RESAERCH ON THE PARALLIZATION OF TWO CLASSICAL GAME'S AI ALGORITHM BASED ON INTEL'S MULTICORE MULTIPROCESSOR

A Thesis Submitted to

Southeast University

For the Academic Degree of Master of

Engineering

BY

Zou Feng

Supervised by

Prof. YANG Quan-sheng

School of Computer Science an Engineering

Southeast University

东南大学学位论文独创性声明

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，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

研究生签名：[signature] 日期：2016.5

东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括以电子信息形式刊登）论文的全部内容或中、英文摘要等部分内容。论文的公布（包括以电子信息形式刊登）授权东南大学研究生院办理。

研究生签名：[signature] 导师签名：[signature] 日期：2016.5

摘要

目前，随着多核处理器的迅速发展，单核时代已经成为历史，尤其是由多个多核处理器组成的多处理器系统，更是为应用软件性能改善提供了硬件基础。然而我们的软件却大多停留在单核时代，并没有充分利用多核多处理器架构所带来的优势，对自身性能加以提升。对现有软件的改进以及并行程序的开发势在必行。因此与多核技术相结合，把过去串行化的程序并行化将能充分利用多核多处理器的性能优势，提高程序的效率。

现今，游戏产业也是迅猛发展，在多核多处理器架构的 PC 上应用很广泛，然而现在的很多游戏没能充分发挥多核多处理器架构的优势。本文以棋类游戏中的五子棋和球类游戏中的足球作为例子，对这两个游戏的两种经典 AI 算法——博弈树搜索算法和遗传算法进行并行化研究。

有关提高树搜索算法的性能的研究，一直以来都是学术界的热点问题，新的算法层出不穷。而研究树搜索算法最重要的实验温床是在研究如何在二人棋类游戏中找到最优解，即游戏树的搜索。自上世纪 90 年代以来，学者和研究员们对于树搜索算法的研究转向了并行化并提出了数量众多的并行树搜索算法。同时，遗传算法在人工智能中发挥了重要作用，遗传算法的并行化研究也日趋重要。我国有关遗传算法和并行计算的研究，从 20 世纪 90 年代以来一直不断地发展，特别是近年来，遗传算法和并行计算的应用在许多领域取得了很好的效果，成就瞩目。

本文首先分析了多核多处理器架构的特点，提出了线程绑定到核的并行技术。在此基础上对博弈树搜索算法中的 Alpha-Beta 剪枝搜索算法和遗传算法进行了并行化研究，并分别在五子棋游戏和足球游戏中对这两个进行了实现和验证。最后，论文对多核多处理器结构下的 PLCN 模型进行了进一步的研究，对其适应范围作了描述。

关键字： **多核多处理器，并行算法，Alpha-Beta 剪枝搜索算法，遗传算法**

Abstract

Recently, with the fast development of multi-core processor, the single-core age has become a history, and the multi-processor system composed of several multi-core processors provides hardware foundation for the improvement of software performance. However, the software which has not taken full advantage of multi-core multi-processor architecture still stays in the single-core age. It's imperative to make improvement to existing software and develop the parallel of program. So with the combination of multi-core technique, paralyzing the past serial program will take full advantage of multi-core multi-processor's performance advantages and improve the efficiency of program.

At present, the game industry is also developing fast, and has wide application on multi-core multi-processor architecture PC. However, most games have not taken full advantage of multi-core multi-processor. This paper takes the gobang and soccer ball as example, investigates the classic AI algorithm and Game tree search algorithm and genetic algorithm of the two games.

The research on improving the performance of tree search algorithm is always a hot issue of academic. The important part of tree search algorithm is the research on how to find the optimal solution of the two-people game. Since the 90's of last century, scholars and researchers shift their attention to parallelization and bring forward lots of parallel tree search algorithms. Meanwhile, the genetic algorithm weighs much in AI and its parallelization is becoming more and more important. The research on genetic algorithm and parallel algorithm in China has been developed fast, especially in recent years, genetic algorithm and parallel computing applications in many areas have every good results and catch much attention.

This paper first analyzed the characteristics of multi-core multi-processor architecture and proposed the technique of binding thread to single-core processor. It has done some parallelization research on Alpha-Beta tree of search algorithm and genetic algorithm on this basis. Finally, this paper implemented and verified the two in gobang game and soccer game.

Key Words: Multi-core and Multi-processor, Parallel algorithm, Alpha-Beta search algorithm, Genetic Algorithm

目录

第一章 绪论

1.1 研究背景及意义

多核处理器也称为片上多处理器(chip multi-processor，CMP)，或单芯片多处理器[1]。自 1996 年美国斯坦福大学首次提出片上多处理器(CMP)思想和首个多核结构原型，到 2001 年 IBM 推出第一个商用多核处理器 POWER4，再到 2005 年 Intel 和 AMD 多核处理器的大规模应用，最后到现在多核成为市场主流，多核处理器经历了十几年的发展。在这个过程中，多核处理器的应用范围已覆盖了多媒体计算、嵌入式设备、个人计算机、商用服务器和高性能计算机等众多领域，多核技术及其相关研究也迅速发展，比如多核结构设计方法、片上互连技术、可重构技术、下一代众核技术等。然而，多核处理器的技术并未成熟，多核的潜力尚未完全挖掘，仍然存在许多待研究的问题。

如何有效地利用多核技术，对于多核平台上的程序员来说是个首要问题。多核时代的到来需要软件开发者必须找出新的开发软件的方法，选择程序执行模型。在多核系统中，核心是通过内存共享数据和通信。为了充分利用多核，程序需要同时做很多事情。并行程序执行指令流的速度将比传统的串行程序要快很多，因为它能将工作负载按照不同需求分配给处理器的不同核心。由此我们可以看出在多核平台上进行程序并行化研究的必要性和重要性。

现今，游戏业蓬勃发展，游戏种类也很多包括：角色扮演，策略游戏，即时战略游戏，射击游戏，竞速游戏，体育游戏和棋类游戏等。这些游戏基本包含多个子系统，例如物理引擎、离子系统、声音混合、人工智能和渲染等。在多核多线程模式下，有些子系统可以放到单独的线程中拥有自己的数据独立执行，这些单独的线程就是所谓的模块线程，例如渲染，物理引擎等。有些可以把任务分解开，让多个线程来处理，这就是所谓的数据并行线程，这样，不同的线程可以在同一时间处理数据的不同部分，从而提高效率，通常在游戏中的人工智能算法会采用这种方式进行加速。本论文所涉及到的为体育类游戏和棋类游戏这两类游戏，它们的共同特点是都涉及到人工智能模块。因此，本文所采用的并行方法即为数据并行方法。

本论文以体育类游戏中的简单足球游戏和棋类游戏中的五子棋为例子，这两款游戏都用到了人工智能。五子棋游戏中的需要用到博弈树搜索算法，例如极大极小值算法、深度优先算法和负极大值算法等。简单足球游戏中的球员站位策略、确定传球路线、寻找最佳射门角度和障碍回避等多个方面都可以应用遗传算法。本文中的五子棋程序使用了 Alpha-Beta 剪枝搜索算法，简单足球游戏中球员站位策略使用了遗传算法，并结合多核多处理器结构进行并行化研究。

1.2 研究现状

本文以五子棋算法和足球游戏中的寻径算法作为例子，研究树搜索算法和遗传算法在多核处理器上的并行化问题，博弈树搜索算法和遗传算法并不是新算法，都有一定的发展历史。计算机博奕，简单的说，就是让计算机像人一样从事高度智能的博奕活动。研究者们从事研究的计算机博奕项目主要有国际象棋，围棋，中国象棋，五子棋，西洋跳棋，桥牌，麻将，Othello，Hearts，Backgammon，Sarabble 等[1-4]。其中，二人零和完备信息博奕的技术性和复杂性较强，是人们研究博奕的集中点。二人零和随机性研究的一个代表是 Backgammon，并且产生了很大的影响。桥牌是研究不完备信息下的推理的好方法。

高随机性的博奕项目趣味性很强，常用于娱乐和赌博，是研究对策论和决策的好例子。近代计算机博奕的研究是从四十年代后期开始的，国际象棋是影响最大、研究时间最长、投入研究精力最多的博奕项目，成为计算机博奕发展的主线。1950年C.Shannon发表了两篇有关计算机博奕的奠基性文章(Programming A Computer for Playing Chess 和 A Chess-playing Mechine)。1951 年 A.Turing 完成了一个叫做 Turochamp 的国际象棋程序，但这个程序还不能在已有的计算机上运行。1956 年 Los Alamos 实验室的研究小组研制了一个真正能够在 MANIAC-I 机器上运行的程序(不过这个程序对棋盘、棋子、规则都进行了简化)。1957 年 Bernstein 利用深度优先搜索策略，每层选七种走法展开对局树，搜索四层，他的程序在 IBM704 上操作，能在标准棋盘上下出合理的着法，是第一个完整的计算机国际象棋程序[2]。

1958 年，人工智能界的代表人物 H.A.Simon 预言[3]："计算机将在十年内赢得国际象棋比赛的世界冠军。"当然，这个预言过分乐观了。

1967 年 MIT 的 Greenblatt 等人在 PDP-6 机器上，利用软件开发工具开发的 MacHack VI 程序，参加麻省国际象棋锦标赛，创出了计算机正式击败人类选手的记录[2]。

从 1970 年起，ACM(Association for Computer Machinery)开始举办每年一度的全美计算机国际象棋大赛。从 1974 年起，三年一度的世界计算机国际象棋大赛开始举办。

1997 年，IBM 公司的超级计算机"深蓝"战胜了国际象棋世界冠军卡斯帕罗夫，成为人工智能领域的一个里程碑。

在其它的博奕项目上，1959 年 Samuel 等人利用对策理论和启发式搜索技术编制了一个西洋跳棋程序，这个程序具有学习能力，它可以在不断的对奕中改善自己的棋艺。4 年后，这个程序战胜了设计者本人。又过了 3 年，这个程序战胜了美国一个保持 8 年之久的长胜不败的冠军。这个程序向人们展示了机器学习的能力，提出了许多令人深思的社会问题和哲学问题[5]。1979 年 H.Berliner 的程序 BKG9.8 以 7 比 1 战胜了 Backgammon 游戏的世界冠军 Luigi Villa，1980 年美国西北大学 Mike Reeve 的程序 The MOOR 战胜了 Othello 世界冠军[1]。

1989 年，第一届计算机奥林匹克大赛在英国伦敦正式揭幕，计算机博奕在世界上的影响日益广泛。

与博弈算法相比，遗传算法的发展历史和研究历程都相对较晚。20 世纪 70 年代中期，美国 Michigan 大学的 Hulland 教授出版其著作"Adaptation in Natural and Anifieial systems"，为遗传算法奠定 T 理论基础。在随后的 30 年里，遗传算法不断的被改进和完善，尤其是上世纪 80 年代中期以来遗传算法得到了飞速蓬勃的发展。1985 年，在美国卡耐基·梅隆大学召开的第一届国际遗传算法会议 ICGA'85 以后每两年举行一次。现在与之并行的国际会议很多，如 Intemational Conference on Evolutional Programming、IEEE International Conference on Evolutionary Computation 等。在欧洲，"Parallel Problem solving from Nature : FPSN"国际会议自 1990 年开始在德国举行以来，在比利时和德国隔年轮流举行。另外日本新的计算机发展规划 RWC 计划(Real World Computing Program)也把遗传算法作为其主要支撑技术之一。德国 Dortmund 大学早在 1993 的一份研究报告表明，遗传算法已在 16 个大领域、250 多个小领域获得了应用遗传算法在机器学习、过程控制、经济预测、工程优化等领域取得的成功引起了数学、物理学、化学、生物学、计算机科学、社会科学等领域的极大兴趣。

我国有关遗传算法和并行计算的研究，从 20 世纪 90 年代以来一直不断地发展。特别是近年来，遗传算法和并行计算的应用在许多领域取得了很好的效果，成就瞩目。有关遗传算法和并行计算领域的论文、学术著作逐年上升。陈国良、王煦法等于 1996 年出版发行了《遗传算法及其应用》；国防科技大学周明、孙树栋的著作《遗传算法原理及其应用》对遗传算法进行了更详尽细致的介绍和探讨；2002 年，同济大学王小平、曹立明合著的《遗传算法...理论、应用与软件实现》在前两部作品的基础上将遗传算法的实现推到了软件和硬件的结合级上，并且提出了遗传算法并行化的思想[7]。在并行计算方面，陈国

良院士集十年时间完成了其著作一并行计算三部曲《并行算法实践》[8]《并行计算机体系结构》《并行算法:结构·算法·编程》[9]。随着“十一.五”计划的不断深入和国家对科研的不断投入，国内许多专家、学者也正试图将并行计算应用到遗传算法中以期更快更好更高效的解决各个领域的问题。毫无疑问，遗传算法和并行计算已经站在了当今计算机领域最前沿最高端的位置，如何将并行计算更加合理更加有效的应用到遗传算法中以期更快更好的将成果应用到人工智能自动控制、以及数学、物理、经济、生物学等各个领域必将成为科技世界中的焦点。

上个世纪 70 年代兴起的并行计算进入 2000 年以来，取得了令人瞩目的成就，虽然并行计算在 20 世纪 90 年代经历了 10 余年的低谷期，但进入新世纪后，随着计算机硬件、软件、通信网络的跨越式发展，计算密集型应用需求的增加，双核、四核乃至多核处理器的不断普及，并行计算又成为了研究热点。IBM、HP、Mrr 等著名商业和学术研究中心都在致力于并行计算的研究并取得了巨大成功，例如并行计算中共享存储的模型，包括共享存储的 SIMD 同步 PRAM 模型和共享存储的 MIMD 异步 APRAM 模型；分布式存储模型，包括分布存储的 MIMD 同步 BSP 模型和异步 LogP 模型等；分布共享存储模型，包括分布存储的 MIMD 均匀存储层次 UMH 模型和扩充 logP 存储 Memory-logP 模型以及分布式存储层次 DRAM(h)模型等。其中 PRAM 是算法界最常用的抽象模型，但不太实用；BSP 和 LogP 等能反映大规模并行机的通信特性；UMH、Memory-logP 和 DRAM(h)等能反映近代主流并行机的多层次存储特性。此外，对于松散耦合的并行系统(如基于局域网连接的 PC 机群等)，也提出了异构非独占使用方式的分时计算模型[10]。所有这些成果为我们将遗传算法并行化提供了极好的基础和条件。

