第32卷 第23期
Vol.32 No.23
计 算 机 工 程
Computer Engineering
2006年12月
December 2006
·人工智能及识别技术· 文章编号：1000—3428(2006)23—0193—03 文献标识码：A 中图分类号：TP301.6

基于混合负载平衡的并行启发式搜索算法

袁 源[1]，李炳法[1]，杨 杰[2]，丁 莹[1]，彭代毅[1]
(1. 四川大学计算机学院，成都 610065；2. 代尔夫特理工大学计算机系，荷兰)

摘 要： 在分析了迭代加深启发式搜索(Iterative Deepening A*)算法及其可并行性后，提出了一种新的基于混合负载平衡的并行迭代加深启发式搜索算法。该算法综合了静态负载平衡和动态负载平衡的优点，可以在多结点的并行搜索计算中获得很高的加速比和效率。给出了该算法的Java RMI实现。通过在72个结点的并行机上的试验表明，该算法可以极大地提高并行搜索算法的加速度和效率。

关键词： 负载平衡；迭代加深；并行；搜索算法

Parallel A* Searching Algorithm Based on Mixed Load Balancing

YUAN Yuan[1], LI Bingfa[1], YANG Jie[2], DING Ying[1], PENG Daiyi[1]
(1. Dept. of Computer Science, Sichuan University, Chengdu 610065;
2. Dept. of Computer Science, Delft University of Technology, The Netherlands)

【Abstract】 This paper presents a new parallel iterative deepening A* searching algorithm by discussing iterative deepening A* algorithm and its parallel properties. This algorithm has the advantages in both static and dynamic load balancing, and achieves very high speedup and efficiency. It shows how to realize Java RMI with this algorithm. The experimentations on 72-node parallel machine indicate that this algorithm can improve the speedup and efficiency of parallel searching algorithm heavily.

【Key words】 Load balancing; Iterative deepening; Parallel; Searching algorithm

搜索是计算机运算中的一个基本问题，尼尔逊曾把它列为人工智能研究中的4个核心之一[1]，在人工智能领域有着广泛应用。怎样才能加快搜索的速度和效率，一直是计算机界科学家们所重点研究的问题。

在各种搜索算法中，迭代加深启发式搜索(Iterative Deepening A*，IDA*)是一种加速比和效率都很好的算法。IDA*是A*算法的一个变形，很好地综合了A*算法的人工智能性和回溯法对空间的消耗较少的优点，在一些规模很大的搜索问题中会起到很好的效果。

然而单个处理机的计算能力毕竟是有限的，它无法满足例如科学和工程问题中数值建模和模拟这些需要高计算速度的领域。而多处理机的优点正体现在此，它的使用常常可以使一个很大的问题能在合理的时间内完成。但是如何设计高效的并行算法，则成了解决这些问题的关键。

本文分析了IDA*算法的可并行性，给出了并行IDA*算法的程序结构。为了加快并行IDA*程序的运行速度，本文给出了一种综合静态负载平衡和动态负载平衡的算法，描述了该算法的Java RMI实现，并在72个结点的并行机上进行试验，通过试验结果证明，该算法可以极大地提高并行搜索算法的速度和效率。

1 IDA*及其可并行性分析

1.1 A、A*和IDA*

当搜索的状态空间十分大，且在不预测的情况下，广度和深度优先搜索的效率实在太低，甚至不可完成。这时就要用到启发式搜索。启发式搜索就是在状态空间中的搜索，对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。

在启发式搜索中，估价函数是十分重要的。估价函数的一般形式是

$$f(n)=g(n)+h(n) \tag{1}$$

其中$f(n)$是节点的估价函数，$g(n)$是在状态空间中从初始节点到n节点的实际代价，$h(n)$是从n到目标节点最佳路径的评估代价。在这里$h(n)$主要是体现了搜索的启发信息，$g(n)$则代表了搜索的广度优先趋势。

