计算机科学1996 Vol. 23 №. 4

# SIMD 模型上深度优先的并行搜索算法*)

马绍汉 谢青松

(山东大学计算机科学系 济南 250100)

**摘 要** 近年来人们开始研究并行模型上的搜索算法，并取得了一些结果。本文介绍了 SIMD 模型上的两种深度优先的并行搜索算法，并对其进行了简要的比较分析，而且详细讨论了负载平衡问题。

**关键词** SIMD，并行搜索，负载平衡

在人工智能和计算机科学中普遍存在着搜索问题，特别是人工智能中搜索问题大都是 NP 难解的，如果用盲目搜索(即深度优先搜索或宽度优先搜索)的方法来求解，就会引起组合爆炸，所以人们广泛采用启发式搜索算法。A* 就是一种最佳优先(bestfirst)的启发式搜索算法，但其空间复杂性却是指数级的，在目前结构的单个处理器上还不能解决象15数码这般规模的问题[1]。Korf 等人在1985年提出 IDA* 算法，克服了 A* 在存储需求上的困扰，这是一种代价界定的启发式深度优先的迭代方法，其空间复杂性只是求解深度的线性函数，其运行时间，对于大多数用最佳优先的方法来求解的问题，是渐近最优的[2]，它能找到15数码问题的最优解，但对个别实例，在目前结构的单处理器上仍要花几十个小时。另一种可以提高搜索效率的办法就是采用并行处理技术[3,14]，近几年来在研究并行的启发式搜索算法方面已经得到了一些结果[5—10]。本质上并行化搜索的途径有四[1]：一是并行化对单个节点的操作的方法；二是所谓的并行窗口搜索；三是在搜索空间的状态和处理器之间进行匹配的方法；还有一种就是我们在本文中所用的对搜索树进行分解的方法，其基本思想是：先把原先串行算法的搜索树分解成多棵不同的子树，并把它们交给并行机上不同的处理器，然后让这些处理器并行地搜索各自获得的子树。由于通常的搜索树及其子树都是不规则的，所以要提高处理器利用率就必须在各处理器之间合理地分配工作负荷，使之尽量均衡，我们称之为负载平衡，这是需要花费很大代价的。目前在 MIMD 上实现的搜索算法已经有许多，如文[7]、[10]、[12]和[13]等。SIMD 上的处理器相对来说数量多，局部存储器容量小，功能弱，且是同步操作的，即在任一运行时刻每个微处理器要么和其他活动处理器执行同样的指令，要么它就根本不执行指令，而处于待激活状态，这使得对一些直截了当的任务的编程和校验更加容易，但是复杂任务在这种机器上的编程仍然很麻烦。直到目前 SIMD 计算机还主要用于矩阵运算、图像处理等并行的数字计算。但树搜索主要是不断地进行节点扩展和估值计算的过程，这就意味着 SIMD 可以用来进行高效的启发式搜索[1]。虽然在 SIMD 上进行负载平衡所花的代价比在 MIMD 上大，但是 SIMD 有更好的性能价格比，所以如果能解决好负载平衡问题，使之少占运行时间，从而提高总的搜索效率的话，那么用 SIMD 计算机进行并行搜索还是很有实际意义的。本文介绍了 SIMD 上的两种深度优先的并行搜索算法，同时还讨论了搜索中的负载平衡问题。在介绍我们的并行算法之前，先简介几种常见的串行深度优先搜索算法。

## 1 深度优先的搜索算法

一般，一个搜索问题可以形式化为一个三元组(I,O,S)，其中 I 是初始状态，O 是目标状态集合，S 是问题空间[11]。对问题的求解就是要找出一个能把初始状态映射到一个目标状态的操作序列。比如人们所熟悉的八数码难题里，问题的状态即九格方框内八个数码方块的可能的排列，所允许的操作即把邻接空格的数码方块移到空格位置，目标即要找出一个合法的移动序列，以使得九格方框能从初始排列变成目标排列，而解的代价即为所需移动的总的

