序找第一台未被处理的机

器，放在这台机器上处理

4 **While** *i* 不是 入口节点

5 j=fpred(i)

6 **If** j 不是唯一前驱

7 **if** last(i)-lact(j)>=C(j,i) 或者 j 已经被分配过

8 在 i 的前驱中找到一个 k，其中 T(k,p)是最小的。

9 j=k

10 **Endif** line 7

11 **Endif** line 6

12 将 j 加入到这台机器上

13 i=j

14 **Endwhile** line 4

15 **Endwhile** line 2

另外，与同构 TDS 算法不同的是，异构 TDS 算法还考虑了任务复制的策略，考虑当前调度方案中的处理机 p 上的任务序列 $x_1, x_2, x_3, ...x_k$，可以知道 $x_1, x_2, x_3, ...x_k$ 是符合 DAG 图中的拓扑序列的。当前的任务 $x_i (1 < i \le k)$，如果存在 $fpred(x_i) \ne x_{i-1}$，则算法就选择在 x_i 和 x_{i+1} 之间进行任务复制。复制的方案是首先将任务 x_1 到任务 x_{i+1} 会被移到另外一个按照 *fproc(i,1),fproc(i,2),...,fproc(i,m)*的顺序第一台未分配任务的机器上，然后将 $fpred(x_i), fpred(fpred(x_i))$ 等等任务拷贝到机器 p 上，重复此过程，直至到达入口节点。一旦拷贝任务后，makespan 变大，则放弃此次复制，然后继续找下一个合适的位置进行复制。

3.2.2 TANH 算法

TANH 算法[23]（Task Duplication-Based Scheduling Algorithm for Network of

Heterogeneous Systems）改进了异构 TDS 算法的相关参数定义，如下为参数的数学定义，其他参数的定义和 TDS 算法中的式 3.4, 式 3.5, 式 3.6 类似：

$$fproc(j,1)=\{w \mid ect(k,w)+T(j,w)<ect(k,q)+c(k,j)+T(j,q) \underset{k\in PRED(j),w=fproc(k,1),q\neq fproc(k,1)}{}\} \quad (式3.21)$$

$$est(j)=est(j,w)=\min_{k\in PRED(j)}\{\max_{\substack{l\in PRED(j)\\ l\neq k}}\{^{est(k)+T(k,w)}_{ect(l)+C(l,j)}\}\} \quad (式3.22)$$

$$ect(j)=est(j)+T(j,fproc(j,1)) \quad (式3.23)$$

$$fpred(j)=\{k \mid est(k,fproc(k,1))+T(k,fproc(j,1))\underset{k\in PRED(j),l\in PRED(j)}{}\geq ect(l)+C(l,j)\} \quad (式3.24)$$

式 3.21 定义当前任务 j 对机器的喜爱程度的时候，对于不同的前驱可能会计算出对机器的喜爱程度不同，遗憾的是，TANH 并没有说明在遇到这种情况下作何处理。

式 3.24 计算当前任务 j 的关键前驱的时候，用了它的前驱任务 k 在最喜爱的机器上的最早开始时间和任务 j 在最喜爱的机器上的执行时间来定义数据到达最晚的时间，但是我们可以发现任务 j 和任务 k 最喜爱的机器有可能不同，因此传输延迟不能忽略，因此这个参数的定义并一定能准确合理的估计最关键前驱任务。

TANH 生成初始调度方案和异构 TDS 完全相同，但在算法 3.2 中的第 7 行，条件不等式：

$$last(i)-lact(j)\geq C(j,i),\ j\in PRED(i) \quad (式3.25)$$

式 3.25 中条件不等式表明的意思是说对于当前任务 i，如果关键前驱 j 和 i 不放在同样的机器上的时候也不会使得任务 i 的开始时间延迟，则我们可以选取另外一个不满足这个条件不等式的前驱使其与当前任务 i 放在同样的机器上，这个判断条件其实是对 $fpred(i)$ 的拓展，因为在计算 $fpred(i)$ 的时候我们是按照在 i 最喜爱的机器上时数据到 i 最晚的那个前驱，一旦最喜爱的机器已经被占用，则关键前驱就可能不一样了。但是这个不等式不一定能够准确的选出合适的前驱任务 j，因为 $lact(i)$ 和 $last(i)$ 不一定能准确的估算出任务的开始时间和任务的结束时间，在实际的测试用例中，$last(i)$ 和 $lact(i)$ 完全可能小于对于的 $est(i)$ 和 $ect(i)$，甚至有可能是负数。

另外，TANH 算法考虑在机器数目不够用情况下的解决方案，它加入了虚拟机器的概念，并且在虚拟机器的计算时间为 m 个真实的处理机器上的计算时间的平均值。最后在归并阶段将虚拟机器归并到真实处理机器当中去，从而得到合法的调度方案。

同时，TANH 算法是也是基于任务复制方案的，任务复制的情况和任务归并的情况属于不同的分支，而分支的判断条件就是处理器数目是否足够，TANH 算法在任务复制阶段采取的策略跟异构 TDS 算法完全一致，不足的是，对于任务复制的顺序并没有明确指定，而任务复制的顺序对调度结果影响很大。

3.2.3 DCPD 算法

DCPD 算法[32]（Dynamic Critical Path Duplication）是基于列表优先级和任务复制的，算法首先定义任务的优先级，然后按照优先级的序列开始调度。算法首先定义了下面几个参数，任务 i 在机器上的平均处理时间 $avge(i), b\_level(i), priority(i)$，具体如下：

$$avge(i) = \sum_{1}^{m} T(i,m)/m \qquad \text{(式 3.26)}$$

$$b\_level(i) = \max_{j \in SUCC(i)} (b\_level(j) + avge(j) + C(i,j)) \qquad \text{(式 3.27)}$$

$$ect(i) = \min_{p \in m} \{drt(i,p) + T(i,p)\} \qquad \text{(式 3.28)}$$

$$u\_level(i) = \max_{k \in PRED(i)} \{\text{ect}(k) + C(k,i)\}. \qquad \text{(式 3.29)}$$

$$priority(i) = b\_level(i) - u\_level(i) + avge(i) \qquad \text{(式 3.30)}$$

式 3.28 中的 $drt(i,p)$ 表明的任务 i 可以开始时所有前驱数据需要准备的时间.

式 3.27 定义了 $b\_level(i)$ 值，意思是表明在考虑传输时间的情况下当前任务 i 到出口任务的最长路径长度。

式 3.29 定义了 $u\_level(i)$ 值，意思是表明在考虑传输时间的情况下从入口任务到当前任务 i 的最长路径长度。

式 3.30 定义了任务的优先级 $priority(i)$,意思是指在当前未被处理的任务集当中，执

行时间长，距离出口路径越长，距离入口距离越短的任务最有可能是关键路径上的任务，有理由相信如果提前这种关键路径上的任务就可以提前整个任务集合的调度长度。

DCPD 算法首先对任务进行了如下划分，任务包括已被处理和未被处理，其中在未被处理的任务集合中又包括已经准备好可以马上调度的任务(*ready_set*)，部分准备好的任务(*partially ready set*), 没准备好的任务（*unready_set*）。在 *ready_set* 的任务集合中，表明的是任务的所有前驱数据已经全部到达，可以马上开始执行。

DCPD 算法的流程具体如下：

算法 3.3：DCPD 算法

Input：DAG 图，任务在机器集上的执行时间

Output：任务在机器集上的调度结果

1 入口节点进队列
2 **While** 当前队列不为空
3 dt=当前队列中 priority 最大的那个任务
4 寻找处理机器 pdt，使得任务 dt 在 pdt 上有最早的完成时间
5 寻找关键前驱 dp
6 **If** 复制当前 dp 到 pdt 上提前了 dt 在 pdt 上的结束时间
7 复制 dp 到 pdt 上
8 **Endif** line 6
9 将 dt 移除队列，加入新的 ready_set 且求出 ready_set 的 u_level 值，然后压入队列中
10 **Endwhile** line 2

DCPD 算法的主要思想就是先处理关键路径上的任务，并且复制一些关键任务是有利于 makespan 减少的。算法的第六行也包括了寻找关键前驱，因为 DCPD 算法的优先级计算是根据当前的调度动态计算的，这样就比 TANH 或者和 TDS 算法系列参数在调度前就计算好来的更准确合理，因此在寻找关键前驱 *dp* 时，找到的 *dp* 更加准确

合理。而实验也表明了 DCPD 算法比 TANH 算法效果更好。

DCPD 算法根据当前调度动态计算相关参数比 TANH 等静态计算相关参数要好，但是算法的缺陷任务优先级的先后关系对算法的效果起着至关重要的作用，一旦某一个任何优先级不合理就会对任务的调度起到连锁反应，进而影响算法的结果。

3.2.4 HEFT 算法

HEFT 算法[6](Heterogenous Earliest-Finish-Time)也是基于列表的任务调度算法，算法也是首先定义了任务的优先级 *b_level(i)*，和 DCPD 算法定义不同的是，式 3.27 其中的 *avge(j)*被替换成了 *avge(i)*。

算法首先按照优先级从小到大排序，然后根据此列表进行调度，另外需要注意的是 HEFT 算法是基于任务插入的思想，对当前分配任务 *i*，是从处理机集合 *m* 中枚举，使得分配到那台机器上任务有最早的开始时间。HEFT 算法相对于 DCPD 算法，一是任务优先级有微小差异，二是在选取机器的原则不同，HEFT 是选取导致有最小开始时间的机器，而 DCPD 算法是计算导致有最小结束时间的机器。三是 DCPD 算法考虑了任务复制策略而 HEFT 算法并没有考虑复制策略。

3.2.5 HNPD 算法

HNPD 算法[36](Heterogenrous N-Predessoror Duplication)也是基于列表和任务复制的调度策略。HNPD 算法定义任务优先级的方式结合了 DCPD 算法和 HEFT 算法 *b_level* 的定义，HNPD 算法将两种 *b_level* 的值的和作为 HNPD 定义任务优先级顺序的方式。这里 HNPD 和 HEFT 算法只是将 *b_level* 的值作为任务处理的优先顺序，而 DCPD 算法是动态计算任务优先级的顺序的，而且定义任务优先集的时候是优先考虑关键路径上的任务，DCPD 算法同时考虑了节点权和边权。

HNPD 算法和 DCPD 算法流程基本类似，主要有如下几点不同，首先加入任务到队列的方法不同，假设当前处理的是未被分配的任务 i，则 HNPD 算法是将任务 *i* 的前驱当中未被分配的也加入到队列当中，而 DCPD 算法是按照拓扑序列的顺序，判读当