目前，多核技术已经成为最受关注的话题和研究方向。多核体系结构为性能提高和节能计算等领域开辟了新的方向。然而，现在的多核处理器还没有统一的标准，基本上处于探索阶段。核与核之间的连接方式、通讯协调方式、同一处理器中核与核间结构的差异、器件资源分配策略、任务调度策略、节能策略、软硬件协同设计策略等方面都处于研究探索之中。多核必将带来影响整个计算机行业方方面面的巨大变革，包括体系结构研究、嵌入式系统设计和解决方案设计、编译技术、操作系统核心算法、应用软件设计等计算机系统的各个领域。

在单核处理器研究中，主要集中在提高频率，提高指令级并行度等方面。而在多核体系中，更加关注核与核之间的协作、共享资源的分配、提高线程级并行度等方面。

多核处理器必然带来一个问题是，需要提高程序的并行度，因为单线程程序是无法发挥多核处理器的优势的。通过编译优化可以把原先单线程的代码编译成多线程的形式。OpenMP 提供了一种方法，程序员根据需要把可以并行处理的代码加上合适的标记，编译器根据这些标记把相应代码编译成多线程的程序段。多线程程序开发涉及到多线程调试的难题，这在多核处理器上会变得更加困难，所以多核体系导致程序开发模式发生巨大变化。

如何有效地利用多核技术，对于多核平台上的应用程序员来说是个首要问题。多核时代的到来需要软件开发者必须找出新的开发软件的方法，选择程序执行模型。在多核系统中，核心是通过内存共享数据和通信。为了充分利用多核，程序需要同时做很多事情。并行程序执行指令的速度将比传统的串行程序要快很多，因为它能将工作负载按照不同需求分配给处理器的不同核心。

1.3 本文的主要工作

随着多核技术的迅猛发展，单核时代的程序已无法充分利用多核技术所带来的性能提升，同样单核时代的游戏程序无法充分利用多核计算机的性能。基于多核多处理器架构的程序并行化研究日趋重要。因此，本文基于多核多处理架构对两个经典游戏 AI 算法进行了并行化研究。

本文主要完成了以下三个方面的工作：

（1）基于多核多处理器架构提出了线程分配到每个核的算法。

（2）基于多核多处理器架构对 Alpha-Beta 剪枝搜索算法进行并行化研究，并应用于五子棋程序中。

（3）基于多核多处理器架构对遗传算法进行并行化研究，并应用于简单足球游戏中。

课题来源于教育部-英特尔信息技术专项科研基金项目“基于 Intel 多核架构的程序并行优化技术研究”（项目编号：MOE-INTEL-08-12）。本文正是在此项目下展开的研究。

1.4 论文组织

本文围绕着 Intel 多核架构，利用数据隔离和线程绑定核技术对五子棋游戏应用的 Alpha-Beta 剪枝搜索算法和足球游戏中应用的遗传算法进行并行化研究。下面简述一下本文的安排和主要研究结果。

第一章，阐述了本课题的研究背景及其意义。给出了现今博弈树技术和遗传算法的研究状况。同时，阐述了多核技术的发展情况。

第二章，介绍如今 Intel 多核多处理器的架构，以及在现今多核架构下的 OpenMP 并行技术；提出了在并行计算中采用线程绑定核的线程分配技术。

第三章，介绍了一些基本的博弈树搜索技术。提出了将 Alpha-Beta 剪枝搜索算法并行化的方法，并就如何结合 Intel 多核架构将其应用在五子棋中进行了探讨。

第四章，介绍了遗传算法的基本原理，给出了遗传算法并行化的方法。结合 Intel 多核架构，阐述了将该并行遗传算法应用在足球游戏中的实现技术。

第五章，测试与分析，验证并行算法与现今多核架构相结合所体现出的性能优势。

最后，对全文进行了总结和展望。

第二章 多核多处理器架构下的并行技术

为了便于后面论文的叙述，本章有必要对多核处理器架构和多核架构下的算法并行化技术进行介绍。

2.1 Intel 多核多处理器架构概述

Intel Xeon®系列多核多处理器属于同构多核 CPU。同构的各个 core 可以通过共享存储器互连，也可在 Cache 层面（局部存储器）互连。在连接方式上也有不同，可以通过总线连接，总线为它们通信提供协议支持；也可以各单元直接向连，这就要求在每个单元内部有负责通信的电路。这些区别完全由具体的硬件设计情况决定[11]。

2.1.1 Intel 双核处理器

图 2-1 所示为 Intel 发布的 Pentium D 内部结构图，可以清楚的看到在一个 CPU 内有两个完全相同的内核，它将两颗 Pentium 4 处理器进行改造后封装在一起，支持超线程技术。采用了独立 L2 Cache 的做法，C0 和 C1 之间用总线相连。

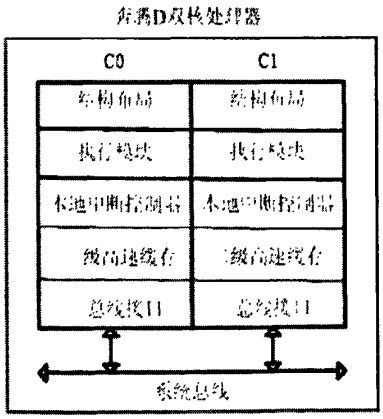

图 2-1 Pentium D 处理器内部结构图

Intel Core Duo 架构如图 2-2 所示，采用了 Smart Cache 技术，将 2 个独立的内核封装在一起。由于采用了共享 L2 Cache 的做法，有效加强了多核心架构的效率，大幅提高了 L2 Cache 的命中率，在确保 L2 Cache 的数据一致性的同时使得缓存的可用空间最大化，从而可以减少通过前端串行总线和北桥进行数据交换带来的传输代价。比起 Intel 之前采用的需要通过主板北桥芯片迂回的方法相比，不但大幅度降低了缓存数据的延迟，而且还不必占用前端总线资源，减少了不必要的延迟。此外，采用单一外部总线控制器能够有利于降低成本，简化总线控制设计。

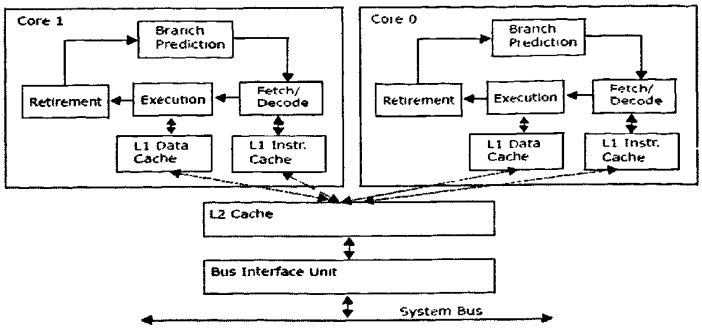

图2-2 Intel Pentium E2160处理器结构图[12]

2.1.2 Intel 双核双处理器

多核 CPU 可以是局部共享高速缓存相连并且相互独立的处理单元之间通过总线连接起来，共享一个内存空间。图 2-3 为 Intel 双核双处理器（四核）结构：主频 1.6GHz，4M 的二级缓存，功耗 65W，不支持超线程技术。该双核双处理器系统实际上是将两个酷睿处理器[13]集成在一块主板上，两个处理器 $C1$-$C2$ 与 $C3$-$C4$ 之间通过外连总线相连，每个处理器间的两个核即 $C1$ 和 $C2$、$C3$ 和 $C4$ 是共享 L2Cache 结构。

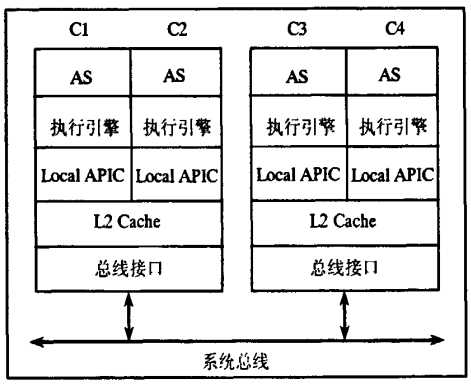

图 2-3 Intel Xeon® 5110 至强双核双处理器结构示意图

2.1.3 Intel 四核双处理器

图2-4为Intel四核双处理器（八核）结构，集成了两个至强四核处理器，这两个四核处理器之间通过外连总线相连，而四核处理器内部结构采用双双核方式，即两个核用L2 Cache组成一个双核组，然后再用片内总线将两个双核组连接起来，封装到一起，成为一个四核处理器。如图2-4中，C1与C2组成一个双核组，它们与C3-C4双核组通过片内总线连接成一个四核的处理器。而四核双处理器系统则是用系统总线将两个这样的四核处理器相连，形成一个8核的双处理器系统。

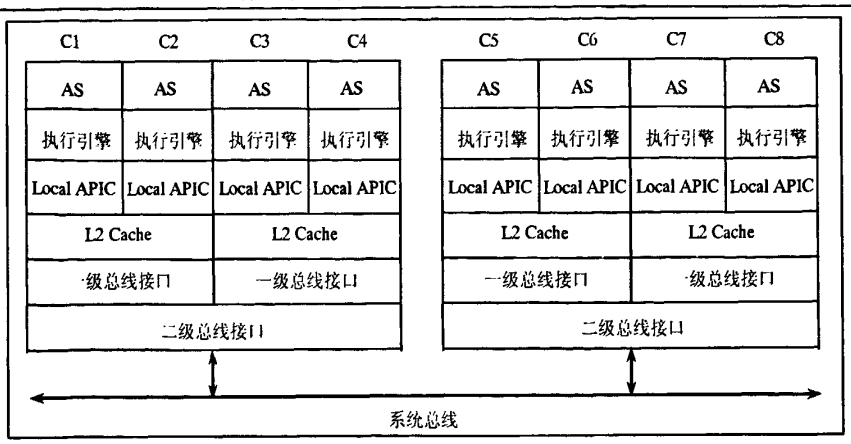

图2-4 Intel Xeon® E5310至强四核双处理器结构示意图

2.2 OpenMP 并行编程模式

OpenMP 是作为共享存储标准而问世的。它是为在多处理机上编写并行程序而设计的一个应用编程接口。它包括一套编译指导语句和一个用来支持它的函数库。OpenMP 是通过与标准 Forturn，C 和 C++ 结合来工作的[14]。

OpenMP 程序设计模型提供了一组与平台无关的编译指令、指导命令、函数调用和环境变量，可以显式地指导编译器如何利用应用程序中的并行性。对于很多循环来说，都可以在其开始之前插入一条编译指导，使其以多线程执行，开发人员不需要关心实现细节，这是编译器和 OpenMP 线程库的工作，开发人员只需要认真考虑哪些循环应该以多线程方式执行，以及如何重构算法以便在多核处理器上获得更好的性能等问题。

OpenMP 执行模型采用 fork-join 的方式。程序开始是以一个单进程运行，称为执行的主线程。主线程顺序运行到第一个并行块结构时就生成一个线程组，原来的主线程成为线程组的主线程。程序中被并行块包围起来的所有语句在线程组中并行执行，一直到并行块执行完后，线程组中的线程中止，而主线程继续执行。一个程序中可以定义任意数目的并行块，因此，在一个程序的执行中可以分叉、合并若干次。

2.2.1 OpenMP 指令格式

OpenMP 的指令格式为：#pragma omp directive-name [clause [[,] clause]...]

OpenMP 的所有编译指导语句以#pragma omp 开始，后面跟着具体的功能指令，其中 directive 部分就包含了具体的编译指导语句，包括 parallel、for、parallel for、section、sections、single、master、critical、flush、ordered 和 atomic。这些编译指导语句或者用来分配任务，或者用来同步。后面的可选字句 clause 给出了相应的编译指导语句的参数，字句可以影响到编译指导语句的具体行为，每一个编译指导语句都有一系列适合它的子句，其中有 5 个编译指导语句（master，critical，flush，ordered、atomic）不能跟相应的子句。

2.2.2 对 for 循环并行化

在 C 程序中固有的并行操作常常以 for 循环的形式表现。OpenMP 可以方便的对其进行指示一个 for 循环的迭代可以被并行地执行。

For 循环的编译指导语句是#pargma omp parallel for 例如以下程序段指示编译器将 for 循环并行化：

#pargma omp parallel for

```
for ( i = begin; i < end; i ++) a[i] = i;
```

在 for 循环并行执行的过程中，主线程创建若干派生线程，所有这些线程协同工作共同完成循环的所有迭代。每个线程有各自的执行现场，也就是上下文：一个囊括所有这个线程将访问的变量的地址空间。执行现场包括静态变量，堆中动态分配的数据结构以及运行时堆栈中的变量。

执行现场包括线程本身的运行时堆栈，这个堆栈保存着调用函数的框架信息。其他变量或者是共享的，或者是私有的。共享变量在所有线程的执行现场中的地址都是相同的。所有线程都可以对共享变量进行访问。私有变量在各个线程的执行现场中的地址不同。一个线程可以访问它自己的私有变量，但是不能访问其他线程的私有变量。

在 parallel for 编译指导语句中，变量默认设置为共享，而循环编号变量除外，它是私有变量。如下程序段中 for 循环的迭代被分配到两个线程中进行。循环编号 i 为私有变量-每个线程拥有一份自己的副本。剩下的变量 b 和变量 cptr，以及堆中分配的数据，均为共享变量。

```
int main(int argc, char* argv[])
{
    int b[3];
    char* cptr;
    int i;
    cptr = malloc(1);
  #pragma omp parallel for
    for(i = 0; i < 3; i ++)
      b[i] = i;
```