*)本课题得到国家自然科学基金资助。

次数。在这样的问题模型里，搜索算法所花的时间和空间分别与它所扩展的状态个数和所必需存储的状态个数成正比，而我们只对问题的渐近复杂度感兴趣，所以将用两个参数：问题空间分枝因子 b 和问题求解深度 d 来表示问题的复杂度。这里 b 是整个问题空间中对任一给定状态应用一个操作规则后所产生的新状态的平均个数，也叫问题空间节点分枝因子。d 是能从初始状态映射到一个目标状态的最短的操作序列的长度。一个搜索算法的时间代价可以定义为所扩展的状态节点的个数。不失一般性，为简单起见，我们假设 b 在整个问题空间中是个常量。传统的深度优先搜索算法的基本思想是：

从起始状态 I 开始扩展，以后每次总是选最新扩展节点的一个后继节点先来扩展，直到到达某个深度界限值 B 或者最新扩展节点没有后继节点为止，然后回溯到上一个最新产生的深度最大的未扩展的节点继续扩展之。如此继续下去，直到产生了某个目标节点 $t\in O$ 为止。

由于深度优先搜索在执行过程中只有从初态到目标状态路径上的节点是必须存储的，所以其空间复杂性是 O(d)。而其时间复杂性是 $O(e^d)$，这里 e 是边界分枝因子，即对一个给定状态平均可用的不同的操作规则的数目[2]。可见实际上深度优先搜索具有指数时间复杂性，这是它的一个缺陷，它的另一个缺陷是要求有一个合适的但很难找到的深度阈值[3]。

有一种叫作深度优先纵深迭代的搜索算法（以下简记为 DFID 算法）可以克服深度优先搜索的时间受限的缺陷，该算法实际是一种不断改变深度阈值的深度优先的搜索，其算法描述为：

(1)深度阈值置初值1；

(2)从起点 I 开始，执行一次深度优先搜索；

(3)丢弃上次搜索中产生的节点，深度阈值增1，再从 I 开始，执行一次深度优先搜索；

(4)重复执行(3)，直到到达一目标状态 $t\in O$ 为止。

由于 DFID 法先扩展了某一深度层上的所有节点后才开始扩展下一层的节点，所以它与宽度优先搜索算法类似，必能找到一条最短的解路径（如果存在的话）；又因为在任何给定的时间内它都在执行深度优先搜索，且决不会搜索到 d 深度层以下，故其所用空间为 O(d)，DFID 搜索算法保留了宽度优先搜索和深度优先搜索的优点，克服了它们的一些缺点，但其运行时间和深度优先搜索差不多，因为它必须搜索到任一给定深度的所有可能的路径[2]。

启发信息的引入可以大大减小搜索的空间。虽然可采纳的 A* 算法能找到最优解（若最优解存在的话），但它在搜索规模大的问题时却与宽度优先搜索类似，也是存储受限的。把 DFID 的思想用于 A*，可以去掉 A* 的这种存储受限的空间复杂性的缺陷，这样产生的新方法就是纵深迭代的 A* 算法，以后简记为 IDA* 算法。与 DFID 算法相比，这里搜索迭代中不断改变的不是深度阈值，而是节点的代价阈值。IDA* 的算法描述为：

(1)代价阈值置初值为起始状态 I 的代价的估值；

(2)开始一次迭代，即执行一次深度优先搜索，当一个搜索路径上最后一点的代价（f(n)=g(n)+h(n)）超过这次迭代的阈值时，就删掉这个分枝；

(3)代价阈值增值为：当前超过上一个阈值的所有节点的代价 f 值的最小值；

(4)重复执行(2)和(3)，直到选中一目标状态节点 $t\in O$ 来扩展为止。

和 A* 算法一样，若启发函数是单调可采纳的且最优解存在的话，则 IDA* 算法一定能找到目标节点而终止。因为在任一给定时间内 IDA* 都在执行深度优先搜索，所以它的空间复杂性只是求解深度的线性函数，这是 IDA* 的主要优点，同时它还继承了 A* 算法的一些好的特性。文[2]证明了 IDA* 算法对于大多数现有的最佳优先的树搜索来说是渐近最优的。

