第二届数学中国数学建模网络挑战赛
地址：内蒙古数学会　　网址：www.tzmcm.cn
电话：0471-4343756　　邮编：010021　　Email：ceo@madio.cn

第二届“数学中国杯”数学建模网络挑战赛

承　诺　书

我们仔细阅读了第二届“数学中国杯”数学建模网络挑战赛的竞赛规则。

我们完全明白，在竞赛开始后参赛队员不能以任何方式（包括电话、电子邮件、网上咨询等）与队外的任何人（包括指导教师）研究、讨论与赛题有关的问题。

我们知道，抄袭别人的成果是违反竞赛规则的，如果引用别人的成果或其他公开的资料（包括网上查到的资料），必须按照规定的参考文献的表述方式在正文引用处和参考文献中明确列出。

我们郑重承诺，严格遵守竞赛规则，以保证竞赛的公正、公平性。如有违反竞赛规则的行为，我们将受到严肃处理。

我们允许数学中国网站(www.madio.net)公布论文，以供网友之间学习交流，数学中国网站以非商业目的的论文交流不需要提前取得我们的同意。

我们的参赛报名号为：1418

参赛队员　(签名)：

队员 1：徐涛

队员 2：张立峰

队员 3：张健

参赛队教练员 (签名)：

参赛队伍组别：本科组

第二届数学中国数学建模网络挑战赛
地址：内蒙古数学会 网址：www.tzmcm.cn
电话：0471-4343756 邮编：010021 Email：ceo@madio.cn

第二届“数学中国杯”数学建模网络挑战赛

编 号 专 用 页

参赛队伍的参赛号码：（请各个参赛队提前填写好）：1418

竞赛统一编号（由竞赛组委会送至评委团前编号）：

竞赛评阅编号（由竞赛评委团评阅前进行编号）：

第二届数学中国数学建模网络挑战赛
地址：内蒙古数学会　　网址：www.tzmcm.cn
电话：0471-4343756　　邮编：010021　　Email：ceo@madio.cn

2009 年 第二届“数学中国杯”
数学建模网络挑战赛

题　　目　**串行算法的并行化处理**

关 键 词　串行算法，并行算法，划分，通信，组合，映射，负载平衡，快速傅立叶变换（FFT）

摘　　要

随着信息时代的到来，需要处理的信息量越来越庞大。需要解决的问题越来越复杂，使得计算量剧增。通过提高单个处理器的运算速度和采用传统的“顺序（串行）”计算技术已难以胜任。需要有功能强大的新的计算机系统和计算技术来支撑，因此，并行计算机及并行处理技术应运而生。

本文首先对行算法的并行化处理做了复杂性分析，较充分地考虑了串行算法的并行化处理需要解决的四个阶段即：划分，通信，组合和映射，然后建立了数学模型，以及通过快速傅立叶变换对模型进行的并行优化处理。

文中通过在UNIX操作系统的环境下使用并行计算平台PVM对设计的算法编程实现这一实例，论证了并行算法相对于串行算法的优越性，最后给出了模型评价，分析了文中采用算法的优缺点。

参赛队号　1418

所选题目　A

参赛密码 _____
（由组委会填写）

第二届数学中国数学建模网络挑战赛
地址：内蒙古数学会 网址：www.tzmcm.cn
电话：0471-4343756 邮编：010021 Email：ceo@madio.cn

英文摘要（选填）

Abstract

With the advent of the information age, the amount of information we need to address is ever-growing. The complexity of problems we need to address is increasing, thus leads to dramatic increase in computation. So hard can we handle them by increasing computational speed of individual processor and the use of the ”sequence ” computing technology, that we need a powerful new computer system and computing technology to support them. Therefore, parallel computers and parallel processing technologies come into being.

This article first does a complex analysis for serial parallel processing algorithms. Taking four phases that serial parallel processing algorithms need to address into fully account, namely: division, communications, composition and mapping, then establishes a mathematical model as well as complete parallel optimization of the model through the Fast Fourier Transform.

This article demonstrates the advantages of parallel algorithms over serial algorithms through the instance being achieved by the use of parallel computing environment for the design of PVM platform programming algorithm in UNIX operation system, gives the final evaluation of the model, and analyses the pros and cons of the algorithms adopted by the article.

报名号　#　1418

一．问题概述

并行计算，是将一个计算任务分摊到多个处理器上并同时运行的计算方法。由于单个 CPU 的运行速度难以显著提高，所以计算机制造商试图将更多个 CPU 联合起来使用。台式机和笔记本电脑现在也已广泛地采用了双核或多核 CPU。双核 CPU 从外部看起来是一个 CPU，但是内部有两个运算核心，它们可以独立进行计算工作。运行一个以常规的串行代码写成的程序时，如何将计算任务拆分成多个部分并分解到多个核心上同时运行，是很困难的事情。有时甚至需要人工来读懂原来代码的含义，并以适合并行计算的语言重写程序。这需要耗费大量的人力物力。

最容易被并行化的计算任务称为“易并行”的，它可以直观地立即分解成为多个独立的部分，并同时执行计算，但并不是所有程序都能均匀分配个每个 CPU。由此，在软件行业中很自然地提出这个问题：希望设计一种方法，能够将一个常规的串行程序分解成两个部分，使之能够在 CPU 的两个核心上并行运算，并且希望能够尽量发挥双核心的计算能力。一般来说，如果两个核心的运算量基本相当，那么总的运算效率较高。如果出现某个核心空闲等待的状态，双核 CPU 的运算能力就没有被充分利用起来。

问题：

（1） 第一阶段问题：

问题：请你们建立合理有效的模型，并依据模型对现成的串行算法进行处理。将能够使用双核心并行处理的部分分解开，并分配到两个核心上同时运行。以期达到比单核 CPU 处理更快速的目的。

具体要求：假设算法是使用 C 语言写成的，代码里只有顺序执行、分支、循环三种结构。为简单起见，我们假设只对整型变量和整型数组进行操作，不需要调用已有的库函数。程序中所有的语句只包括简单的代数运算、赋值、条件分支语句（if-else 语句），循环语句（while 语句）。不包括其他语句。

这里的关键是如何通过分析代码，从总的计算任务中尽量识别可独立运算的部分，并估计每部分的计算量，来尽量使双核 CPU 发挥出超过单核的效能。

（2） 第二阶段问题：

问题：在合理划分任务的前提下，双核处理器比单核处理器的运算效能要高。在算法级别上的划分，主要目的是进行数据的划分，以及顺序执行和循环的分解。分解的方法也就是第一阶段问题的目的。一个任务虽然可能能够分解到两个核心上分别处理，但不一定能较好地达到负载均衡的要求。由于算法的种类极多，所以分解方法也可能有试用范围的限制。请建立合理的模型，对你们设计的分解方法的效果作出评价，并根据你的评价，有针对性地对分解方法作出优化。

注：解决这类问题，按照软件行业的要求，最终产品应是一个代码预处理器，也就是可以把源代码进行分解的计算机程序。但完成本问题不需要进行编程，也不需要考虑语法分析等详细问题。只需要在算法的层面上进行分析，给出确切的分析方法和模型即可。

二．模型背景及假设

双核 CPU 中，要很好地发挥出多个 CPU 的性能的话，必须保证分配到每个 CPU 上的任务有一个很好的负载平衡。否则一个 CPU 在运行，另外一个 CPU 处于空闲，无法发挥出双核 CPU 的优势来。

要实现一个好的负载平衡通常有两种方案，一种是静态负载平衡，另外一种是动态负载平衡。

1、静态负载平衡

静态负载平衡中，需要人工将程序分割成多个可并行执行的部分，并且要保证分割成的各个部分能够均衡地分布到各个 CPU 上运行，也就是说工作量要在多个任务间进行均匀的分配，使得达到高的加速系数。

静态负载平衡问题从数学上来说是一个 NP 完全性问题，Richard M. Karp, Jeffrey D. Ullman, Christos H. Papadimitriou, M. Garey, D. Johnson 等人相继在 1972 年到 1983 年间证明了静态负载问题在几种不同约束条件下的 NP 完全性。

虽然 NP 完全性问题在数学上是难题，但是这并不是标题中所说的难题，因为 NP 完全性问题一般都可以找到很有效的近似算法来解决。

2、动态负载平衡

动态负载平衡是在程序的运行过程中来进行任务的分配达到负载平衡的目的。实际情况中存在许多不能由静态负载平衡解决的问题，比如一个大的循环中，循环的次数是由外部输入的，事先并不知道循环的次数，此时采用静态负载平衡划分策略就很难实现负载平衡。

