



University
of Glasgow | School of
Computing Science

Towards Faster Combinatorial Search: Performance-Driven Workstealing Policy in YewPar

Hao Xie

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of
the Degree of Master of Science at The University of Glasgow

24th August 2023

摘要

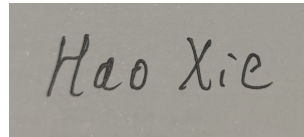
abstract goes here

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Hao Xie

Signature:

A rectangular box containing a handwritten signature in dark ink. The signature is written in a cursive style and reads "Hao Xie".

Acknowledgements

acknowledgements go here

目录

1	Introduction	5
2	Survey	7
2.1	组合搜索	7
2.1.1	并行组合搜索(Parallel Combinatorial Search)与YewPar	7
2.2	workstealing	7
2.3	为什么要在YewPar中设计并使用新的workstealing策略	8
3	Design and implementation	9
3.1	总体设计	9
3.2	性能数据收集与传输	10
3.2.1	节点所有worker的负载情况	10
3.2.2	各节点剩余任务量与获取花费时间收集	13
3.3	最优窃取目标计算与缓存	14
3.4	Refresh data and provide tasks mechanism	16
4	Evaluation	18
4.1	Search Applications	18
4.2	Experimental Setup	18
4.3	Isolated Performance Evaluation of Different Policies	19
4.3.1	Isolated Performance Evaluation of Maxclique	19
4.3.2	Isolated Performance Evaluation of NS-hivert	22
4.4	Simultaneous Performance Evaluation under Resource Contention	23
4.4.1	Simultaneous Performance Evaluation of Maxclique	24

4.4.2	Simultaneous Performance Evaluation of NS-hivert	25
5	Conclusion	26
5.1	future work	26
A	First appendix	27
A.1	Section of first appendix	27
B	Second appendix	28
	参考文献	29

Chapter 1: Introduction

精确组合搜索对于包括约束编程、图匹配和计算代数在内的广泛应用都是必不可少的。而组合问题是通过系统地探索搜索空间来解决的,这样做在理论上和实践中都很难计算,其中精确搜索则是探索整个搜索空间并给出可证明的最佳答案。概念上精确的组合搜索通过生成和遍历代表备选选项的(巨大的)树来进行。结合并行性、按需树生成、搜索启发式和剪枝可以减少精确搜索的执行时间。由于巨大且高度不规则的搜索树,并行化精确组合搜索是极具挑战性的。

而其中有名为YewPar[4]的框架,这是第一个用于精确组合搜索的可扩展并行框架。YewPar旨在允许非专业用户从并行中受益;重用编码为算法骨架的并行搜索模式(to reuse parallel search patterns encoded as algorithmic skeletons);并能在多个并行架构上运行。

与此同时,随着并行计算和多核处理器的普及,有效的任务调度变得越来越重要。能并行加速搜索是YewPar的一个关键特性,这是通过各个节点的多个worker在本地任务池无任务时向其它节点的任务池窃取任务来实现节点空闲资源的利用。而YewPar在本地任务池无任务时,是随机选取节点来窃取任务的,其中Workstealing是一种被广泛研究和应用的并行调度策略,它允许空闲的处理器从繁忙的处理器中“窃取”任务。然而,很多如YewPar这样的workstealing调度器往往采用偏随机窃取的策略,这往往会导致不必要的开销和延迟,浪费了大量试探窃取任务的时间,同时导致各节点的相对负载不均衡,延长了最终完成的时间。目前也有很多关于workstealing的改进,但是不能很好的兼顾到低开销,去中心化,高负载均衡,高性能,高可扩展性等特性。

相比之下,本文提出了Performance-Driven Workstealing Policy,并为之设计了新的框架能定期监测与传输多项有价值的数据,并计算缓存最优窃取目标节点,它能够以低成本搜集多项性能指标,如各节点的任务执行时间、处理器的工作/空闲时间比例和任务池获取的耗时,并通过自研的Time-Optimized Workstealing Strategy算法计算出最优窃取目标节点并缓存和定时刷新,当有worker空闲时便能直接从缓存获取最近一段时间的最优窃取目标,从而能很好的在多节点去中心化的环境下以较低成本缩短完成全部任务所需的时间。

本文对具体的设计与实现细节进行了剖析,其中Performance-Driven Workstealing Policy在搜集性能参数时采用了平台无关与去中心化的设计,一方面并不涉及具体的系统参数调用命令,而是从YewPar内部和底层的HPX[10]框架进行数据收集,从而能够在不同的硬件平台上正常运行;另一方面没有单一的节点负责收集所有节点的性能参数,而是各节点各自收集本地的性能参数,并通过HPX的分布式通信机制分享本地处理后的数据,同时获取其它节点数据进行本地最优窃取目标计算,从而避免了单一节点的性能瓶颈。其中Time-Optimized Workstealing Strategy的核心思想是计算各节点执行本地任务的所需时间的预期和获取节点任务池任务的耗时的预期,并优先选取所需时间预期最大的节点同时尽量缩短不必要的获取任务的额外耗时,从而降低各节点的worker空闲率的同时缩短完成全部任务所需的时间。其中定时刷新最优窃取目标缓存的任务由一种动态调整刷新时间的自

动刷新任务来主要负责,各节点都会部署一个这样的刷新器,它通过间隔一个动态时间后执行刷新性能参数信息并计算最优窃取目标最后将最优目标进行缓存来实现刷新最优窃取目标缓存的目的.而空闲的worker则会在试图获取缓存目标任务失败时进行一次称为辅助刷新的操作,帮助此时可能处在休眠的刷新器进行刷新最优窃取目标缓存的任务.这些工作结合起来便能够实现Performance-Driven Workstealing Policy的加速效果.

本文同时对改进workstealing策略后的YewPar进行了评估,评估在具有多核机器的Beowulf集群上进行,结果表明,改进后的YewPar在不同节点数量和线程下相比原YewPar平均能够获得更好的性能,能在不影响搜索结果的情况下不同程度的有效缩短执行完全部任务所需的时间.

Chapter 2: Survey

2.1 组合搜索

2.1.1 并行组合搜索(Parallel Combinatorial Search)与YewPar

随着现代计算机硬件的发展,特别是多核处理器和分布式系统的普及,利用并行性来加速组合搜索已经成为研究的热点. 并行组合搜索的目标是将搜索空间分解成多个部分,以便可以在多个处理单元上同时执行,从而加速解决方案的发现。

YewPar是一个专为组合搜索设计的并行框架. 它提供了一套强大的工具和策略,允许开发者轻松地并行化他们的搜索算法. YewPar的主要特点是其灵活性和可扩展性,使其能够应对各种复杂的搜索场景. 其中,workstealing是YewPar中用于任务调度的核心策略,它允许处理器在本地任务队列为空时从其他处理器窃取任务,确保所有处理器都能保持忙碌,从而提高整体性能。

Performance Anomalies

搜索算法依赖于排序启发式方法尽早找到有用的节点,例如决策问题中的目标节点,或者分支限界优化问题中的一个强约束节点。为了利用这些启发式方法,搜索按从左到右的顺序进行(在树的所有深度上). 因此,访问节点的顺序搜索具有关于搜索树的完整信息,例如来自它左侧的所有节点的当前最佳解,并且搜索是确定性的。并行搜索在没有完整的左侧信息的情况下推测性地搜索子树,并可能从从右到左的信息流中受益。但是,这种推测意味着并行搜索可能比相同的顺序搜索执行更多的工作。因此,并行搜索因性能异常而臭名昭著[7]. 有害的异常是当 w 个工作线程上的运行时间超过 $w-1$ 个工作线程时发生的。在这里,额外的工作可能超过额外计算资源的好处,或者额外的计算资源可能会破坏搜索启发式方法。加速异常是超线性的加速,通常是由于从右到左的知识流动,通过允许比顺序搜索中更多的修剪来减少总体工作量。异常的存在使得推理搜索应用的并行性能变得困难。Yew-Par旨在避免有害的异常,同时允许加速异常; [2]报告了一个专门的搜索框架,该框架仔细控制异常以提供可复制的性能保证。

2.2 workstealing

Workstealing 是并行编程中的一个核心概念。其主要优点在于分散调度,让每个处理器自主地管理其任务队列,从而显著降低了全局同步所带来的开销。众多并行框架和库,如Cilk、TBB 和 OpenMP,都已经采纳了这种方法。

在workstealing策略中,系统内的每个处理器都有自己的待执行任务队列. 每个任务都由一系列指令组成,这些指令需要按顺序执行。但在执行的过程中,一个任务还可能生成新的子任务. 这些子任务被初步放入生成它们的任务所在的处理器队列. 当某处理器的任务队

列为空时,它会尝试从其他处理器的队列中“窃取”任务. 这样,workstealing 实际上将任务调度到了空闲的处理器上,并保证了只有在所有处理器都繁忙时才会发生调度开销.[6]

与workstealing形成对比的是工作共享策略,它是动态多线程调度的另一种方法. 在工作共享中,新产生的任务会被立即调度到一个处理器上执行. 相较于此,workstealing减少了处理器间的任务迁移,因为当所有处理器都繁忙时,这种迁移是不会发生的.[5]

尽管如此,传统的随机窃取策略在多节点复杂环境中往往有较大的开销. 这些开销主要来源于“窃贼”在集群中随机地探测节点以寻找“受害者”. 加之集群规模的不均匀分布,这种问题变得更为严重,导致系统消息量过大以及“窃贼”由于多次窃取失败和网络延迟而产生的长时间饥饿状态. 尽管关于workstealing的优化尝试从未停歇,例如通过建立固定的通道来匹配任务饥饿和任务丰富的节点[13],但至今仍未有一种通用的去中心化、低开销、跨平台且高效的策略出现. workstealing仍处于针对各种应用环境进行持续优化的阶段.