## 2 并行的深度优先的搜索算法

采用树分解技术，借鉴 IDA* 的思想，现在我们给出一个在 SIMD 上实现的 IDA* 算法，以下简记为 SIDA* 算法。该算法由两大阶段构成，首先是一个在诸处理器之间进行节点初始分配的阶段，其次是一系列 IDA* 的迭代过程。在每个 IDA* 的迭代过程中，每个处理器独立地对分给它的节点下的子树进行深度优先搜索，只有当所有处理器都完成了它们的搜索任务时，该算法的一次 IDA* 的迭代才算完成。在每个 IDA* 的迭代过程中，当有足够多的处理器完成了对分给它们的子树的搜索而变得空闲时，整个搜索暂时停下来，负载平衡开始，即在诸处理器之间重新分配剩余的工作负载，然后搜索继续。所以每个 IDA* 迭代过程由一系列搜索和负载平衡阶段的交替进行而组成。当算法完成了一次 IDA* 的迭代之后，又要进行一次负载平衡来重新调整分给各处理

器的节点个数，以使下一次迭代开始时每个处理器的负载尽量均等，接着要增大阈值，开始下一次IDA*的迭代。这个过程继续下去，直到一个目标节点被选来扩展。算法执行时，每个处理器用一个栈来存放搜索路径上的状态节点。SIDA*的算法描述为：

(1)含根节点的处理器扩展根节点，把其第一个子节点留给自己，其余的逐个分给其它空闲的处理器，若发现目标节点则算法成功结束；

(2)for 每个含当前最小f值节点的处理器 do

扩展该节点，把其第一个子节点留给自己，其余的逐个分给其它空闲的处理器，若发现目标节点则算法成功结束；

(3)重复执行(2)，直到每个处理器都有一个起始点为止；

(4)代价阈值初始化为所有处理器起始点的最小的f值；

(5)repeat

for 每个处理器 do

if 起始节点的f值≤代价阈值 then 该起始节点压栈；

repeat

执行一次深度优先的搜索，当有"足够多"的处理器变得空闲时，搜索暂停，进行迭代过程中的负载平衡。

until 每个处理器搜索完自己的子树或者一个目标节点被找到；

根据上次迭代产生的节点负荷信息，进行一次两个相继的搜索迭代过程之间的负载平衡；

修改代价阈值；

until 一个目标节点被找到。

注意，为了保持较高的处理机利用率，算法中有三个地方需要负载平衡。一是在对处理器进行初始节点分配时，使各处理器的子树的大小尽量接近；二是在一次IDA*迭代中，使搜索中出现的空闲的处理器尽可能得以利用；还有一次是在两次相继的IDA*迭代过程之间，使下一次IDA*搜索迭代开始时各处理器的负载尽量平衡。然而，这些是要花大量处理器之间的通讯代价的，特别是IDA*搜索迭代过程中的负载平衡，其执行频率最高，对算法的效率影响最大。实验表明，处理器初始填充时的负载平衡以及两次相继的IDA*搜索迭代过程之间的负载平衡对SIDA*算法总的执行效率的影响很小，故可忽略之；而IDA*搜索迭代过程中的负载平衡则是必不可少的[1]。本文后面第3部分将着重讨论SIDA*算法中IDA*搜索迭代过程中的负载平衡的有关问题，至于另两处的负载平衡的具体情况请参阅文[1]。实验证明SIDA*算法尽管扩展的节点比串行算法多，但它对搜索15数码难题等规模大的问题所花的时间远比串行算法少。