动态负载平衡中对任务的调度一般是由系统来实现的，程序员通常只能选择动态平衡的调度策略，不能修改调度策略，由于实际任务中存在很多的不确定因素，调度算法无法做得很优，因此动态负载平衡有时可能达不到既定的负载平衡要求。

3、负载平衡的难题在那里？

负载平衡的难题并不在于负载平衡的程度要达到多少，因为即使在各个 CPU 上分配的任务执行时间存在一些差距，但是随着 CPU 核数的增多总能让总的执行时间下降，从而使加速系数随 CPU 核数的增加而增加。

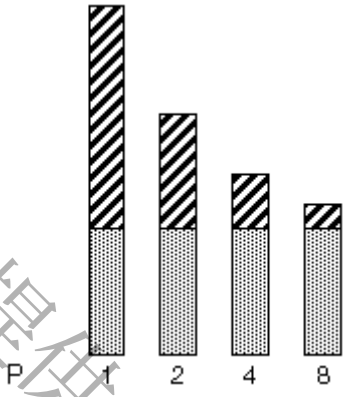
固定问题大小加速通常遵守 Amdahl 定律。Amdahl 定律明确说明：给定问题中的并行加速量受问题的顺序部分的限制。以下方程式描述问题的加速，其中 F 是在顺序

区域花费的时间，剩余时间可均匀分摊到 P 个处理器。如果方程式的第二项减小到零，则总加速受第一项约束，保持固定。

$$\frac{1}{S} = F + \frac{(1-F)}{P}$$

下图以图表方式说明此概念。深色阴影部分表示程序的顺序部分，并且对于 1、2、4、8 个处理器均保持不变。浅色阴影部分代表程序的并行部分，可均匀地分摊在任意多个处理器之间。

图 2 - 1 固定问题加速

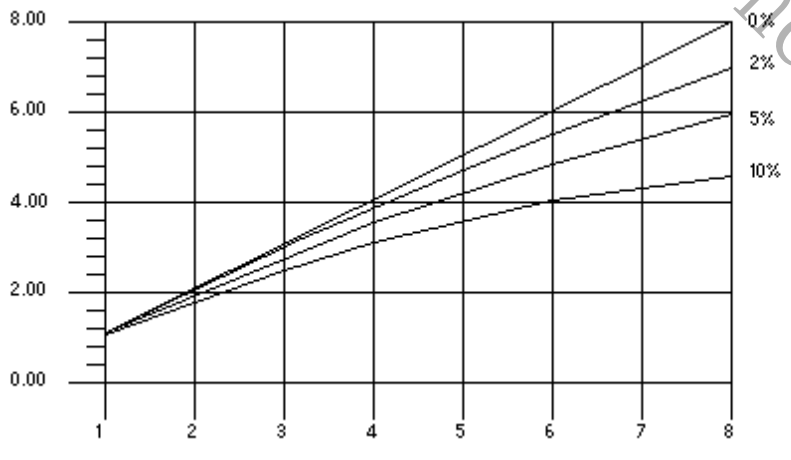


我们可以看到，随着处理器数目的增加，每个程序并行部分所需的时间会减少，而每个程序串行部分所需的时间保持不变。

然而，实际上可能会发生由于通信以及向多个处理器分配工作而产生的开销。对于使用的任意多个处理器，这些开销可能固定，也可能不固定。

下图 2-2 说明一个包含 0%、2%、5% 和 10% 顺序部分的程序的理想加速。此处假定无开销。

图 2 - 2 Amdahl 定律加速曲线



显示一个包含 0%、2%、5% 和 10% 顺序部分的程序的理想加速（假定无开销）的图。x 轴表示处理器的数目，y 轴表示加速。

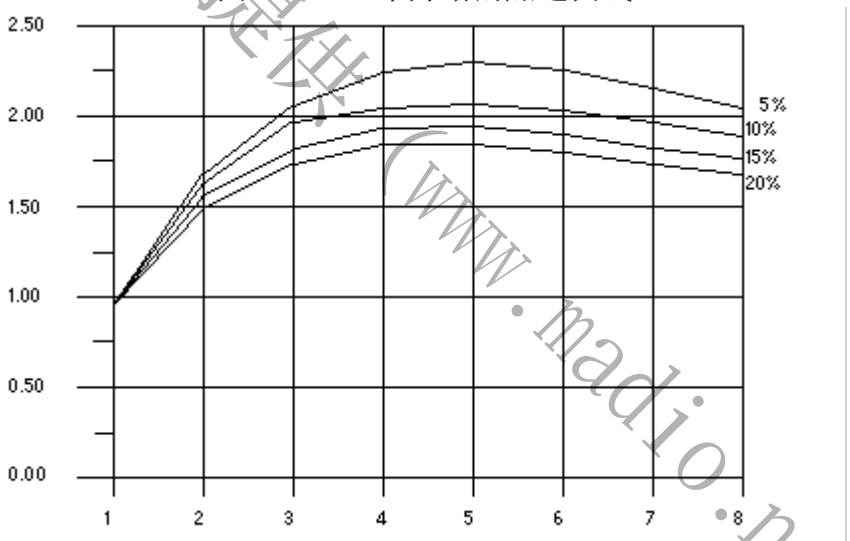
若考虑开销问题，将开销加入此模型中，加速曲线会发生很大的变化。为了方便说明，我们假定开销包括以下两部分：独立于处理器数的固定部分，以及随使用的处理器数呈二次增长的非固定部分：

$$\frac{1}{S}=\frac{1}{F+(1-\frac{F}{P})+K_1+K_2P^2}$$

1 除以 S 等于 1 除以下列数值之和：F 加上左括号 1 减去 F 除以 P 右括号加上 K（下标 1）加上 K（下标 2）乘以 P 的平方。

在此方程式中，K_1 和 K_2 是固定因子。在这些假定下，加速曲线如下图所示。值得注意的是，在此情况下，加速达到峰值。在某个点之后，增加更多处理器会降低性能，如下图 2-3 所示。

图 2 - 3 带开销的加速曲线



此图显示：使用 5 个处理器时所有程序到达最高加速，然后添加到多达 8 个处理器时即会失去此加速优势。x 轴表示处理器的数目，y 轴表示加速。

4、负载平衡的应对策略

对于运算量较小的软件，即使放到单核 CPU 上运行速度也很快，负载平衡做得差一些并没有太大影响，实际中负载平衡要考虑的是大运算量和规模很大的软件，这些软件需要在多核上进行负载平衡才能较好地利用多核来提高性能。

三．问题的分析

从题目的要求中，知道了题目要求串行算法的并行化处理及对算法做出优化处理，以提高计算机执行效率，即将一个常规的串行程序进行分段，产生并行子程序，分别放入双核 CPU 中运行，使 CPU 尽量发挥双核的计算能力，并要考虑到对做出的算法有化，达到提高计算效率的目的。在串行程序并行化及分配 CPU 的过程中，我们必须解决以下四个问题；

1、 如何将一个串行程序划分成并行子程序

2、 划分所产生的并行执行的任务，一般而言都不能完全独立执行，一个任务中的计算可能需要用到另一个任务中的数据，从而就产生了通信要求。所谓通信，就是为了进行并行运算，诸任务之间所需进行的数据传输。

3、 在设计的第一和第二阶段，划分了任务并考虑了任务间的通信，不过所得到的算法仍是抽象的，因为并未考虑它在任何特定的并行机上的执行效率，在第三阶段，即组合阶段，将从抽象转到具体，即重新考察在划分和通信阶段所做的抉择，力图得到一个在某一类并行机上能有效执行的并行算法。

4、对划分的子程序，如何分配两个 CPU 上,使 CPU 负载均衡，以达到总的运输效率较高即负载均衡效率最大，即下面所说的映射问题。开发映射的主要目的是减少算法的总的执行时间，其策略有二：一是把那些能够并发执行的任务放在不同的处理器上以增强并行度；二是把那些需频繁通信的任务置于同一个处理器上以提高局部性。这两个策略有时会冲突，这就需要权衡。

对于问题 1，这里我们采用快速傅立叶变换算发来实现串行程序并行化。

对于问题 2，在域分解中，通常难以确定通信要求，因为将数据划分为不相交的子集并未充分考虑在数据上执行操作时可能产生的数据交换，在功能分解时，通常容易确定通信要求，因为并行算法中诸任务之间的数据流就相应于数据要求。我们可采用局部/全局通信的策略等方法。