2.2.3 私有变量声明

对于较为复杂的循环结构，例如下面的双重循环，我们都可以进行并行的执行。并行外重循环还是内重循环就会涉及到变量的私有化。以下介绍三个变量私有化子句。

```
for(i = 0; I < BLOCK_SIZE(id,p,n); i++)
    for(j = 0; j < n; j ++)
        a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

1. private 子句

Private 子句指导编译器将一个或若干个变量私有化。语法如下：

private (<variable list>)

指导语句告诉编译器为每个执行这条指导语句的代码段的线程分配一个这个变量的私有副本。如下例一个并行 for 循环。变量 j 的私有副本只有在 for 循环内部才访问到。在循环的入口和出口处此变量都是未定义的。

一个使用 private 子句的双层潜逃循环的 OpenMP 实现如下：

```
#pragma omp parallel for private(j)
for(i = 0; i < BLOCK_SIZE(id,p,n); i++)
  for(j = 0; j < n; j ++)
    a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
```

即使 j 在并行的 for 循环区之前已经被提前赋值了，也没有一个现场能够访问到那个值。同样的，无论在并行执行 for 循环时线程为 j 赋值了什么值，共享的 j 的值不会受到影响。另一方面，当进入并行结构的时候一个私有变量的默认值为未定义，当并行结构结束的时候这个变量的值也是未定义。

由于削减了共享变量和与其对应的私有变量之间不必要的复制，私有变量的默认情况将节省执行时间。

2. firstprivate 子句

Firstprivate 子句有以下语法：

firstprivate（<variable list>）

它将指导编译器在进入循环时创建各个私有变量，并使得它们与由主线程所控制的相应变量有相同的值。

下面是正确使用该子句的并行循环代码：

```
X[0] = complex_function();
#pragma omp parallel for private(j) firstprivate(x)
for(i = 0; i < n; i ++)
{
    for(j = 1; j < 4; j ++)
        x[j] = g(i, x[j – 1]);
    answer[i] = x[1] – x[3];
}
```

在 firstprivate 变量表中，各个变量的值将在线程创建时初始化，而非进行每次迭代时均初始化。如果某个线程执行了该循环的多次迭代，并对其中的某个变量进行了改动，那么之后的各次迭代所得到的该变量的值将是改动后的值。

3.lastprivate 子句

Lastprivate 子句将指示编译器产生可以在循环的并行执行结束时将其私有变量复制回主线程中的对应变量的代码。

如下程序段为了使 x[3]能在 for 循环外访问到，我们必须将 x 声明为 lastprivate 类型的变量。

```
#pragma omp parallel for private(j) lastprivate(x)
for(i = 0; i < n; i++)
{
    x[0] = 1.0;
    for(j = 1; j < 4; j++)
        x[j] = x[j-1]*(i+1);
    powers[i] = x[0] + x[1] + x[2] + x[3];
}
temp = x[3];
```

2.2.4 部分 OpenMP 函数

1. omp_get_thread_num

返回线程的标志数。如果有 t 个活动线程，线程的标志数将为 0~t-1。

2. omp_get_max_threads

返回运行系统所允许的程序可以创建的线程数目最大值。

3. omp_get_num_threads

返回当前活动的线程数。如果此函数在串行部分被调用，它将返回 1。

4. omp_set_num_threads

设置后续的并行块所期望的线程数。线程数可能超过可用的处理器数，此时多个线程会被映射到同一个处理器上。此函数必须在串行部分被调用。

5. omp_get_num_procs

返回并行环境中可用的处理器数。

2.3 多核多处理器的 PLCN 模型

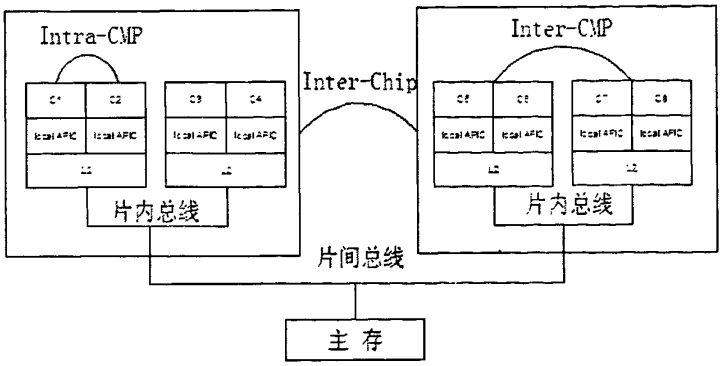

图 2-5 Intel Xeon® E5310 至强四核双处理器系统层次结构示意图

图 2-5 是 Intel 的四核双处理器系统的层次结构示意图，从该示意图上来看，我们根据处理器执行核心的连接情况区分两个层次，(1) 片内总线相连 (2) 片间总线相连。在通过大量的实验、计算和分析的基础上，我们提出了当前多核体系结构的模型——PLCN 模型[11]。下面我们结合该图讨论该模型。首先定义：

- P：表示消息处理开销。即一个处理器核处理一个消息所用的时间消耗。在模型中用 P_k 表示在第 k 个核对数据的处理开销。在同构多核下，由于 $P_1 = P_2 = ... = P_k$，故可定义 P_c 为单个核对数据处理的开销。
- N：数据传输层次关系的集合，$N = \{N_1, N_2, ..., N_n\}$，显然 $|N| = n$。比如针对图 2-5，n=3，即将相互通信的数据传输分为 3 个层次，3 个分量分别记为 N_1，N_2 和 N_3，其中 N_1 表示共享 L2 Cache 的两个 core 之间通信；N_2 表示通过片内总线互连的两个 core 之间的通信；N_3 表示通过片间总线互连的两个 core 之间的通信。

L：表示消息传输延迟的集合，即各传输层次上从源到目的地传输消息所用的时间的集合，显然|L|=N 。在图 2-5 中，$|L| = 3$，其中，L_1 表示数据在 Intra-CMP 中传输的时间消耗；L_2 表示数据在 Inter-CMP 中传输的时间消耗；L_3 表示数据在 Inter-Chip 中传输的时间消耗。不同传输层次上的传输延迟 T_{N_i} 可以表示为：

$$T_{N_i} = \sum_{i=1}^{n} L_i \qquad \text{公式 2-1}$$

- C：表示参与点对点通信的核的数目。

(1) 当 C=2 时，不同结构之间进行通信的时间开销为：

$$T = 2 \cdot P_c + T_{N_i} \qquad \text{公式 2-2}$$

(2) 当 C≥3 时，各核之间存在通讯环路的时间开销通信为：

$$T = C \cdot P_c + \sum^{C} T_{N_i} = C \cdot P_c + \sum^{C} \sum_{i=1}^{n} L_i \qquad \text{公式 2-3}$$

为了验证该模型的正确性，我们分别测试三核情况下小数据的传输、四核情况下小数据和大数据传输三种情况，对比模型计算值和实际测试值之间的差异，结果见图 2-6、图 2-7、图 2-8 所示。。

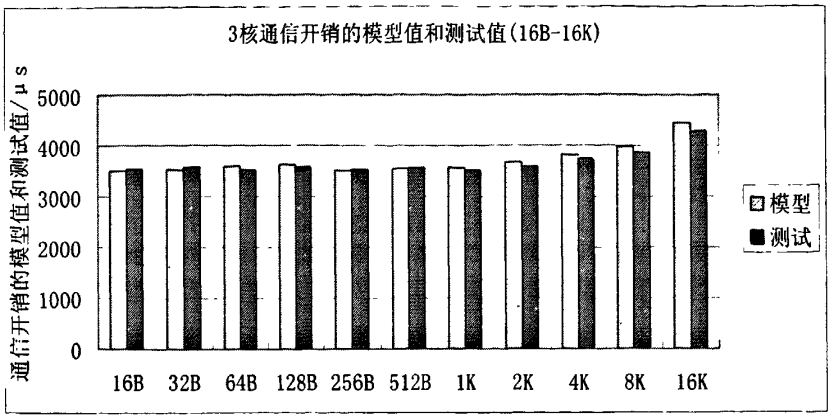

图 2-6 三核情况下对于小数据的传输

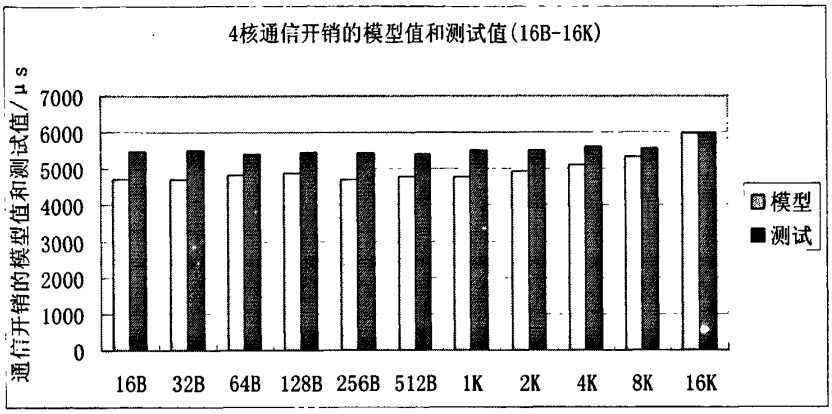

图 2-7 四核情况下对于小数据的传输

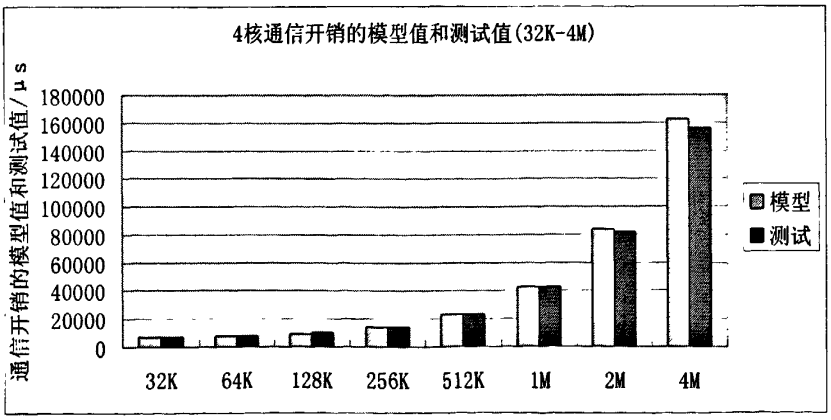

图 2-8 四核情况下对于大数据的传输

图 2-6 表示了三个核通信的情况下对于小数据的传输情况，最大相对（与测量值比较）误差 3.7%，最小相对误差 0.2%，平均相对误差 1.73%。图 2-7 表示了四个核通信的情况下对于小数据传输的情况，最大相对误差 14.5%，最小相对误差 0.2%,平均相对误差 9.96%。图 2-8 表示了四个核通信的情况下对于大数据传输的情况，最大误差 6.69%，最小误差 0.41%，平均误差 2.66%。

从实际测量与模型对比分析中可以表明 PLCN 并行计算模型比较适合于目前由 Intel 多核结构处理器组成的多处理器系统。

根据公式 2-3 可以看出在多核多处理中不同核之间的通信开销由两部分组成：在处理器上的处理时间和在层次上的传输时间。根据模型可知要减少通信的开销可从两个方面进行：一方面减少处理核上的

处理时间；另一方面要减少层次间的通信过程。从减少处理时间这个角度讲，该模型指出了并行执行的必要性，可以通过创建多个子任务来减少每个处理核上所处理的数据规模，进而减少处理时间，从而提高系统性能。但从另一角度来看，任务划分的越多，处理核上分配的运算量虽然变得越少，但是由于并发任务数量的增加反而会带来通信代价开销的增加，系统性能不会有明显的提升反而有可能会下降。因此对应模型的后半部分，在有效划分子任务的同时，应该作好任务的核分配，尽量减少多层次间数据传输的代价，来提高系统性能。

根据上面的模型，我们在编程中可以应用数据隔离的方法（程序划分成多个线程，使线程之间没有相互的依赖关系），同时和线程绑定核这一技术相结合，这种并行的方法来提高并行程序的效率。

2.4 线程绑定核并行技术

OpenMP 是现在运用很广的并行编程模式，另外还有 MPI 以及 Intel 的 TBB 等，但这些并行编程模型均没有考虑到多核多处理器之间不同的连接方法带来的数据通信间的代价是不同的这一问题。这一节，我们先介绍我们在不同结构的多核多处理器上建立的一个模型，并由此提出一个线程绑定核的并行技术。

线程绑定核并行技术的基本思想：首先，把程序段分为几个线程并行执行。其次，按照线程分配的方法把每个线程分配到不同的核上执行。最后，用线程绑定技术来指定线程在某个核上的执行。

2.4.1 线程分配模型

1.模型的建立

假设多核多处理器有 N_{node} 个处理器 P_0、P_1、…$P_{Nnode-1}$，每个处理器包含 N_{core} 个处理器核 C_0、C_1、…$C_{Ncore-1}$。待分配的程序有 N 个线程 T_0、T_1、…T_{N-1}。

待分配的多线程程序可以表示为一个任务关系图(Task Interaction Graph)，它是一个无向图 G=(V，E)，其中 V 是节点的集合$\{V_i\}$，V_i 对应 T_i，T_i 为节点对应的线程号；E 是无向边的集合$\{E_{ij}\}$，连接节点 V_i 和 V_j 的边 $E_{ij}\in E$ 表示线程 T_i 和 T_j 之间有依赖关系，用权值 W_{ij} 表示，由于建立的是无向图，则 $W_{ij}=W_{ji}$。T_i 线程的执行时间用 t_i 表示。

如果线程 T_i 和 T_j 之间没有依赖关系，则权值 $W_{ij}=0$。如果 T_i 和 T_j 之间有依赖关系，则有以下四种情况[13]。

(1)数据依赖：设 u 与 v 为给定程序中两条不同的语句，如果有一条从 u 到 v 的执行路径，同时存在一个在 u 处定义、v 处引用的变量，且该变量在从 u 到 v 的执行路径上的其他位置没有被重新定义，则称 v 数据依赖于 u。

(2)控制依赖：设 u 与 v 为给定程序中两条不同的语句，如果从 u 至少可以引出两条执行路径，其中一条导致 v 的执行，而另一条可能导致 v 不执行，则称 v 控制依赖于 u。

(3)同步依赖：设 u 与 v 为给定程序不同线程中的两条语句，如果 u 执行的开始或终止通过线程间同步直接决定 v 执行的开始或终止，则称 v 同步依赖于 u。

(4)通信依赖：设 u 与 v 为给定程序不同线程中的两条语句，如果 u 计算的变量的值通过线程间通信直接影响在 v 计算的变量的值，则称 v 通信依赖于 u。

在本论文中主要考虑第一种数据依赖的情况，如果线程 T_i 和 T_j 之间数据依赖的大小用权值 W_{ij} 来表示，数据依赖大权值就大，数据依赖小权值就小。

任务的分配可以表示为一个函数 f，通过该函数，每个线程都被映射到一个处理器核上。函数 f 可以形式化地表示为：

f: $\{T_i\}$ ->$\{C_j\}$，i=0，1，…N-1，j=0，1，… $N_{node} * N_{core}$-1

任务分配算法的目标是将任务关系图划分成 $N_{node} * N_{code}$ 个不相交的子图，包含在子图 G_{ij} 中的节点就对应了分配给处理节点 P_i 中处理器核 C_j 的线程[15]。

2.线程分配原则

根据多核多处理器以上模型特性，得出了以下需要遵循的分配原则：

(1)将相互通信频繁的线程分配给同一处理器中的不同核中，以利用共享 Cache 或片内总线来降低通信开销 。

(2)各处理器及处理器核的负载基本均衡。

A)尽量利用每个核。

B)每个核上的任务量尽量保持均衡。

C)将运行时间长或任务量特别重的的几个线程分配到不同的核。

3.线程分配算法

算法分为两步，第一步是线程到处理器上的划分，第二步是将已经划分到处理器上的线程再划分到每个核上。

第一步划分，线程到处理器上的划分：

(1)分配到每个处理器上的线程个数基本相等，线程个数符合线程总数除以处理器个数 N/N_{node}。这主要是为了负载平衡。

(2)分配给不同处理器的线程之间的权值之和在所有的划分方案中最小。把通信频繁的线程尽量划分到一个处理器上，这是因为同一个处理器上核之间的通信开销比较小。

第二步划分，线程到处理器核上的划分：

(1)每个核上分配的线程个数 $N/(N_{node}*N_{core})$。

(2)执行时间 ti 长的线程 Ti 不分配到同一个核上。这样不至于把执行时间长的线程都划分到一个核上。

5.线程分配算法的有效性验证

例如有待分配线程数 N=6，处理器个数 N_{node}=2，每个处理器上的核数 N_{core}=2。它们之间的权值分别为：$W_{12}=1, W_{13}=7, W_{14}=4, W_{15}=0, W_{16}=9, W_{23}=4, W_{24}=8, W_{25}=8, W_{26}=2, W_{34}=4, W_{35}=5, W_{36}=5, W_{45}=1, W_{46}=7, W_{56}=1$。总权值 W=66。每个线程的执行时间分别为：$t_1=14, t_2=10, t_3=6, t_4=2, t_5=11, t_6=9$。

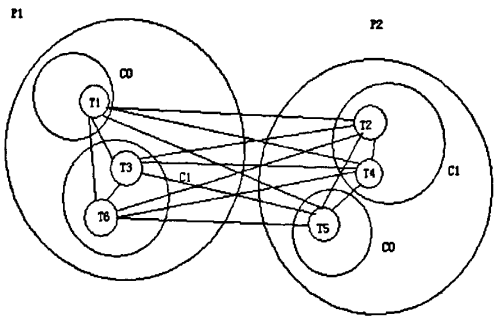

图 2-9 线程分配算法分配后的结果

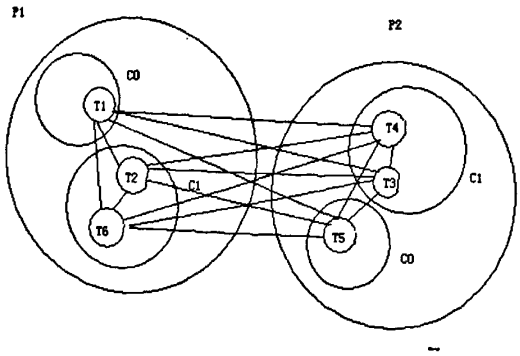

图 2-10 改进的随机分配算法分配后的结果

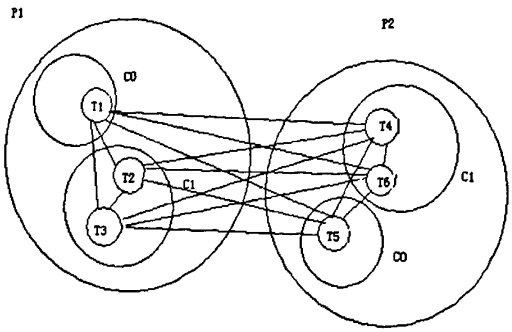

图 2-11 随机分配算法分配后的结果

图 2-9 描述了经过线程分配算法把线程分配到不同处理器过后处理器之间的线程权值总和为 28 (W_{12}+ W_{14}+W_{15}+W_{23}+W_{26}+W_{34}+W_{35}+W_{46}+W_{56})。图 2-10 描述了经过改进的随机分配算法把线程分配到不同处理器过后处理器之间的线程权值总和为 44(W_{13}+ W_{14}+W_{15}+W_{23}+W_{24}+W_{25}+W_{36}+W_{46}+W_{56})。图 2-11 描述了未经改进的随机分配算法把线程分配到不同处理器过后处理器之间的线程权值总和为 45(W_{14} +W_{15}+W_{16}+W_{24}+W_{25}+W_{26}+W_{34}+W_{35}+W_{36})。通过比较我们发现经过线程分配算法分配线程后处理器之间线程依赖最低，分配后的性能最佳。

表 2-1 测试比较结果

总权值	线程分配算法	改进的随机分配算法	随机分配算法
292	171	190	222
323	203	214	258
319	180	237	243
296	180	209	213
281	158	208	207

表 2-1 中“总权值”表示所有线程间权值之和，通过表 2-1 我们可以知道，线程通过以上线程分配算法分配到核上过后，在同一处理器上的线程与其他处理器上的线程间依赖关系最小的，同时运行时通信代价就最小，并行效果最好。

2.4.2 线程绑定核实现方法

线程绑定核思想就是希望根据上述线程分配算法，将线程独分配到合适的核资源，在该核上独立执行。以下是线程绑定核是实现代码：