既然处理器初始填充阶段的负载平衡和两次相继的IDA*搜索迭代间的负载平衡对SIMD上并行深度优先搜索算法效率的影响小得可以忽略，而IDA*迭代过程中的负载平衡又是必不可少的，现在我们给出一个SIMD上的最佳优先的分支限界的搜索算法，以下简称为IDPS算法，它只含深度优先搜索迭代过程中的负载平衡。该算法与SIDA*算法类似，主要也由两大阶段构成，一是搜索阶段，二是负载平衡阶段，这两大阶段交替进行，直到一目标节点被找到为止。但该算法又与SIDA*不一样，因为虽然每个处理器也用一个栈来进行节点的扩展，但该栈不存储搜索路径，当搜索中某点的子女全部产生后便丢弃这个节点，另外该算法所用的具体的搜索和负载平衡策略均异于SIDA*中相应的策略，而且它不严格遵从深度优先的性质。这里我们使用了Mahanti等人提出的两种扩展节点子程序：部分扩展子程序PE和完全扩展子程序FE，以及两种负载平衡子程序：共享一点子程序SS和共享多点子程序GS[4]。SS与串行深度优先搜索过程类似，每次产生被扩展节点的一个子节点，然后沿这个子节点继续往深处搜索，其算法描述为：

```
while 栈不空 do
  弹出栈顶节点 n
  产生 n 的一个新的子节点 c
  if n 还有未产生的子节点 then 把 n 压入堆栈
  if f(c)≤阈值 then
    if c 是一个目标节点 then 成功返回
    else 把 c 压入堆栈
  endwhile
```

FE在扩展节点时似宽度优先搜索算法，其算法描述为：

```
while 栈不空 do
  弹出栈顶节点 n
  产生 n 的一个新的子节点 c
  if f(c)≤阈值 then
    if c 是一个目标节点 then 成功返回
    else 把 c 压入堆栈
  if n 还有未产生的子节点 then 把 n 压入堆栈
endwhile
```

SS 负载平衡算法的思想很简单：它把空栈标记为穷栈，把含多个节点的栈标记为富栈，把仅含一个节点的栈标记为满足栈，然后匹配穷富栈，让富栈分给穷栈一个节点，继续此过程，直到没有栈被标为穷栈或者当没有栈被标为富栈为止，没有穷栈存在时，就把所有的栈都标记为满足栈，可见，一旦初始标记的穷栈都获得一个节点后，负载平衡过程便暂停了。该算法使得单独执行一次负载平衡的花费最少，尽管它只是最小限度地满足了各处理器的工作需要，但它仍然提高了下一个搜索阶段的处理器利用率，其算法描述为：

```
repeat
  标记所有的空栈为穷栈，并依次编号
  标记所有的含多个节点的栈为富栈，并依次编号
  for 每一个有相应(编号相同的)穷栈的富栈 do
      从富栈贡献一个节点给穷栈
until 没有栈被标记为穷栈或者没有栈被标记为富栈。
```

GS 负载平衡算法的思想与 SS 相反，它不在乎一次负载平衡的费用，而是要在一次执行中尽可能地在所有处理器间均分节点，这样就有必要算出栈尺寸平均值的上整数界(即把这个平均值向上取整的结果，记为 ceil)和下整数界(即把这个平均值向下取整的结果，记为 floor)，以便在每一个栈恰好含有 ceil 或 floor 个节点时，这个负载平衡的迭代过程能暂停。该算法的每一次迭代中也标记三种栈：含少于 floor 个节点的栈标记为穷栈，多于 ceil 个节点的栈标记为富栈，其余的标记为满足栈。当没有穷栈且至少有一个富栈时，把所有含 floor 个节点的满足栈标记为穷栈；当没有富栈且至少有一个穷栈时，把所有含 ceil 个节点的满足栈标记为富栈；同样当所有的栈都是满足栈时，该算法的迭代过程终止。为了防止在搜索过程中一个栈可能会无界地增长，这里给出了穷栈尺寸的一个界限值 $B^{[4]}$，GS 虽然增加了一次负载平衡的费用，却降低了搜索过程中触发负载平衡的频率，因为栈空的可能性减小了。实验表明，GS 的效能比 SS 好。GS 的算法描述为：

```
floor←[栈尺寸平均值]
ceil ←[栈尺寸平均值]
repeat
  for 每个含少于 min(B-1,floor)个节点的栈 do
      把这个栈标记为穷栈，并在编号表中顺次登记
      其标识符
  for 每个含多于 min(B,ceil)个节点的栈 do
      把这个栈标记为富栈
  if 没有栈被标记为穷栈 then
    for 每个恰好含 min(B-1,floor)个节点的栈 do
        把这个栈标记为穷栈，并在编号表中顺次登
        记其标识符
  else if 没有栈被标记为富栈 then
    for 每个恰好含 ceil 个节点的栈 do
        把这个栈标记为富栈
  for 每个其栈被标为富栈的处理器 do
      顺次读编号表，获得一个唯一的相应穷栈的标
      识符
  for 每个有一个相应穷栈的富栈 do
      从富栈贡献一个节点给穷栈
until 没有栈被标记为穷栈或者没有栈被标记为富
    栈。
```