对于问题 3，用增加计算和通信的粒度的办法可减少通信成本，组合时要保持足够的灵活性同时要减少软件工程代价。这几个相互矛盾的准则在组合阶段要仔细考虑。增加粒度

对于问题 4 对于两个处理器而言，映射问题有最佳解；当处理器的数目大于等于 3 时，此问题是 NP 完全问题，但我们可以利用关于特定策略的知识，用启发式算法来获得有效的解。在进行映射时，对于许多用域分解技术开发的算法，有固定数目的等尺寸任务，有结构化的局部/全局通信，此时映射很简单；对于更复杂的域分解算法，每个任务的工作量可能不一样，通信也许是非结构化的，有效的组合就不那么容易，此时可能要采用负载平衡算法。在基于功能分解的算法中，常常会产生一些由短暂任务组成的计算，他们只在执行的开始与结束是需与别的任务协调，此时我们可以用任务调度（Task-Scheduling）算法来分配任务个那些可能处于空闲状态的处理器。

报名号　#　1418

四．模型的建立与求解

（一）傅立叶模型及算法描述

1）快速傅里叶变换（FFT）模型

一维离散序列的傅立叶变换是指 $y_k=\sum_{n=0}^{N-1}x_n w_N^{kn}$,k=0,1,…, N−1, 其中 y_k ,x_n 均为长为 N 的复序列，$w_N=e^{-2\pi i/N}$ 为 N 次单位原根

也可以记作：Y=F_NX。其中

X=$(x_1,x_2,\ldots,x_n)^T$

Y=$(y_1,y_2,\ldots,y_n)^T$

$$F_{N=}\begin{Bmatrix}1 & 1 & \cdots & 1\\ 1 & w_n & \cdots & w_N^{N-1}\\ . & \vdots & & \vdots\\ 1 & w_N^{N-1} & \cdots & w_N^{(N-1)^2}\end{Bmatrix}$$

快速傅立叶变换的基本思想是分而治之，不妨假设 N 为 2 的幂，令：

$$U=F_{N/2}\begin{pmatrix}x_0\\ x_2\\ \vdots\\ x_{N-2}\end{pmatrix},V=F_{N/2}\begin{pmatrix}x_1\\ x_3\\ \vdots\\ x_{N-1}\end{pmatrix}$$

其中 $F_{N/2}=(\omega_{N/2}^{i\times j})_{i,j},i,j=0,1,2,\cdots,N/2-1$，而 $\omega_{N/2}=\omega_N^2$ 为 N/2 次单位原根。

因为：

$$y_i=\sum_{n=0}^{N-1}x_n\omega_N^{ni}=\sum_{n=0}^{N/2-1}x_{2n}\omega_N^{2ni}+\sum_{n=0}^{N/2-1}x_{(2n+1)}\omega_N^{(2n+1)i}$$

$$=\sum_{n=0}^{N/2-1}x_{2n}\omega_{N/2}^{ni}+\omega_N^i\sum_{n=0}^{N/2-1}x_{(2n+1)}\omega_{N/2}^{ni}$$

因此我们有：

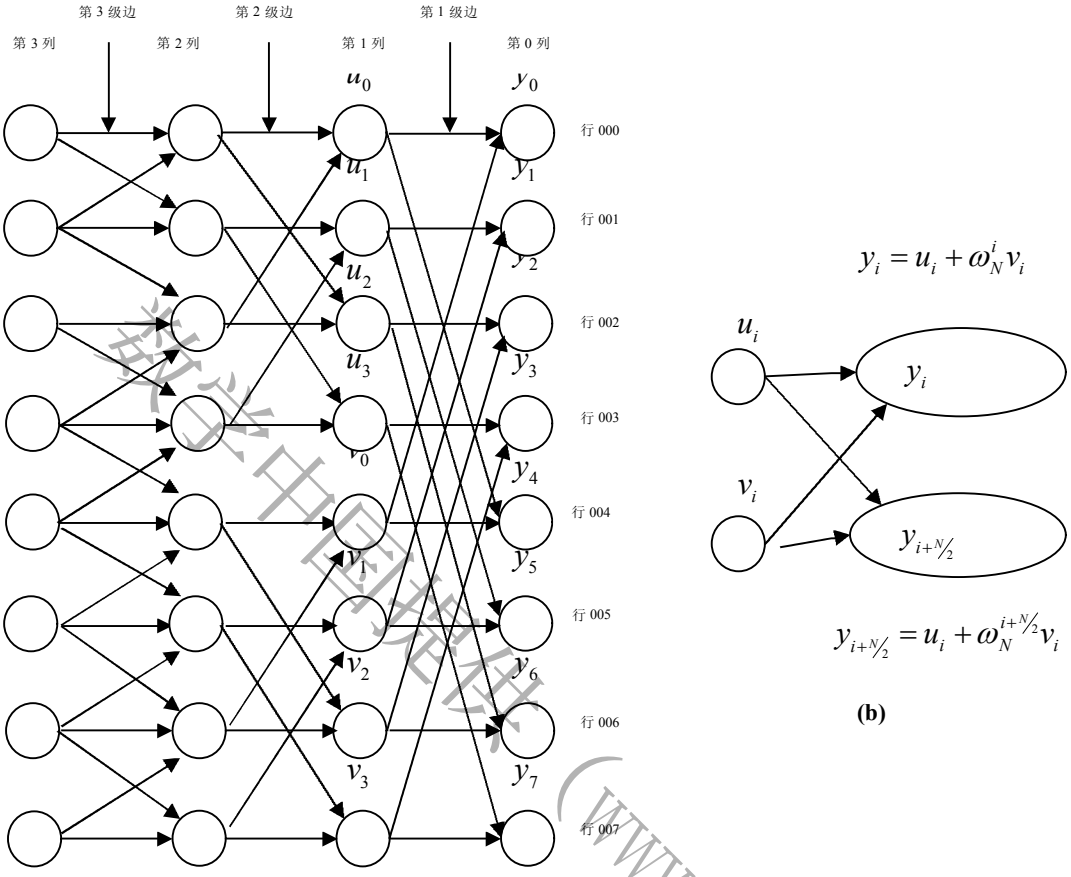
$$y_i=\begin{cases}u_i+\omega_N^i\upsilon_i(0\le i<N/2)\\ u_{i-N/2}+\omega_N^i\upsilon_{i-N/2}(N/2\le i<N)\end{cases}$$

其中 u_i,v_i 为向量 U 和 V 的第 i 个分量，它是一种递推计算格式，因向量 U,V 实际上是两个输入都是 N/2 的 DFT 问题，又可采用上述格式递推求解。这就是快速傅立叶变换（FFT），其总计算量已经证明为 O(NlogN).

FFT 的基-2 并行计算过程如图 4-1 所示。利用蝶网的第一级边，只需一步便可计算出向量 Y。如图 4-1b 所示，第 0 列的节点{i,0}（0≦i<N/2）将接受直线边传来的数据 u_i 与交叉边传来的数据 v_i，计算

$y_i=u_i+w_N^i\ v_i$(设 w_N^i 已存储在其内部)；节点{i,0}（N/2≦i≦N-1）将接受直线边传来的数据 $v_{i-N/2}$ 与交叉边传来的数据 $u_{i-N/2}$，计算 $y_i=u_{i-N/2}+w_N^i\ v_{i-N/2}$,即可得到向量 Y。

以此类推，向量 U,V 又可利用第二列的蝶网边进行计算，直至第 logN 层变为用原始向量 X 计算。可见，经过 logN 个并行步后，即可以计算出一个 N 点一维 FFT。相对于串行 FFT 的 O(NlogN)计算时间来说，已经到了线性加速比，是一个非常高效的并行算法。



(a)　　(b)

图 4-1 快速傅里叶变化并行计算蝶网示意图

2）并行算法描述

并行算法采用主进程/子进程模型，基于上述算法思想，具体算法描述如下。

（1） 主进程的算法描述

主进程的算法要点描述

主进程算法的伪码表示如下：

主进程（）

{

1）接受用户输入参数，即数据量的大小 DataSize，和并行计算的节点数目 TaskNum；

2）根据输入参数 DataSize，程序产生 DataSize 大小的随机数（供算法测试使用，也可接受用户的输入数据）；

3）进行初始化，产生 TaskNum 个进程，分布到各个节点处理机上运行；

4）传送初始化参数给各个子进程，即 TaskNum，DataSize 和各个子任务的 Tid 号；

5）将产生的随机数（输入数据）分块，传送给各个子任务作为运算的数据源；

6）等待接收子进程传送来的结果数据；

7）等待善后处理，程序结束跳出。

}

（2） 子进程的算法描述

子进程算法的要点描述

子进程算法的伪码表示如下：

子进程（）

{

1） 接受主进程传来的初始化参数，即 TaskNum，DataSize 和各个子任务的 Tid 号；

2） 接收该子任务 计算的随机数据块（输入数据块）；

3） 按照前面描述的算法思想，若令： Ptask_Point=DataSize/TaskNum, 则先对 Ptask_Point 个数据进行串行计算，等待进入并行计算的第一个阶段；