```
DWORD WINAPI ThreadProcess1(LPVOID p)
{
    ......;
    return 0;
}
DWORD WINAPI ThreadProcess2(LPVOID p)
{
    ......;
    return 0;
}
void main
{
    char buff[20];
    DWORD id1,id2;
```

```
//创建线程，并将其挂起
HANDLE                                    hHandle1                    =
CreateThread(NULL,0,ThreadProcess1,&buff,CREATE_SUSPENDED,&id1);
 HANDLE                                   hHandle2                    =
 CreateThread(NULL,0,ThreadProcess2,&buff,CREATE_SUSPENDED,&id2);
//设置线程亲和性，也就是线程绑定核
SetThreadAffinityMask(hHandle1,0x00000001);
SetThreadAffinityMask(hHandle2,0x00000002);

//唤醒线程
ResumeThread(hHandle1);
ResumeThread(hHandle2);

//判断线程是否执行结束，如果没有则一直等待
WaitForSingleObject(hHandle1,INFINITE);
WaitForSingleObject(hHandle2,INFINITE);

//关闭线程
    CloseHandle(hHandle1);
    CloseHandle(hHandle2);
```

以上代码实现了两个线程 ThreadProcess1 和 ThreadProcess2 分别绑定到 0x00000001 和 0x00000002 两个核上，实现了充分的并行化。

2.5 本章小结

本章介绍了现今 Intel 多核多处理器的一些架构，例如 Intel 双核处理器、Intel 双核双处理器和 Intel 四核双处理器，同时介绍了 OpenMP 并行编程模式和多核多处理器的 PLCN 模型。最后提出了线程绑定核的并行技术，此种技术的线程分配模型和线程绑定核的实现方法。

第三章 并行 Alpha-Beta 剪枝搜索算法及其在五子棋中的应用

为了更好的阐述并行博弈树搜索技术在五子棋中的应用，本章首先介绍一些基本的串行搜索技术，以及并行 Alpha-Beta 剪枝算法的基本原理。最后介绍如何把并行 Alpha-Beta 剪枝算法应用到五子棋游戏中。

3.1 基本搜索技术

本节主要介绍三种基本的搜索技术：博弈树，极大极小值算法和 Alpha-Beta 剪枝算法。

3.1.1 博弈树

博弈树是从根部向下递归产生的一棵包含所有可能的对弈过程的搜索树，这里称为完全搜索树[17]。

为了方便博弈树的研究我们利用最简单的一种博弈来作为研究对象。这种博弈具有三个特性：二人零和，全信息，非偶然。具体的说这样的博弈具有如下的特点：

(1) 对垒的 A，B 双方轮流采取行动，博弈的结果只有三种情况：A 方胜，B 方败；A 方败，B 方胜；双方战成平局。

(2) 在对垒过程中，任何一方都了解当前的格局及过去的历史。

(3) 任何一方在采取行动前都要根据当前的实际情况，进行得失分析，选取对自己最为有利而对对方最为不利的对策，不存在“碰运气”的偶然因素。即双方都是很理智地决定自己的行动。

在博弈过程中，任何一方都希望自己取得胜利。因此，在某一方有多个执行步骤可选择时，它总是挑选对自己最为有利而对对方最为不利的那个执行步骤。此时，如果我们站在 A 方的立场上，则可供 A 方选择的若干执行步骤之间是“或”关系，因为主动权操在 A 方手里，他或者选择这个执行步骤，或者选择另一个执行步骤，完全由 A 方决定。但是，若 B 方也有若干个可供选择执行步骤，则对 A 方来说这些执行步骤之间是“与”关系，因为这时主动权操在 B 方手里，这些可供选择的执行步骤中地任何一个都可能被 B 方选中，A 方必须考虑到对自己最不利的情况发生。

若把上述博弈过程用图表示出来，得到的是一棵“与/或”树。这里要特别指出，该“与/或”树是始终站在某一方(例如 A 方)的立场上得出的，决不可一会儿站在这一方的立场上，一会儿又站在另一方的立场上。

博弈树有如下的特点：

(1) 博弈的初始格局是初始节点。

(2) 在博弈树中，“或”节点和“与”节点是逐层交替出现的。自己一方扩展的节点之前是“或”关系，对方扩展节点之前是“与”关系。双方轮流地扩展节点。

(3) 所有能使自己一方获胜的终局都是本原问题，相应的节点是可解节点；所有使得对方获胜的终局都是不可解节点[18]。

3.1.2 极大极小值算法

在搜索过程中，一般是使用一个估价函数，对每个格局(即博弈树中的节点)进行“估价”，假设估价函数值越大，表示对计算机走棋越有利，那么计算机在走下一步棋时，只要搜索出估价函数值最大的格局即可。当然对手走棋会选函数值低的格局，这样非终端结点的函数值就由其下层结点的这种最大最小交替递归调用得到，称为极大极小算法搜索算法。

极大极小值搜索属于完全搜索，在最大最小树搜索的过程中，实际相当一部分结点的搜索并不会影响最终搜索树的值。采用这种搜索方法的搜索引擎的搜索效率非常低。“蛮力搜索”肯定是不可取的，而

应进行有选择的搜索。主动放弃没有意义的分支(剪枝)，不能遗漏存在最优解可能的节点，在最有希望的方向上局部加深，这便是最基本的搜索策略[19]。

3.1.3 Alpha-Beta 剪枝算法

Bruno 在 1963 年首先提出了 Alpha-Beta 算法，1975 年 Knuth 和 Moore 给出了 Alph-Beta 的数学正确性证明。Alpha-Beta 算法通过下界(Alpha)和上界(Beta)对搜索树值的最终范围进行了划定，当某些子树其值被证明会在上述界限之外，无法影响整颗树的值时，便可进行剪枝。Alpha-Beta 剪枝的示意图如图 3-1 所示。

Alpha-Beta 剪枝的原则归纳为以下两个结论[20]：

对 min 层，若 min 值小于等于其父节点之值，则剪掉子节点(Alpha 剪枝)；

对 max 层，若 max 值大于等于其父节点之值，则剪掉子节点(Beta 剪枝)；

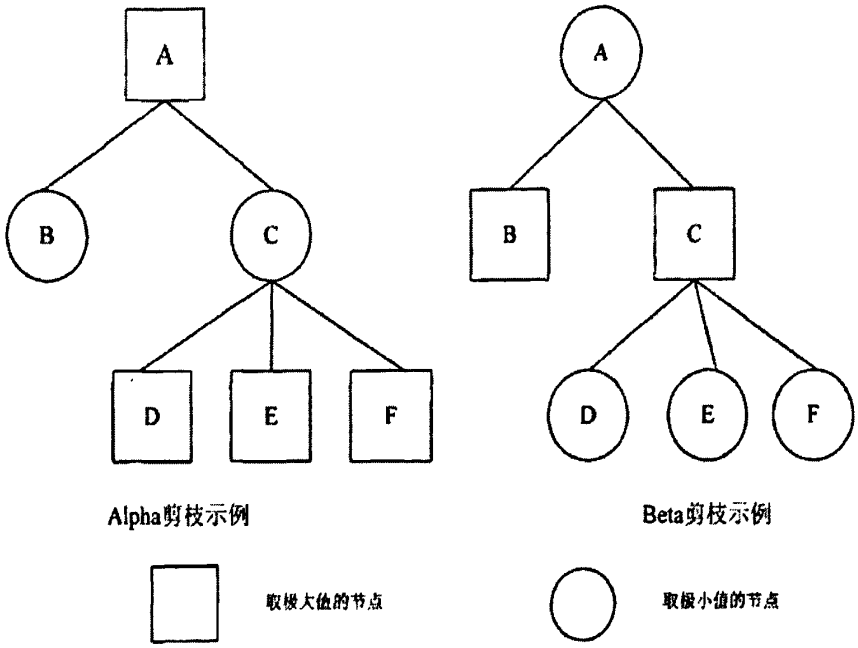

图 3-1 Alpha-Beta 剪枝示意图

对于图 3-1 中的 Alpha 剪枝示例，如果节点 B 的值为 20，节点 D 的值为 17，由此我们可以判断节点 C 的值将小于等于 17(取极小值)；而节点 A 的值为节点 Max(B，C)，为 20，也就是说不再需要估节点 C 的其他子节点如 E、F 的值就可以得出父节点 A 的值了。这样将节点 D 的后继兄弟节点减去称为 Alpha 剪枝 (alphaCutoff)。

对于图 3-1 中的 Beta 剪枝示例，如果节点 B 的估值为 10，节点 D 的估值为 16，由此我们可以判断节点 C 的值将大于等于 16(取极大值)；而节点 A 的值为节点 Min(B，C)，为 10。也就是说不再需要求节点 C 的其他子节点如 E、F 的值就可以得出父节点 A 的值了。这样将节点 D 的后继兄弟节点减去称为 Beta 剪枝 (betaCutoff)。

Alpha-Beta 搜索能够让我们忽略许多节点的搜索。对于每一个被忽略的非叶子节点来说，这都意味着不仅节点本身，而且节点下面的子树也被忽略掉了。这就导致了 Alpha-Beta 搜索需要遍历的节点远少于极大极小算法所遍历的节点。这也同时意味着对搜索同一棵树来说，Alpha-Beta 搜索所花费的时间远少于极大极小算法所花费的时间。

同极大极小搜索算法一样，Alpha-Beta 剪枝算法也有点繁琐，我们不仅要在奇数层进行 alpha 剪枝，而且还要在偶数层进行 beta 剪枝。不过只要将负极大值的形式套用上去，这样在任何一层都只进行 beta 剪枝，它就会同负极大值算法一样简洁。

负极大值形式的 alpha-beta 搜索算法的伪代码如下：