有了 PE、FE、GS 和 SS 后，我们的 IDPS 算法的基本描述为：

(1)选 PE 或 FE 作为扩展节点子程序

(2)选 SS 或 GS 作为动态负载平衡子程序

(3)进行处理器初始填充：从起始根节点出发进行完全扩展，使每个处理器有一个起始点(子树的根)

(4)repeat

```
for 每个处理器 do
  if 起始节点的 f 值≤代价阈值 then 该起始节点
  压栈
repeat
  执行所选的扩展节点子程序
  if 某个栈已空 then 执行所选的动态负载平衡子程
    序
  until 每个栈均空或一个目标节点被找到
  修改代价阈值
until 一个目标节点被找到。
```

该算法略作修改，便得到一个效率更高的算法：在处理器初始填充阶段，可对搜索树进行启发式扩展，使每个处理器在开始搜索迭代前都有 $S_0 \geq 1$ 个其 f 值≤阈值的起始点，则上述算法中第一个 if 判断可去掉，改为起始点直接入栈，这样算法后面栈空的机会和修改阈值的次数都会减少，总的效率得以提高。这相当于进行了处理器初始填充阶段的负载平衡。另外，GS 被引发时不仅可以立即给穷栈提供节点，而且可以阻止本来快要变成空闲的处理器很快变成空闲，所以说 GS 比 SS 性能好，故算法第二步中的 SS 可以去掉。实验表明用了 PE 和 GS 的 IDPS 比较适于处理器较少(比如少于8K)的机器，而用了 FE 和 GS 的 IDPS 比较适于处理器较多(比如多于16K)

的机器[4]。

注意，设最大的搜索路径长度是L，最大的节点分枝因子是b，则用了PE和FE的IDPS的存储需求分别大约是L和bL。与SIDA*类似，尽管IDPS扩展的节点总数一般比串行的IDA*解同一问题时所扩展的节点多得多，但它找到解所花的时间一般要比IDA*算法少得多。

扩展的IDPS算法与SIDA*算法相比，本质上都是基于IDA*的SIMD上的并行搜索算法，都用了初始填充阶段的负载平衡和搜索迭代过程中的负载平衡，但由于它们所用的具体的搜索和负载平衡策略不同，其主要差别如下：

①IDPS没用两次相继搜索迭代过程之间的负载平衡，而SIDA*用了；

②IDPS的搜索中每当有一个处理器空闲时就进行负载平衡，而SIDA*在有"足够多的"处理器变得空闲时才进行搜索迭代过程中的负载平衡，SIDA*对负载平衡的触发更智能化，详情请见本文第3部分；

③IDPS在作搜索中负载平衡时，每个穷栈可从相应富栈获得多个节点，而SIDA*在作搜索中负载平衡时，每个穷栈只能共享相应富栈的一个节点；

④IDPS不保存完整的搜索路径，当某点的子女节点在搜索中被完全产生后，该点便被丢弃，IDPS不严格遵从深度优先搜索产生搜索树的性质；而SIDA*严格遵从深度优先的性质，每个处理器有一个用于深度优先搜索的栈，用来保存完整的搜索路径信息，SIDA*能适应的问题面更广一些。

## 3 负载平衡

由于初始分给各处理器的子树往往是不规则的，尺寸大小不一，在算法后面的IDA*迭代搜索过程中常常会出现这样的情况：某些处理器还在搜索自己的子树而另一些处理器却早已搜索完它们的子树而变得空闲了。这时若把活动处理器的一部分工作交给那些空闲的处理器，则处理器的利用率会显著提高，整个并行搜索算法的效率也会提高。尽管在处理器初始填充阶段和两次相继的IDA*搜索迭代之间进行负载平衡也能提高处理器总的利用率，但这两处的负载平衡对我们的并行算法的效率的影响总的来说是很小的，故在此我们只讨论我们的并行算法IDA*搜索迭代过程中的负载平衡。