4） 对并行层进行循环{

a) 找到要进行两两数据交换的子任务，合成一个组；

b) 在这个组内进行数据交换。每个子任务在数据互交过程中，先发送数据给对方，再接收对方发送来的数据；

c) 利用自己已有的数据和同组成员发送来的数据根据 FFT 计算方法，得出结果数据；

d) 解散已形成的组；

}//并行层循环结束

5） 将子进程计算的结果数据发送个主进程；

6） 退出。

}

（二） 通信

在讨论通信时，将通信分成以下四种模式：

1 局部/全局通信：局部通信时，每个任务只与较少的几个近邻通信；全局通信中，每个任务与很多别的任务通信。

2 结构化/非结构化通信：结构化通信时，一个人物和其近邻形成规整结构（如树、网格等）；非结构化通信中，通信网则可能是任意图。

3 静态/动态通信：静态通信，通信伙伴的身份不随时间改变，动态通信中，通信伙伴的身份则可能由运行时所计算的数据决定且是可变的。

4 同步/异步通信：同步通信时，接收方和发送方协同操作；异步通信时，接收方获取数据无需与发送方协同。

点到点通信开销的测量 对于点到点的通信，测量开销使用**乒-乓方法**：节点 0 发送 m 个字节给节点 1；节点 1 从节点 0 接受 m 个字节后，立即将消息发送回节点 0。总的时间除以 2，即可得到点到点通信时间，也就是执行单一发送或接收操作的时间。用乒-乓方式测量延迟的代码见附录：

乒-乓方法可一般化为热土豆法，也称为救火队法：节点 0 发送 m 个字节直接点 1，节点 1 再将其发送给节点 2，以此类推，最后节点 n-1 再将其返回给节点 0，最后时间除以 n 即可。

点到点通信开销的表达式 Hockney 对于点到点的通信，给出了如下所示的通信开销 t(m)的解析表达式，它是消息长度为 m（字节）的线性函数：

$$t(m)=t_0+m/r_\infty$$

其中，t_0 是通信启动时间(μs)；r_∞ 是渐进带宽(Mb/s)，表示传送无限长的信息时的通信速率。Hockney 也同时引入了两个附加参数：半峰值长度 $m_{1/2}$（字节），表示达到一半渐进带宽（即 $\frac{1}{2}r_\infty$）所需要的消息长度；特定性能 π_0(Mb/s),表示短消息带宽。4 个参数 t_0、r_∞、$m_{1/2}$、π_0 中只有两个是独立的，其他两个可使用如下关系式导出：

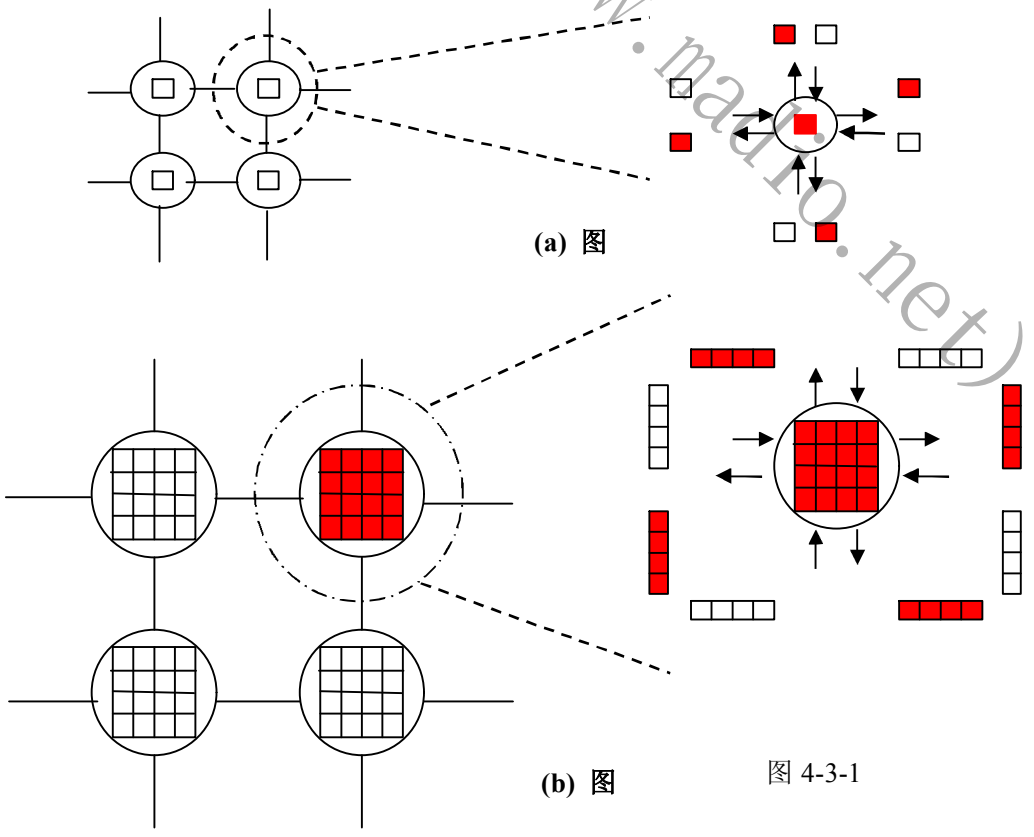
$$t_0 = m_{1/2}/r_\infty = 1/\pi_0$$

（三） 组合

组合的母系是通过合并小尺寸的任务来减少任务书，但它仍可能多于处理器的数目，理想的情况是，在组合是就将任务数减少到恰好每个处理器上一个，从而得到一个 SPMD 的程序，这是映射也宣告完成，另外，组合时我们也要考虑是否值得进行数据和/或计算的重复（Replication）。

在设计过程的划分阶段，致力于定义尽可能多的任务以增大并行执行的机会。但是定义大量的细粒度任务不一定能产生一个有效的并行算法，因为大量细粒度任务有可能增加通信代价和任务创建代价。

表面-容积效应（Surface-to-Volume Effects）如果每个任务的通信伙伴是少的，则增加划分粒度常能减少通信次数，同时还能减少总通信量参照图来阐述所谓“表面-容积效应”：一个任务的通信需求比例于它所操作的子域的表面积，而计算需求却比例于子域的容积。在一个二维问题中，“表面积”比例于问题的尺寸，而“容积”比例于问题尺寸的平方。因此一个计算单元的通信/计算之比随问题尺寸的增加而减少。



(a) 图

(b) 图

图 4-3-1

图 4-3-1（a）（b）分别示出了细粒度和粗粒度的二维网络上 5 点法计算有限差分：在（a）中，2*2 的网格上，计算被划分成 2*2=4 个任务，每个任务负责计算一个格点；而在（b）中，统一计算问题被划分成 2*2=4 个任务，每个任务负责 16 个格点的计算。在（a）中，共需要 4*4=16 次通信，每个任务通信 4 次，总共传输 16 个数据值；而在（b）中，金要求 4*4=16 次通信，总共传输 16*4=64 个数据值，可见同一计算问题，粗粒度划分的通信次数和通信量均比细粒度划分时有所下降。

表面-容积效应启发我们，在其他条件等同的情况下，高位分解一般更有效。因为相对于一个给定的容积它减少了表面积。因此从效率角度，增加粒度的最好办法是在所有的维组合任务。

重复计算（Replication Computation）它也成为冗余计算。有时候可以采用不必要的多余的计算的方法来减少通信要求和/或执行时间。仍以二叉树上的求和 $\sum_{i=0}^{N-1} x_i$ 为例来说明之。假定在二叉树上用 N 个处理器来求 N 个数的全和，且要求每个处理器上均保持最终的全和。为此如图 4-3-2 所示，必须先自叶向根逐级求和；然后自根向叶将全和逐级播送给每个处理器，供需 2logN 步。