```
int alphabeta(int nPly, int alpha, int beta)
{
    if(Gameover)
        return eval();  //胜负已分，返回估值
    if(nPly <= 0)
        return eval();  //叶子节点返回估值
    for(each possible move m)  //对每一可能的走法
    {
        Make move m:  //产生子节点
                //递归搜索子节点
        Score = -alphabeta(nPly-1，-beta，-alpha)
        unmake move m;   //撤销搜索过的子节点
        if(score >= alpha)
            alpha = score;  //保存最大值
        if(alpha >= beta)
            break;          //beta 剪枝
     }
     return alpha;  //返回极大值
}
```

3.2 Alpha-Beta 剪枝搜索算法并行化的途径

Alpha-Beta 搜索有很多可以进行并行执行的操作。一种方法是对位置估值和移动产生进行并行化。另一种是对搜索过程进行并行化。

3.2.1 并行渴望搜索

对 Alpha-Beta 剪枝搜索的一个直接的并行化方法就是进行并行的渴望搜索。如果有三个核可用，那么可以给每个核分配下面的窗口之一：(-∞，v-e),(v-e,v+e),(v+e，∞)。在理想的情况下搜索(v-e,v+e)的核会得到结果，但是在任何情况下并行搜索的速度都比单个核搜索(-∞，∞)要快。我们还可以通过搜索更加狭窄的窗口来使用更多的核[14]。

3.2.2 并行子树估值

另一种方法是让每个核并行搜索不同的子树。当采用这种方法的时候，我们必须考虑搜索开销与通信开销。搜索开销是指由于引入并行性所导致的搜索节点个数的增加。通信开销是指用于协调各个进程进行搜索所需要的开销。通过让每个进程都知道当前的搜索窗口(a，ß)，可以将搜索开销转化为通信开销。通过允许各个核搜索过时的搜索窗口，可以将通信开销转化为搜索开销。

并行Alpha-Beta剪枝搜索算法讲就是从树根处划分博弈树，并给每个核分配相同大小的子树。让每个核在其子树上执行Alpha-Beta剪枝搜索。每个核都以(-∞，∞)为初始窗口搜索，并且各个核不通知其他核自己的搜索窗口的改变。显然这种方法使得通信开销降到了最低。

3.3 基于线程绑定核技术的 BC-Alpha-Beta 剪枝搜索算法

基于线程绑定核技术的并行 Alpha-Beta 剪枝搜索算法简称 BC-Alpha-Beta 剪枝搜索算法。本节讨论了 BC-Alpha-Beta 剪枝搜索算法的算法思想和算法实现。

3.3.1 算法思想

BC-Alpha-Beta 剪枝搜索算法，主要是对串行 Alpha-Beta 剪枝搜索算法进行了并行化，从树根处划分博弈树，根据棋盘的特性可以将博弈树划分为多个子树，把每个子树创建一个线程独立执行，线程之间并无相关性，当所有线程搜索结束，比较这多个结果把最好的结果作为最终结果。根据线程分配算法，同时又由于线程间没有依赖关系，相当于线程间的权值 W_{ij} 可以被视为 0，那么我们在分配线程时可以有一定的随意性，原则上分配到任何一个处理器上都可以，相当于线程分配算法中最简单也是最特殊的情况，但考虑到主线程对所有子线程间都有一定的数据传送这一特点，所以我们分配时，尽量使子线程向主线程靠拢，即按照 PLCN 模型中的核间通信代价，取代价较小的核分配。在这样的一种情况下我们尽量的使线程个数和处理器的核数相等，使每个线程都能在一个核上执行，同时每个核也被充分利用起来，到达充分利用多核多处理器每个核的目的。

整个 BC-Alpha-Beta 剪枝搜索算法分两步走，第一步，根据多核多处理器的核数划分博弈树的子树个数，例如双核双处理器有 4 个核，则我们划分 4 个子树，在每个子树上执行 Alpha-Beta 剪枝搜索。第二步，把划分好的子树搜索采用线程实现，并把每个线程分配到一个核上去执行。我们这里划分的子树个数等于核数，这样能最大化的利用多核多处理器[21]，有效的并行化 Alpha-Beta 剪枝搜索算法。

3.3.2 BC-Alpha-Beta 剪枝搜索算法

基于线程定核分配的并行 Alpha-Beta 剪枝搜索算法如下。

Step1. 检测系统处理器拓扑结构，得到核数 N。

Step2. 根据核数 N 把棋盘大小平均划分成 N 块，每块分别建立一个子博弈树。

Step3. 为每个子博弈树创建一个线程，一共创建 N 个线程，每个线程执行串行的 Alpha-Beta 剪枝搜索算法（即上文中所提到的负极大值形式的 Alpha-Beta 剪枝搜索算法），每个子博弈树搜索层次为 M。

Step4. 主线程应用上文中所提出的线程绑定核技术把线程分配到各个核上并行执行，主线程挂起。

Step5. 各子线程在各自数据集上同时执行 Alpha-Beta 剪枝搜索算法。

Step6. 子线程搜索结束后返回各自的最佳值到主线程。所有子线程返回后主线程被唤醒。

Step7. 由主线程判断是否满足结束条件。如满足，则算法结束；否则转 step8

Step8. 主线程比较这 N 个返回值，得出最终最佳值，确定下棋位置，下子。返回 step2。

3.3.3 BC-Alpha-Beta 剪枝搜索算法的关键问题

BC-Alpha-Beta 算法有两个值得关注：第一，如何分配线程到核。第二，搜索层次的确定搜索层越多节点规模越大花费时间越多。需要有一个合理的值。

根据第二章中的线程分配算法，此处由于各线程间是相互独立的，则显然各线程间的权值 Wij=0。可以按照第一步把线程平均分配到各个处理器上。由于划分的每个子树的搜索规模相同，因此每个线程的执行时间相差不大。同时，线程数量和核数个数相同，根据线程分配算法第二步，则分配到每个核上的线程数为 1。由此保证搜索任务被均衡的分配到各个核上，不会出现负载不均衡的情况。

至于第二个问题。我们在 Intel 双核双处理器机器上做相关实验，测试结果如图 3-2 所示。

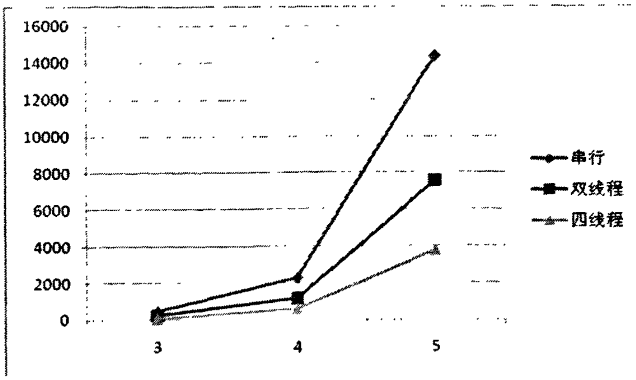

图 3-2 搜索层次与搜索时间

图 3-2 中纵坐标为运行时间（单位 ms），横坐标表示搜索层数，“串行”表示执行的串行程序，“双线程”表示执行的双线程程序，“四线程”表示执行的四线程程序。从图中可以得知随着搜索层数的增加执行时间随着指数时间增长，因此在实际应用中层次越深下子等待时间越长，并不是搜索层次越深越好，而是取一定搜索层数比较恰当。如果是串行搜索，从图上可以看到，搜索层次为 3 是比较合适的，但如果是在双核或者 4 核上进行并行搜索，则搜索层次可以到 4，如果 4 秒延迟能够容忍的话，四核上甚至可以将搜索层次进一步加深到 5。这也进一步说明了，采用并行算法后，能获得两方面的好处，不仅能提高算法执行速度（这在本章最后一节会有测试），还能增加搜索深度，从而更有可能取得更优解。

3.4 BC-Alpha-Beta 剪枝搜索算法在五子棋中的应用

3.4.1 棋盘表示

棋盘表示主要探讨的是使用什么数据结构来表示棋盘上的信息。一般说来，这与具体的棋类知识密切相关。通常，用来描述棋盘及其上棋子信息的是一个二维数组。要让计算机知道棋盘局势状态，就是要它记住棋盘中哪个位置有黑子，哪个位置有白子以及哪个位置是空点。对于五子棋程序而言，因为棋盘是 15 行，15 列，因此可以将棋盘状态的描述用一个 15×15 的二维数组表示。

本程序的数据将基于如下所示的数据表示：

```
char chessOld[N][N]={{'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','1','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','2','2','2','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','1','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','1','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','1','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     {'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'},
                     };   //棋盘表示，0 表示无子，1 表示黑子，2 表示白子。
```

（1）棋盘状态数据由一个 15×15 的二维数组表示。

（2）用字符“1”和“2”来表示不同的棋子，黑色棋子用“1”表示，白色棋子用“2”表示。字符“0”表示无子。

3.4.2 静态估值

为了给某一局势估分，我们提取局势中一些特殊的棋型，给它们分别估分，并累加求和，得局势的总得分。

由于五子棋规则中，横、竖及斜方向上的同色五子相连都为赢，因此在寻找特征时，横、竖、斜方向上都要寻找。我们称棋盘上某一行、某一列或某一对角斜线为一路。

我们给每个局势的估分如下所示，‘0’表示无子，‘1’表示黑子，‘2’表示白子。

黑子：

(1) 连五	11111	->	100000
(2) 死四 B	11101	->	3000
(3) 死四 C	11011	->	2600
(4) 死三 C	10011	->	600
(5) 死三 D	10101	->	550
(6) 活四	011110	->	50000
(7) 死四 A	211110	->	2500
(8) 死三 A	211100	->	500
(9) 死三 B	010110	->	800
(10) 死二 A	211000	->	150
(11) 死二 C	010010	->	200
(12) 活三	0011100	->	3000
(13) 死二 B	0010100	->	250
(14) 活二	00011000	->	650

白子：

(1) 连五	22222	->	100000
(2) 死四 B	22202	->	3000
(3) 死四 C	22022	->	2600
(4) 死三 C	20022	->	600
(5) 死三 D	20202	->	550
(6) 活四	022220	->	50000
(7) 死四 A	122220	->	2500
(8) 死三 A	122200	->	500
(9) 死三 B	020220	->	800
(10) 死二 A	122000	->	150
(11) 死二 C	020020	->	200
(12) 活三	0022200	->	3000
(13) 死二 B	0020200	->	250
(14) 活二	00022000	->	650

本程序中使用 int evaluate(int x, int y, char chessflag, char chess[][N])函数来实现局势的估分。其中，chessflag 为 1 表示黑子，2 表示白子。x,y 为下子的位置。chess[][N]为棋盘形势。

3.4.3 走法产生

走法产生是指将一个局面的所有可能的走法罗列出来的那一部分程序。也就是用来告诉其它部分下一步可以往哪里走的模块。各种棋类随规则的不同，走法产生的复杂程度也有很大的区别。一般说来，在一种棋类游戏种，双方棋子的种类越多，各种棋子走法的规则越多，则在程序中，走法产生的实现就越复杂。

在五子棋的对弈程序中，由于双方只有黑白各一个子，并且在走子的过程中，没有吃子、提子等规则的存在，双方轮流走子，只要有一方首先在棋盘的水平、垂直、斜线方向上使己方的五个子连成一条直线，就获得胜利。因此，对于五子棋的走法产生来说，棋盘上的任意空白点都是合法的下一步。这样在五子棋的走法产生模块里，只要扫描棋盘，寻找到所有的空白，就可以罗列出所有符合规则的下一步。

如何确定走出最好的走法，就需要用到搜索技术。本程序所用的搜索算法为 Alpha-Beta 剪枝算法，我们从根节点处划分博弈树，结合现今多核多处理器计算机，给每个核分配相同大小的子树，让每个核执行 Alpha-Beta 剪枝搜索。我们使用线程绑定核并行技术把 Alpha-Beta 剪枝搜索实现并行化。

本五子棋程序中线程绑定核并行的 Alpha-Beta 剪枝搜索核心代码如下：