一般的启发式搜索算法，记为A。其基本特点是任意寻找并设计一个与问题有关的$h(n)$，并通过式(1)构造$f(n)$，然后以$f(n)$的大小来排列待扩展状态的次序，每次选择$f(n)$值最小者进行扩展。

A*算法就是对估价函数加上一些限制后得到的一种启发式搜索算法。A*算法的估价函数可表示为

$$f'(n)=g'(n)+h'(n) \tag{2}$$

把式(1)中的$f(n)$与式(2)中的$f'(n)$相比，$g(n)$是对$g'(n)$的一个估计，$h(n)$是对$h'(n)$的一个估计。A*算法是对$g(n)$和$h(n)$采取以下限制的算法：

$$\left\{\begin{array}{l}g(n)\text{是对}g'(n)\text{的估计，且}g(n)>0\\ h(n)\text{是对}h'(n)\text{的下界，即对任意结点}n\text{均有}h(n)\le h'(n)\\ \text{估价函数}f(n)\text{是一个递增的函数}\end{array}\right\} \tag{3}$$

IDA*算法是A*算法的一个变形。IDA*的基本思路是：首先将初始状态结点的$f(n)$值设为阈值bound，然后进行深度优先搜索，搜索过程中忽略所有$f(n)$值大于bound的结点；如果没有找到解，则加大阈值bound，再重复上述搜索，直到找到一个解。在保证$f(n)$值的计算满足A*算法的要求下，可以证明找到的这个解一定是最优解。在程序实现上，IDA*

作者简介： 袁 源(1982－)，男，硕士生，主研方向：并行算法，分布式系统设计等；李炳法，教授；杨 杰，研究助理；丁 莹、彭代毅，硕士生

收稿日期： 2005-11-17 **E-mail：** minixer@163.com

要比 A* 方便，因为不需要保存结点，不需要判重复，也不需要根据 $f(n)$值对结点排序，占用空间小。

1.2 IDA*算法的可并行性分析

和其它的搜索算法一样，IDA*也是一个 n 叉树的搜索。并且因为 $f(n)$是一个递增的函数，即使在搜索树的某个分支没有最优解，或者根本没有解的情况下，这分支的 $f(n)$也在不断增大。由于阈值 bound 的存在，随着搜索深度的加深，一旦扩展后的某个结点的 $f(n)$大于 bound，便不再向下扩展。所以说 IDA*搜索的深度是有限制的，即使一个分支的结点不知道另一个分支的任何情况，也是可以单独扩展下去的。

这样就决定了 IDA*可以被很好地并行化。一般的并行方法是给每个分支分配一个处理机，最后再对从各个处理机上收集到的结果进行处理。

2 IDA*算法的静态负载平衡算法

由于 IDA*算法的特殊性，当使用简单的处理器分配时，因每个分支上的搜索计算量是不一样的，就导致了一些处理器运行得比其它处理器快，使一些处理器比其它处理器先完成任务而变得空闲。这样就会浪费计算器的计算能力，并使整个计算速度降低。

使用普通的并行 IDA*算法，将一个 15 数码问题划分为 16 个搜索任务，交给 16 个处理器计算，每个处理器的运行时间如图 1 所示。

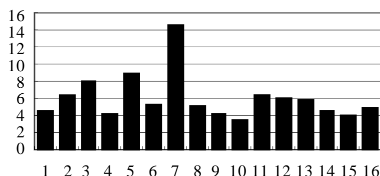


图 1 普通并行 IDA*算法中各处理器的运行时间

由图 1 可以看出，不同处理器的运行时间差别很大。在一些处理器还在计算的时候，有可能很多处理器已经处于空闲状态，这样就浪费了计算器的计算能力，并使最后的计算时间变得很大。为了解决这样的问题，需要负载平衡技术。负载平衡包括静态负载平衡和动态负载平衡。