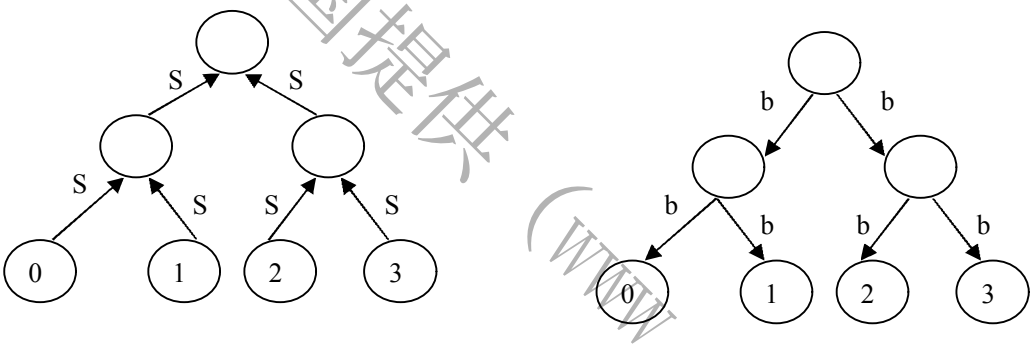


图 4-3-2　　二叉树上求全和

以上述方式求和是，处理器的利用率是逐级减半的，如果在属的每一级都充分利用所有的处理器参与计算，即在树的每一级，每个处理器（任务）均接收两个数据，执行一次局部求和，然后在发送结果至下一级的两个处理器，那么经过 logN 级后，每个处理器中均积累了全和 $\sum_{i=0}^{N-1} x_i$，如图 4-3-3 所示，在实际上就是蝶式通信结构，他利用了重复计算的方法，只需 logN 步就可在个处理器中积累了最终的全和，但却以总共施行 O（NlogN）次运算和通信操作为代价。

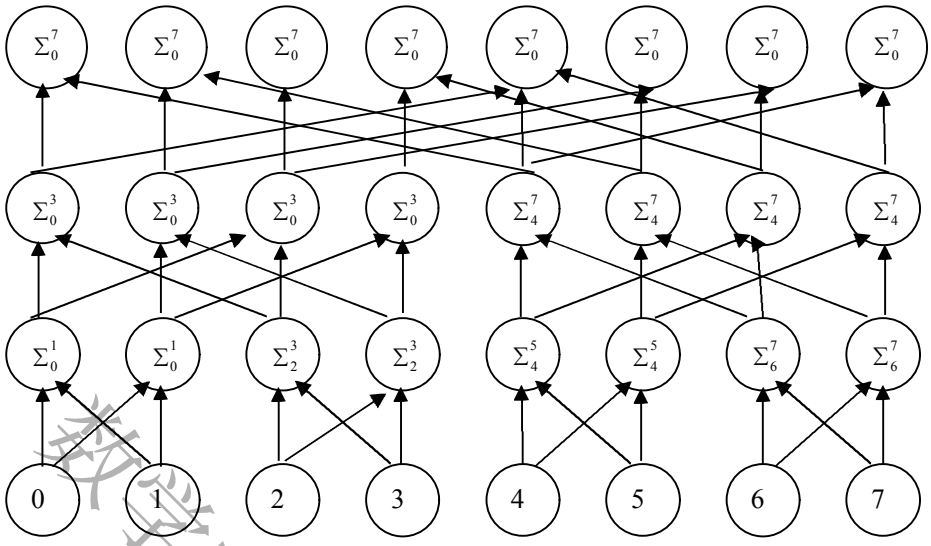


图 4-3-3 使用蝶式通信结构求全和

当对通信需求分析揭示了一组任务不能同时执行时，组合总是有益的，例如，用图所示二叉树和图所示蝶式图进行求和时，只有在树的和蝶式图的同一级中的任务方能同时执行，这样不同级中的任务可组合到一起而不会减少并行执行的机会，同时也可能产生优化的通信结构。

（四） 映射

在设计的最后阶段，我们要指定每个任务到哪里去执行，此即映射（Mapping）。

负载平衡算法：

基于域分解技术的算法有很多专用和通用的负载平衡技术，它们都是试图将划分阶段产生的细微粒任务组合成一个处理器一个粗粒度的任务。下面简单介绍几种有代表性的方法。

递归对刨：递归对刨技术用来将一个域划分成计算成本大致相等的子域，同时试图使通信代价最小，为此常使用分治方法：域首先在一维方向上分割成两个子域；然后分割以递归的方式在两个子域中进行，直至子域数和所要求的任务数一样多。

局部算法　上述描述的技术是很昂贵的，因为它们都要求计算状态的全局知识。相反，局部负载平衡算法，只是使用邻近处理器的负荷信息，来周期性地调整自己的负载，或转移到邻居，或从邻居迁入。例如，对于下图 4-4-1 所示的网格问题，每个处理器周期地与周围的处理器比较它们的计算负载，如果它们的差异超过了某个阈值就施行计算负载的迁移

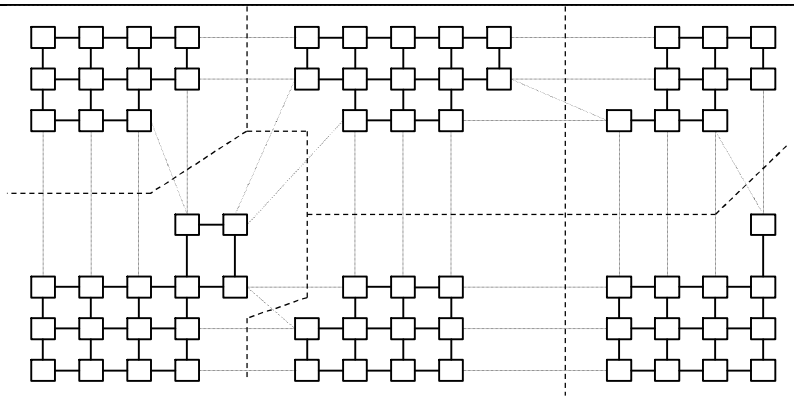


图 4-4-1 网格问题中的负载平衡

循环映射　如果知道每个格点计算负荷是不变化的，且呈现明显的空间局部性，则我们可以使用所谓的循环指派法，即采用某种枚举方法，轮流地将各处理器分配给诸计算任务。这种方法牺牲了负载平衡，但牺牲了局部性且通信可能会增加。

任务调度算法：

任务调度算法最关键之处是任务分配策略。策略的选择应在独立运算和计算状态的全局知识之间折中，常用的调度模式有经理/雇员方式和非集中方式，在非集中调度方案中，怎样检测整个问题的求解是否已经完成也是一个必须考虑的问题。

经理/雇员模式　如下图 4-4-2 所示，中心任务负责任务分配，每个雇员重复地从经理那里请求并执行具体任务。此策略的有效性依赖于雇员数和执行任务的成本，使用预取方法和换村方法可以改善效率。这种方案的一种变体是层次经理/雇员模式，它将诸雇员分成不相交的集合，每一个都有一个小经理，雇员们从小经理那里领取任务，并周期地与经理和小经理施行通信。

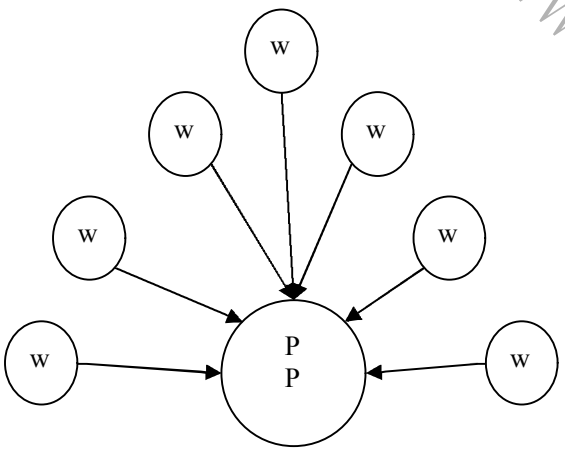


图 4-4-2 经理/雇员调度模式

非集中模式　它也就是无中心管理者的分布式调度法。在每个处理器中均维持一个任务池，这些任务池实际上就变成了可供请求者异步访问的分布的数据结构。

结束检测　任务调度算法需要一种机构检测何时操作结束，否则，空闲的雇员们将永不停止地发出工作请求，这种检测机构在集中式算法中容易实现，因为经理能容易地决定何时雇员们都空闲了；在分布式算法中则比较困难，因为没有一个中央机构记录雇员的空闲状况

（五）平衡负载算法讨论

在动态负载平衡策略中，比较有代表性的算法是轻载结点请求算法和重载结点请求算法。在嵌入式多处理机系统中，一般情况下，根据系统当前的负载情况选用其中之一，可以有效地平衡负载；但是，当系统负载发生变化后，可能会由于原先选用的算法不合适而导致附加开销陡增，并且无法有效地平衡负载。因此，考虑到嵌入式系统本身的特点(例如资源有限等)，轻载结点请求算法和重载结点请求算法不加改进而直接用于嵌入式多处理机系统是不合适的。综合这两种算法的优缺点，就有了双向请求算法。

（1） 轻载结点请求算法

轻负载结点会尝试向重载结点请求任务，每个结点都定义了相关域，相关域的定义是把所有与之相邻的结点作为相关域成员。结点只与其相关域中的结点进行交互和任务传递。如果请求到任务，则中止请求，否则就继续询问下一个相邻结点。