前 *i* 的后继是否是 *ready_set*，也就是说将 *i* 的所有后继已经准备好的加入到队列中。

其次，HNPD 算法是按照任务 *i* 在机器上的最早开始时间来选取机器，而 DCPD 算法是按照任务 *i* 在机器上的最早结束时间在来选取机器。

最后，HNPD 算法和 DCPD 算法选取任务复制的方式也有所不同，HNPD 算法是尝试复制当前任务 *i* 的各个前驱到当前任务 *i* 所分配的机器上，看能否使任务 *i* 提前，而 DCPD 算法是找寻关键前驱，只对关键前驱判定是否复制到当前任务 *i* 所分配的机器上。

3.2.6 LDCP 算法

LDCP 算法(Longest Dynamic Critical Path)和 DCPD 算法更加相似，也考虑了关键路径上的任务，也定义了 *b_level*，但是定义的方式不同，LDCP 定义了在处理器上的 *b_level(i,p)*，如下：

$$b\_level(i,p) = T(i,p) + \max_{k \in SUCC(i)} (C(i,k) + b\_level(k,p)) \qquad \text{(式 3.31)}$$

LDCP 算法处理的流程和 DCPD 以及 HNPD 算法类似，LDCP 每次从队列中选取最大的 *b_level(i,p)*作为当前处理的任务，然后选择合适的处理机，选取的策略和 HNPD 算法相同，也是使得任务 i 有最早的开始时间。然后加入任务到队列的方式和 HNPD 算法相同，不过在加入前驱任务 j 到队列的同时要更新 *b_level(j,p)*，将 *T(i,p)*改为 *T(j,p)*，同时如果前驱 *j* 的 *b_level* 也在 *p* 上有最大值，则将边权 *C(j,i)*改为 0.

3.3 本章小结

这一章主要介绍了同构情况下和异构情况下的多任务调度算法，其中包括同构 TDS 算法，MJD 算法，DCP 算法。异构情况下主要介绍了异构 TDS 算法，TANH 算法，DCPD 算法，HEFT 算法，HNPD 算法，LDCP 算法。

同构 TDS，MJD，DCP 算法都是采用了基于聚簇和任务复制的策略，MJD 和 DCP 又是一脉相承的，算法都定义了粒度的概念，并给出了算法的性能下届并给出了分析和证明。

异构 TDS 算法，TANH 算法也是结合了基于聚簇和任务复制的策略，而 DCPD 和 HNPD，LDCP 算法又是一脉相承的，都是采用列表优先级和基于任务复制的方式，而 HEFT 算法仅是采用了简单的基于优先级列表的调度策略。

TDS 算法和 TANH 算法基本流程类似，但是相关参数定义不同，如关键前驱，最早开始时间，最喜爱的机器等的定义不同，因此在生成初始聚簇的时候就会有很大的差异。总的来说，TANH 计算参数的方式更加合理。在对关键前驱的定义上，异构 TDS 算法是在都考虑传输时间的情况下，数据到达的最晚的那个前驱。而 TANH 算法计算的是在不考虑传输时间的情况下，分别假设前驱和当前任务都在自己所最喜爱的机器上处理时数据到达最晚的那个前驱，这两种定义的方式都有可取之处。

DCPD 算法和 LDCP 算法以及 HEFT，HNPD 算法在定义优先级的方式上都有不同，其中 HEFT 和 HNPD 算法计算优先级采用了简单的 *b_level* 的值，其中 HNPD 算法继承了 HEFT 的 *b_level*，DCPD 算法和 LDCP 算法在计算优先级的时候是动态计算的，首先也是将动态关键路径上的任务先处理，然后又更新任务的优先级，优先级高的则就是所谓的关键路径上的任务。从这几个算法可以发现，基于列表优先级和任务复制策略算法的主要不同在于定义优先级的方式和如何选取任务复制的情形，研究表明，考虑动态关键路径的方式定义优先级比静态先算好优先级在生成调度的方式更加优秀。另外，选择任务复制的情形总是好于不结合任务复制的方式。

4 基于任务复制和插入的聚簇和归并算法

本章根据第二章给出的问题的形式化描述，提出了基于任务复制和插入的聚簇和归并算法（Task Duplicate Insertion-Based Clustering and Merging Algorithm, TDICMA）。TDICMA 主要在基于 TDCMA(Task Duplication-Based Clustering and Merging Algorithm)的基础上做了相关优化，包括在调度最后判断任务是否可插入，以及调度过程中处理器数目不够的情况下该如何应对，算法主要流程包括如下几个步骤，参数计算，生成初始聚簇，任务复制，任务合并，任务复制插入。

本章先给出相关参数定义，然后具体描述各个步骤的算法思想和流程。第一个阶段，TDICMA 首先根据计算的参数构造初始调度，接下来采用任务复制的策略调整修正调度结果，第三个阶段，TDICMA 尝试归并处理机上的调度，第四个阶段，对任务可否复制插入进行判断，以求可以得到更优的调度结果。

4.1 参数计算

定义 4.1 传输代价(Communication cost). 对任意任务 $j \in PRED(i)$，如果任务 j 和任务 i 分别在机器 q 和 p 上被处理，则 j 和 i 之间的所需要的传输代价为

$$\underset{(j,i)\in E}{\delta(j,i,q,p)} = \begin{cases} 0, & q=p \\ C(j,i), & q\neq p \end{cases} \quad \text{(式 4.1)}$$

定义 4.2 最早可能开始时间（Earlisest Starting Time）. 对于 $i \in V$ 且 $p \in P$，$est(i,p)$ 表明了 i 分配在机器 p 上时最早的可能开始时间

$$est(i,p) = \begin{cases} 0, & i=entry-node \\ \max\limits_{j\in PRED(i)}\{\min\limits_{q\in P}\{ect(j,q)+\delta(j,i,q,p)\}\}, & i\neq entry-node \end{cases} \quad \text{(式 4.2)}$$

式 4.2 其实表明的是任务 i 如果放在 p 上最早可以开始所需要的数据准备时间，它首先应该满足它的所有前驱都已经执行完成，这也就是式 4.2 外面 *max* 的含义，另外对于每一个前驱来说，可以在机器集合内的任意一台机器上执行，则里面的 *min* 的含义就是找到最合理的那台机器，使得那个前驱数据最早达到 i。

下面我们来看一下计算所有任务的最早可能开始时间 *est* 的时间复杂度是多少，计

算 *est(i,p)*的时间复杂度是 *O(EP)*，所以计算 *est* 的时间复杂度是 $O(EP^2)=O(em^2)$，但我们稍微分析一下可以发现，我们在计算 $est(i,p)$ 的时候并不需要都枚举机器集合一遍，因为对于任务 *i* 的前驱任务 *j* 来说，最早可能结束的时间是发生在最喜爱的机器上，因此我们只需要比较 $ect(j,p)$ 和 $ect(j,fproc(j,1))+C(j,i)$ 的大小就可以得到当前前驱任务 *j* 数据可能最早到达任务 *i* 的时间。因此我们重新定义如下计算方式：

$$est(i,p)=\begin{cases}0, & i=entry-node\\ \max\limits_{j\in PRED(i)}\{\min\{ect(j,fproc(j,1))+C(j,i),ect(j,p)\}\}, & i\neq entry-node\end{cases} \quad \text{(式 4.3)}$$

定义 4.3 最早可能完成时间（Earlisest Completion Time）. 对于 $i\in V$ 且 $p\in P$，*ect(i,p)* 表明了 *i* 分配在机器 *p* 上时最早的可能完成时间

$$ect(i,p)=est(i,p)+T(i,p) \quad \text{(式 4.4)}$$

定义 4.4 第 r 喜欢的机器. 对于 $i\in V$ 且 $r\in[1,m]$，*fproc(i,r)*表明了任务 *i* 第 *r* 程度喜欢的机器，它满足如下条件：

$$ect(i,fproc(i,1))\le\ldots\le ect(i,fproc(i,r)) \quad \text{(式 4.5)}$$

这里计算任务 *i* 第 *r* 喜欢的机器时，只需要先计算出任务 *i* 在各个机器上最早可能完成时间(*ect(i,p)*)，然后按照非降序的顺序排序则可以得到对机器喜爱程度的排名。

定义 4.5 关键前驱(Critical Predecessor). 对于 $i\in V$,*cpred(i)*表明的是所有前驱任务中数据到达 *i* 最晚的那个前驱，这里计算关键前驱的方式是假定当前任务 *i* 和前驱任务都在各自最喜爱的机器上执行，具体公式计算方式如下：

$$\begin{aligned}&cpred(i)=\\&\{\ j\ \mid ect(j,fproc(j,1))+\delta(j,i,fproc(j,1),fproc(i,1))\\&\qquad\ge ect(k,fproc(k,1))+\delta(k,i,fproc(k,1),fproc(i,1))\}\end{aligned} \quad \text{(式 4.6)}$$

*cpred(i)*的定义是基于 TANH[23]中对 *fpred(i)*的定义的，在上一章的式 3.24 中可以发现，TANH 也是假定当前任务 *i* 和前驱任务都在最喜爱的机器上执行时数据到达任务 *i* 最晚的那个任务为关键前驱， 但 TANH 中计算前驱任务 *j* 数据到达任务 *i* 的时间时有这样的计算方式：

$$est(j, fproc(j,1)) + \max_{j \in PRED(i)} T(j, fproc(i,1))$$

它是用任务 *j* 在最喜爱的机器的开始时间加上它在任务 *i* 最喜爱的机器的执行时间来定义前驱数据 *j* 到达任务 *i* 的时间，很明显这样定义不是很准确，因为前驱任务 *j* 和当前任务 *i* 最喜爱的机器可能不同，那么必要的传输代价就不能忽略，而在我们的式 4.6 中，我们就考虑到了这一点。

定义 4.6 任务优先级. TDICMA 使用了 B-Level[17]的计算方式来定义任务的优先级，对 $i \in V$，定义 *level(i)*是从当前任务 *i* 到出口任务的最长路径（考虑了传输代价）主要的计算方式如下：

$$level(i) = \max_{k \in SUCC(i)} \{level(k) + C(i,k)\} + \max_{q \in P}(T(i,q)) \qquad \text{(式 4.7)}$$

可以看到，这里定义 *level(i)*的方式与 DCPD，以及 HEFT 定义 *level* 的方式有点类似，只是将 HEFT 中的 *avge(i)*替换成了 $\max_{q \in P}(T(i,q))$。