静态负载平衡是指在任何进程执行之前的负载平衡技术。具体地说，就是在各处理器开始运算之前把需要解决的问题划分为许多任务(Job)，然后给每个处理器分配不同数目的任务去执行。这样每个处理器得到的任务将会平均一些，可以较好地避免图 1 情况的发生。

负载平衡技术一般采用任务池的方法。任务池技术是指有一个主进程维护一个任务队列。当其它的从进程空闲时，主进程就负责把任务发送给从进程并接收从进程的计算结果。图 2 给出了任务池的模型。

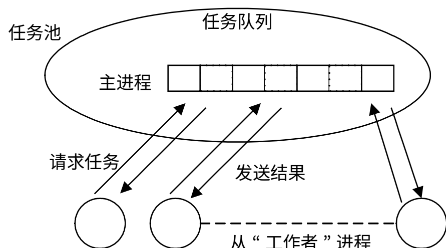


图 2 任务池模型

而初始任务队列的生成，成为静态负载平衡技术的关键。其中有两个关键要素：任务的数目和任务的粒度。

因为从进程在向主进程请求任务并接收任务时，需要网络传输的时间和与其它的从进程同步，如果任务数目过多，这将消耗大量时间，导致运算效率的降低。所以任务的总数目不能太多。相反，如果任务的数目太少，每个任务的粒度就会太大，各个任务的运算时间就会不平衡，这将导致各个处理器之间运算时间的不平衡，出现如图 1 的情况：

对于任务的分配，常用的算法是在确定了分配任务的数量后，将初始任务按照一定深度划分任务进行分配。但是 IDA*算法有它的特殊性，因为它的估价函数 $f(n)$值可以近似代表任务的规模。如果仅仅按照深度划分的分配方式，会使划分出的任务在规模，也就是粒度方面造成不一致。但如果仅仅按照估价函数 $f(n)$的值进行分配，可能在估价函数不变时，结点的深度会一直增加，这样会划分出粒度很小的任务。而这些粒度小的任务可能所需的计算时间比它在网络中的传送时间还小，这样就会极大地降低程序的效率。

综合按结点的估价函数 $f(n)$值和深度的划分任务的方法，本文给出了一种较好的 IDA*初始任务划分算法(图 3)。

```
JobExpand(Job job)                    // 函数首先传入的初始的job
{   扩展 job 得到儿子结点数组 jobs[n];
    FOR(jobs[n]中的每一个结点 jobs[i])
{IF jobs[i]中的估计函数值 > jobs[i]中的阈值 bound
// 不满足扩展条件
    CONTINUE;
ELSE IF jobs[i]中的阈值 bound  > 某个设定的阈值 initBound
    将 jobs[i]加入到任务池中;
ELSE IF jobs[i]中的深度 depth  > 某个设定的最大深度 maxDepth
    将 jobs[i]加入到任务池中;
ELSE
JobExpand(jobs[i]);                   // 继续扩展任务
  } // end FOR
} // end JobExpand
```

图 3 IDA*初始任务的划分算法

3 并行 IDA*的混合负载平衡算法

静态负载平衡可以很大提高并行计算的效率。但是因为没有实际执行程序的各个分段，很难准确估计程序各个部分的执行时间，还是有可能会出现某些处理器忙，而其它处理器空闲的情况。这时，为了进一步提高并行程序的效率，就必须采用动态负载平衡算法。

本文提出了一种新的集中式接收者启动的动态负载平衡算法：在静态负载初始化的任务还没有完成时，各个处理器都处于忙的状态，这时先不进行负载平衡处理。一旦有 1 个任务在向主进程申请任务时发现任务池为空，主进程就随机发消息给 m 个从进程，让它们划分当前正在计算的任务，并把新划分的任务发送给主进程，主进程把收到的任务放入到任务池中，再发送给空闲的从进程进行计算。