启动时，所有结点几乎都是轻载结点。经过一段时间以后，结点开始检查自身是否仍是轻载结点，如果仍是，就试图在相关域中找出重载结点，并请求该结点上的任务。具体来说，设该轻载结点的负载为 WORK(p)，相关域中有 k 个结点 WORK(a+1)、WORK(a+2)……WORK(a+k)，则该部分的平均负载 Wavg′为：

$$\mathrm{Wavg}' = \left[\mathrm{WORK}(p) + \sum_{a+1}^{a+k}\mathrm{WORK}(i)\right]\cdot\frac{1}{k+1}$$

为达到任务的均匀分布，应求得相关域中重载结点应该传递给该结点的负载量(设为 Mk)，但是必须对每个结点引入阈值 H(j)，以避免任务从负载更轻的轻载结点迁移过来。若 WORK(j)>Wavg′，则 H(j)一 WORK(j)一 Wavg′；否则，H(j)=0。

$$M_k = \frac{[\mathrm{Wavg}' - \mathrm{WORK}(p)]\cdot H(j)}{\sum_{1}^{k} H(j)}$$

然后，该结点就可以按照计算出的 Mk 值，向各个相关结点发出接收任务请求。轻载结点请求算法流程如下：

①判断自己是否为轻载结点；

②如果是，则找出附近的重载结点，并对它发出任务请求；

③接收到请求算法的重载结点向该轻载结点发送任务

在系统整体负载较轻时，使用轻载结点请求算法反而会造成较大的额外开销，不利于系统的整体性能。因此，轻载结点请求算法适合在整个系统负载较重时使用。

（2） 重载结点请求算法

重负载结点会尝试向轻载结点发送任务，至于发送任务给哪个结点，则取决于该结点相关域中结点的负载状态。因此，该策略需要交换处理器的负载信息。一个结点有多种方法向邻接结点通知它的负载情况，例如定期询问、当任务数发生变化时、接收到执行任务请求时、响应请求或者当任务数超过一定阈值时等。

启动一段时间以后，各结点开始检查自身是否是重载结点，如果是，就试图在相关域中均匀地分布任务。与轻载结点请求算法类似，首先计算域内的平均负载 Wavg′，然后计算所要转移的任务量 Mk。

设该重载结点的负载为 WORK(p)，相关域中有 k 个结点 WORK(a+1)、WORK(a+2)……WORK(a+k)，则该部分的平均负载 Wavg′为：

$$\mathrm{Wavg}' = \left[\mathrm{WORK}(p) + \sum_{a+1}^{a+k}\mathrm{WORK}(i)\right]\cdot\frac{1}{k+1}$$

对每个结点引入阈值 H(j)，以避免任务从负载更轻的轻载结点迁移过来。若

WORK(j)>Wavg′，则 H(j)=WORK(j)一 wavg′；否则，H(j)=0。则 Mk 的值为：

$$M_k = \frac{[\mathrm{WORK}(p) - \mathrm{Wavg}'] \cdot H(j)}{\sum_{1}^{k} H(j)}$$

然后该结点就可以按照计算出的 Mk 值向各个相关结点发送任务。

重载结点请求算法流程如下：

①判断自己是否为重载结点；

②如果是，则找出附近的轻载结点，并对它发出任务转移请求；

③得到同意后，重载结点向该轻载结点发送任务。

系统整体负载较重时，如果使用重载结点请求算法，那么重载结点在自身已经高负荷的情况下，还要负担额外的处理负载平衡调度的负担，发出任务转移请求。由于重载结点数目较多，多数任务转移请求无法得到满足，重负载结点会在将来继续发出请求，这些请求反而会造成较大的额外开销。因此，重载结点请求算法适合在整个系统负载较轻时采用。

（3） 双向请求算法

一个需要解决的问题是：怎么样判断系统负载的轻与重，即怎样决定何时使用重载结点请求算法，何时使用轻载结点请求算法。这是非常关键的，如果解决得不恰当，那么双向请求算法就不是结合重载结点请求算法与轻载结点请求算法的优点，而是结合了二者的缺点。

（4） 双向请求算法的改进

改进算法中，每个结点记录其相关域中其他结点的状态信息，它维护 2 个集合，分别是轻载集θ和重载集φ。轻载集中保存的是其相关域中轻载结点的信息，而重载集中保存的是其相关域中重载结点的信息。每当结点状态发生变化时，发消息给相关域中的各结点，各结点相应地改变其轻载集和重载集。

比较两个集合的大小来决定采用重载结点请求算法还是轻载结点请求算法。当系统处于重负载时，会有大量的重负载结点，因而轻载集较小，而重载集较大，那么就采用轻载结点请求算法，在轻载集中找到接收者，由接收者主动申请结点的任务；当系统处于轻负载时，会有大量的轻负载结点，因而重载集较小，而轻载集较大，那么就采用重载结点请求算法，在重载集中找到发送者，由发送者主动迁移任务给结点。

各结点的状态分为 R(轻载，即任务接收者)和 S(重载，即任务发送者)，阈值记为 Mk。系统刚启动时，各结点负载都比较轻(即均为 R)，因此，重载集合为空，轻载集合则等于结点全集。当产生新任务时，只要结点负载不超过阈值 Mk，这个新任务就在本地运行，结点状态仍旧是 R。此时的系统处于低负载，使用重
载结点请求算法。随着一个个新任务的到来，结点负载增大，当超过阈值 Mk 时，结点状态变为 S，并通知其他结点改变它们所维护的重载集与轻载集。

然后，比较结点轻载集和重载集的大小：若重载集小于轻载集，则继续采用重载结点请求算法，按重载结点请求算法遍历其轻载集中的结点，找出最合适执行新产生任务的结点，并发送任务；若重载集大于轻载集，则改用轻载结点请求算法，按轻载结点请求算法遍历重载集中的结点，并发送请求任务的信号

图 5-5-1 为改进的双向请求算法流程。

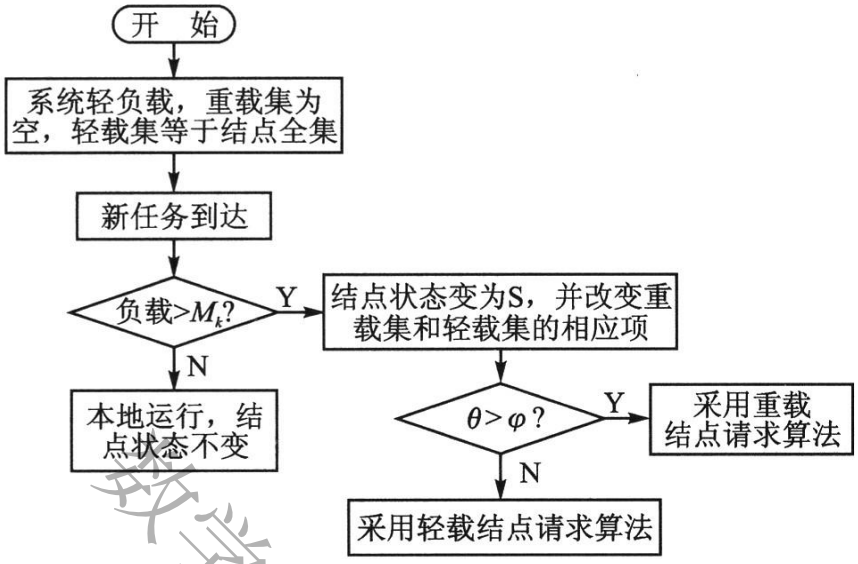


图 5-5-1 改进的双向请求算法流程

要实现任务的再次分配，一般是采取进程迁移的方式。但是进程迁移开销较大，而且选择可迁移进程的标准和策略是实现动态抢先式负载平衡的关键问题，若选择了不该迁移的进程进行迁移，则可能会抵消负载平衡所带来的性能的改善。

定义 2 进程从开始执行到最终结束所花费的 CPU 周期数称为“进程生存周期数”，进程当前已经耗费掉的 CPU 周期数称为“进程已生存周期数”。

最简单的方法是选择最新生成的任务，导致处理器工作负载超出门限值，这些任务相对来说迁移的代价不大。也可以选择已运行的任务，然而，可能的结果是迁移运行任务的代价抵消了作业运行时间的减少。因此，选择生存期长、已生存周期数较少的进程更有利，可以使迁移开销有时间得以补偿。在本模型中，选择前一种迁移策略。