参数 *est*，*ect*，*fproc*，*cpred* 都是按照 DAG 图的拓扑序列顺序进行计算，而 *level* 的值是按照逆拓扑的顺序计算。TDICMA 算法的参数定义跟 TANH 有点类似，其中 *ect* 与 *fproc* 的定义和 TANH 的定义一样，但是 TDICMA 定义了不同的 *est*，因此也会产生不同的 *ect* 和 *fproc* 值，另外 TDICMA 不在使用 *last* 和 *lact* 这个参数，关于 *level* 的定义和 TANH 相同。

定义 4.7 关键前驱链(Critical Predecessor Trail). 关键前驱链的含义有点类似关键路径，我们对定义任务 *i*，以及 *cpred(i),cpred(cpred(i)*，…直到入口节点这样的链为关键前驱链。

4.2 生成初始聚簇

TDICMA 算法在根据上面参数定义完成相关参数计算之后，接下来开始生成初始聚簇。首先按照式 4.7 中的 *level* 值将任务非降序排列，接下来定义如下不等式：

$$ect(k, p) \le ect(k, fproc(k,1)) + C(k,i) \qquad \text{(式 4.8)}$$

然后按如下算法流程开始调度：

算法 4.1：TDICMA 初始聚簇

Input：DAG 图，任务在机器集上的执行时间

Output：任务在机器集上的初始聚簇调度结果

1 将任务按照排过序之后的 level 压入任务队列中
2 **For** 任务队列的中每个任务 *i*
3 **If** 任务 *i* 已经被处理过，回到 line3 继续循环
4 *CurProc*=按照 *fproc*(*i*,1...*m*) 的顺序选出第一个未分配任务的处理器
5 把任务 *i* 分配在 *CurProc* 这个处理器上，并设置任务 *i* 的状态已经被处理过
6 **While** 任务 *i* 不是入口节点
7 *j=cpred(i)*
8 **if** {*i* 的前驱不止一个} 且 {任务 *j* 已经被处理过或者不满足式 4.8}
9 则重找满足式 4.8 的 *k*，若有多个，找 *ect(k,Curproc)*最小的那个
10 *j=k*
11 If *k* 不存在 退出 line 6 的 while 循环
12 **Endif** line 8
13 将 *j* 分配在机器 *CurProc* 上，并设置任务 *j* 的状态已经被处理过
14 *i=j*
15 **Endwhile** line 6
16 设置处理器 curProc 已经被分配
17 **Endfor** line 2

TDICMA 在生成初始调度流程和 TANH 基本类似，主要有以下几点不同：

1 首先,在 TANH 中，如果当前处理的任务 *i* 有多个前驱，且任务 *i* 的关键前驱任务 *j=fpred(i)*未被分配过，则 TANH 就马上把当前这个关键前驱任务 *j* 分配到当前的处理机器上，但是在 TDICMA 当中，我们还需要判定是否满足式 4.8（式中的 *k* 替换成 *j*），

如果不满足，则有理由相信将关键前驱任务 *j* 在当前机器上并不构成瓶颈，我们可以将任务 *j* 放在最喜爱的机器上，这样数据比分配在当前机器 *p* 上，数据到来的时间来还要更早。

2 其次，在 TANH 中，如果不满足式 3.25：$last(i)-lact(j)\geq C(j,i), j\in PRED(i)$，则就重新寻找另外一个前驱，而此时重新寻找另外一个前驱的方法仅仅是找一个使得 T(k,CurProc)最小的那个前驱，首先在 3.2.2 节讲到，在 TANH 中 last(i)和 lact(i)并不能准确的描述出任务 i 的开始时间和结束时间，因此在 TDICMA 当中用的是不等式 4.8 $ect(j,p)\leq ect(j,fproc(j,1))+C(j,i)$，另外如果不满足这个不等式的情况下，TDICMA 重新寻找前驱的方法也和 TANH 不同，它寻找满足这个条件的前驱任务 k，如果多个满足，则使得 ect(k,Curproc)最小。

3 接下来，TANH 当前处理器上构造任务调度方案时，结束的条件是直到入口节点才退出本轮的循环，而通过研究发现，有些任务的数据其实可以是通过其他处理器上的传输数据直接传过来的，因此 TDICMA 采用的终止每次迭代的条件是如如果关键任务 j=cpred(i)不被选取，且找不到合适的前驱任务 k 满足 Eq.39，则就终止此次循环。

4 TDICMA 在处理器数目不够的情况下，也就是算法流程中找不到未分配的处理器的情况下，算法采取的策略是首先计算出当前处理器的负载情况，负载小的机器优先考虑，以及当前需要分配的任务对机器的喜爱程度，喜爱程度高的优先考虑。TDICMA 给与这两种情况不同的权重，最后综合选取合适的机器。

4.3 任务复制

TDICMA 在完成初始聚簇调度后，开始任务复制阶段，前面说到在任务复制过程中存在链式反应，本轮无效的复制在下一轮可能就变得有效。因此我们选取了迭代 4 次，至于为什么将迭代次数选为 4 将会在第五章实验部分说明。

TDICMA 复制的策略跟 TANH 相似，它的主要思想主要是因为每个任务的关键前驱任务是当前任务最早可能开始时间的瓶颈，因此我们考虑是否可以将当前任务的关键前驱链(定义 4.7）复制到当前任务 *i* 前面，算法首先枚举每个已经分配的处理器，

找到可以选择复制的位置，即当前处理器上任务 i 的前面一个任务 j 满足 $j \neq cpred(i)$，则这样的位置就是我们要找的可复制的候选位置。假定 $X_p = (x_1, x_2, ..., x_k)$ 是指机器 p 上的任务序列，由上面可以知道，该序列是符合拓扑序的。TDICMA 按照如下方式找候选位置：

(1) 首先在这些任务序列中找到是存在 x_i 满足 $cpred(x_i) \neq x_{i-1}$，如果是这样，则定位到这个任务 x_i，任务复制可以在 x_i 和 x_{i-1} 中进行。

(2) 然后我们检查 x_1 是否是入口节点，如果不是，则在 x_1 前面可以加入关键前驱链。TDICMA 在找到上面(1)所说的这样位置的时候，首先将 x_1 到 x_{i-1} 的任务移到另外一个处理器 q 上，处理器的选择是 x_{i-1} 按照机器的喜爱程度的排名，第一台未被分配的处理器 q，然后任务 x_i 的关键前驱链将继续加到当前处理器 *CurProc* 中，接着判断调度长度是否减少，如果减少了，则采用这次复制方案。

接下来，TDICMA 按照上述 2 的方式继续复制任务，这个过程不需要移动任何任务到其他处理器上，只需要判定调度长度（Makespan）是否减少了，如果减少了，则采用这次复制方案。

TDICMA 具体的复制流程如下所示：

算法 4.2：TDICMA 任务复制

Input：任务在机器集上的初始聚簇调度结果

Output：任务在机器集复制过后的调度结果

1 **For** 迭代次数从 1 到 4

2 **For** 处理器 p 从 1 到 m

3 **For** 下标 i 从 $|x_p|$ 到 2

4 **If** $X_p(i-1) \neq cpred(X_p(i))$

5 nextProc=从 $fproc[X_p(i-1),1...m]$ 中找第一个未被分配处理器

6 将当前调度 schedule 拷贝到 newSchedule 上

7 将 $X_p(1,2,...i-1)$ 移动到 nextProc 上

8 在 CurProc 上将 $X_p(i)$ 的关键前驱链复制过来

9 **If** newSchedule 的 makespan 变小了

10 schedule=newSchedule

11 **Endif** line 9

12 **Endif** line 4

13 **Endfor** line 3

14 **If** $X_p(1)$ 不是入口节点

15 拷贝当前调度到 newSchedule 上，并将 $X_p(1)$ 按照 line9 到 line11 循环一遍

16 **Endif** line 14

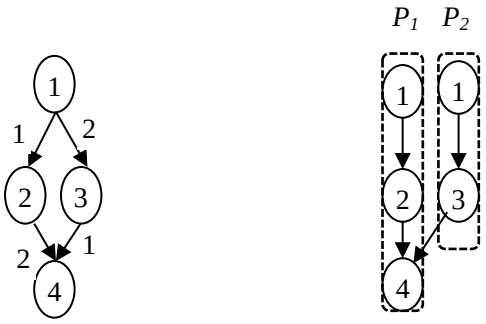
17 **Endfor** line 2

18 **Endfor** line 1

4.4 任务归并

TDICMA 在任务复制阶段之后，开始进行任务归并阶段，看调度长度(Makespan)是否能够继续减少，首先如下图 Fig.1(a)所示的 DAG，Table 1 描述了任务在机器集上的处理时间。

从中可以看到，很明显是将任务 1,2,3,4 都分配在机器 P_1 上处理能够得到最优调度长度，TANH 在机器数目足够的情况下是不会进行任务归并阶段的，所以得到的调度不可能是全部任务都在机器 P_1 上处理，如图 Fig.1(b)所示是 TANH 得到的调度方案。



(a) (b)

Fig. 1 DAG 和不采取归并时的调度

TABLE 1

Fig. 1 上任务在机器上的计算时间

Processor / Node	P_1	P_2
1	1	1000
2	1	1000
3	1	1000
4	1	1000

在采取任务归并策略中，我们通过从 1 到 m 的处理器集合中枚举每一个处理器 p，考虑当前处理器 p 上的最后一个任务 x，假设 p 是任务 x 第 r 程度喜爱的机器，那么将当前处理器 p 上的所有任务和任务 x 第 r-1 程度喜爱，第 r-2 程度喜爱,….最喜爱的机器上的所有任务进行归并处理，另外一个比较简单的方案就是仅仅将当前处理器 p 上的所有任务和任务 x 最喜爱的机器上的所有任务合并。从中可以知道，前一个方案的结果总是好于后一个方案，但是前一个方案算法的时间复杂度很大。因此 TDICMA 用了后一种策略，将当前处理器 p 和任务 x 最喜爱的机器合并，判定是否能够减少调度长度。下面是任务归并的具体流程，流程中迭代了 4 次是因为复制过程迭代了 4 次，因此这里归并过程相应的也迭代 4 次。

算法 4.3：TDICMA 任务归并

Input：任务复制阶段过后的调度结果

Output：任务归并阶段之后的调度结果

```
1   For 迭代次数从 1 到 4
2       For 处理器 p 从 1 到 m
3           拷贝当前调度 schedule 到 newSchedule
4           X=处理器 p 上的最后一个任务
5           在 newSchedule 上归并处理器 p 和 fproc(x,1)
6           If 调度长度减少
```

```
7                schedule=newSchedule
8        Endfor line 2
9    Endfor line 1
```

4.5 任务插入

在 TDICMA 的最后一个阶段，继续考虑基于任务插入的策略，看是否能够继续减少调度长度，正如 Fig.2 经典的 EZ 示例所示，这是个同构示例，圆圈内数字表示任务在机器上的计算时间，现在有任务$T6$ 分配在机器$P2$上，任务$T7$分配在机器$P1$上，如果不采取任务插入的策略，那么得到的调度长度是 8.5, TDICMA 检查到可以将任务$T6$插入到$P1$机器上任务$T2$与$T7$之间，继而继续减少了调度长度得到最优解8.0.