这样做的好处是一旦第 1 个任务处于空闲，也就意味着马上将有更多的任务也处于空闲状态。这个时候划分任务可以更好地平衡各个处理器的计算量，能更好地使所有的进程在相同时间内结束，从而减少整个任务的运行时间。

当然，在从进程收到划分任务的消息时，它必须判断当前任务的粒度是否太小，如果太小，则自己接着计算，等搜到一个粒度比较大的任务再划分并发送任务。这样，对于很小的任务，可以避免因在网络中传送它们所需要的时间。

本文提出的混合负载平衡，就是将初始任务分配的静态负载平衡和程序开始运行后的动态负载平衡结合起来。图 4 给出了混合负载平衡算法的模型。

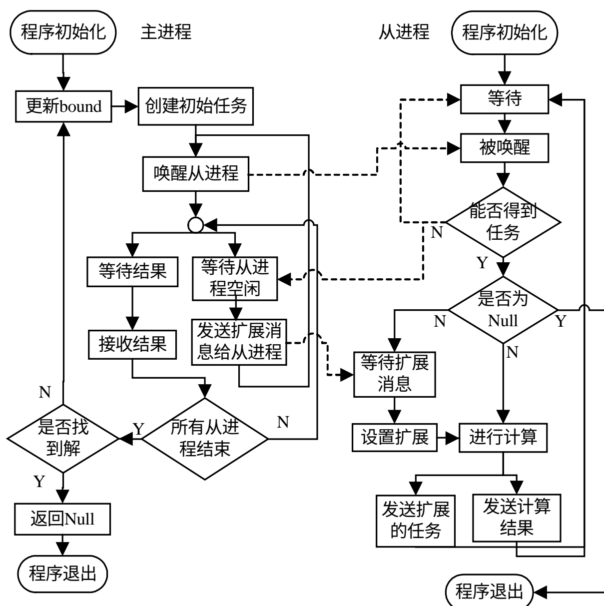


图 4 基于混合负载平衡的并行迭代加深启发式搜索算法模型

4 试验数据和性能分析

4.1 试验环境和测试用例

本文使用阿姆斯特丹自由大学(Vrije Universiteit, Amsterdam)的 DAS-2 并行机作为试验的平台。该并行机拥有 72 个本地结点，每个结点包括 2 个 Pentium III 1.0 GHz CPU、1 024 MB RAM、1 个 20 GB 硬盘、1 个 Myrinet interface card 和一个 Fast Ethernet interface card。在本文的测试中，最多使用了 33 个结点进行程序性能的测试(1 个主进程和 32 个从进程)。

本文使用经典的 15 数码问题作为并行 IDA*算法的测试用例。15 数码的题目是：在 4*4 的棋盘上，摆有 15 个棋子，每个棋子分别标有 1~15 的某一个数字。棋盘中有一个空格，空格周围的棋子可以移到空格中。现给出初始状态和目标状态，要求找到一种移动步骤最少的方法。图 5 给出了一个简单的需要 2 步到达目标状态的 15 数码问题的例子。

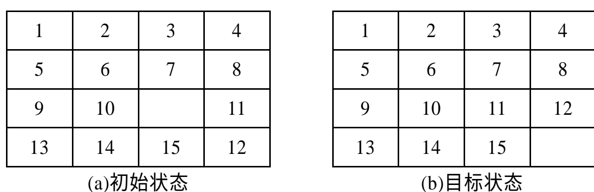


图 5 15 数码问题

但是 IDA*算法解决 15 数码的速度很块，一般找到最优解的时间都会小于 1s，这样不利于并行程序的性能测试。为了更好地测试并行 IDA*算法的性能，本文修改计算的目标，把找到一种移动步骤最少的方法改为：找到估价函数 $f(n)$ 相等且最小的所有解的数目。