仿真测试基于卡内基·梅隆大学的负载平衡测试框架，设置了 5 个结点。输入具有代表性的任务集之后，分别在系统负载较轻、较重和正常的情况下进行仿真测试。每个结点的剩余负载能力不同，分别记为：20，90，30，20，40。不妨假设，在负载平衡前，负载是平均分配到 5 个结点上的，使用本文中的策略进行负载平衡后，剩余负载能力较强的结点将负担更多的负载。由于篇幅所限，这里只能列出部分测试结果，分别如图 5-5-2~图 5-5-4 所示。

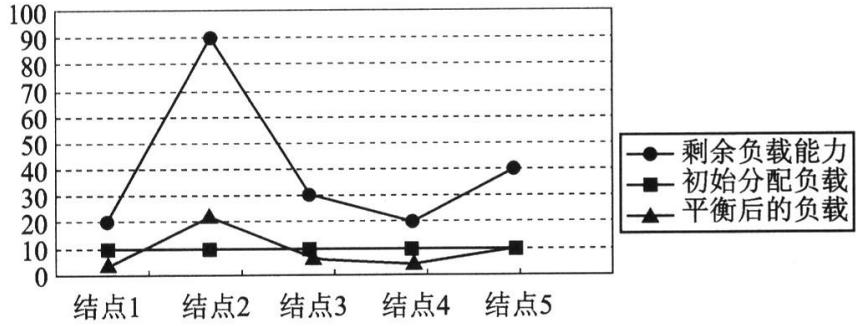


图 5-5-2 负载较轻时（负载指数 50）

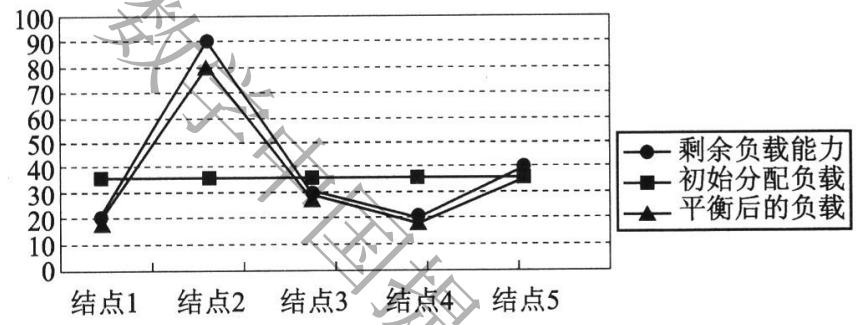


图 5-5-3 负载较重时（负载指数 180）

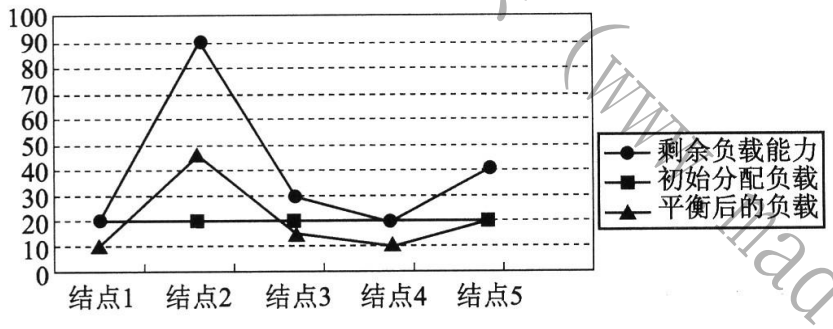


图 5-5-4 正常负载时（负载指数 100）

五．模型分析与结果检验

5.1 模型分析

（一）FFT 并行求解过程分析

如前分析，群集系统中，对 FFT 的算法设计就是将 FFT 分块处理，例如：数据量为 8，节点机为 4，每个节点处理 2 个数据，如图 5-1 所示。一般地，若数据量大小为 Datasize；并行处理机节点数为 TaskNum，则在 $\log_2^{DataSize}$ 至 $\log_2^{TaskNum}$ 列都是单机的穿行处理，从 $\log_2^{TaskNum}$ 至 0 列都是并行处理，各子任务之间进行任务通讯，这样完成 FFT 并行计算。

在并行处理第一阶段，Task1 和 Task2 交换数据，Task3 和 Task4 交换数据，进行 FFT 计算。这样递归计算，直到把结果数据得出。

现采用主进程/子进程模型，主进程读入数据，将数据整理，分成 TasNum 块，每块

数据传给处理所属的子进程，等到每个子进程完成任务，将结果返回给主进程。在这其中，不但有主进程和子进程之间的数据传递，且子进程之间也要进行数据传递，且在每一个阶段，都要进行同步，才可以进行下一步。

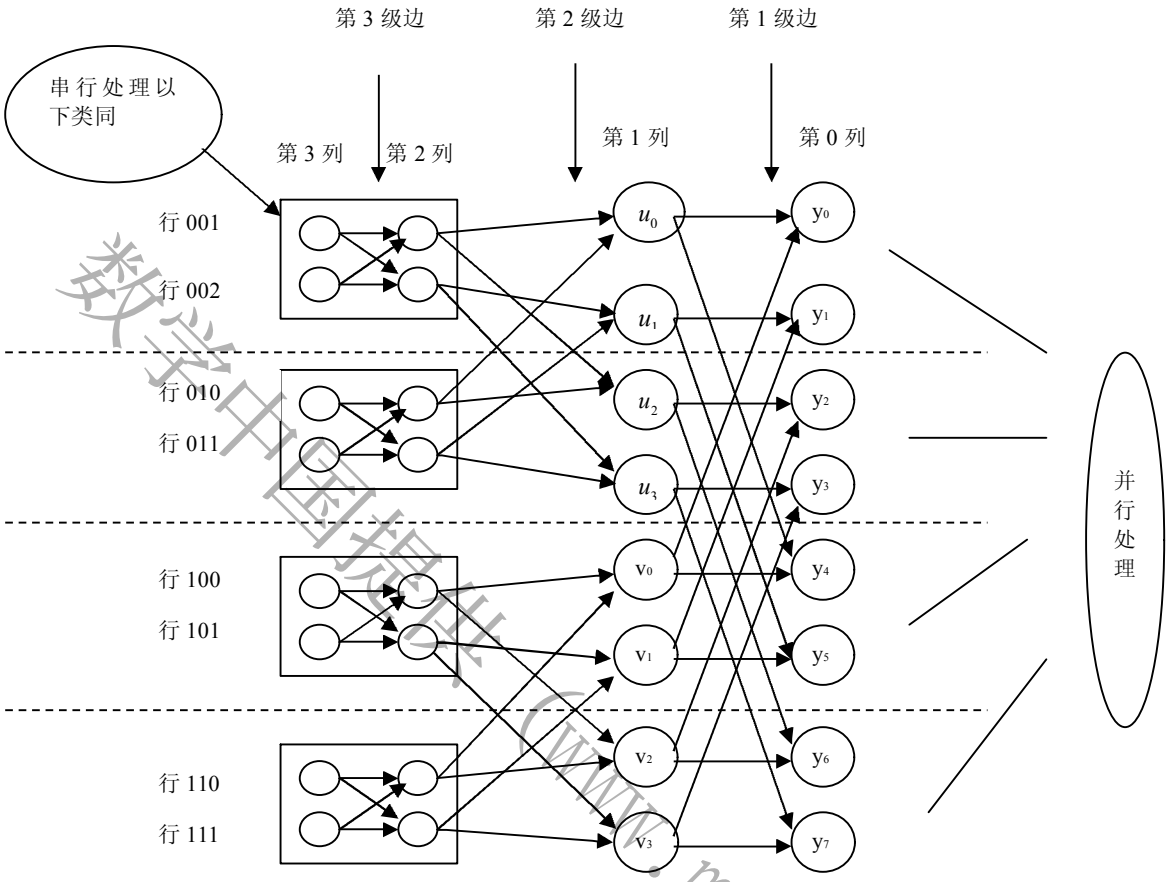


图 5-1 快速傅立叶变换在集群系统中的算法设计

（二）算法分析

假设消息启动时间和单位数据的消息传递时间分别为 $t_\alpha + Nt_\beta$。

假设节点机数为 p。

记 FFT 的串行计算时间为 T_s，P 台节点机的并行时间为 T_p,且记其中的计算时间和通信时间分别为 T_{comp},T_{comn}。

（1） 串行算法分析

假设有 n 个数据，计算一个数据的时间是 t_{elem}

FFT 算法模型层数为：LevelNum=$\log_2^n$

每层共有 n 个数据，则，FFT 串行计算的时间为：

$T_s = t_{elem} n \log_2^n$

(2)并行算法分析

假设设计的并行算法达到了较好的负载平衡，各子任务分别计算 FFT 模型中的一块，各个子任务的计算时间为一部分的串行计算时间加上并行算法计算时间。

假设有 n 个数据，处理机数为 p。仍然假设计算一个数据的时间是 t_{elem}。

设串行部分计算时间为 t_1，并行部分计算时间为 t_2。

串行计算的层数为：$\log_2^n - \log_2^p$

每层有 n/p 个数据，所以：

$t_1 = t_{elem}\, n(\log_2^n - \log_2^p)/p$

每层计算的层数为：$\log_2^p$

每层 n/p 个数据，所以：

$t_2 = t_{elem}\, n\log_2^p /p$

$T_{comp} = t_1 + t_2 = t_{elem}\, n\log_2^n /p$

下面分析并行算法的通信时间。主任务要有 n 个数据传送给子任务，且每儿歌子任务之间要进行 n 个数据交换以及主任务要接收来自子任务的结果数据。

所以：

$T_{comn} = p(t_\alpha + t_\beta /p) + p(t_\alpha + t_\beta n/p)\log_2^p + p(t_\alpha + t_\beta n/p) = p(t_\alpha + t_\beta n/p)(2+\log_2^p)$