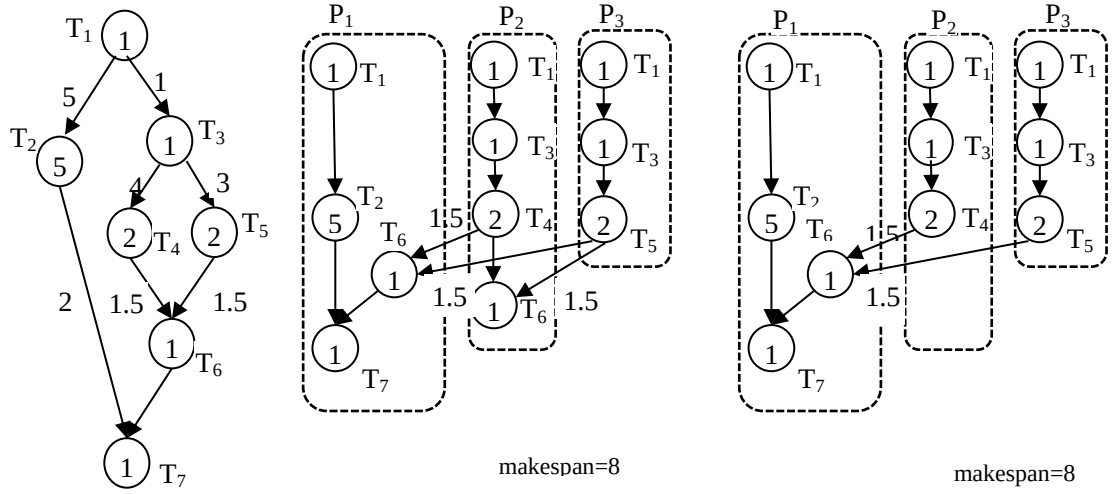


Fig. 2 DAG 和 TDICMA 调度结果

在之前的研究过程中，这个 EZ 算例的最优解一直被认为是 8.5[4]，何琨等人在论文[34]中给出了 EZ 示例的最优解 8.0 的证明。

在任务插入阶段，我们事先已经知道当前调度，因此 TDICMA 首先找到所有跨处理器的边，然后判定每个跨边上的源节点任务插到目的节点任务所在的机器上去是否可以提前目的节点任务的最早完成时间，如果提前了，则选择复制插入源节点任务

到该机器上，如图 Fig.2(b)所示，最后判定当前源节点任务是否是它所在处理器上的最后一个任务且在当前调度中其他任务不以这个任务为数据依赖，如果满足这样的条件，则我们可以在源节点任务的机器上删除该任务，如图 Fig.2 (c)所示，流程如下：

算法 4.4：TDICMA 任务插入

Input：任务归并阶段过后的调度结果

Output：任务插入阶段之后的调度结果

1 **For** 每条跨处理器的边 e

2 e_{fro}->t_i， e_{to}->t_j， p_i为任务t_i所在的机器， p_j为任务t_j所在的机器

3 **If** 插入t_i到p_j上的t_j前面位置使得t_j的最早完成时间提前

4 则将t_i插入到p_j上的t_j前面位置

5 **If** t_i是p_i的尾任务，且对其他节点任务不构成数据依赖

6 删除t_i

7 **Endif** line 3

8 **Endfor** line 1

4.6 算法用例演变

这个部分给出一个DAG的范例来解释TDICMA具体的流程，DAG如图Fig.3所示，Table.2 表示的是任务在机器集上的计算时间（*computation cost*）.算法首先计算相关参数如下表 Table 3 所示：

举例来说，在计算 est(10,1)的时候按照式 4.3 有如下计算流程。

est(10,1) =max{min{ect(8,1),ect(8,fproc(8,1))+C(8,10)},min{ect(9,1),ect(9,fproc(9,1)+C(9,10)}

=max{min{22,21+10},min{21,18+5}}

=max{22,21}=22

计算 *cpred(8)*的时候，因为任务 8 有两个前驱任务 5 和任务 6，任务 8 最喜爱的机器是第三台机器，任务 5 和任务 6 最喜爱的机器分别是第五台和第一台，则因为

ect(5,5)+C(5,8)=19<23=ect(6,1)+C(6,8)

所以任务 8 的关键前驱 cpred(8)=5.

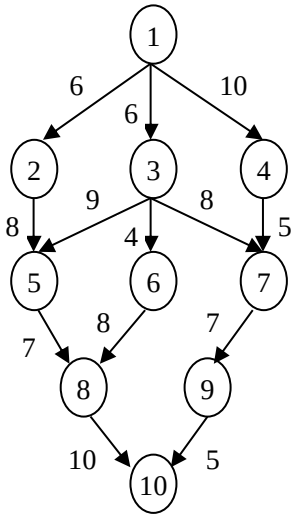


Fig.3 样例 DAG 模型

TABLE 2

Fig.3 DAG 节点在机器集上的计算时间

Processor / Node	P_1	P_2	P_3	P_4	P_5
1	3	6	5	5	4
2	6	3	5	7	3
3	6	5	7	7	6
4	5	6	4	4	4
5	7	2	4	2	2
6	6	7	4	6	5
7	6	4	5	4	6
8	6	6	5	5	6
9	4	5	3	2	3
10	3	1	7	1	2

TABLE 3

s* for *est*, cfor *ect*, *fp* for *fproc*, *cpr* for *cpred*, *lev* for *level

No	s_1	s_2	s_3	s_4	s_5	c_1	c_2	c_3	c_4	c_5	fp_1	fp_2	fp_3	fp_4	fp_5	*cpr*	*lev*
1	0	0	0	0	0	3	6	5	5	4	1	5	3	4	2	0	65
2	3	6	5	5	4	9	9	10	12	7	5	1	2	3	4	1	52
3	3	6	5	5	4	9	11	12	12	10	1	5	2	3	4	1	53
4	3	6	5	5	4	8	12	9	9	8	1	5	3	4	2	1	41
5	9	11	12	12	10	16	13	16	14	12	5	2	4	1	3	3	37
6	9	11	12	12	10	15	18	16	18	15	1	5	3	2	4	3	38
7	9	12	12	12	10	15	16	17	16	16	1	2	4	5	3	3	30
8	16	18	16	18	15	22	24	21	23	21	3	5	1	4	2	6	23
9	15	16	17	16	16	19	21	20	18	19	4	1	5	3	2	7	17
10	22	24	21	23	21	25	25	28	24	23	5	4	1	2	3	8	7

接下来根据计算出的参数生成初始聚簇，按照 *level* 的顺序根据任务生成初始聚簇的流程来生成初始调度，首先从任务 10 开始，然后选择最喜爱的机器是第 5 台机器，因为 *cpred(10)*=8,且任务 8 满足不等式 4.8，所以把任务 8 也加入到第 5 台机器中去，然后按照这种方式继续算法流程直至到入口节点任务 1.按照这种方式可以得到初

始调度情况，如图 Fig.4 所示。

在任务复制阶段，我们发现第二台机器上任务 5 的关键前驱不是任务 2，因此首先将任务 1,2 移动到 2 最喜爱的机器 p_1 上面去，然后将 5 的关键前驱链复制到机器 p_2 上，因为新得到的调度长度等于 34，小于之前的调度长度 36，因此采用了这次复制方案，得到如图 Fig.5 所示的调度结果。

接下来继续任务归并阶段，因为在 p_1 机器上任务 2 最喜爱的机器是第五台机器，因此尝试将 p_1 机器上的任务和 p_5 机器上的任务合并，但是由于调度长度变大，因此不采用这次合并，同样的方式发现可以合并 p_2 和 p_5 上的任务，因为这次得到的调度长度变小，所以采用这次合并。同理，按照这种方式继续寻找其他可以合并的可能，最终得到如图 Fig.6 所示的调度结果。

最后在任务复制插入阶段，按照复制插入的流程思想，在任务复制阶段之后，我们从调度结果中可以看到，只在机器 p_4 和机器 p_5 上任务之间存在空闲，但是由于插

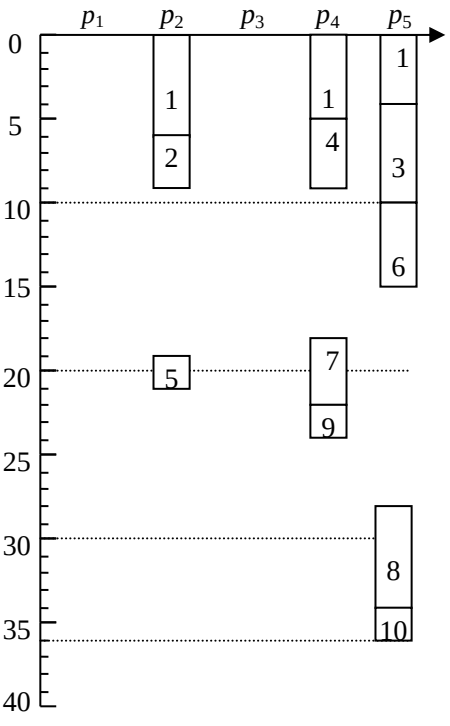


Fig.4 初始聚簇
（makespan=36）

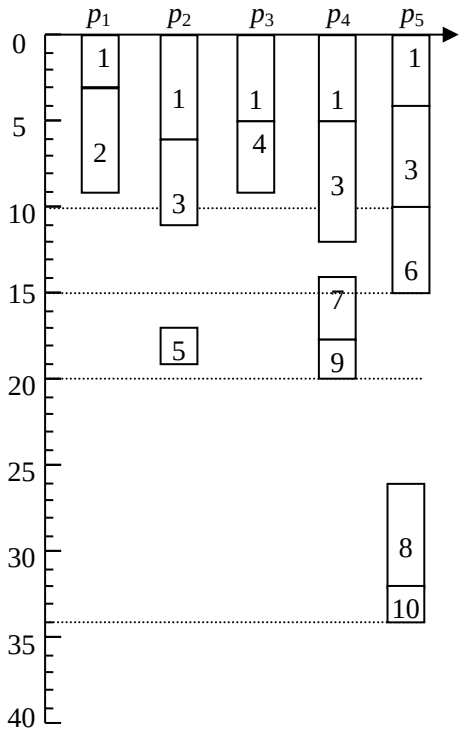


Fig.5 任务复制过后
（makespan=34）

入过后导致调度长度变长，因此不采用这次任务插入，如 p_2 机器上的任务 4 插入到 p_4 机器上的任务 7 的前面，发现调度长度变长。因此 TDICMA 得到的最终调度结果为 27，如图 Fig.6 所示。Fig.7 是 TANH 调度得到的最终结果为 31.