4.2 试验数据分析

本文使用加速比和效率来表示并行程序的性能。加速比和效率定义为

$$S(p)=\frac{\text{使用单处理器运算时间(使用最好的顺序算法)}}{\text{使用具有p个从进程的多处理机的运算时间}} \quad (4)$$

$$E=\frac{\text{使用单处理器的运算时间}}{\text{使用多处理器的运算时间}\times\text{从进程个数}}=\frac{S(p)}{p} \quad (5)$$

为了更好地进行测试，本文选用了一个初始状态，用 Java 的顺序算法在单个结点上需要计算 61.572s。试验使用 Java 的 RMI 作为任务的发送和结果的接收方法。程序使用纯静态负载和混合负载平衡技术进行对比。试验平均结果见表 1、表 2。

表 1 静态负载平衡试验结果

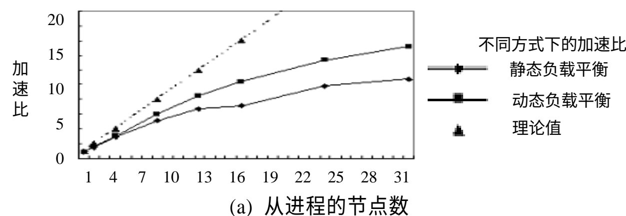
静态负载平衡	顺序算法	不同从进程结点数的并行运算							
		1	2	4	8	12	16	24	32
平均时间(s)	61.572	71.780	39.664	21.313	12.035	9.182	8.600	6.242	5.725
加速比	1	0.858	1.552	2.889	5.116	6.706	7.159	9.864	10.755
效率	100%	85.8%	76.1%	72.2%	63.95%	55.8%	44.7%	41.1%	33.6%

表 2 图 7 混合负载平衡试验结果

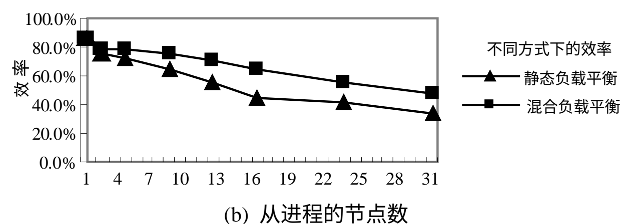
混合负载平衡	顺序算法	不同从进程结点数的并行运算							
		1	2	4	8	12	16	24	32
平均时间(s)	61.572	72.120	39.062	19.759	10.227	7.188	5.906	4.609	4.070
加速比	1	0.854	1.576	3.116	6.021	8.566	10.425	13.359	15.128
效率	100%	85.4%	78.8%	77.9%	75.3%	71.4%	65.2%	55.7%	47.3%

由表 1 和表 2 可以看出，尽管 Java 程序的整体数值运算速度比 C++等编程语言要低。但通过并行化处理，一样可以得到较好的计算速度。尤其是在结点数目较多的情况下，使整个程序的时间从顺序算法的 61.572s 降低到 5.725s 和 4.07s。

但仅使用静态平衡的并行算法和使用混合负载平衡的并行算法，在运算的时间、加速比和效率上还是有很大的区别。图 6 直观地反映出了这种差别。



(a) 从进程的节点数



(b) 从进程的节点数

图 6 不同负载平衡方式下的加速比和效率

由图 6 可以看出，动态负载平衡的加入，可以在很大程度上提高计算的加速比和效率，使加速比更加靠近理论的加速比。在 32 个从进程节点的情况下，整个程序的效率还维持在 50%左右，同时使 32 个结点的计算时间降低到了 4.07s，这足以证明本文提出的针对并行 IDA*算法提出的混合负载平衡技术的高效性。

5 结论

本文在分析 IDA*算法的可并行性基础上，给出了并行 IDA*算法的程序结构。为了加快并行 IDA*程序的运行速度，本文提出了一种综合静态负载平衡和动态负载平衡的算法。最后用 Java RMI 技术实现了该算法，并在 72 个结点的并行机上进行试验。通过试验结果证明该算法可以极大地提高并行搜索算法的速度和效率。 (下转第 215 页)