```
DWORD WINAPI ThreadProcess1(LPVOID p)  //定义线程 1 所要完成的操作，下同
{
    printf("\nthis is from 1 thread ! the thread num is %d\n",GetCurrentThreadId());
    int score;      //得分
    int i, j, i1 = 0, j1 = 0;
    int depth = SearchDepth;
    int alpha = -1000, beta = 100001;
    int x, y;
    bool turn = true;
    if(depth == 0)
    {
        return evaluate(i1, j1, blackorwhite(chess1), chess1); // 叶子节点返回估值
    }
    if(turn == false)     //极小值点
    {
        for(i = 0; i < N/2; i ++)
        {
            for(j = 0; j < N; j ++)
            {
                if(chess1[i][j] == '0')   //表示棋盘i,j位置没有子
                {
                    chess1[i][j] = blackorwhite(chess1);        //搜索到的位置
                    score = alphabeta(alpha, beta, depth - 1, chess1, true, &x, &y, 0, 0);
                    chess1[i][j] = '0';      //撤销搜索位置
                    if (score < beta)
                    {
                        beta = score ; // 取极小值
                        if (alpha >= beta)
                            return alpha ; // 剪枝,抛弃后继节点
                    }
                }
            }
```

```
        }
        return beta ; // 返回最小值
    }
    else
    {       //极大值点
        for(i = 0; i < N/2; i ++)
        {
            for(j = 0; j < N; j ++)
            {
                // 生成新节点
                if(chess1[i][j] == '0')
                {
                    chess1[i][j] = blackorwhite(chess1);        //搜索到的位置
                    score = alphabeta(-1000, 100001, SearchDepth, chess1, true, &x, &y, 0, 0); // 递归搜索
子节点
                    chess1[i][j] = '0';      //撤销搜索位置
                    if (score > alpha)
                    {
                        if (depth == SearchDepth)
                        {
                            x = i;
                            y = j;
                        }
                        alpha = score ; // 取极大值
                        if (alpha >= beta)
                            return beta ; // 剪枝,抛弃后继节点
                    }
                }
            }
        }
        return alpha ; // 返回最大值
    }
    return 0;
}
```

线程绑定核代码段：

```
HANDLE hHandle1 = ::CreateThread(NULL,0,ThreadProcess1,&buff,CREATE_SUSPENDED,&id1);
//创建线程，并将其挂起
SetThreadAffinityMask(hHandle1,0x00000001);      //设置线程亲和性，也就是线程绑定核
ResumeThread(hHandle1);          //唤醒线程
WaitForSingleObject(hHandle1,INFINITE); //判断线程是否执行结束，如果没有则一直等待
CloseHandle(hHandle1);//关闭线程
HANDLE hHandle2 = ::CreateThread(NULL,0,ThreadProcess2,&buff,CREATE_SUSPENDED,&id2);
SetThreadAffinityMask(hHandle2,0x00000002);
ResumeThread(hHandle2);
WaitForSingleObject(hHandle2,INFINITE);
CloseHandle(hHandle2);
```

3.5 并行 Alpha-Beta 剪枝搜索算法的测试与分析

如图 3-3 所示，Alpha-Beta 剪枝搜索的层数为 3，纵坐标表示运行时间（单位 ms），横坐标表示每一组的测试数据。程序运行平台为 Intel 双核双处理器。“并行绑定四核四线程”表示 Alpha-Beta 剪枝搜索算法经过线程绑定核技术创建四个线程并行后的测试数据，“串行”表示 Alpha-Beta 剪枝搜索算法的串行测试数据。从图中可以分析出，经过线程绑定核并行，创建四个线程并行执行效率明显比串行执行效率高，加速比为 3.84。

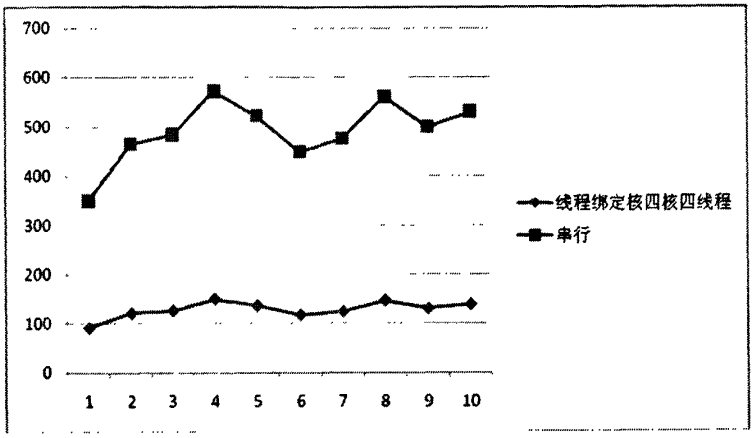

图 3-3 线程绑定核四核四线程

如图 3-4 所示，Alpha-Beta 剪枝搜索的层数为 3，纵坐标表示运行时间（单位 ms），横坐标表示每一组的测试数据。程序运行平台为 Intel 双核处理器。“并行绑定双核双线程”表示 Alpha-Beta 剪枝搜索算法经过线程绑定核技术创建两个线程并行后的测试数据，“串行”表示 Alpha-Beta 剪枝搜索算法的串行测试数据。从图中可以分析出，经过线程绑定核并行，创建两个线程并行执行效率明显比串行执行效率高，加速比为 1.90。

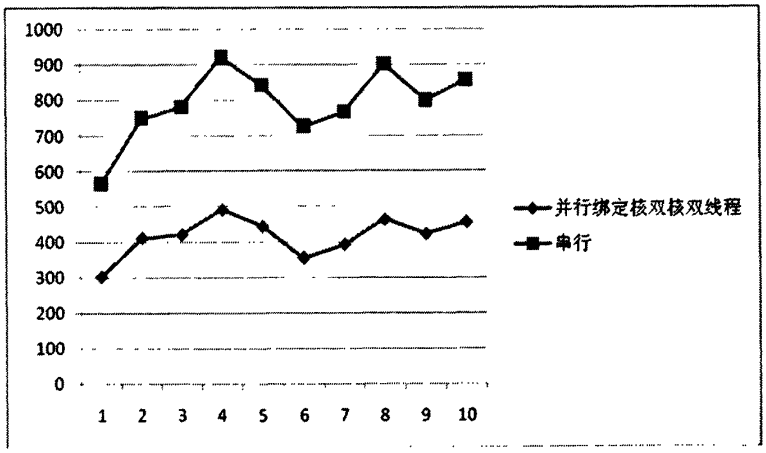

图 3-4 线程绑定核双核双线程

程序对 Alpha-Beta 剪枝搜索算法的并行进行了数据隔离，每个子树独立运行，相互之间并无依赖关系。经过分析与对比得到了不错的性能加速比，同时由图 3-2 又表明，算法并行后，可以适当增加搜索层次，从而可能找到更好的解。以上说明本文对该算法的并行化是成功的。

3.6 本章小结

本章首先介绍了基本的搜索技术，其次介绍 Alpha-Beta 剪枝搜索算法的两种并行化途径，最后提出了基于线程绑定核技术的 BC-Alpha-Beta 剪枝搜索算法并应用于五子棋中进行了验证和测试。

第四章 并行遗传算法在足球游戏中的应用

本章首先阐述遗传算法的基本原理，在此基础上，给出了遗传算法的并行化方法，并应用于足球游戏中。

4.1 遗传算法基本原理

遗传算法是从代表问题可能潜在的解集的一个种群（population）开始的，而一个种群则由经过基因（gene）编码的一定数目的个体(individual)组成。每个个体实际上是染色体(chromosome)带有特征的实体。染色体作为遗传物质的主要载体，即多个基因的集合，其内部表现（即基因型）是某种基因组合，它决定了个体的形状的外部表现，如黑头发的特征是由染色体中控制这一特征的某种基因组合决定的。因此，在一开始需要实现从表现型到基因型的映射即编码工作。由于仿照基因编码的工作很复杂，我们往往进行简化，如二进制编码，初代种群产生之后，按照适者生存和优胜劣汰的原理，逐代（generation）演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度（fitness）大小选择（selection）个体，并借助于自然遗传学的遗传算子（genetic operators）进行组合交叉（crossover）和变异（mutation），产生出代表新的解集的种群。这个过程将导致种群像自然进化一样的后生代种群比前代更加适应于环境，末代种群中的最优个体经过解码（decoding），可以作为问题近似最优解[22]。

4.1.1 遗传算法术语说明

由于遗传算法是由进化论和遗传学机理而产生的搜索算法，所以下面是我们将会用来的一些术语说明：

（1）染色体(Chromosome)

染色体又可以叫做基因型个体(individuals),一定数量的个体组成了群体(population),群体中个体的数量叫做群体大小。

（2）基因(Gene)

基因是串中的元素，基因用于表示个体的特征。例如有一个串 S=1011，则其中的 1，0，1，1 这 4 个元素分别称为基因。它们的值称为等位基因(Alletes)。

（3）适应度(Fitness)

各个个体对环境的适应程度叫做适应度(fitness)。为了体现染色体的适应能力，引入了对问题中的每一个染色体都能进行度量的函数，叫适应度函数. 这个函数是计算个体在群体中被使用的概率。

4.1.2 遗传因子

复制、交叉和变异三个遗传算子是遗传算法能够找到最优解的途径。这三个遗传算子模拟了自然界的物种交配和生殖的方式[23]

1.复制

复制操作，也叫选择操作，根据个体的适应度函数值所度量的优、劣程度决定它在下一代是被淘汰还是被遗传。一般地说，选择将使适应度较大（优良）个体有较大的存在机会，而适应度较小（低劣）的个体继续存在的机会也较小。

2.交叉

交叉操作是将被选择出的两个个体 P1 和 P2 作为父母个体，将两者的部分码值进行交换。假设有如图 4-1 所示的八位长的二个体：

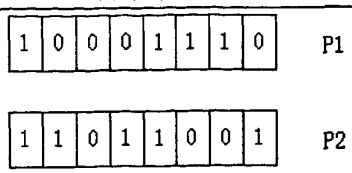

图 4-1 父母个体

产生一个在 1 到 7 之间的随机数 c，假如现在产生的是 3，将 P1 和 P2 的低三位交换：P1 的高五位与 P2 的低三位组成数串 10001001，这就是 P1 和 P2 的一个后代 Q1 个体；P2 的高五位与 P1 的低三位组成数串 11011110，这就是 P1 和 P2 的一个后代 Q2 个体。其交换过程如图 4-2 所示：

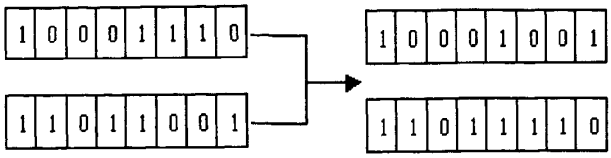

图 4-2 交叉操作

3.变异

变异操作是改变数码串的某个位置上的数码。我们先以最简单的二进制编码表示方式来说明，二进制编码表示的每一个位置的数码只有 0 与 1 这两个可能，比如有如图 4-3 所示的二进制编码表示：

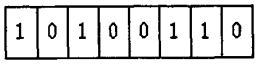

图 4-3 变异前

其码长为 8，随机产生一个 1 至 8 之间的数 k，假如现在 k=5，对从右往左的第 5 位进行变异操作，将原来的 0 变为 1，得到如图 4-4 所示的数码串：

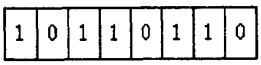

图 4-4 变异后

二进制编码表示时的变异操作是将 0 与 1 互换：0 变异为 1，1 变异为 0。

4.1.3 遗传算法的特点

遗传算法是解决搜索问题的一种通用算法，对于各种通用问题都可以使用。搜索算法的共同特征为：

（1）首先组成一组候选解；

（2）依据某些适应性条件测算这些候选解的适应度；

（3）根据适应度保留某些候选解，放弃其他候选解；

（4）对保留的候选解进行某些操作，生成新的候选解。

在遗传算法中，上述几个特征以一种特殊的方式组合在一起：基于染色体群的并行搜索，带有猜测性质的选择操作、交叉操作和变异操作。这种特殊的组合方式将遗传算法与其它搜索算法区别开来。

4.1.4 遗传算法的步骤

1.创建一个随机的初始状态

初始种群是从解中随机选择出来的，将这些解比喻为染色体或基因，该种群被称为第一代，这和符号人工智能系统的情况不一样，在那里问题的初始状态已经给定了。

2.评估适应度

对每一个解(染色体)指定一个适应度的值，根据问题求解的实际接近程度来指定(以便逼近求解问题的答案)。不要把这些“解”与问题的“答案”混为一谈，可以把它理解成为要得到答案，系统可能需要利用的那些特性。

3.繁殖(包括子代突变)

带有较高适应度值的那些染色体更可能产生后代(后代产生后也将发生突变)。后代是父母的产物，他们由来自父母的基因结合而成，这个过程被称为“杂交”。

4.下一代

如果新的一代包含一个解，能产生一个充分接近或等于期望答案的输出，那么问题就已经解决了。如果情况并非如此，新的一代将重复他们父母所进行的繁衍过程，一代一代演化下去，直到达到期望的解为止。

4.2 并行遗传算法的并行模型

将遗传算法改写成适合多核多处理器的并行算法可以采用以下两种方法：1）让每个处理器独立地运行于个体的一个隔离子种群上，通过迁移与其他处理器周期地共享它的那些最好的个体。2）让每个处理器在一个公共的种群上完成算法中的每一步的一部分——复制、交叉和变异。

4.2.1 主从模式

主从式模型是最直接的遗传算法并行化方案，并不改变遗传算法的基本结构特点，它只有一个群体，复制、交叉和变异等全局操作由主处理器节点串行进行，而适应度的评价和计算则由各从处理器节点并行执行。主处理器节点和从处理器节点的通信表现在两个方面，一方面主处理器节点把选择的群体部分个体发送给从处理器节点；另一方面从处理器节点把适应度的评价结果发送给主处理器节点。

可以看出，主从模式的计算时间主要集中在适应度的评估和计算上，易于并行实现。但是会出现主处理器节点和从处理器节点负载不均衡的情况，这情况主要表现在主处理器节点把任务分配给各个从处理器后导致自己处于等待状态，等待从处理器节点返回数据。这就导致主处理器节点的资源浪费，出现不均衡情况。同样这样的情况也会出现在多核多处理器架构中，没法充分利用处理器的每个核。

4.2.2 粗粒度模型

粗粒度模型同时又被称作分布式模型或孤岛模型，是适应性最强和应用最广泛的遗传算法并行模型。它将群体依照处理器节点的个数分成与处理器节点个数相同的多个子群体，各个子群体在各自的处理器节点上并执行遗传算法，每经过一定的进化代数，各个子群体间将交换若干个体，一方面用来引入更优良的个体，另一方面丰富了种群的多样性，达到了防止未成熟早收敛现象[24]。对于粗粒度模型，重点是什么时候交换个体，交换哪些个体，采用什么方式交换，因此目前对粗粒度模型的研究，主要是解决这三个方面的问题：迁移规模、迁移策略和迁移拓扑。

4.2.3 细粒度模型

细粒度模型又称作邻域模型，在整个进化过程中虽然保持一个群体，但要求子群体的划分要非常细小，最理想状态是每个处理器节点只处理一个个体，要求各处理器节点具有极强的通信能力，对于每个染色体的选择和交叉操作都只在所处的处理器节点及其邻域中进行。由于整个进行过程中，不需要或者需要很少的全局操作，因此充分发挥了遗传算法的并行特性。可以看出细粒度模型的一个明显缺点，处理器节点间的通信频繁，对处理器的要求比较高，也不适合多核处理器架构的计算机。

4.3 基于线程绑定核技术的并行遗传算法 BC-Genetic

基于线程绑定核技术的并行遗传算法简称 BC-Genetic 算法。本节讨论了 BC-Genetic 算法的算法思想和算法实现。

4.3.1 算法思想

BC-Genetic 算法采用的并行化模型为粗粒度模型，它将一个种群划分为多个子群。为每个子群创建一个子线程，根据线程分配算法划分到多核多处理器的各个核上执行。每个子线程之间经过一定的进化代数，相互之间进行最优个体的交换。因此在线程间存在有一定的数据交换，即线程间依赖关系 $W_{ij}>0$，然而每次交换的个体数目相等，通信代价差不多，则我们视为各个线程之间依赖关系相等。根据 PLCN 模型，我们尽可能的把线程分配到同一个处理器的核中，以减少线程在核之间的通信代价。我们通过线程分配算法，在满足一个线程在一个核上执行的条件下，将线程分配到同一个处理器中，再将同一个处理器中的线程分配到各个核上执行。

本文中的足球游戏把 BC-Genetic 算法应用在球员站位中，它把球员站位的组合视为一个染色体，一组染色体为一个群体。群体依照多核多处理器核的个数分成与多核多处理器核个数相同的多个子群体，各个子群体在各自的核上执行遗传算法，每经过一定的进化代数，各个子群体间将交换若干个体，一方面用来引入更优良的个体，另一方面丰富了种群的多样性。

4.3.2 算法步骤

基于线程定核分配的并行遗传算法如下。

Step1. 检测系统处理器拓扑结构，得到处理器个数 C 和单个处理器的核数为 N，总核数为 C·N。

Step2. 给球员站位编码，组成一染色体，初始化一个种群，种群大小为 M。

Step3. 根据核数划分为 C·N 个子种群，每个种群大小为 M/C·N。为每个子种群创建一个线程，一共创建 C·N 个线程，每个线程执行串行的遗传算法，经过复制、交叉和变异产生新的子种群。

Step4. 主线程应用上文中所提出的线程绑定核技术按下列子步骤把线程分配到各个核上并行执行，主线程挂起。

Step 4.1 从 C·N 个子种群中，选择 N 个线程，由于所有的 W 值一样，所以可以任意选择。剩余线程数设为 T。

Step 4.2 将 N 个线程分配到处理器 0 中

Step 4.3 如果处理器中的 N 个核又分成 K 个组，每组用共享 Cache 链接，组间用片内总线连接，则从 N 个线程中选取 N/K 个线程分配到核 0，剩余的线程按此办法依次分配到其他核。如果处理器 0 中的核全部采用共享 Cache 或者片内总线，则直接将 N 个线程分配到 N 个核上。

Step 4.4 如果 T=0，转 Step 5；否则，从剩余的 T 个线程中选择 N 个线程，分配到处理器 1 中，剩余的线程数 T=T-N，转 Step4.3。

Step5. 各子线程在各自数据集上同时执行遗传算法。

Step6. 子线程经过一定代数的复制、交叉和变异。相互之间交换较优的个体。所有子线程执行结束后返回各自的最佳站位到主线程。所有子线程返回后主线程被唤醒。

Step7. 由主线程判断游戏是否结束。如满足，则程序结束；否则转 step8

Step8. 主线程比较这 N 站位方案，得出最终的站位方案。

Step9. 根据当前各个球员的站位进行编码，产生一个新种群，种群大小为 M，返回 step3。

该算法步骤在划分的线程数小于核数的时候(比如在多任务系统中有些核比较忙，不能再分配任务)，能够尽量使用通信代价小的核进行并行工作。

4.4 BC-Genetic 算法在足球游戏中的应用

本节讲述并行遗传算法应用在足球游戏中的球员位置产生。简单的介绍了实现方法。

4.4.1 足球游戏环境和规则

足球游戏的比赛规则如下：

1. 游戏环境有下列项组成。
   - 1个足球场
   - 2个球门
   - 1个球
   - 2个球队
   - 8个场上队员，每队各4名队员。

2. 场地规则。

由于足球的球场四边被墙围住，所以球不会出界，但会简单地从墙上回弹。这就意味着：没有角球，没有投球，没有越位。球门区域在球场内，球员不得在该区域内活动。

3. 比赛规则。

双方给派四名队员，没有守门员。以某方球员将球踢入对方球门区域为得分，每次得一分。任意一方先得20分则比赛结束。

为了方便描述，给出以下定义：

定义1：球员跑动的区域R（又称活动区域）为n个相邻且连续分布的子区域组成的集合，即$R=\{r_i \mid 0 \leqslant i \leqslant n-1\}$，$r_i$表示球场中球员能跑动的第i块子区域，且$r_0 \cap r_1 \ldots \cap r_{17} = \Phi$，显然$|R|=n$，为子区域个数。为了方便后续的算法设计，我们约定所有子区域大小相等（除球门前的两个区域除外），且是按从下到上，从右到左编号。R被分成两大区域，$RA=\{r_0,r_1,\ldots r_{(n/2-1)}\}$，$RB=\{r_{n/2},r_{(n/2+1)},\ldots r_{(n-1)}\}$

定义2：球门内的区域Q为2个子区域集合$Q=\{q_0,q_1\}$，显然$|Q|=2$，且$q_0 \cap q_1=\Phi$；

定义3：球场$P=R \cup Q$，且$R \cap Q=\Phi$。

定义4：定义球员S为8个球员的集合，即$S=\{s_i \mid 0=i=7\}$，s_i表示第i个球员。S被分成两个子集SA和SB，分别表示两个队的球员，其中$SA=\{s_0,s_1,s_2,s_3\}$，$SB=\{s_4,s_5,s_6,s_7\}$。假定MA中的球员将RA作为自己半场，SB中球员将RB作为自己半场。

定义5：函数$f(s_i)$表示球员i在场上的位置，即在R中的某个子区域。显然，$f(s_i) \in R$。

定义6：定义球的位置为ball，$ball \in R$。

定义7：$dist(r_i,r_j)$为r_i与r_j之间的距离，我们定义从r_i到r_j之间经过的区域块数的最小值为其距离。每经过一个区域，距离加1。球场的长度为len，定义为最左端区域到最右端区域之间的距离。

定义8：函数$goals(s_i)$为球员i的进球数，$0=i=7$；$kicks(s_i)$为球员i的得球数。这两个数越大，说明该球员经验越丰富，拿球儿率越大。

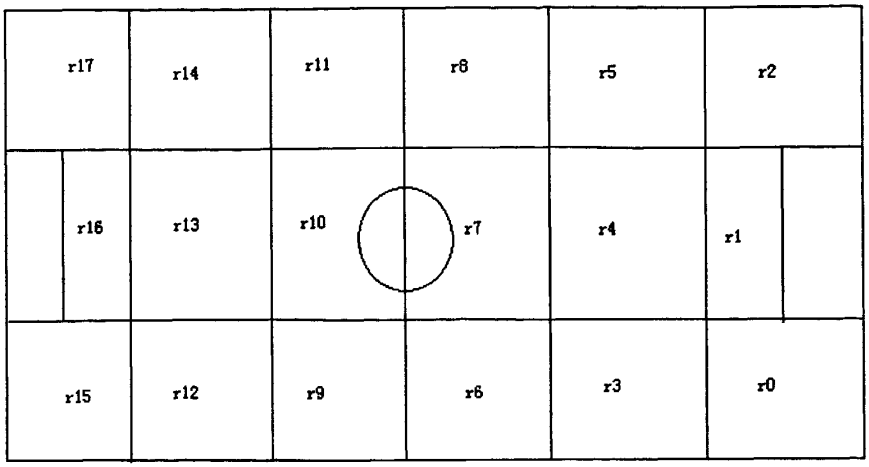

图4-5 球场区域

如图4-5中，将球场分割为18块大小相等的活动区域R{r_0,r_1...r_{17}}，$r_0 \cap r_1 \cap \ldots \cap r_{17} = \Phi$，$r_i$表示球场中的第i块区域，其中$r_1$与$r_{16}$两个区域的大小略小，减除的部分是两个球门区域的大小。球门区域Q={{q_0,q_1}。Dist(r0,r1)=2，Dist(r0,r7)=4。len=6

4.4.2 染色体编码

染色体编码的方式有很多种，常见的是二进制方式和十进制字方式，也有字符串方式。如著名的旅行商问题(TSP)里，假设有20个城市以[0...19]编码，那么[7，6，9，8…19，3，0，4]这个包含20个元素的序列A就可以看作是一个染色体，每一个元素A_j(0≤j<20)就是染色体的一个基因。这个染色体可以解码为从编号为7的城市出发，依次到达城市6、城市9、……，最后到达城市4完成20个城市的遍历。显然，这个序列是TSP的一个可能解，因此染色体就是问题的可能解的表示方式。

当一个球员不处于进攻状态（Attacking）、助攻（SupportAttacker）、逐球（ChaseBall）、运球（Dribble）、踢球（KickBall）及返回（ReturnToHomeRegion）时，他就进入Wait状态——等待球队发出的下一个行动指令。显然，就像人类进行足球比赛时需要排兵布阵一样，游戏中球员站在哪个位置也相当重要，能否组织起有效的进攻或者防守，决定因素之一就是在合适的位置有没有球员可以快速有效地执行命令。使用遗传算法来对球员的站位进行决策分析，可以找出对当前局势有利的位置编排方案。从而使得球队策略更加有效。

在足球游戏中，我们期望获得某一队的球员的合适站位，这需要一种方式来描述某个队的所有4个队员（守门员位置忽略）的站位，因此给出如下定义。

定义9：序列TA[f(s_0),f(s_1), f(s_2),f(s_3)]表示SA队所有队员的站位情况，序列TB[f(s_4),f(s_5), f(s_6),f(s_7)]表示SB队所有队员的站位情况，.比如，TA(14,11,12,6)，则表示SA队中0号球员站在14号的区域中，即f(s_0)=r_{14}，1号球员站在11号区域中,即f(s_1)=r_{11}，2号球员站在12号区域中,即f(s_2)=r_{12}，3号球员站在6号区域中,即f(s_3)=r_6。在本文的实现中，我们仅对SA队的队员站位情况使用并行的遗传算法，SB队作为参照组，依然采用传统算法。因此在下面的讨论中，我们只考虑SA队及序列TA。

称序列TA为一个可能解，其编码方式即是我们设计的球员站位染色体的编码方式——十进制编码方式的一个序列。具体实现中，将染色体封装成类Chromosome，它的成员变量m_Geneme是一个基因序列，用std::vector容器来保存；成员变量m_iScore是这个染色体对“自然环境”的适应值，由评估模块评定。友元类GT实现了两个Chromosome的大小比较；还定义了三个友元函数分别实现交叉、复制及变异三个遗传算子。具体的C++代码如下：