2.3 为什么要在YewPar中设计并使用新的workstealing策略

虽然YewPar已经采用workstealing作为其核心的调度策略,但随机发现一个有任务的节点并一直窃取其任务的方法可能导致不必要的开销和计算资源的空闲,尤其是在不均匀的任务分布和各节点具有不均衡的计算资源下. 考虑到组合搜索的特性,任务之间可能存在巨大的执行时间差异,这使得随机窃取策略可能并不是最优选择.

为了更好地利用每个处理器的计算能力并减少不必要的通信开销, YewPar需要一种更加智能的workstealing策略. 基于各节点性能的新的Performance-Driven Workstealing Policy正是为了解决这一问题而提出的. 通过低成本评估每个节点的性能如平均任务执行时间等,新策略可以更精确地刷新最优窃取目标,从而提高整体性能.

Chapter 3: Design and implementation

3.1 总体设计

本文为了Performance-Driven Workstealing Policy设计了一套新的框架, 框架包含Scheduler Channel(主要负责监测与统计与传输节点所有workers的负载状态), Performance Monitor(主要负责统计各项性能数据与发送各项处理后的本地性能数据,并计算和缓存最优窃取目标节点的id), 和Performance Policy(在本地任务池无任务时,主要负责从Performance Monitor获取最优窃取目标).

组件之间的关系大致如Figure 3.1所示.

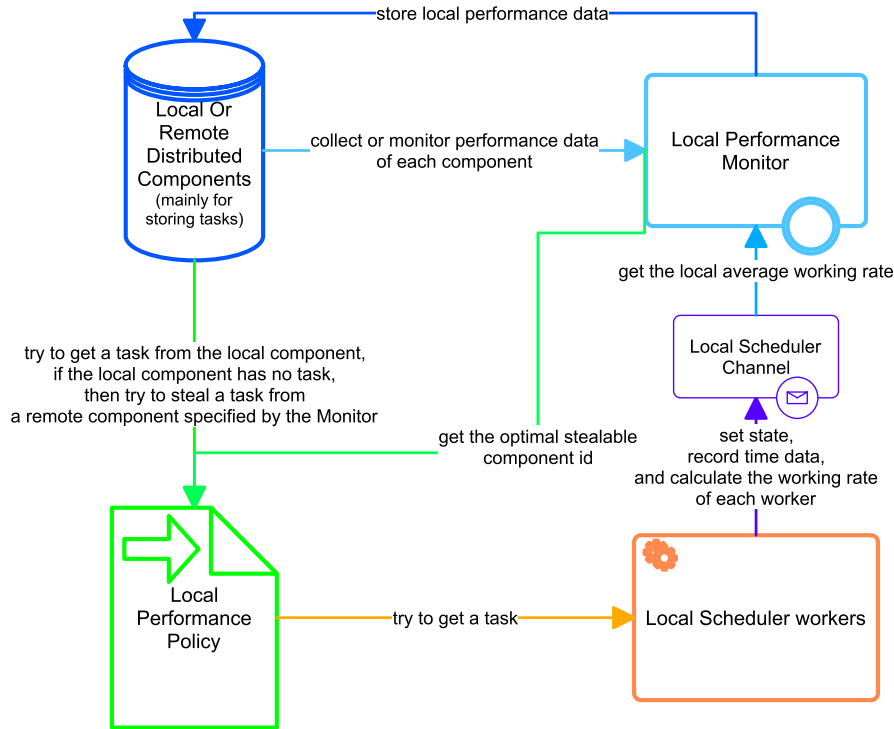


图 3.1: Performance-Driven Workstealing Policy Architecture

其中Performance Monitor主要负责各项性能数据的统计和传输,并计算和缓存最优窃取目标节点的id; distributed component负责存储可供远程和本地访问的任务池和性能等数据,通过hpx框架的action提供访问与改写操作; Scheduler Channel提供接口让workers能更新自己当前负载状态并计算自己的负载数据, workers负责从Performance Policy获取可执行的任务; Performance Policy负责调取本地任务池任务或者从Performance Monitor获取最优窃取目标节点后窃取其任务池任务.

3.2 性能数据收集与传输

数据收集包含三部分:各节点负载情况,各节点剩余任务量,各节点获取任务池任务的耗时. 这三个参数将会为后续的最优窃取目标计算提供有力的数据支持. 同时,为了保障能以较低成本收集到更有效的数据,本文对三部分都进行了仔细的设计与优化.

3.2.1 节点所有worker的负载情况

首先,YewPar的System Stack大致如 Figure 3.2 [3] 所示.

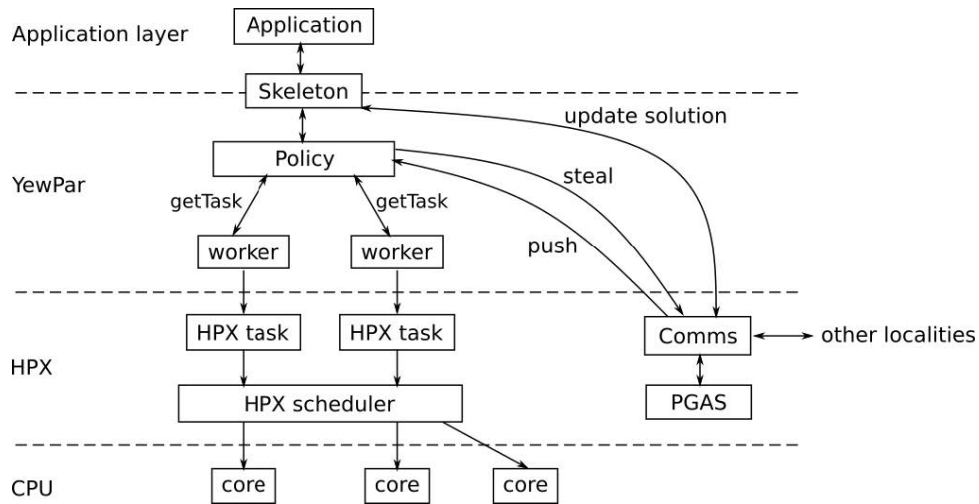


图 3.2: YewPar System Stack

其中Policy负责被worker调用时,通过自己的策略试图获取任务 (最终会从本地池获取或从其它节点任务池偷取任务), 真正负责执行任务的是YewPar的worker, 所以如果想要不依赖系统底层函数获得各节点的负载情况,就需要获取各节点的worker的负载情况. 本文对worker的工作流程进行了分析,如Figure 3.3

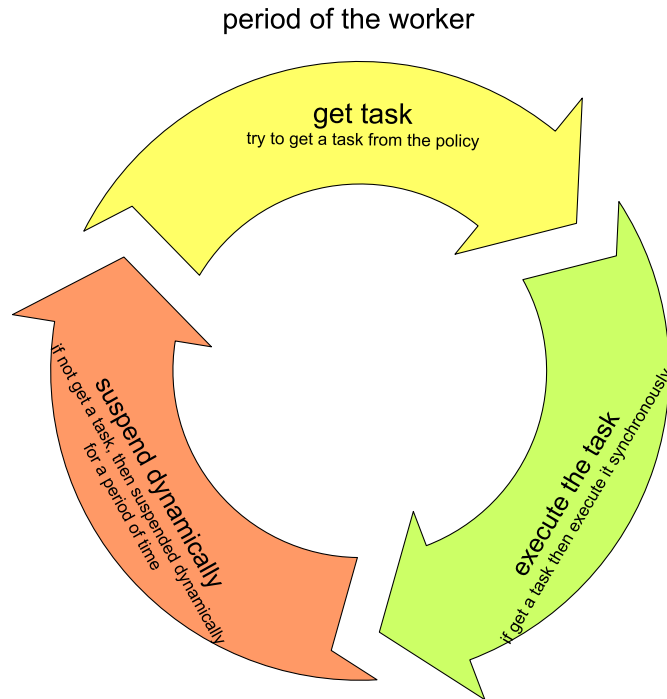


图 3.3: period of the worker

可以看出每个worker从创建到销毁之间, 它的生命周期都处在一个循环中,其中包含了三个可能的阶段:

- 1.调用Policy的getWork函数获取任务;
- 2.如果获取到任务则执行该任务;
- 3.如果没有获取到任务则休眠一段时间, 这段时间随着没有获取到任务的次数的增加而增加,当再次获取到任务时则时间重置

这三个阶段中,第一阶段是每次循环都会经历的阶段, 而第二阶段和第三阶段则是互斥关系,每次循环只会经历其中一个阶段.