SIMD上并行解决单个搜索问题时常见的负载平衡模式有两种，一种是传统的负载平衡模式，即"全局阈值变量"模式[5]。在此模式下，并行的基于IDA*的搜索似串行的IDA*算法，所有处理器在同一个阈值下搜索自己的子树，集合起来看，它们就好象是在一棵完整的树上进行纵深迭代的IDA*搜索，当有某些处理机变得空闲时就进行负载平衡。

另一种模式是"局部阈值变量"模式。在此模式下，各处理器以不同的阈值搜索自己的子树。当某个解被找到时（一般不是最优的），搜索还要继续，直到所有更浅层的阈值下的搜索都完成为止，这时才能产生最优解。

经过实验和比较，近年来人们的兴趣主要在上述第一种负载平衡模式上，本文所介绍的两种并行算法中的负载平衡均以这种模式为基础。具体的全局阈值变量模式的平衡负载的办法可以有多种。在SIDA*算法中，我们通过匹配活动处理器与空闲处理器来平衡负载的负载平衡策略是一种集合分配方法。这种方法把所有活动处理器和空闲处理器分别逐个顺序编号，所有活动的处理器从栈顶到栈底查找并标记它们的第一个（最浅层的）具有未扩展子女的节点，这个节点在处理器的子树中最高，它反映了处理器剩余的负荷，被选为要共享的"传输节点"。因为开始负载平衡时活动处理器可能比空闲处理器多，所以我们开始选负载最大的那些处理器进行节点传输。SIDA*的搜索迭代过程中的负载平衡的算法描述为：

(1)将所有的活动处理器逐个顺序遍号

(2)将所有的空闲处理器逐个顺序遍号

(3)if 活动处理器比空闲处理器多 then do

①具有最小f值的传输节点的活动处理器把该传输节点的拷贝送给编号相同的空闲处理器

②具有次最小f值的传输节点的活动处理器把该传输节点的拷贝送给编号相同的空闲处理器，如此继续，直到没有空闲处理器或者所有活动处理器都已输送一个点给相应的空闲处理器为止。

else for 每个活动处理器 do

把传输节点的拷贝送给编号相同的空闲处理器。

有了负载平衡算法后，在SIMD上进行深度优先的搜索过程中何时进行负载平衡才好呢？这也有两种基本办法。一是采用常量触发值的办法；一是采用变量触发值的办法。所谓采用常量触发值的办法即令X(0<X<1)为常量触发值，P为处理器总个数，在处理器初始填充后，每当活动处理器个数下降到XP个时，即每当(1−X)P个处理器已经变得空闲时就引发负载平衡过程。因为负载平衡的费用与其执行频率成正比，所以要找出一个最好的常量触发值是非常困难的。另外，即使一个常量触发值对某

个问题是最好的，它对其它的问题也不一定会是最好的，而且它对那个特定问题也不一定导致最好的综合性能。例如，如果负载平衡的费用相对于剩余工作量比较低的话，那么该触发值应取得大一些，以鼓励负载平衡，反之，它应当取得小一些，但是，在单一一个问题的整个求解过程中，剩余工作量是趋于减少的，尽管负载平衡的费用近似不变。可见用这种常量触发值的方法来决定搜索中引发负载平衡的时机是不太好的，但它实现起来最简单。最好的方法应该是采用动态触发值的方法，即让负载平衡的触发值随时间的变化而变化。我们在 SIDA* 中所用的动态触发值的方法就是一种近似最优的方法：设 A(t)是 t 时刻的活动处理机个数，W(t)是目前搜索阶段前 t 秒所完成的总的工作量，它是 A(t)从当前搜索阶段开始到 t 时刻的积分，L 是上一次负载平衡花的时间(秒)，P 是总的处理机个数，则每当 $A(t)\leqslant W(t)/(L+t)$ 或 $A(t)\leqslant P/2$ 且 $t\geqslant L/2$ 时，就进行迭代过程中的负载平衡。这是一种最大值化每个从上次负载平衡开始到当前搜索阶段结束期间的平均工作率的贪婪策略，文[1]证明了当 $A(t)=W(t)/(L+t)$ 时，当前的负载平衡与搜索的循环圈内的平均工作率 $W(t)/(L+t)$ 取得最大值。因为 A(t)是离散变化的，所以一旦 $A(t)\leqslant W(t)/(L+t)$ 就应当进行负载平衡，这种取局部最优的负载平衡触发时机的方法是近似最优的[1]，可适用于任何搜索和负载平衡分开进行的问题的并行求解中。这里，我们的负载平衡触发值是随时间而变化的，由于每个富栈在一次负载平衡中最多只能送给相应穷栈一个共享节点，所以在这个负载传输期间活动处理器个数不会超过原来的两倍。为让更多的处理器保持活动状态，在我们的触发机制里每当 $A(t)\leqslant P/2$ 时也进行负载平衡。考虑到随着搜索的推进，当所剩工作很少时，这种机制可能导致负载平衡过于频繁，故我们加了 $t\geqslant L/2$ 的限制。当活动处理器少于 P/2 时，如果 $t<L/2$，即若再进行负载平衡会得不偿失时，就停止进行负载平衡。这种 $t\geqslant L/2$ 的限制(即两次负载平衡至少要间隔 L/2 秒的限制)也适于常量负载平衡触发值机制[12]。