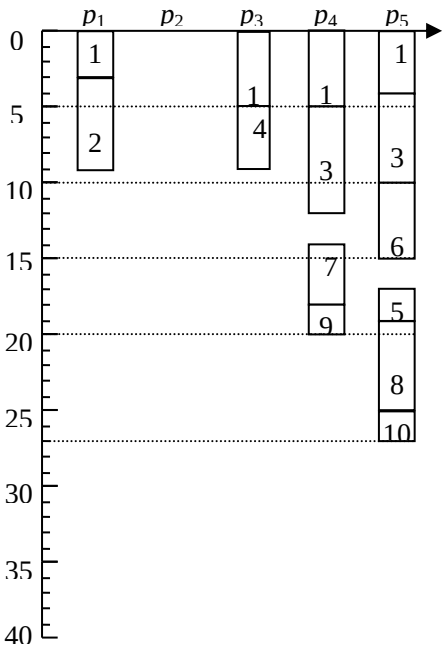


Fig.6 TDICMA 调度最终结果
（makespan=27）

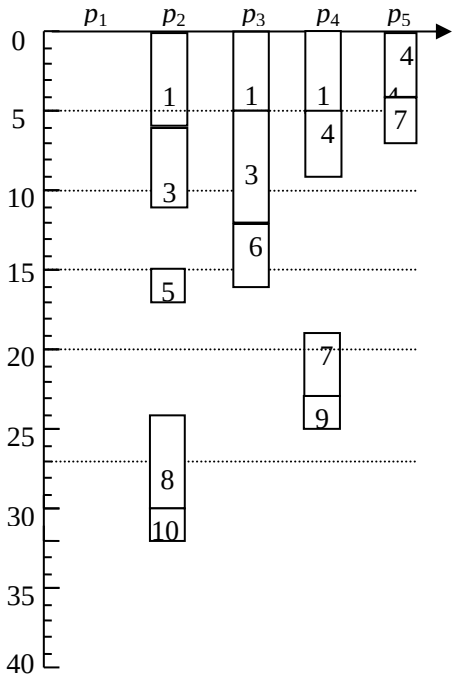


Fig.7 TANH 调度结果
（makespan=31）

4.7 算法时间度分析

TDICMA 算法计算 *est* 参数时间复杂度是 *O(me)*, 计算 *fproc* 的时间复杂度是 *O(mnlogm)*,计算其余 *ect*，*level*，*cpred* 参数需要 *O(e)* .所以 TDICMA 在参数计算的时间复杂度是 *O(me+mnlogm)*。

在初始生成聚簇阶段，首先需要根据 *level* 值从小到大排序任务，这部分的时间复杂度是 $O(n\log n)$，对于每个任务来说，都需要找一个未被分配的机器，因此需要 *O(mn)*，另外，在不符合不等式 4.8 的情况下，需要枚举当前任务 *i* 的所有前驱，需要 $O(e)$ 的时间复杂度，所以在初始生成聚簇阶段，总的时间复杂度是 *O(mn+nlogn+e)*

在任务复制阶段，至多有$O(n)$个候选复制位置，每轮复制要重新计算调度长度（*makespan*），而重新计算 *makespan* 的需要花费$O(me)$的时间，因此任务复制阶段总共的时间复杂度是$O(mne)$。

在归并阶段，枚举所有处理器需要$O(m)$，每次合并也需要重新计算 *makespan*，因此在任务归并阶段的时间复杂度是$O(m^2e)$.

在任务插入阶段，至多有$O(n)$个候选插入位置，每次通过判断任务之间是否有空闲时间来决定是否插入，判定需要$O(1)$的代价，所以任务复制插入阶段，总的时间复杂度是$O(n)$.

因此综合上述 TDICMA 的各个步骤，算法需要的总的时间复杂度是$O(mne+m^2e)$。

4.8 本章小结

本章具体介绍了基于任务复制插入的聚簇和归并算法（TDICMA）的各个阶段的思想，算法首先改进了 TANH 部分相关的参数定义，同时也优化了 TANH 在生成初始聚簇时的局限，在任务复制阶段，通过迭代 4 次来避免链式反应带来的连锁反应，在任务归并阶段，只将当前任务和它最喜爱的机器上的任务进行归并以此减少时间复杂度，任务复制插入阶段，通过判断当前调度上任务之间是否有空闲可供插入来更深层次的优化调度方案。并用一个实例重点阐释了算法的各个流程，最后本章还给出了 TDICMA 算法的时间复杂度分析。

5 实验与分析

这一部分我们将 TDICMA 算法与经典的基于列表的 HEFT[6],基于任务复制的 TANH[23],基于列表优先级和任务复制的 DCPD 算法进行了比较。首先，实验生成了一个 DAG 模型的随机生成器，总共产生了 50625 个测试用例，然后参照对比了不同参数下对算法性能的影响程度。

5.1 DAG 模型生成器

实验部分首先设计了一个 DAG 图的生成器来自动生成测试用例，DAG 图包括很多属性，如节点个数，图的拓扑结构，异构程度，边权等。

实验按照如下步骤生成 DAG 图上任务的计算时间，并设计衡量处理器异构性的参数 *h*：

Step1. 对于每一个任务 *i*，首先产生在所有处理器上的平均计算时间 *meanCompCost(i)*.

让 *meanCompCost* 表示所有任务在所有处理机器上的平均计算时间，且 *meanCompCost(i)*服从如下均匀分布：

$$U[meanCompCost-\delta, meanCompCost+\delta]$$

则由概率学知识可以知道，*meanCompCost(i)*的方差是$\delta^2/3$，标准差为$\delta/\sqrt{3}$，它表明了在任务 *i* 在不同机器上的平均处理时间的差异度。

Step 2. 对于任务 *i*，产生在机器 *p* 上的计算时间 *T(i,p)*.

同理，让 *T(i,p)*服从如下均匀分布：

$$U[meanCompCost(i)-\gamma, meanCompCost(i)+\gamma]$$

则我们可以知道，$\gamma/\sqrt{3}$衡量了任务 *i* 在不同机器上的计算时间的差异程度，因

此我们定义参数 $h=\frac{\gamma}{meanCompCost}$ 来表明处理器的异构程度。

5.1.1 产生 DAG 图不同层的节点

首先假设 DAG 图共有 *L* 层，节点个数为 *n*，则我们按照如下方式分配每层上的节点，首先第一层和第 *L* 层节点个数都为 1，然后有：

$$(n-2)\div(L-2)=avge\cdots\cdots r$$

其余第 2 层到第 *L*-1 层先平均得到 avge 个节点个数，剩于的 *r* 个节点从第二层到 *L*-1 层逐层分配一个直至全部分配完毕。

5.1.2 产生 DAG 图的边

实验用了两种方式产生了 DAG 图的边，一种是相邻层的边，还有一种是跨层的边。

首先介绍第一种生成边的方式，相连层的边，假设层 *i* 和层 *i*+1 是相邻的两层，$i\in[2,L-2]$，我们保证层 *i* 上的每个节点至少都有一天出边，层 *i*+1 上的每个节点至少都有一天入边。因为第 1 层和第 *L* 层都只有一个顶点，因此我们增加从入口节点到第二层每个节点的所有边，增加从第 *L*-1 层到出口节点的所有边，对于其它层上的相邻边，让 *numberofNode(i)*表示第 *i* 层上节点的个数。并定义：

$$\max(i)=\max\{numberofNode(i),numberofNode(i+1)\}$$

首先产生 *max(i)*条相邻边，让第 *i* 层的第 1 个节点指向第 *i*+1 层的第一个节点，i 层上的第 2 个节点指向 *i*+1 层的第二个节点，以此类推。如果当前层数节点个数不够了，则又从当前层的第 1 号节点循环开始。这一步骤如图 Fig.8（a）所示.

接下来我们做 *maxEdgeChange* 次边的而随机交换，每次交换我们随机两条边，并交换两条边的目标顶点。如图 Fig.8(b)所示，我们选择边(1,4)和边（3,3）以及边（1,1）和边（2,5），然后分别按照如上方式作交换。

最后我们在随机加上 *moreEdge(i)*的边，如图 Fig.8（c）所示，*moreEdge(i)*，

maxEdgeChange 值的选取有如下规则：

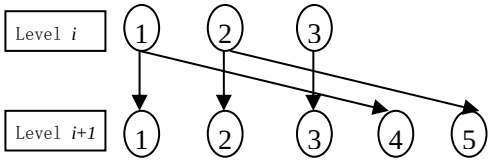
$$\max EdgeChange = randow(0, \max(i))$$
$$moreEdge(i) = random(0, \max MoreEdge(i))$$
$$\max MoreEdge(i) = numberofNode(i) \times numberofNode(i+1) - \max(i)$$

对于第二种类型的边，定义 *maxCrossEdge* 为最多可能的交叉边，*adjEdges* 为相邻层的边，则

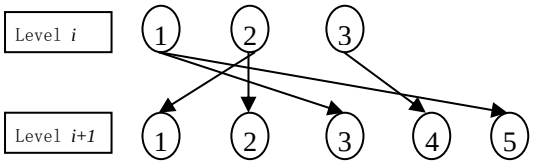
$$\max CrossEdge = n \times (n-1)/2 - adjEdges$$

接下来，从不同层分别选取源节点和目标节点并添加边。选取增加的交叉边的数目是 *crossEdge*，大小为如下：

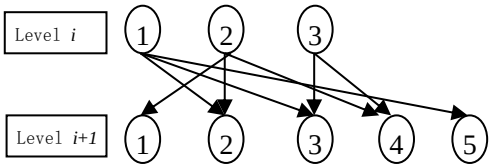
$$crossEdge = randow(0, \max CrossEdge)$$