如果需要分析worker的真实负载情况, 就需要以第二阶段为核心进行统计, 因为第二个阶段是worker真正执行任务的阶段, 它所消耗的时间是worker执行任务的时间,也就是有效的负载时间, 而其它阶段所消耗的时间则可以归类为worker的空闲时间, 因为这段时间并没有执行任何任务,也就是无效的负载时间.

由于原先一个周期未必能经历第二阶段, 所以为了避免更新无效数据, 本文对worker的生命周期判定进行了重定义, 以一个任务的执行完毕为一个新周期的开始, 以一个任务的开始为一个周期的结束, 这样便能保证每个周期都能经历第二阶段,也就是执行任务的阶段.

在此理论上,本文设计了如Figure 3.4的一套完整的数据收集方案,

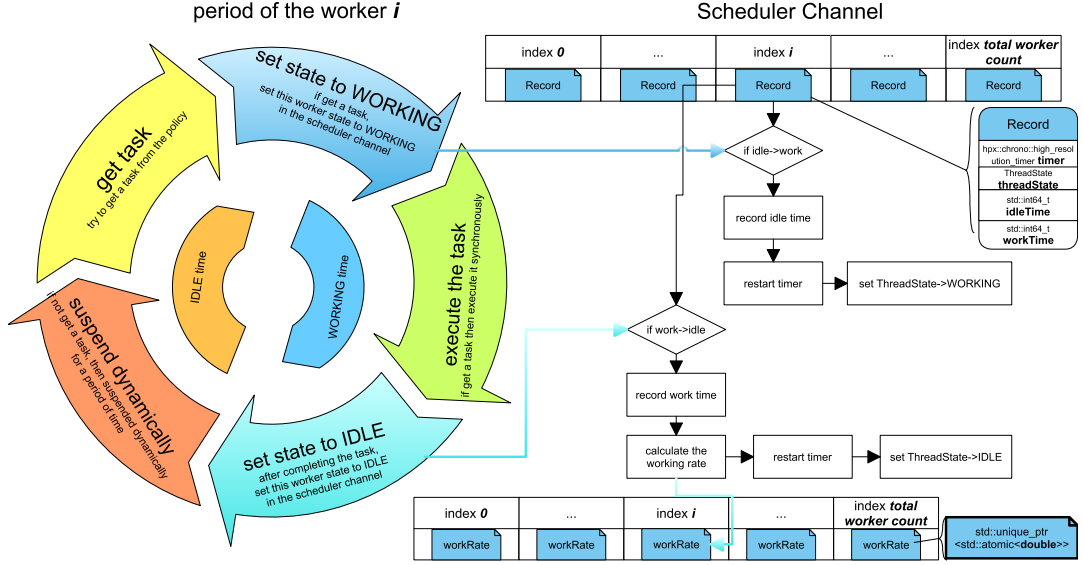


图 3.4: collect worker state to calculate the work load

通过在任务开始执行之前和执行完成后的位置插入探针代码, 让worker在执行任务前和完成任务后都去Scheduler Channel(一个辅助传输状态的channel)中更新自己的状态, 并计算自己的负载数据, 方便后续Performance-Driven Policy获取这些数据进行计算.

其中Scheduler Channel的统计数据主要由两组长度为workers数量的数组组成, 一组存储着特定的Record数据结构, 由一个计时器timer, 一个当前状态threadState, 一个当前周期的空闲时间idleTime和一个当前周期的工作时间workTime组成, 方便workers记录自己的具体各状态持续时间等数据, 由于设计采用一个record对应一个worker的方式, 所以record可以不采用锁机制以提升性能; 另一组存储着采用特定算法计算得出的worker负载率数据workRate, 由被unique_ptr包装的原子double类型组成, 由于可能有worker和Moninter同时访问, 所以需要采用原子类型以保证在无锁机制下数据的正确性.

在”set state to WORKING”阶段, worker主要在channel更新自己这轮周期的空闲时间, 并更改状态为WORKING; 而在”set state to IDLE”阶段, worker主要在channel更新自己这轮周期的工作时间, 并根据空闲与工作时间与历史workRate数据计算出当前的workRate, 并更改状态为IDLE.

而计算最新的workRate的算法如eq. (3.1)所示:

$$\begin{aligned}
 \text{workRate} = & \left(\ln \left(2.72 + \frac{\text{workTime}}{\text{workTime} + \text{idleTime}} \right) \right. \\
 & \times \ln (2.72 + (\text{workTime} + \text{idleTime})) \times 0.65) \\
 & + \text{workRate} \times 0.35
 \end{aligned} \tag{3.1}$$

公式目的是为了计算当前worker是否繁忙和计算速度是否较慢, 主要分为三部分:

1. $\ln \left(2.72 + \frac{\text{workTime}}{\text{workTime} + \text{idleTime}} \right)$, 是由于在每个周期中, 如果WORKING状态的持续时间相

对IDLE状态的持续时间比值越大, 则说明当前任务量较大或者任务较为复杂, 所以需要更多的时间来执行任务. 同时为了避免数据过于敏感, 所以采用了对数函数来降低数据的敏感度, 使用了 $(2.72 + \dots)$ 是为了保证结果永远为正数.

2. $\ln(2.72 + (\text{workTime} + \text{idleTime}))$, 是因为如果当前周期的总持续时间越长, 则说明当前任务较为复杂或者worker的性能较低, 所以需要更多的时间来执行任务. 同时也采用了对数函数来降低数据的敏感度.
3. $(\dots \times 0.65 + \text{workRate} \times 0.35)$, 这是采用了一种称为指数平滑[9]的算法, 通过给与历史数据影响程度一定的系数, 防止短期的波动给结果带来较大的影响, 从而使得数据更加平滑和更具参考价值. 而0.65和0.35则是经过多次测试后得出的一个较为合理的系数, 既保留了一些历史参考, 也较好地体现了最近的数据变化.

最终, 计算出的各个worker的workRate数据会通过提供的本地接口在计算总和并除以worker的数量后递交给Performance Monitor, Performance Monitor会将本地workRate数据提交给local distributed component以供其它节点查询.

3.2.2 各节点剩余任务量与获取花费时间收集

性能数据还需要对各节点剩余任务量和获取花费时间进行收集, 具体实现如Figure 3.5所示.

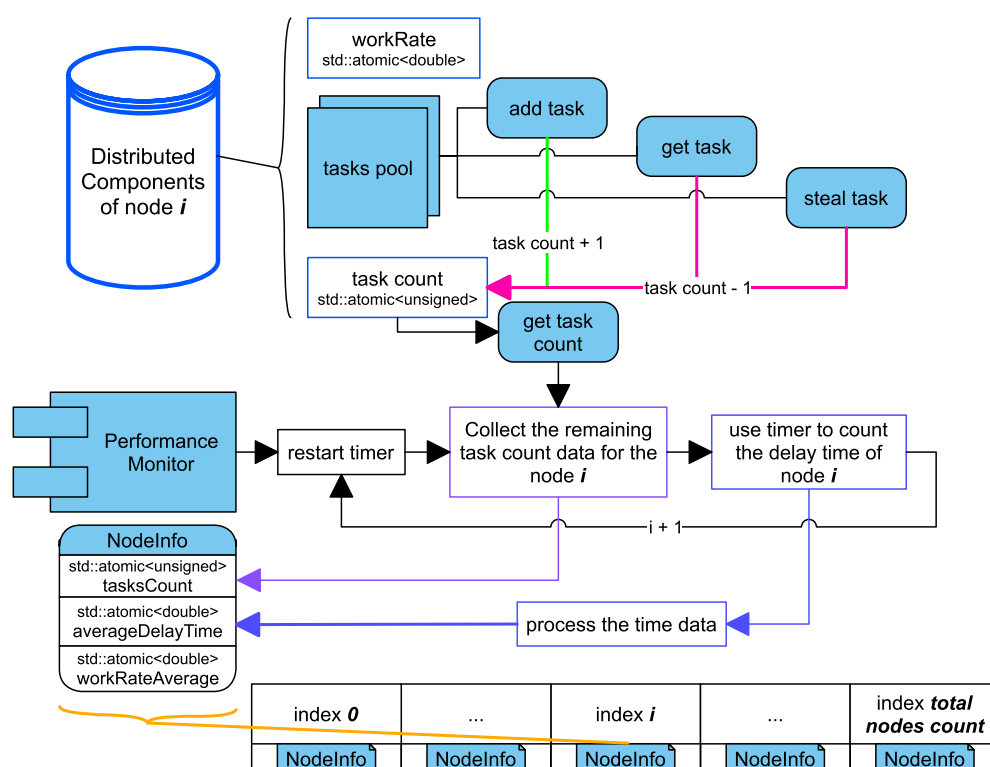


图 3.5: collect tasks count and delay time

在hpx分布式组件中包含了任务池与剩余任务数量, 任务池有三种分布式操作包含添加任务, 获取任务与窃取任务, 所以在调用这三种操作时会进行检测并对剩余任务数进行更新.