```
class Chromosome{
private:
        std::vector<Genetype> m_Geneme;
```

```
        double m_iScore;
public:
        Chromosome();
        ~Chromosome(){};
        const std::vector<Genetype>& GetGeneme()const{return m_Geneme;}
        void SetScore(double iScore){m_iScore = iScore;}
         double GetScore(){return m_iScore;}
        friend void Intercross(const Chromosome& p1, const Chromosome& p2,
                                        Chromosome& c1, Chromosome& c2);
        friend void Agamogenesis(const Chromosome& p, Chromosome& c);
        friend void Mutant(const Chromosome& p, Chromosome& c);
         friend void Mutant(Chromosome& c);
        friend class GT;
};
class GT{
public:
        inline bool operator()(const Chromosome* c1, const Chromosome* c2)const{
                return c1->m_iScore > c2->m_iScore;
            }
```

4.4.3 遗传算子

交叉、复制和变异三个遗传算子是遗传算法能够找到最优解的途径。这三个遗传算子模拟了自然界的物种交配和生殖的方式，各个算子的实现如表4-1所示。

表4-1 遗传算子

遗传算子	输入的染色体	输出的染色体
交义(Intercross)	TA[17,9,4,3] TA [16,13,7,9]	TA[17,9,7,9] TA [16,13,4,3] 假定元素2为交叉点
复制（选择） (Agamogenesis)	TA [17,9,4,3]	TA [17,9,4,3]
变异(Mutant)	TA [17,9,4,3]	TA [17,9,7,3] 假定元素2为变异点

遗传算子声明为染色体类Chromosome的友元函数是为了方便操作它的私有变量，其实现代码如下：

```
void Intercross(const Chromosome& p1, const Chromosome& p2,Chromosome& c1, Chromosome& c2)
{
        unsigned int IntercrossPoint = RangeRandom<unsigned int>(0, GeneLen);
        unsigned int i = 0;
        for(; i < IntercrossPoint; ++i){
                c1.m_Geneme[i] = p1.m_Geneme[i];
                c2.m_Geneme[i] = p2.m_Geneme[i];
        }
        for(; i < GeneLen; ++i){
                c1.m_Geneme[i] = p2.m_Geneme[i];
                c2.m_Geneme[i] = p1.m_Geneme[i];
```