(a) 初始分边



(b) 交换相邻层的边



(c) 添加边

Fig.8 产生相邻层的边

在产生边阶段的最后，因为有可能有重复边，所有需要去掉重复边。

综上几个步骤，边的生成已经全部完成，然后接下来开始生成边的权重，也就是传输时间的大小，定义参数 *CCR* 如下：

$$CCR = \frac{\underset{(i,j)\in E}{avg}\ C(i,j)}{\underset{i\in V}{avg}(\underset{p\in P}{avg}(T(i,p)))} = \frac{meanCommCost}{meanCompCost}$$

其中 *meanCommCost* 是表示所有任务之间的平均传输时间，*meanCompCost* 表示所有任务在所有处理器的平均计算时间，*CCR* 越大表明在任务调度过程中传输时间对调度长度影响越大，*CCR* 越小表明机器的计算时间对调度长度影响越大。同理，让 *meanCommCost* 也服从如下均匀分布：

$$U[meanCommCost-\sigma, meanCommCost+\sigma]$$

则$\sigma/\sqrt{3}$代表了边权上的差异程度。

至此，DAG 图用例已经全部生成完毕，实验重点分析了如下参数对算法性能的影响，包括节点个数 *n*，机器个数 *m*，层数 *l*，异构性 *h*，边权因子 *CCR*，并且我们给定一组标准配置，值如下：

Table 4 参数标准配置

n	*l*	*m*	*CCR*	*h*
50	7	100	2.0	0.7

然后采用参照对比的方式，只变化一个参数，固定其余参数的方式来比较不同算法 TANH，HEFT，DCPD，TDICMA 算法产生的效果。

实验部分，这些参数的取值分别如下：

n = {30,35,40,45,50,55,60,65,70,75,80, 85,90,95,100}

l = {3,4,5,6,7,8,9,10,11,12,13,14,15,16,17}

CCR={0.1,0.3,0.5,0.7,1.0,1.2,1.5,1.8,2.0,2.5,3.0,5.0, 7.0,10.0,15.0}

h={0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.1,1.2,1.3 1.4}

机器个数 *m* 的取值设定为 *DAG* 图的最大宽度，$\max Width(G)$，其中$\max Width(G)$定义任意相邻两层的最大边数。这样保证机器个数总是足够的。

5.2 算法比较与分析

实验总共产生了 50625 组测试用例，并分别产生了 TDICMA，TANH，HEFT，DCPD 在这些测试用例上的调度结果。如图 Fig.9 所示，表明各个算法在总调度长度大小的排名情况，从中可以看到 TDICMA 算法调度长度最小，排在第一名的用例共有 29434 组，占 58.14%,接下来排在第二名的用例有 10410，占 20.56%，另外还可以发现，紧靠 TDICMA 后面的是 DCPD 算法，可以看到 DCPD 绝大多数都排在了第二名，HEFT 大多数用例都集中在第三名。因此，实验表明，TDICMA 算法调度结果普遍优于 DCPD，TANH，HEFT 等算法。

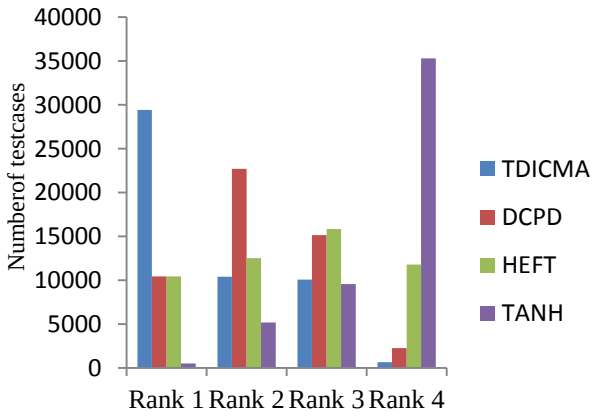


Fig.9 算法在用例上调度结果的排名情况

下面我们来具体分析不同参数下对各个算法的表现。实验采用参照对比的方式，首先按照 Table 4 的标准配置，然后变化其中的某一个参数，每个参数的取值在上一节中已经给出。对于某一个参数来说，虽然其余参数的配置相同，但是由于边的权重以及处理器的计算时间等值是随机产生的并不相同，因此我们对于每个参数的每次变化进行了多次测试，然后取它的平均值，图中阴影部分代表实验数据的方差大小。

5.2.1 节点个数 n 比较

如图 Fig.10 所示，TDICMA，DCPD，TANH，HEFT 算法都随着节点个数的增加，调度长度变大，并且基本都呈现线性增长，且 TDICMA 总是好于 DCPD，TANH，HEFT。

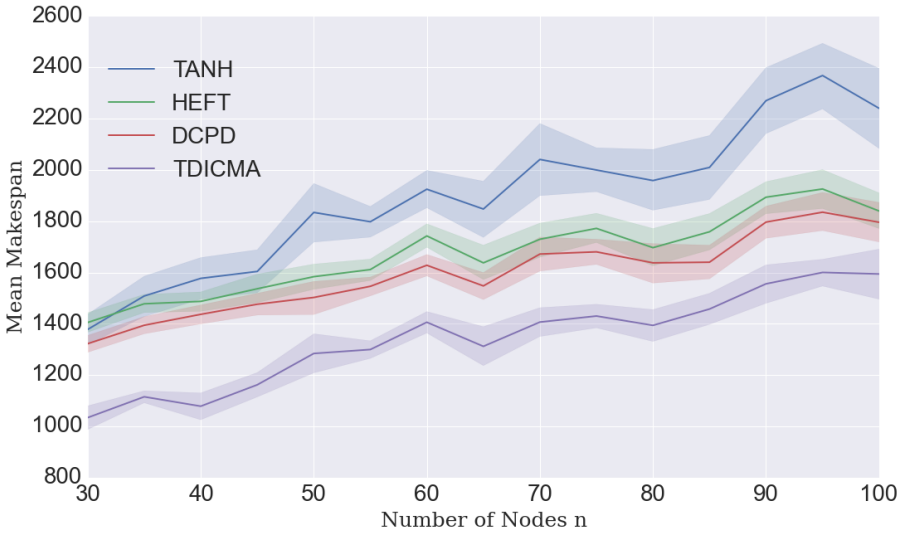


Fig. 10 不同的节点个数下算法的调度长度平均值

5.2.2 边权因子 CCR 比较

如图 Fig.11 所示，TDICMA，DCPD,HEFT,TANH 算法随着 CCR 的增加，调度长度也呈现线性增长趋势，且在 CCR 非常小的情况下或者非常大的情况下，如 CCR<0.3 和 CCR>13 的时候，DCPD 算法要好于 HEFT 以及其他算法，但是在 CCR 属于[0.3~13]的范围区间可以发现 TDICMA 算法普遍好于其他算法。可以知道当 CCR 很小的情况下，即调度长度主要受机器的计算时间影响，机器之间的传输时间基本可以忽略不计，在 CCR 很大的情况下，调度长度主要受机器之间的传输时间影响，机器的计算时间基本可以忽略不计，这两种情况都属于现实情况中的极端情况，在实际应用中并不常见。因此，可以看出，TDICMA 算法在实际的应用中有很好的应用性能。

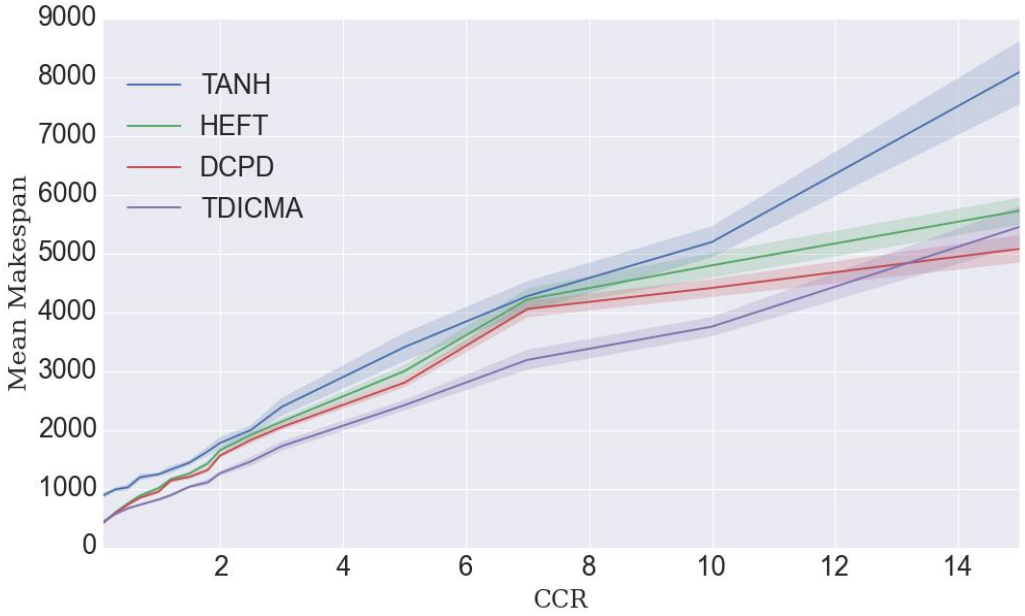


Fig. 11 不同的 CCR 下算法的平均调度长度

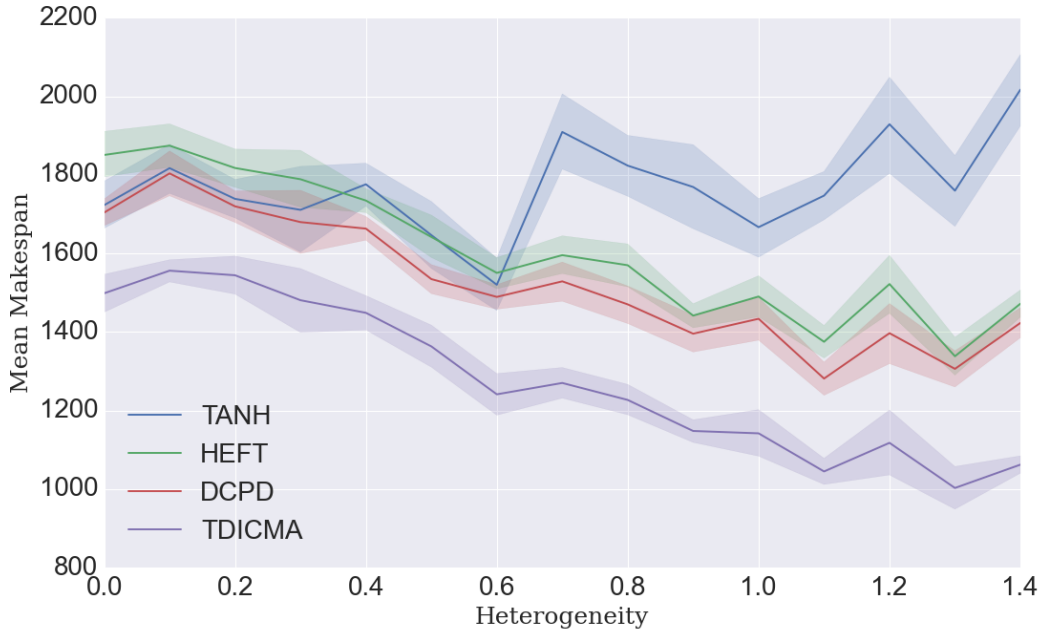


Fig. 12 不同异构程度下不同算法的平均调度长度比较

5.2.3 异构性 h 比较

如图 Fig.12 所示，可以看到 TANH 受 *h* 变化影响最大，最不稳定，TDICMA，DCPD，HEFT 在 *h*<1 的时候都随着 *h* 的增大，调度长度都呈现线性减少的趋势，说明算法有效的利用了不同机器之间的不同的处理能力，但是随着 *h* 大于 1.1 的情形，因为机器的计算时间也可能会变得很大，可以看到当 *h*>1.1 时实验数据方差变大，调度结果也呈现增长趋势。