而在Performance Monitor中, 会定时遍历并调用各节点的distributed component的接口获取节点的剩余任务数. 同时在获取时通过计时器计算获取节点任务数的耗时, 并加以处理后进行存储, 存储会存储在一个长度为节点总数的数组的对应节点编号的位置的一个称为NodeInfo的数据结构中, 也采用了无锁的原子数据以低成本保证线程安全.

这里耗时反应了节点线程池的繁忙程度和节点之间的通讯延迟时间, 因为一方面任务数量是采用原子类型, 在有数据修改时都会将其它修改或者读取请求延后, 那么当任务池繁忙时, 往往会导致获取任务数的耗时增加; 另一方面由于各节点之间的通讯是通过hpx的分布式通讯机制进行的, 当进行“get task count”分布式操作时, 中间的耗时也必然反应了节点间的通讯延迟情况. 后续通过这两项数据也会对最优窃取目标的计算有重要影响.

由于获取任务数耗时是一项时间数据, 可能具有较大的数值与波动, 所以需要处理后再存储, 处理公式为

$$\begin{aligned} \text{averageDelayTime}_i = & \ln(2.72 + \text{delayTime} \times \text{worker_count}) \times 0.65 \\ & + \text{nodeInfoVector}[i] \rightarrow \text{averageDelayTime} \times 0.35 \end{aligned} \quad (3.2)$$

其中*i*为节点编号, delayTime为获取任务数的耗时, worker_count为节点worker数量, 公式首先通过将延迟时间乘以本地的worker数量来反应综合节点的通讯耗时, 再同时参考eq. (3.1)也通过对数函数与指数平滑来降低数据的敏感度, 并使数据更加平滑和更具参考价值

3.3 最优窃取目标计算与缓存

在Performance Monitor通过性能收集机制获得了包括各节点负载情况, 各节点剩余任务量, 各节点获取任务池任务的耗时等性能数据后, Performance Monitor会通过各节点的数据计算出最优窃取目标节点, 并将其缓存起来以供后续Performance Policy获取. 其实现如Figure 3.6所示.

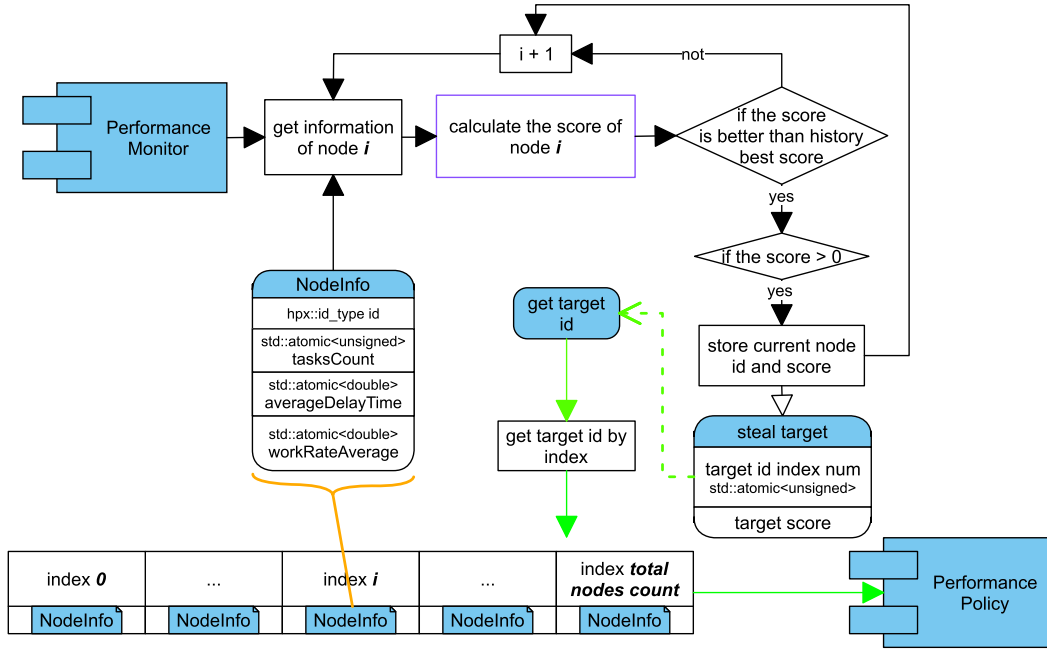


图 3.6: refresh the optimal steal target

这里整体流程也采用了无锁化的设计,通过的互相独立的节点结构和原子数据以低时间成本保证线程安全.

其中首先通过遍历各节点的NodeInfo数据结构中性能数据,来计算各节点的steal-worthiness score, 得分越高则说明越值得窃取, 其计算公式为

$$\begin{aligned} \text{score}_i = & \max(\text{nodeInfoVector}[i] \rightarrow \text{workRateAverage}, 0.0001) \\ & \times \text{nodeInfoVector}[i] \rightarrow \text{tasksCount} \\ & - \text{nodeInfoVector}[i] \rightarrow \text{averageDelayTime} \end{aligned} \quad (3.3)$$

公式的核心思想是尽量减少预计耗时最长的节点的任务池任务数的同时一定程度减少浪费在窃取其任务上的时间, 所以先通过 $\max(\text{nodeInfoVector}[i] \rightarrow \text{workRateAverage}, 0.0001)$ 来避免过小的无意义数据对计算的干扰, 之后通过 $\times \text{nodeInfoVector}[i] \rightarrow \text{tasksCount}$ 来预估该节点执行完任务的大致剩余时间, 最后通过 $- \text{nodeInfoVector}[i] \rightarrow \text{averageDelayTime}$ 来减去一定的获取任务预估的损耗时间. 最终得到的结果越大则一定程度上说明该节点越值得窃取.

在遍历完所有节点后,一般会得到一个最优的节点目标编号索引, 会将这个最优的节点目标编号索引数进行缓存, 当Performance Policy需要获取最优窃取目标的id.type时, 会通过提供的查询接口, 通过索引再去节点数组里获取最优窃取目标节点的id.type, 这样同时也保障了在hpx的id.type类型无原子操作的情况下仍能保持无锁机制.

在判断最优窃取目标时,我们也需要判断窃取的成本是否高过了收益, 而这种情况往往发生在窃取过程的消耗时间大于了帮助目标节点完成任务所节省的时间, 这种情况下score往往会小于或者等于0, 所以设置了额外的判断条件, 在score小于或者等于0的情况下不会缓

存最优窃取目标.

3.4 Refresh data and provide tasks mechanism

在建立了一套从数据收集到计算最优窃取目标和缓存最优窃取目标的机制后, 我们还需要的问题,就是以何种频率去刷新各项数据. 虽然每一次刷新数据的开销都是相对很小的,但是YewPar的worker数量会尽量占满本地的计算资源, 那么当worker繁忙时,过于频繁的刷新会导致worker的计算资源被占用, 从而一定程度影响YewPar的性能. 但是如果刷新的频率过低,则会导致数据的可靠性降低, 因为数据的实时性降低, 提供给其它节点的数据和缓存的其它节点数据都将不够准确, 甚至可能导致窃取到其它任务饥饿的节点,加剧不均衡的情况.

为解决这一问题,本文设计了一种双向动态刷新的刷新机制, 它由两种刷新方式组成, 一种是自动动态刷新任务方式,另一种是通过Performance Policy进行辅助刷新方式. 大致流程如Figure 3.7所示.