因此，最终计算得：

$T_p = T_{comn} + T_{comp} = t_{elem}\log_2^n /p + p(t_\alpha + t_\beta n/p)(2+\log_2^p)$

进一步，根据 $S = T_s / T_p$ 及 $E = S/p$ 计算加速比和效率有

$$S = T_s / T_p = \frac{t_{elem} n \log_2^n}{t_{elem}\log_2^n /p + p(t_\alpha + t_\beta n/p)(2+\log_2^p)}$$

$$E = S/p = \frac{1}{p}\,\frac{t_{elem} n \log_2^n}{t_{elem}\log_2^n /p + p(t_\alpha + t_\beta n/p)(2+\log_2^p)}$$

由最后 E 和 S 的表达式可得出：

当 $n \to \infty$,p 固定时，S=O(p),E=O(1)。

这是因为，随着数据的增大，计算在并行算法中的比重将逐渐增强，从而使得加速比逐渐接近线性加速比，效率达到最优

当 n 固定，$p \to \infty$ 时，初始时 S 将增大，但当 p 增大到一定值时，S 将减小，效率 E 却始终是减小的。

5.2 模型结果检验

实验环境：在 UNIX 操作系统的环境下使用并行计算平台 PVM 对设计的算法编程实现，并测试其结果。

表 1，表 2 为 FFT 串行，并行算法在 PC 机群集中的单机模拟、多机模拟的实验结果。其中列出了子任务数 P=2 和 P=4（即子进程的个数）的单机模拟和多机模拟情况。在单机模拟的情况下，由于只有一台节点机来运行程序，所以这种情况下的并行算法的执行时间实际上大约表示为多个子任务所执行时间之和。因此，若拿单机模拟的并行运算时间和多机模拟的并行运算时间进行比较，应该将单机模拟的并行运算时间除以子任务数，即拿一个任务所需的时间来比较。

报名号　#　1418

表 5-2-1 快速傅立叶变换串行、并行在 PC 机群中的单机模拟结果

问题规模 N(2 的幂次)	串行	并行（p=2）			并行（p=4）		
	时间（s）	时间（s）	平均加速比	平均效率	时间（s）	平均加速比	平均效率
13	0.040	0.056	0.710	0.355	0.030	1.332	0.333
14	0.090	0.102	0.880	0.440	0.060	1.504	0.376
15	0.200	0.172	1.164	0.582	0.104	1.916	0.479
16	0.490	0.393	1.246	0.623	0.213	2.304	0.576
17	1.200	0.762	1.574	0.787	0.448	2.680	0.670
18	2.650	1.864	1.422	0.711	0.906	2.920	0.730
19	5.690	3.887	1.464	0.732	1.998	2.848	0.712

表 5-2-2 快速傅立叶变换串行、并行算法在 PC 机群中的多机模拟结果

问题规模 N（2 的幂次）	串行	并行（p=2）			并行（p=4）		
	时间（s）	时间（s）	平均加速比	平均效率	时间	平均加速比	平均效率
13	0.040	0.050	0.800	0.400	0.029	1.400	0.350
14	0.090	0.098	0.920	0.460	0.054	1.652	0.413
15	0.200	0.174	1.148	0.574	0.100	2.000	0.500
16	0.490	0.350	1.398	0.699	0.184	20668	0.667
17	1.200	0.740	1.623	0.811	0.383	3.132	0.783
18	2.650	1.825	1.452	0.762	0.884	3.000	0.750
19	5.690	4.622	1.231	0.616	2.018	2.820	0.705

从实验数据中，我们可以看出：

（1） 单机模拟效率高于多机模拟效率。因为，单机模拟的进程通信开销要比多机模拟的通信开销少，这样，单机模拟比多机模拟的效率要高；

（2） 随着节点机数目的增加，加速比呈增长趋势，但增加到一定程度，节点机在数目增加的同时也增加了任务之间的通信开销，加速比就呈下降趋势（由于实验条件所限，此处表格未列出更多节点机的情况），效率（加速比/节点机数）始终呈下降趋势；

（3） 随着问题规模的增加，加速比和效率呈上升趋势。由于规模的增加，问题的计算时间占总的运行时间的比重增加，而任务之间的通信时间占总的运行时间的比重相比就要减少。

这一试验结果正好与前一节理论分析的结果一致。

六．模型评价

快速傅立叶变换算法的优点：该算法具有高效、通用、可移植性，最大限度利用了机群系统配置灵活的优势。由于机群系统的以扩展性，用户可根据实际要求选择后端机个数与配置，使用户以较低的价格取得较高的性能。

快速傅立叶变换算法的缺点：算法较为简单，性能略差，最坏情况下，只有一台后端机可参与计算，算法退化成串行算法。

轻载结点请求算法的优点是：不需要相互交换负载信息；当每个结点均处于忙状态时，不会有结点启动轻载结点请求算法，几乎不需要额外调度开销；处理负载平衡问题的许多工作是由空闲结点来完成的，没有给重载结点增加太多的额外负担。

缺点是：在开始和结束阶段时任务数相对较少，大量轻载结点会不断发出任务请求，并且这些请求中的大多数无法得到满足，于是许多轻载结点会继续发出请求。最终，大量的请求增加了系统的额外开销，影响了系统整体性能，同时大量针对重载结点的任务请求会拖延它们本身任务的执行。

重载结点请求的优点是：如果没有过重负载的忙结点，就不会被空闲邻接结点所打扰这在整个系统负载较轻时显得尤为重要。

缺点是：负载过重的重载结点还要额外增加处理负载平衡调度的负担。

负载平衡调度是嵌入式多处理机系统利用处理器资源的一种有效途径，它能让多个处理单元比较平衡地共同承担一系列繁重的任务，从而大大提高了系统的吞吐率与性能。动态负载平衡问题是一个正在蓬勃发展的研究热点，还有许多未知的问题有待进一步地探索和研究。仿真结果表明，本文介绍的改进算法有效地平衡了各结点的负载，提高了整个系统资源的吞吐率与性能。该算法还有待在今后的研究工作中，通过实践的检验，找出该算法所需设置的参数(例如阈值 Mk 和 H(j))的合适值。

双向请求算法综合上面两种算法的优缺点，就有了双向请求算法。发送者和接收者都能转移任务，因此该算法兼有重载结点请求算法和轻载结点请求算法的优点。在系统负载较轻时，使用重载结点请求算法；在系统负载较重时，使用轻载结点请求算法

改进算法采用自适应算法，合理地设置判断负载的阈值，并随着每个结点的任务负荷变化，动态地改变这个评判标准，在系统负载重时采用轻载结点请求算法，在系统负载轻时采用重载结点请求算法。

七．参考文献

【1】 陈国良，《并行计算—结构，算法，编程》，高等教育出版社

【2】苏德富，《并行计算技术及其应用》，重庆大学出版社

【3】于秀敏，《基于机群的快速傅立叶变换并行算法》，《东北农业大学学报》第 36 卷第 3 期，36（3）358~360，2005 年 6 月

【4】王力生，《单片机与嵌入式系统应用》，同济大学

附录：

```
//*乒-乓法测量延迟的代码段*//
for i=0 to Runs-1 do /*发送者*/
    if(my_node_id=0) then
       temp=second()   /*second()为时标函数*/
       start_time=second()
       send an m-byte message to node 1
       receive an m-byte message from node 1
       end_time=second()
       timer_overhead=start_time-timer_overhead
       total_time=end_time-strart_time+timer_overhead
       communication_time[i]=total_time/2
    else if(my_node_id=1) then   /*接收者*/
           receive an m-byte message from node 0
           send an m-byte message to node 0
         endif
    endif
 endfor
```