5.2.4 层级数 l 比较

如图 Fig.13 所示，TDICMA，DCPD，TANH，HEFT 算法都随着层级数的增大，调度长度也随之变大，而且 TDICMA 的调度长度总是要小于其他算法，另外从图中可以发现，TDICMA 调度长度变大的速度要小于其他算法调度长度变大的速度，也就是说算法 TDICMA 对复杂 DAG 图有更强的适应能力。

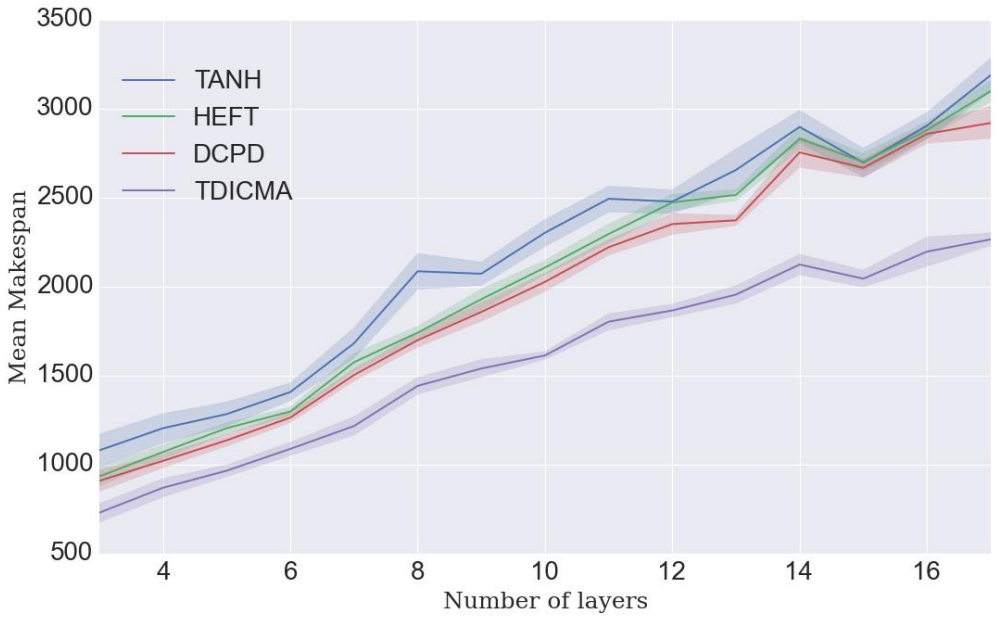


Fig. 13 不同的层级数下算法的平均调度长度比较

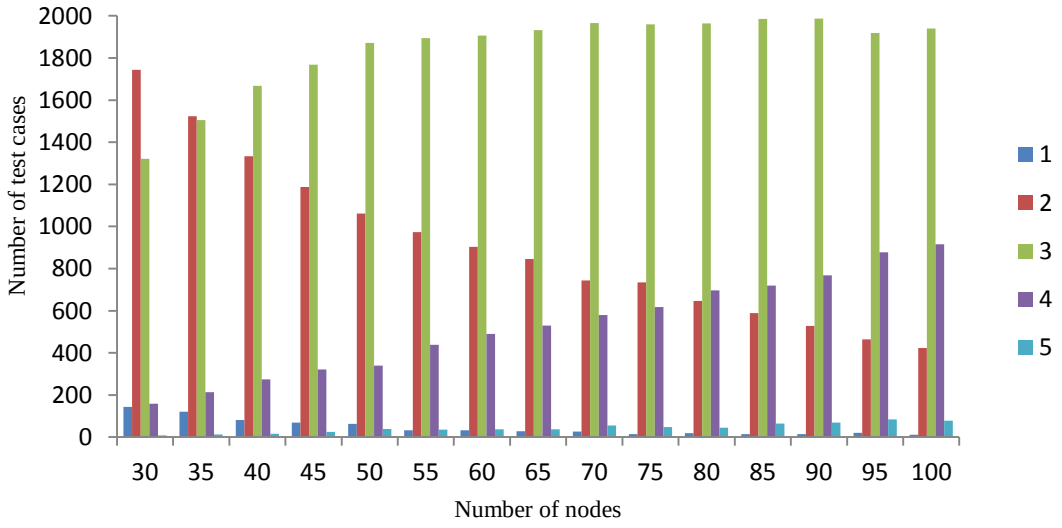


Fig. 14. 不同节点数下达到局部最优所用迭代次数的用例数目比较

5.2.5 任务复制的链式反应现象分析

在上面 4.3 章中说道，任务复制过程选取迭代了 4 次来消除链式反应带来的负面效果，在实验过程中，我们通过在找寻最优解的过程中记录下迭代次数，如果每次迭代能够继续找到候选复制的位置使得调度长度减少，则继续新一轮的迭代，否则就退出迭代。我们在 50625 组测试用例记录得到最优解时的迭代次数，如图 Fig.14

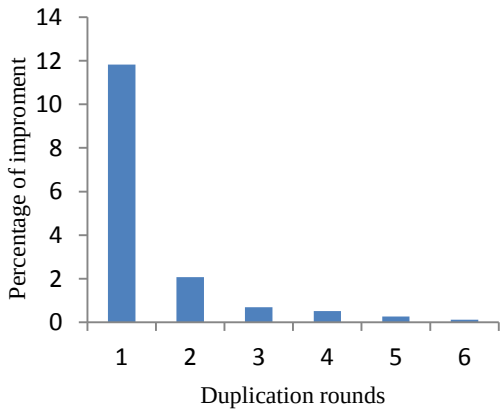


Fig. 15. 每次迭代效果提升的百分比.

所示，表明了测试用例基本在迭代 4 次之后基本上能够找到所有可选的复制候选集，因此我们选取的迭代次数为 4，另外，Fig.15 表明了每次迭代之后调度结果变好的提升百分比。

5.3 本章小结

本章首先介绍了 DAG 图测试用例的生成器，然后比较了算法在 50625 组测试用例上总的调度结果排名情况，接下来对实际 DAG 图的一些属性，如节点个数 n，层数 l，边权因子 CCR，异构程度 h 等参数分别对 TDICMA，TANH，HEFT，DCPD 算法进行了对比，实验表明，TDICMA 在各方面都明显优于其他算法，同时本章还对复制的迭代次数进行了实验研究分析，在最好调度情况下记录下任务复制迭代的次数，找到 TDICMA 算法复制过程中最合适的迭代次数，以此解决链式反应引起的负面效果。

6 总结与展望

6.1 总结

本文重点研究了在分布式系统下具有约束关系的任务的调度问题，提出了基于任务复制及插入的聚簇和归并算法(TDICMA)。

算法流程包括以下 4 个步骤：

1 任务聚簇，在初始生成聚簇调度前，首先计算了相关参数，任务的最早可能开始时间（*est*）,任务最早可能完成时间（*ect*）,关键前驱（*cpred*），喜爱机器的程度（*fproc*），任务优先级（*level*）等，算法借鉴了 TANH 的初始生成聚簇的流程，改进了其中的缺陷，重新设计了 *est*，*cpred*，*fproc* 等参数，以及在关键前驱任务失效时设计了新的关键前驱更换方案，同时在生成初始聚簇的每次循环中，改进了结束条件，TDICMA 在发现通过网络传输数据到达的时间比任务聚簇获取数据来的更早，则此次聚簇循环就会退出。

2 任务复制，TDICMA 通过拷贝关键前驱链来提前任务的结束时间，在此过程中发现了连锁反应的现象，本轮无效的复制可能在下一轮重新变得有效，本文通过实验研究发现了迭代次数对算法效果的影响，并设计了合理的迭代次数来消除链式反应带来的影响。

3 任务归并，TDICMA 重新设计了任务归并的算法流程，这是考虑到分布式系统环境下很强的异构性的特点，对于处理器集合具有很强的处理能力差别的时候，将那些处理能力较弱的机器的任务合并到处理能力强的机器更有利于优化调度结果。

4 任务插入，TDICMA 在解决同构 EZ 经典算例步骤中发现，对于那些不是关键前驱的任务，也有可能存在将其插入到后继所在机器对应位置的空隙处，算法通过寻找这些空隙位置，插入相关任务，进而更一步减少调度长度。

另外，还实现了一个分布式多任务调度问题的综合比较系统，主要实现对比了文献研究过程中的相关算法（DCPD，HEFT，TANH），通过这个系统可以随机生成具

有参考性的 DAG 模型，并与经典调度问题的算法结果进行对比，同时，对于任务复制的链式反应现象也进行了重点研究，过多的迭代次数会消耗更多的计算资源，过少则又可能得不到更优的解。最后通过不断的实验对比和分析，找到了最合理的迭代次数。

最后，定义了 DAG 模型调度问题的一些属性参数，如节点个数 n，边权因子 ccr（通信时间与计算时间的比率），层级数 l，异构性程度 h，通过这个综合比较系统分析了这些参数对算法带来的影响。