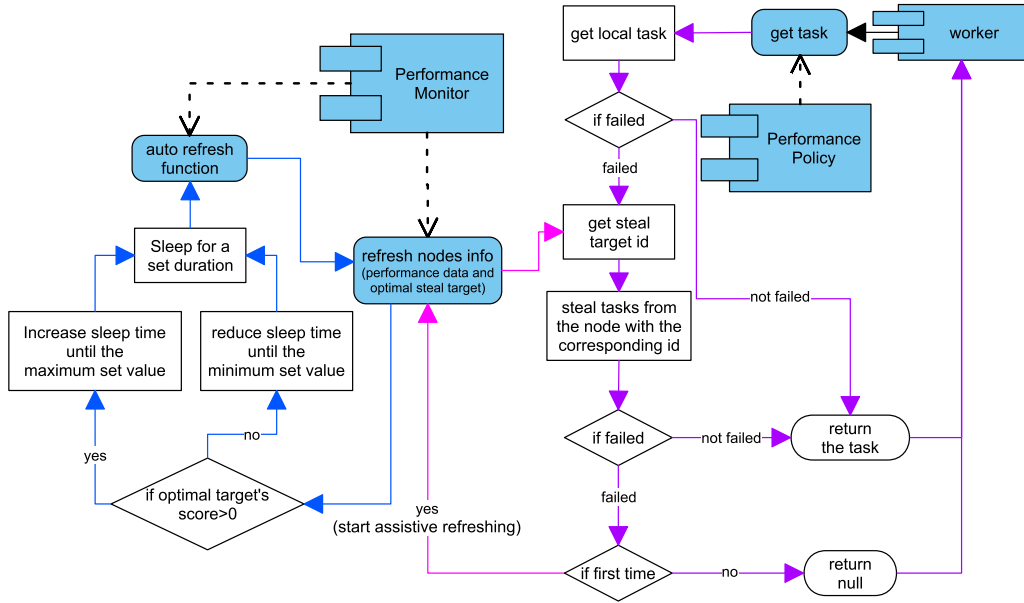


图 3.7: refresh data and provide tasks mechanism

在图的左侧,是在Performance Monitor中部署的自动动态刷新任务, 它的本质上是一个循环,在每次刷新数据后都会根据刷新结果动态调整下一次刷新的时间, 如果刷新数据的结果不理想, 则说明当前很可能仍有worker处在饥饿的状态, 则会大幅缩短下一次刷新的时间; 但如果刷新数据的结果符合预期, 则说明workers很可能都会成功获取到任务进行执行, 所以会逐渐增加下一次刷新的时间, 避免频繁的刷新影响worker的计算性能. 而刷新数据的结果是否理想目前主要通过score进行判断, 目前的算法是如果score小于等于0则说明不理想, 因为根据eq. (3.3)大概率是任务池任务数过少或者获取任务耗时过长导致的, 这种情况下,很可能会导致窃取的成本高于收益, 比如窃取时目标节点已无任务可窃取, 或者窃取的通讯延迟时间和本地执行时间相加不如远程节点自己执行完剩余任务的时间短等. 这里需要说明的是,虽然动态调整时间在实践中收益明显, 但是仍需要设置上下限, 因

为如果不设置上限很可能导致由于落后的数据导致缓存了的错误窃取目标, 而如果不设置下限则会导致频繁的刷新影响了其它仍有任务执行的worker的性能或者影响系统底层的调度, 而如何界定这个上下限的值通过后续的多次实验, 发现在一定范围内既能达到对系统性能影响较低, 又能保障数据的相对实时性, 而之前的通过加入了指数平滑等算法设计, 通过一定程度保留了历史数据的影响, 导致了数据即使有一定的延迟也能较为准确的反应各节点的综合情况.

而上述设计仍有不足之处与优化的空间, 所以还设计了如Figure 3.7的右侧所示的一种通过Performance Policy进行辅助刷新的机制, Performance Policy的本质是获取任务的一个策略, 它会优先获取本地的任务进行执行, 因为如果本地有任务的话会让worker更快地获取到任务并执行, 避免计算资源的浪费, 但是如果本地获取不到任务, 则会优先对Performance Monitor的最优窃取目标的id缓存进行查询, 通过获取的id对远程对应节点的任务池进行窃取, 这样大概率会窃取到有效的任务返回给worker进行执行, 从而尽可能避免worker的饥饿状态, 但是当各节点任务剩余数量都很少等情况下, 有小概率会无法窃取到任务, 这种在初期刚开始逐步生成任务, 或者在各节点都快执行完任务且很少有新任务加入时的情况下会出现的较多. 而自动刷新机制有可能此时在休眠状态无法及时调整刷新时间, 此时被worker调用的Performance Policy就会担任辅助刷新的作用, 主动去同步执行刷新数据的任务, 因为一方面不刷新数据则大概率这个worker还是会处于饥饿状态, 另一方面刷新数据时会占用这个worker的计算资源, 而不会去影响其它worker, 但是如果获取到有效的窃取目标则会给自己和其它worker都带来收益, 当刷新数据完成后会再次尝试根据新的目标窃取一次任务, 如果还是失败则大概率说明当前各节点都很少有任务, 为了避免通讯等资源的占用, 此时会返回空数据让这个worker进入休眠周期避免资源的占用.

而在实际实践中也发现两套机制互相配合能够很好地提升系统的性能.

Chapter 4: Evaluation

4.1 Search Applications

我们对YewPar原有的两种搜索类型的代表性搜索应用和代表性样本上评估改进workstealing策略后的性能, 如下所示.

- **Enumeration:** 非平衡树搜索(UTS)根据给定的分支因子、深度和随机种子动态地构建合成的不规则树工作负载[12]. 数字半群(NS)计算具有特定属数的数字半群有多少[8]. 数字半群 S 是一个包含0且在加法下封闭的自然数的余集; S 的属数是其补数的大小.
- **Optimisation:** 最大团(MaxClique)找到给定图中的最大团, 即最大的两两相邻顶点的集合. 0/1背包问题确定将每个具有利润和重量的物品放入容器中的最佳组合, 使得在给定的重量限制下利润最大化. 旅行商问题(TSP)找到 N 个城市的最短循环之旅.

为了控制变量, 评估使用的应用程序maxclique-16和NS-hivert是YewPar的hpx1.8分支自带的程序, 并未对其进行改动, 但运行时窃取策略分别使用了改动后的Performance-Driven Workstealing Policy 和原版的hpx1.8分支的DepthPool Policy以进行对比. 其中MaxClique[11]、NS[8]的基线实现使用了公开的先进算法. 这些顺序的C++实现是由领域专家提供的. 关于应用程序和实例的完整描述在[1]中.

对maxclique-16应用程序使用了brock800_2.clq这种较大的数据以延长运行时间减小对比误差, 和Depth-Bounded skeleton这种较适合maxclique-16应用并能应用Performance-Driven Workstealing Policy的skeleton, 并设置了参数 $d = 2$ (spawn-depth: Depth in the tree to spawn at), 因为在实践中发现, 当 d 设置为2时, 相对其它参数具有更优的性能和稳定性.

对NS-hivert使用了Budget skeleton这种较适合NS-hivert应用并能应用Performance-Driven Workstealing Policy的skeleton, 并设置了参数 $b = 10^6$ (backtrack-budget: Number of backtracks before spawning work) 和 $g = 47$ (genus: Depth in the tree to count until)以保障结果准确的情况下增加工作量来延长运行时间以便减小统计误差.

4.2 Experimental Setup

我们在多达20台机器上对程序运行情况进行了测量, 每台机器都配备了双8核的Intel Xeon E5-2640v2 2GHz CPU(无超线程), 64GB RAM, 运行Ubuntu 22.04.2 LTS系统. 每台机器在测试前负载均在1%之内, 我们为HPX(版本1.8)和原版的hpx1.8分支的YewPar一样预留了一个核心用于任务管理, 即在16个核心上我们使用15个工作线程. 需要注意的是, 由于

修剪导致的非确定性、寻找替代有效解决方案和原版的YewPar的工作窃取方案具有较高的随机性, 并行搜索的性能分析是非常困难的, 这可能导致性能异常2.1.1, 表现为超线性加速/减速. 为了控制这些因素, 我们对多种不同的节点数量下多次运行每个程序(设置为10次), 报告累计统计数据.

4.3 Isolated Performance Evaluation of Different Policies

为了准确评估基于Performance-Driven Workstealing Policy的YewPar与原版在空闲环境下的性能差异, 我们在各节点均处于空闲状态时, 对两种应用进行了独立的测试评估. 为了减小测试误差, 我们选择交替运行改动版和原版的应用, 即在选定一种应用程序后, 每次运行完基于改动后的Policy框架的程序后, 立即运行基于原版的Policy的程序, 然后再次运行基于改动后的Policy框架的程序, 如此重复, 直到完成所有测试轮数. 在这些测试中, 运行的程序可以独占所有计算资源.

4.3.1 Isolated Performance Evaluation of Maxclique

对于基于Depth-Bounded skeleton的Maxclique程序的每组运行时间测试各进行十次取平均值后的结果如Figure 4.1所示, 其中分为运行时间评估与基于Performance-Driven Workstealing Policy比原来的DepthPool Workstealing Policy的运行速度提升评估两个部分.

速度提升部分的计算公式为

$$\text{Speedup Percentage} = \left(\frac{T_{\text{DepthPool}} - T_{\text{Performance-Driven}}}{T_{\text{Performance-Driven}}} \right) \times 100\% \quad (4.1)$$

其中, $T_{\text{DepthPool}}$ 是基于原版YewPar的DepthPool Workstealing Policy的应用的运行时间, 而 $T_{\text{Performance-Driven}}$ 是基于Performance-Driven Workstealing Policy的应用的运行时间.