Mahanti 等人所提出的两个负载平衡算法与我们在 SIDA* 算法中给出的负载平衡算法本质上是相似的，特别是 SS，但是 IDPS 中对 SS 和 GS 的触发采用的是一种常量触发值的方法，在那里 $X=1-1/p$。GS 只不过是一种为降低负载平衡频率而增加单个负载平衡时间的方法，但在 IDPS 中 GS 比 SS 的效能好[13]。

并行搜索是近几年发展起来的新的研究课题，本文主要介绍了两种 SIMD 模型上的并行深度优先搜索算法，提出并比较了几种扩展节点和负载平衡的方法。我们认为影响 SIMD 机器上深度优先搜索算法效率的主要因素是负载平衡。因此，今后仍需重点研究的问题有：(1)如何更均衡地对处理机进行初始节点分配(2)搜索期间出现处理机空闲的情况下，何时引发负载平衡最好(3)怎样进行高效的负载平衡，使其通讯代价最小，占总的运行时间最少。相信不久的将来一定会有 SIMD 上的更高效的并行搜索算法。

## 参考文献


[1]C. Powley et al., Depth-first heuristic search on a SIMD machine, Artif. Intell. 60(2), 1993

[2]R. E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artif. Intell. 27 (1), 1985

[3]R. E. Korf, Planning as search: A quantitative approach, Artif. Intell. 1987

[4]A. Mahanti et al., A SIMD approach to parallel heuristic search, Same to[1]

[5]C. Powley and R. E. Korf, Single-agent parllel windowsearch, IEEE Trans. Pattern Anal. Mach. Intell. 13(5)1991

[6]M. Evett et al., PRA*: a memory limited heuristic search procedure for the Connection Machine, in: Proc. Third IEEE Symposium on the Frontiers of Massively Parallel Computations. College Park, MD(1990)

[7]V. N. Rao and V. Kumar, Parallel depth-first search, Part I: implementation, int. J. Parallel Program. 16(6), 1987

[8]C. Powley and R. E. Korf, SIMD and MIMD parallel search, in: Proc. AAAI Spring Symposium on Planning and Search, Stanford, CA(1989)

[9]V. Kumar and V. N. Rao, Scalable parallel formulations of depth first search, in: Parallel Algorithms in Machine Intelligence and Vision (Springer, New York, 1990)

[10]V. Kumar et al., Parallel best-first search of state-space graphs: a summary of results, in: Proc. AAAI-88, Saint Paul, MN(1988)

[11]A. Newell et al., Human Problem Solving (Prentice-Hall, Englewood Cliffs, NJ, 1972)

[12]C. Powley et al., Parallel heuristic search: tow approaches, Same to[9]

[13]R. Feldmann et al., Distributed game tree search, Same to[9]

[14]李国杰，并行组合搜索，《智能技术与系统基础》，北大出版社，1990