6.2 展望

接下来更多的改进工作还可以从以下几个方面来考虑：

(1) 对本文提出的 TDICMA 算法给出更多理论上的分析，如从理论上给出性能下界。

(2) 相关参数都是事先静态计算好的，都是基于任务在最喜爱的机器上时做出来的估计，一旦在调度过程中任务不再最喜爱的机器上执行的时候，则参数计算出来的值就不太准确，因此可以借鉴 DCPD 算法策略，根据关键路径动态计算相关参数，，任何结合任务复制，任务插入和任务归并等思想。

(3) 针对同构环境和异构环境对 TDICMA 算法做出特定环境的优化，可以有效利用不同环境下的优势特点。

(4) 可以针对算法的不同类型，以及分布式环境的不同（同构和异构），实现更多任务调度求解算法，从而继续完善本实验的综合比较系统。

(5) TDICMA 的优化目标是最小化调度长度，更进一步的优化目标还包括在最小化调度长度的情况下，最小化计算资源，除去不必要的任务复制的代价。

致 谢

时光匆匆，两年的研究生生涯即将结束，在这期间，收获了很多老师同学的指导和帮助。

首先要特别感谢我的导师何琨老师，何老师认真严谨，一丝不苟的科研精神一直深深感染和激励着我。她定期与我组织课题讨论，在讨论时她强调提高我们的板书能力，要求我们学会深入浅出的讲解问题，要说出算法的真正韵味，要知其然而知其所以然。在做分布式多任务调度求解算法的过程中，何老师给出了很多建设性的指导和帮助，多次组织讨论，让我在已有工作基础上更好的拓展研究问题和优化算法。在我困顿时，也给了我很大的鼓励。在生活中，何老师也会定期举行一些活动，交流各方面的心得，丰富了我的视野。

感谢许贵平老师，在我后期参与到智能车载盒项目当中，许贵平老师在实践方法上给了我很多指导，还要感谢许如初老师，许如初老师多次组织实验室团建活动，给我带来了很多的欢乐和鼓励。

感谢实验室的师兄弟们。曹伟刚，姬朋力，刘燕丽，周阳，叶兵，肖凡，黄梦龙，朱鹏，漆学志，汪光炼，凡义等同学在这两年里给我带来了很大的欢乐和帮助。

最后还要感谢何老师定期组织讨论的课题组的本科生同学们，通过一起讨论，也让我受益颇多。

参考文献

[1] Palis M A, Liou J C, Wei D S L. Task clustering and scheduling for distributed memory parallel architectures. IEEE Transactions on Parallel and Distributed Systems, 1996, 7(2):46-55

[2] T. Cassavant, J.A. Kuhl. Taxonomy of Scheduling in General Purpose Distributed Memory Systems. IEEE Transactions on Software Engineering, 1988, 14(2):141-154

[3] A. Abraham, R. Buyya, B. Nath. Nature's Heuristics for Scheduling Jobs on Computational Grids. in: Proceedings of 8th International Conference on Advanced Computing and Communications. Cochin, India: IEEE Press, 2000. 45-52

[4] S. Darbha, D.P. Agrawal. Optimal Scheduling Algorithm for Distributed Memory Machines. IEEE Transactions on Parallel and Distributed Systems, 1998, 9(1):87-95

[5] A. Ranaweera, D.P. Agrawal. Task Duplication Based Scheduling Algorithm for Heterogeneous Systems. in: Proceedings of 14th International Parallel and Distributed Processing Symposium. Cancun: IEEE Press, 2000. 445-450

[6] H. Topcuoglu, S. Hariri and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Transactions on Parallel and Distributed Systems, 2002,13(3):260-274

[7] H. El-Rewini, T.G. Lewis. Scheduling Parallel Programs onto Arbitrary Target Machines. Journal of Parallel and Distributed Computing, 1990, 9(2):138-153

[8] G.C.Sih, E.A.Lee. A Compile-time Scheduling Heuristic for Interconnection constrained Heterogeneous Machine Architectures. IEEE Transactions on Parallel and Distributed Systems,1993, 4(2):175-187

[9] N. Mehdiratta, K. Ghose. A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessor. in: Proceedings of the International Conference on Parallel Processing. North Carolina State University, NC, USA:CRC Press, 1994.151-154

[10] T. Yang, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of

Processors. IEEE Transactions on Parallel and Distributed Systems, 1994, 5(9):951-967

[11] Y.-K. Kwok, I. Ahmad. Dynamic Critical-path Scheduling: An Effective Technique for Allocating Task Graphs onto Multi-processors. IEEE Transactions on Parallel and Distributed Systems,7(5):506-521

[12] M.A. Palis, J.-C. Liou, D.S.L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. IEEE Transactions on Parallel and Distributed Systems, 1996, 7(1):46-55

[13] A. Radulescu, A.J.C. Gemund. Low-cost Task Scheduling for Distributed-memory Machines. IEEE Transactions on Parallel and Distributed Systems, 2002, 13(6):648-658

[14] S.J. Kim, J.C. Browne. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures. in: Proceedings of the International Conference on Parallel Processing. The Pennsylvania State University, University Park, PA, USA: Pennsylvania State University Press, 1988. 8

[15] I. Ahmad, Y.-K. Kwok. On Exploiting Task Duplication in Parallel Program Scheduling. IEEE Transactions on Parallel and Distributed Systems, 1998, 9(9):872-892

[16] G.-L. Park, B. Shirazi, J.Marquis. DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems. in: Proceedings of 11th International Parallel and Distributed Processing Symposium. Genva: IEEE Press, 1997. 157-166

[17] K. Shin, M. Cha, M. Jang, et al. Task Scheduling algorithm using minimized duplications in homogeneous systems. Journal of Parallel and Distributed Computing, 2008, 68(4):1146-1156

[18] M.K. Dhodhi, I. Ahmad, A. Yatama, et al. An Integrated Technique for Task Matching and Scheduling onto Distributed Heterogeneous Computing System. Journal of Parallel and Distributed Computing, 2002, 62(9):1338-1361

[19] O. Sinnen, L.A Sousa. Communication Contention in Task Scheduling. IEEE Transactions on Parallel and Distributed Systems, 2005, 16(6):503-515

[20] G.Q. Liu, K.L. Poh, M. Xie. Iterative List Scheduling for Heterogeneous Computing. Journal of Parallel and Distributed Computing, 2005, 65(5):654-665

[21] M.I. Daoud, N. Kharma. A High Performance Algorithm for Static Task Scheduling in Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing, 2008, 68(4):399-409

[22] X. Tang, K. Li, G. Liao, R. Li. List Scheduling with Duplication for Heterogeneous Computing Systems. Journal of Parallel and Distributed Computing,2010, 70(4): 323-329

[23] R. Bajaj, D.P. Agrawal. Improving Scheduling of Tasks in a Heterogeneous Environment. IEEE Transactions on Parallel and Distributed Systems, 2004,15(2): 107-118

[24] S. Baskiyar, C. Dickinson. Scheduling Directed A-cyclic Task Graphs on a Bounded Set of Heterogeneous Processors Using Task Duplication. Journal of Parallel and Distributed Computing, 2005, 65(1):911-921

[25] A. Gerasoulis, T. Yang. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. Journal of Parallel and Distributed Computing, 1992, 16(4):276-291

[26] Muhammad Khurram Bhatti, Isil Oz. Noodle: A Heuristic Algorithm for Task Scheduling in MPSoC Architectures. in: Proceedings of the 17th Euromicro Conference on Digital System Design. Verona: IEEE Press, 2014. 667-670

[27] Guoqi Xie, Renfa Li. A High-performance DAG Task Scheduling Algorithm for Heterogeneous Networked Embedded Systems. in: Proceedings of the 28th International Conference on Advanced Information Networking and Applications. Victoria, BC: IEEE Press, 2014. 1011-1016

[28] Ali Javed, Khan Rafuqul Zaman. Optimal Task Partitioning Model in Distributed Heterogeneous Parallel Computing Environment. Technology Research Database,

2012, 2:13

[29] Jing Mei, Kenli Li, Keqin Li. Energy-aware Task Scheduling in Heterogeneous Computing Environments. Cluster Computing, 2014, 17(2):537-550

[30] Abrishami S, Naghibzadeh M, Epema D.H.J. Cost-Driven Scheduling of Grid Workflows Using Partial Critical Paths. IEEE Transactions on Parallel and Distributed Systems, 2012, 23:1400-1414

[31] Hui Cheng. A High Efficient Task Scheduling Algorithm Based on Heterogeneous Multi-Core Processor. in : Proceedings of 2nd International Workshop on Database Technology and Applications. Wuhan: IEEE Press, 2010. 1-4

[32] Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, et al. A Dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems. in: Proceedings of the 12th International Conference on Parallel and Distributed Systems. Washington, DC, USA: IEEE Computer Society, 2006. 365-374

[33] 石威，郑纬民. 相关任务图的均匀动态关键路径调度算法. 2001,计算机学报, 24(9):991-997

[34] 何琨,赵勇,黄文奇. 基于任务复制的分簇与调度算法. 计算机学报,2008, 31(5):733-740

[35] 何琨, 黄文奇. 分布式内存机器中优化调度问题的数学模型. 华中科技大学学报(自然科学版), 2008, 36(2):61-65

[36] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acylic Graphs onto Multiprocessors. Journal of Parallel and Distributed Computing, 1992, 16(4):276-291

[37] 周双娥，袁由光，熊兵周等. 基于任务复制的处理器预分配算法. 计算机学报, 2004, 27(2):216-223

[38] S.S. Pande, D.P. Agrawal, J. Mauney. A Scalable Scheduling Method for Functional Parallelism on Distributed Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1995, 6:388-399

[39] Y.C. Chung, S. Ranka. Application and Performance Analysis of a Compile-Time

Optimization Approach for List Scheduling Algorithm on Distributed Memory Multiprocessors. in: Proceedings of Supercomputing. Minneapolis, MN: IEEE Press, 1992. 512-521

[40] Ralf Hoffmann, Andreas Prell, Thomas Rauber. Dynamic Task Scheduling and Load Balancing on Cell Processors. in: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. Pisa: IEEE Press, 2010. 205-212

附录 1：攻读硕士学位期间发表论文情况

[1] Kun He, Xiaozhu Meng, Zhizhou Pan. Task-Duplication Based Clustering and Merging Algorithm for Heterogeneous Computing Environments. IEEE Transactions on Parallel and Distributed Systems. (2014 年 12 月投稿，目前正在根据审稿意见修改论文)

附录 2：攻读硕士学位期间参与的科研项目

[1] 基于糅合策略的超大规模集成电路布图规划问题的算法研究，国家自然科学基金面上项目，2014.3-2015.7

[2] 基于智能车载盒的叉车资源综合管理系统，2013.1-2014.4