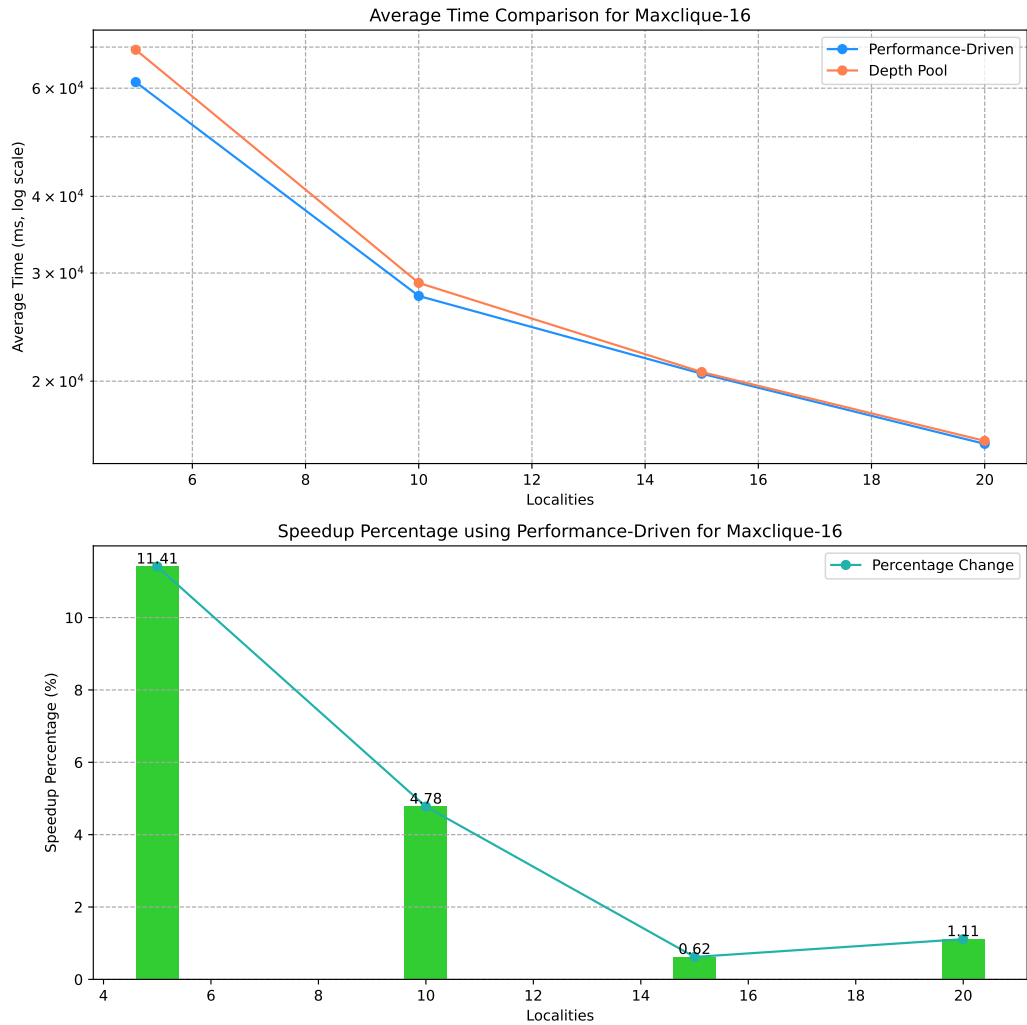


图 4.1: isolated time comparison for Maxclique-16

从图中可以看出在单独运行下,对于不同的节点数量,基于Performance-Driven Workstealing Policy的YewPar都能比原版的DepthPool Workstealing Policy的YewPar有更好的性能,能以更短的时间完成所有任务.

但是从图中也可以看出,在节点数量从5个到15个的区间中,速度的提升从最高的5个节点时的11.41%降低至最低的15个节点时的0.62%,之后到20节点数量时又有所提升,到了1.11%.而理论上节点数量越多,能够优化窃取目标的空间越大,所以速度的提升应该是越来越明显的,所以我们对这一现象进行了分析.

- 对于Performance-Driven Workstealing Policy分析后发现,次要原因包含了由于节点的增加导致了系统总体由于数据收集和计算的开销增加等,但是理论上开销的增加相比对于优化窃取目标带来的提升是微不足道的.
- 从应用实现进行研究和测试的具体数据上分析后,发现应该主要是由于性能异常2.1.1的原因,基于Depth-Bounded skeleton的Maxclique程序相比其它应用具有更高的超线性减速的概率,为了验证这一推测,除了对YewPar代码进行分析外,我们还对基于Depth-

Bounded skeleton的Maxclique程序进行了运行时间的波动情况的分析, 以节点数量为10时的数据为例, 其运行时间的波动情况如Figure 4.2所示, 在评估Maxclique-16应

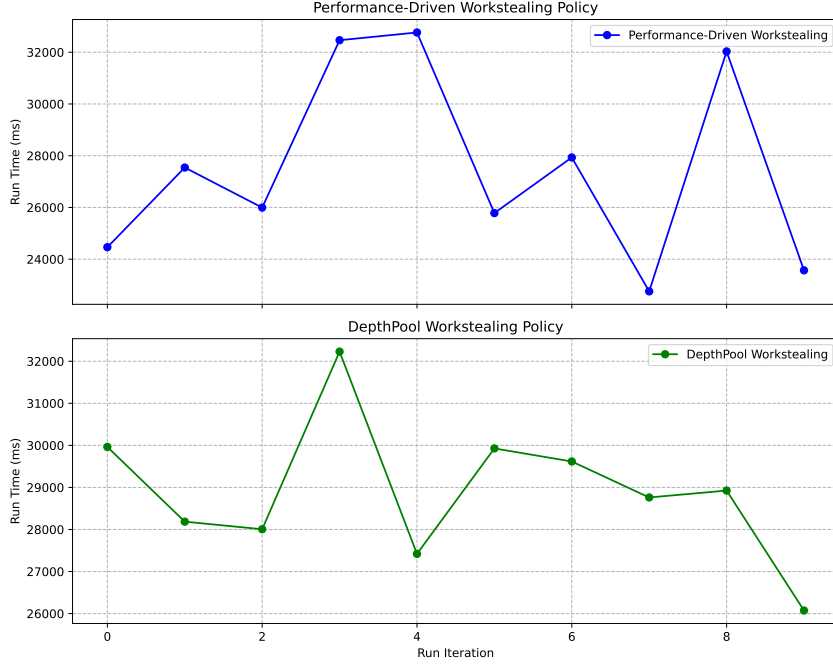


图 4.2: Runtime fluctuations for maxclique-16 application based on two different policies using 10 localities.

用的运行时间波动性时, 我们计算了最大值与最小值之间的差异百分比. 这可以通过以下公式得出:

$$\text{Variability Percentage} = \left(\frac{\text{Max Value} - \text{Min Value}}{\text{Min Value}} \right) \times 100\% \quad (4.2)$$

对于基于Performance-Driven Workstealing Policy的maxclique-16应用, 最大差异百分比为43.98%。而对于基于DepthPool Workstealing Policy的应用, 最大差异百分比为23.61%。不难看出, 基于Depth-Bounded skeleton的Maxclique程序的运行时间已经具有很高的波动性, 而基于Performance-Driven Workstealing Policy的Maxclique程序的运行时间波动性相比前者更高, 几乎是其波动性的两倍. 再结合性能异常2.1.1的内容, 不难发现这里Maxclique程序对于额外的计算资源可能会更多的破坏搜索启发式方法, 造成系统总体增加了更多的额外任务量.

而基于Performance-Driven Workstealing Policy的maxclique-16应用由于对于窃取目标的变化更为频繁, 由于YewPar窃取目标时的设计是从任务池右侧进行窃取, 频繁的变换要窃取的任务池导致窃取的任务不连续, 从而更多的导致在没有完整的左侧信息的情况下推测性地搜索子树, 生成了更多的额外任务拖慢了程序的运行速度. 而且更多节点时越容易导致窃取目标的变动, 所以带来的速度提升会相对降低.

由于这些数据和性能异常中描述的情况相符, 从而验证了我们的推测.

而虽然窃取目标的变动会导致额外的任务生成, 但是我们注意到Figure 4.1在15节点到20个节点数量的区间中, 运行速度还是相对有了提升, 应该这是由于优化窃取目标带来的收益相比于额外任务带来的损耗逐渐变大导致的.

4.3.2 Isolated Performance Evaluation of NS-hivert

对于基于Budget skeleton的NS-hivert程序的每组运行时间测试各进行十次取平均值后的结果如图Figure 4.3所示, 也分为运行时间评估与基于Performance-Driven Workstealing Policy比原来的DepthPool Workstealing Policy的运行速度提升评估两个部分.

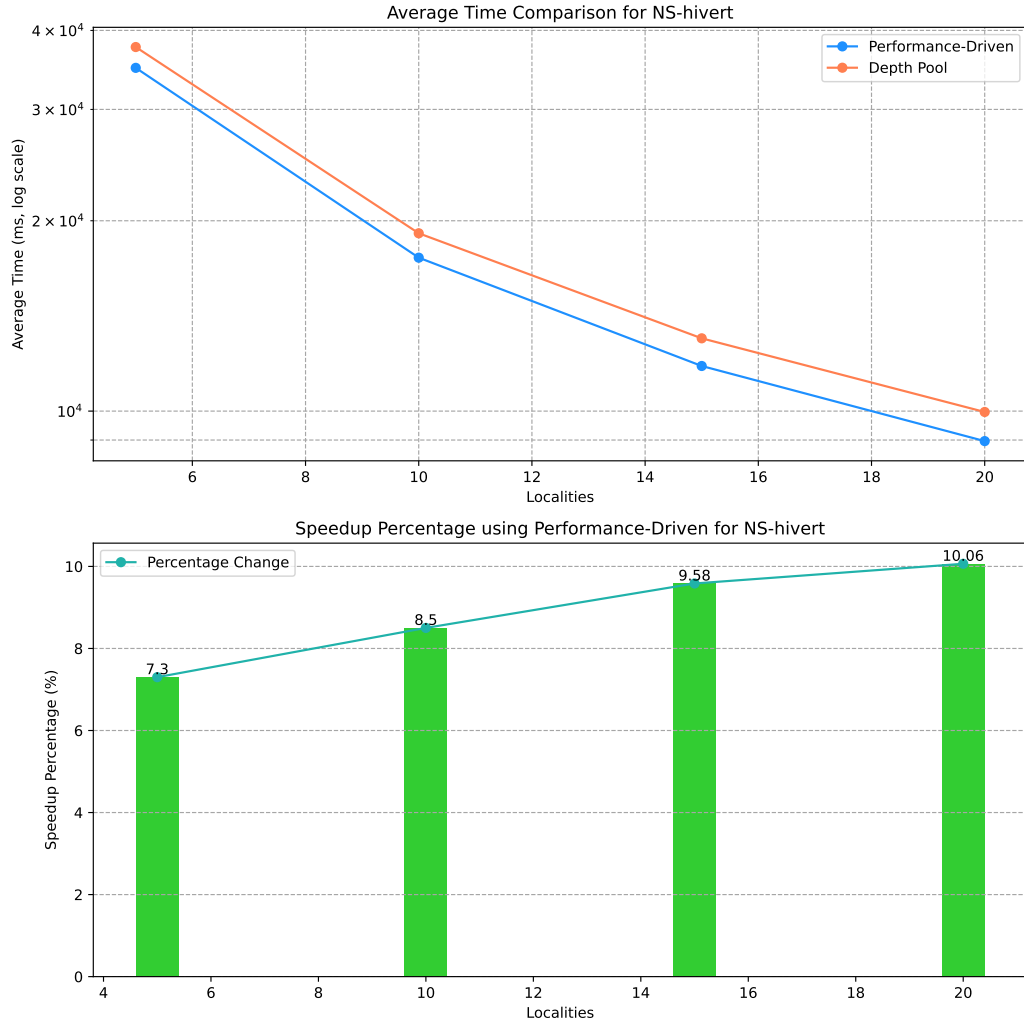


图 4.3: isolated time comparison for NS-hivert

可以看出在单独运行的情况下, 基于Performance-Driven Workstealing Policy的NS-hivert程序在不同节点数量都具有更优的性能, 能以更短的时间完成所有任务. 其中速度的提升随着节点数量的增加而增加, 在节点数量为20时,速度的提升达到了平均10.06%, 考虑到这只是通过调整Workstealing Policy达到的整体速度提升, 并没有对YewPar其它部分进行改动, 并且有性能异常2.1.1等其它因素产生的负面影响, 所以这一结果已经是非常优秀的.

相比于Maxclique程序, 这里节点数量增多时使用Performance-Driven Workstealing Policy带来的速度提升更为明显, 这是由于NS-hivert程序运行时性能异常2.1.1产生的负面影响较小, 同样我们对NS-hivert程序的运行时间波动性进行了分析, 如图Figure 4.4所示.

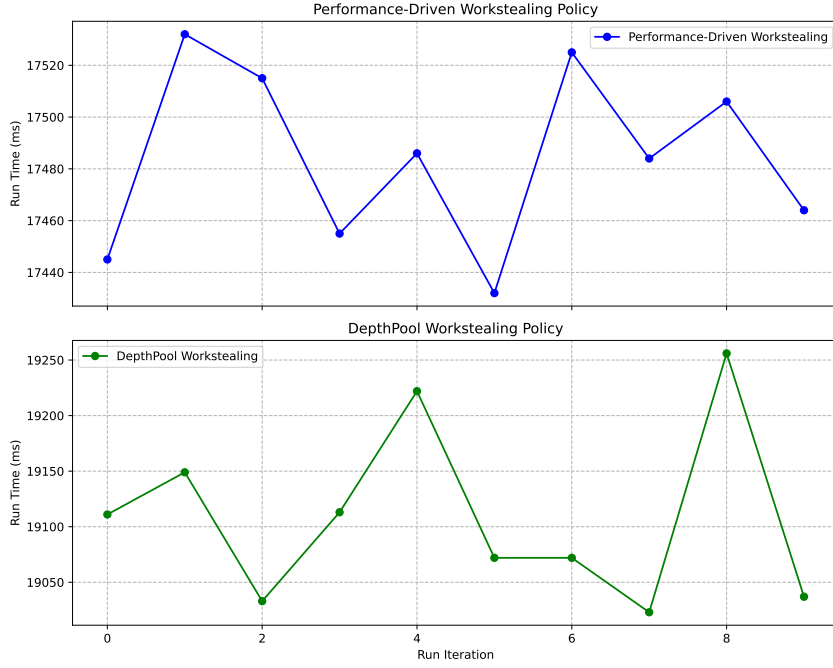


图 4.4: Runtime fluctuations for maxclique-16 application based on two different policies using 10 localities.

通过使用eq. (4.2)进行最大差异百分比计算, 对于基于Performance-Driven Workstealing Policy的maxclique-16应用, 最大差异百分比为0.57%。而对于基于DepthPool Workstealing Policy的应用, 最大差异百分比为1.22%。这相对于Maxclique程序的运行时间波动性是非常小的, 所以推测出NS-hivert程序运行时变更任务执行顺序产生的超线性减速概率较小. 这时速度提升的主要原因应该这是由于优化窃取目标带来的收益导致的。

4.4 Simultaneous Performance Evaluation under Resource Contention

这里主要评估在各节点的计算资源不均衡,有争用的情况下, 基于新的Performance-Driven Workstealing Policy的YewPar与原版在性能上的差异. 因为实际部署环境可能很复杂, 不同节点有不同的计算资源占用情况, 通过对比在资源争用的情况下Performance-Driven Workstealing Policy的表现, 能够更全面地评估其性能。

为了模拟不同节点的计算资源占用情况, 我们使用了一种简单的方法, 通过直接同时运行基于不同Policy的YewPar程序, 让两种程序在同一时间内同时运行, 由于两种程序设置的各节点使用的线程数均和各节点的所拥有的核心数相同, 从而产生较为严重的资源争用, 而由于原版的DepthPool Workstealing Policy所采用的策略是偏随机窃取的, 所以也能够较好地模拟负载不均衡的情况下的资源争用。

但是这样会产生一个问题, 就是当其中一个程序运行完毕后, 剩下的程序会重新独占所有的计算资源, 所以下面评估结果中Performance-Driven Workstealing Policy所产生的性能提升会相对实际情况有所低估, 但是这样的评估结果仍然能够反映出两种Policy在各节点的计算资源不均衡和争用的情况下的性能差异。

4.4.1 Simultaneous Performance Evaluation of Maxclique

对于基于Depth-Bounded skeleton的Maxclique程序的每组同时运行情况下的运行时间测试,各进行十次取平均值后的结果如Figure 4.5所示,其中也分为运行时间评估与基于Performance-Driven Workstealing Policy比原来的DepthPool Workstealing Policy的运行速度提升评估两个部分.

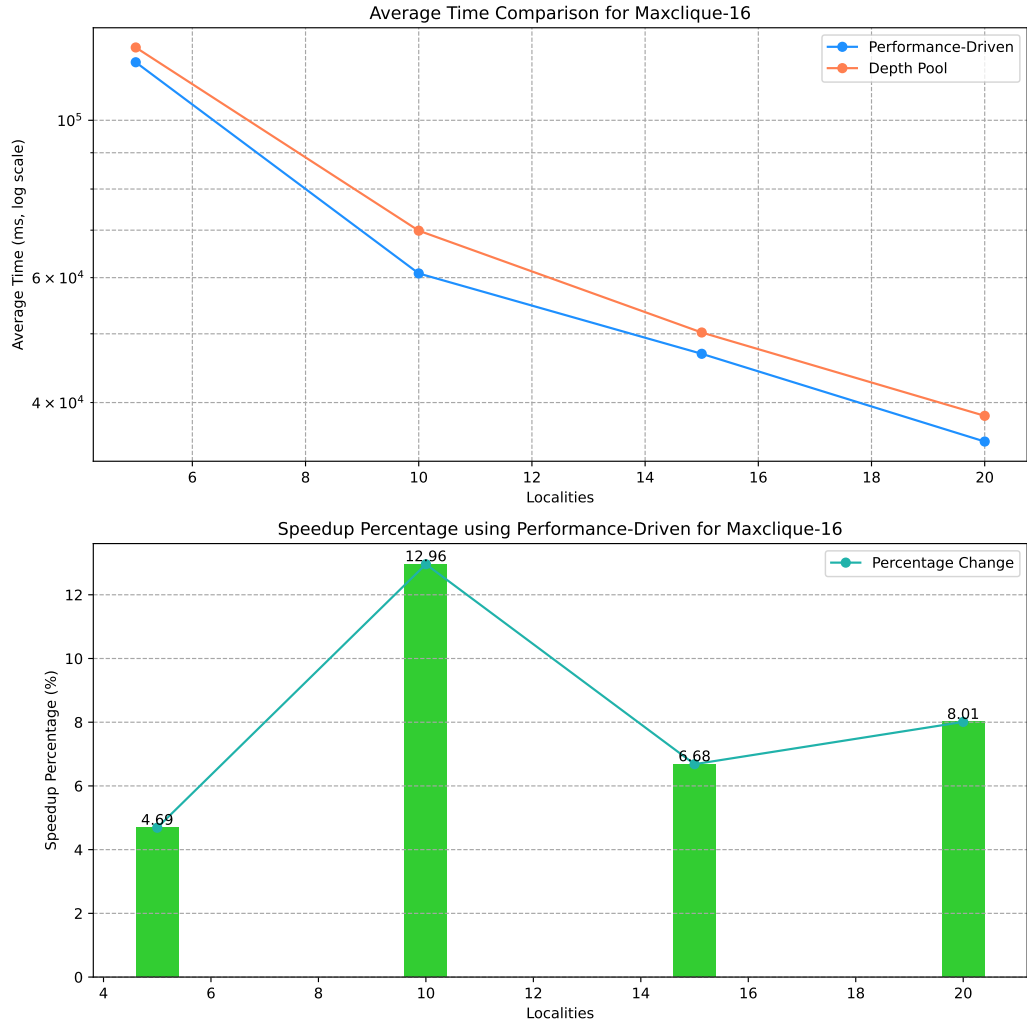


图 4.5: simultaneous time comparison for Maxclique-16

可以看出Maxclique虽然有较为严重的超线性减速的情况,但是在各节点负载不均衡的情况下,基于Performance-Driven Workstealing Policy的Maxclique应用的运行速度优势有了明显的提升.例如在节点数量为20时,速度提升比例从单独运行时的1.11%提升到了同时运行时的8.01%,提升了接近8倍.推测这是由于在负载不均衡的情况下,优化窃取目标带来的优势明显大过了额外任务带来的损耗.同时从图中可以看出,在节点数从10到15时,额外任务的损耗降低了一些运行速度的提升,但是随着节点数的增多,优化窃取目标带来的优势又开始逐渐显现.

4.4.2 Simultaneous Performance Evaluation of NS-hivert

对于基于Budget skeleton的NS-hivert程序的每组同时运行情况下的运行时间测试,各进行十次取平均值后的结果如Figure 4.6所示,其中也分为运行时间评估与基于Performance-Driven Workstealing Policy比原来的DepthPool Workstealing Policy的运行速度提升评估两个部分.

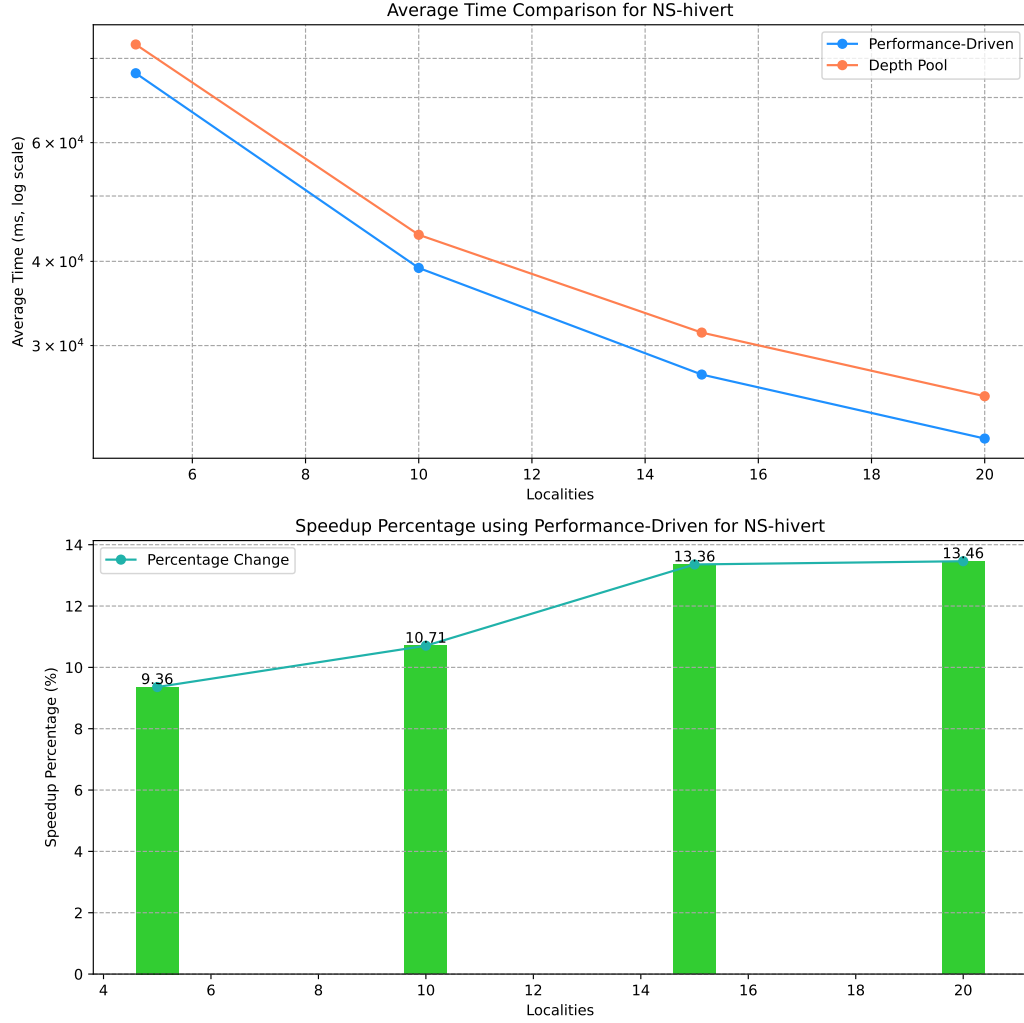


图 4.6: simultaneous time comparison for NS-hivert

从图中可以看出在基于Budget skeleton的NS-hivert程序中, Performance-Driven Workstealing Policy对于性能的提升更加明显,从5个节点数时的9.36%逐步提升至20个节点时的13.46%,考虑到Workstealing Policy相对于整个YewPar框架和NS-hivert程序的优化空间很小,所以达到13.46%的性能提升是非常优秀的.

从图中节点数从15到20时的很小的提升幅度可以推测,此时的性能提升比例已经接近于Workstealing部分可提升的极限,其它时间大多消耗在处理任务和YewPar框架的其它开销上.

Chapter 5: Conclusion

Main conclusions of your project. Here you should also include suggestions for future work.

5.1 future work

附录 A: First appendix

A.1 Section of first appendix

附录 B: Second appendix

参考文献

- [1] Blair Archibald. *Skeletons for Exact Combinatorial Search at Scale*. PhD thesis, University of Glasgow, 2018.
- [2] Blair Archibald, Patrick Maier, Ciaran McCreesh, Robert Stewart, and Phil Trinder. Replicable parallel branch and bound search. *Journal of Parallel and Distributed Computing*, 113:92–114, 2018.
- [3] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. Implementing yewpar: A framework for parallel tree search. In *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings*, pages 184–196, Berlin, Heidelberg, 2019. Springer-Verlag.
- [4] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. Yewpar: Skeletons for exact combinatorial search. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’20*, pages 292–307, New York, NY, USA, 2020. Association for Computing Machinery.
- [5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.
- [6] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA ’07*, pages 105–115, New York, NY, USA, 2007. Association for Computing Machinery.
- [7] A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens. Asynchronous parallel branch and bound and anomalies. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, pages 363–377, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [8] Jean Fromentin and Florent Hivert. Exploring the tree of numerical semigroups. *Math. Comp.*, 85(301):2553–2568, 2016.
- [9] Everette S. Gardner. Exponential smoothing: The state of the art—part ii. *International Journal of Forecasting*, 22(4):637–666, 2006.
- [10] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS ’14*, New York, NY, USA, 2014. Association for Computing Machinery.

- [11] Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013.
- [12] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 235–250. Springer, 2006.
- [13] Hrushit Parikh, Vinit Deodhar, Ada Gavrilovska, and Santosh Pande. Efficient distributed workstealing via matchmaking. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’16, New York, NY, USA, 2016. Association for Computing Machinery.