```
        }
}
void Agamogenesis(const Chromosome& p, Chromosome& c){
        c.m_Geneme = p.m_Geneme;
}

void Mutant(const Chromosome& p, Chromosome& c){
        unsigned int MutantPoint = RangeRandom<unsigned int>(0, GeneLen);
        Genetype NewGene = RangeRandom<Genetype>(0,18);
        c.m_Geneme = p.m_Geneme;
        c.m_Geneme[MutantPoint] = NewGene;
}
void Mutant(Chromosome& c)
{
    unsigned int iMutantPoint = RangeRandom<unsigned int>(0,GeneLen);
    Genetype aNewGene = RangeRandom<Genetype>(0,18);
    c.m_Geneme[iMutantPoint] = aNewGene;
}
```

Intercross，Agamogenesis和Mutant三个函数分别对应交叉、复制和变异三个遗传算子。Intercross函数传入两个Chromosome的实例，随机选择一点进行交叉，组成两个新的染色体用作返回值。Agamogenesis函数传入一个Chromosome实例，返回一个相同的染色体，以保证优势的种群可以壮大，从而使得遗传算法可以在有限的运行时间内收敛。Mutant函数传入一个Chromosome实例，随机选择一个元素（分量）赋以一个随机的r_i区域，返回这一改变后的染色体，变异可以使得遗传算法跳出局部最优，趋近全局最优。

4.4.4 估值模块

估值模块判定每一个染色体对“自然环境”的适应度：如前文关于TSP的染色体，它的估值函数就返回遍历20个城市要走过的路程的总长度，总长度越短的染色体适应度越高，反之则越低。在足球游戏中，估值模块就没有这么简单了，一个染色体就是一个站位组合，这个组合的优劣是与当前局势有很大关系，如球的位置、对方球员的站位、己方球员的站位和控球权等有关。

在足球游戏的算法中，我们主要从以下几方面来对染色体进行评估：

1) 从当前位置到目的位置所要经过的路径的代价
2) 是否有利于进攻或者防守
3) 是否有利于抢球

对于1)，定义有公式4-1所示的路径代价评估函数[25]。

$$g(s_i)=\begin{cases}1.0, & kicks(s_i)=0 \quad goals(s_i)=0\\ 0.5+\frac{dist(f(s_i),ball)}{2.0\times len}, & kicks(s_i)>0 \quad goals(s_i)=0\\ \frac{1.0}{2.0\times goals(s_i)}, & goals(s_i)>0\end{cases} \qquad \text{公式4-1}$$

公式4-1中，对于每个被考察的s_i，均会得到一个值，以取得最小值的s_i作为下一个拿球队员。从公式中可以看到，越是进球数多的球员，拿球的机会越多，其次是过去得球多的球员。从来没有得到过求或进过球的球员，认为其经验少，拿球机会相对也少。

由于拿球队员可以简化的认为他的走向就是进攻，而没拿球的队员则要选择是进攻、还是防守或者抢球，所以对于2)、3)主要是解决没拿球队员的走向问题。这分几种情况，我们假设目前我方是SA，要考察的我方队员是$s_j \in SA$，其下一个跑位位置是$f'(s_j)$。

1）如果$\exists_{S_i,S_i}\in SA$，且$f(s_i)$=ball，则$f'(s_j)>f(s_j)$。即下一位置是助攻。

2）如果$\exists_{S_i,S_i}\in SB$，$f(s_i)$=ball，且$f(s_i)=f(s_j)+2$或$+3$或$+4$，则$f'(s_j)=f(s_i)-3$。即下一位置是企图抢球。

3）如果$\exists_{S_i,S_i}\in SB$，$f(s_i)$=ball，且不满足2）的其他条件，则$f'(s_j)<f(s_j)$。即下一位置是回防。

通过这一简单的估值模块，可以使得遗传算法在淘汰劣质个体时有法可依，从而能够收敛得到较优解。通过精细化估值模块考虑更多因素（如对方球队可能采取的策略等）可使遗传算法的收敛速度加快。

估值模块部分代码如下：

```
double Environment:: Evaluate (const std::vector<Genetype>& candidate)
{
        double iValue = 0;
        for( unsigned int i = 0; i < GeneLen; ++i)
        {
                //减去移动需要的损耗
                iValue -= DistOfTwoRgn(candidate[i], m_CurrGeneme[i]) * m_pPrm->CrossCostPerRgn;
                //有利于防守?
                iValue += GetDefendValue(candidate[i], m_OppGeneme);
                //有利于进攻?
                iValue += GetAttackValue(candidate[i], m_OppGeneme);
                //有利于抢球或者保球?
                if(m_iTeamColor == m_iControllingTeam
                        && m_iBallRgnIdx == candidate[i])
                        iValue += m_pPrm->PlyrKeepBallValue;
                else if(m_iTeamColor != m_iControllingTeam
                        && m_iBallRgnIdx == candidate[i])
                        iValue += m_pPrm->PlyrChaseBallValue;
        }
        return iValue;
}
double Environment::GetDefendValue(const int iPlyrIdx, const std::vector<Genetype>& OppGeneme)
{
        double iValue = 0;
        int OppInMyGround = 0;
        std::vector<Genetype>::const_iterator ci = OppGeneme.begin();
        for(; ci != OppGeneme.end(); ++ci){
                if(IsInMyGround(*ci))
                        ++OppInMyGround;
        }
        if(OppInMyGround > 1 ){
                if(IsInMyGround(static_cast<unsigned int>(iPlyrIdx)))
                        iValue += m_pPrm->DefendValuePerPlyr;
                else
                        iValue += m_pPrm->DefendValuePerPlyr * 0.5;
        }
        else{
```

```
            iValue += m_pPrm->DefendValuePerPlyr * 0.8;
        }
        if(m_iControllingTeam == m_iTeamColor)
            iValue *= 1.2;
        else
            iValue *= 0.8;
        return iValue;
}
double Environment::GetAttackValue(const int iPlyrIdx, const std::vector<Genetype>& OppGeneme){
        double iValue = 0;
        int OppNoInMyGround = 0;
        std::vector<Genetype>::const_iterator ci = OppGeneme.begin();
        for(; ci != OppGeneme.end(); ++ci)
        {
            if( !IsInMyGround(*ci))
                ++OppNoInMyGround;
        }
        if( OppNoInMyGround > 2 ){
            if(IsInMyGround(static_cast<unsigned int>(iPlyrIdx)))
                iValue += m_pPrm->AttackValuePerPlyr * 0.5;
            else
                iValue += m_pPrm->AttackValuePerPlyr;
        }
        if(m_iControllingTeam == m_iTeamColor)
            iValue *= 1.2;
        else
            iValue *= 0.8;
        return iValue;
}
```

4.4.5 球员位置产生的并行遗传算法的实现

本足球游戏中并行遗传算法采用的是粗粒度模型，它将群体依照多核多处理器中核的个数分成与核个数相同的多个子群体。然后采用线程绑定核技术实现并行化。

线程绑定核实现代码如下：

```
void GetBestGenemeByParallelBD(int iBall)
{
    pEn= new Environment(iBall);
    m_LeftBestChromosome = NULL;
    m_RightBestChromosome = NULL;
    std::vector<Genetype> vBest;
    sBround v1,v2;
    v1.down = 0;
    v1.up =   pEn->GetColonySize() /2;
    v2.down =   pEn->GetColonySize() /2 +1;
    v2.up =   pEn->GetColonySize();
```

```
DWORD dwThreadID;
HANDLE hThread = CreateThread(
                    NULL,                       //安全属性使用缺省。
                    0,                          //线程的堆栈大小。
                    ThreadFunc,                 //线程运行函数地址。
                    (void *)&v1,                //传给线程函数的参数。
                    CREATE_SUSPENDED,           //创建标志。
                    &dwThreadID);       //成功创建后的线程标识码。

SetThreadAffinityMask(hThread,0x00000001);      //设置线程亲和性，也就是线程绑定核
ResumeThread(hThread);                          //唤醒线程
WaitForSingleObject(hThread,INFINITE);          //等待线程结束。
CloseHandle(hThread);                           //删除的线程资源。

DWORD dwThreadID2;
HANDLE hThread2 = CreateThread(
                    NULL,                       //安全属性使用缺省。
                    0,                          //线程的堆栈大小。
                    ThreadFunc,                 //线程运行函数地址。
                    (void *)&v2,                //传给线程函数的参数。
                    CREATE_SUSPENDED,           //创建标志。
                    &dwThreadID2);      //成功创建后的线程标识码。
SetThreadAffinityMask(hThread2,0x00000002);     //设置线程亲和性，也就是线程绑定核
ResumeThread(hThread2);                         //唤醒线程
WaitForSingleObject(hThread2,INFINITE);         //等待线程结束。
CloseHandle(hThread2);                          //删除的线程资源。

if(m_LeftBestChromosome && m_RightBestChromosome)//获取两个线程计算出的最好结果
{
    if(m_LeftBestChromosome->GetScore() - m_RightBestChromosome->GetScore() > 0.000001)
    {
        vBest = m_LeftBestChromosome->GetGeneme();
    }
    else
    {
        vBest = m_RightBestChromosome->GetGeneme();
    }
}
 std::vector<Genetype>::iterator ci;
 int iFlag=0;
 /*for(ci=vBest.begin();ci!=vBest.end();++ci)
 {
     cout<<"第"<<iFlag++<<" 的位置: "<<*ci<<endl;
 }*/
 delete pEn;
}
```

4.5 并行遗传算法的测试与实验

如图 4.6 所示，初始种群规模为 1000，纵坐标表示运行时间（单位 ms），横坐标表示每一组的测试数据。程序运行平台为 Intel 双核处理器。“并行绑定双核双线程”表示遗传算法经过线程绑定核技术创建两个线程并行后的测试数据，“串行”表示遗传算法的串行测试数据。从图中可以分析出，经过线程绑定核并行，创建两个线程并行执行效率明显比串行执行效率高，加速比为 1.22。

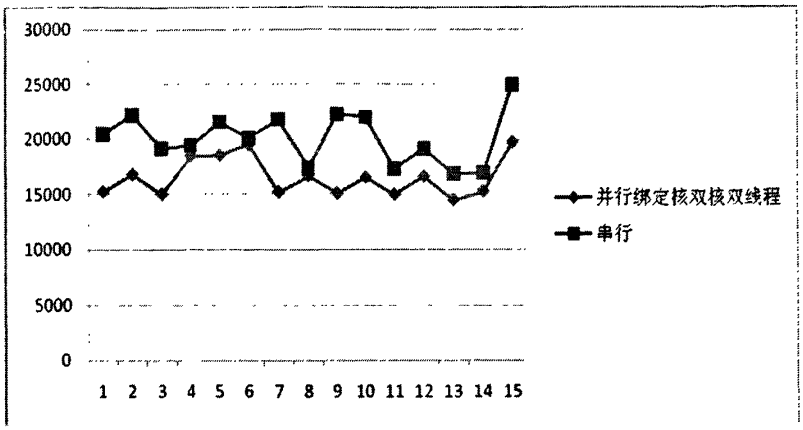

图 4.6 线程绑定核双核双线程

如图 4.7 所示，初始种群规模为 1000，纵坐标表示运行时间（单位 ms），横坐标表示每一组的测试数据。程序运行平台为 Intel 双核双处理器。“并行绑定四核四线程”表示遗传算法经过线程绑定核技术创建两个线程并行后的测试数据，“串行”表示遗传算法的串行测试数据。从图中可以分析出，经过线程绑定核并行，创建四个线程并行执行效率明显比串行执行效率高，加速比为 2.15。

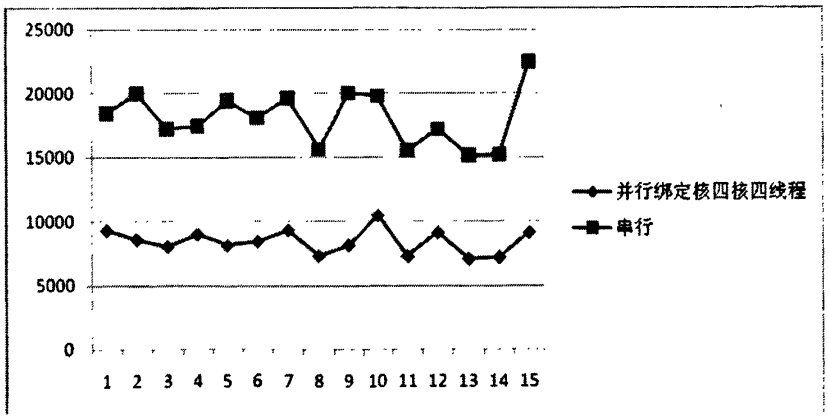

图 4.7 线程绑定核四核四线程

程序对遗传算法进行了并行化。并将并行后的线程指定到相应的核中，经过分析与对比得到了一定的性能加速比，说明本文对该算法的并行化是成功的。

4.6 本章小结

本章首先介绍遗传算法基本原理和现今并行遗传算法的三种并行模型。然后提出了基于线程绑定核技术的并行遗传算法BC-Genetic并应用在足球游戏中。最后对BC-Genetic进行测试与实验。

第五章 对 PLCN 模型的进一步研究

上面几章，我们分析了基于 Intel 多核多处理器系统的模型，和基于该模型提出的线程定核的串行程序并行化思路，并由此修改了两个经典游戏的 AI 算法，取得了一定的效果。但在具体测试与实验中，我们也发现了一些现象，本章将对我们发现的现象进行描述，并简要分析原因，这可作为今后进一步研究 Intel 多核处理器并行算法提供一些参考。

测试的硬件环境是 Intel 双核处理器和 Intel 四核双处理器。软件平台是 Microsoft Visual Studio 2008。测试主要对比同算法采用串行编程、OpenMP 并行化以及和本文中我们提出的线程绑定核的并行化之间效果的差异。

5.1 基于 PLCN 模型的线程绑定核技术再分析

如图 5-1 所示，所测数据为双线程的并行遗传算法在 Intel 四核双处理器架构上的运行情况，Intel 四核双处理器架构如本文中的图 2-4 所示。“C1 核和 C2 核”表示两个线程被分别指派到 C1 核和 C2 核上运行所得到的结果，“C1 核和 C4 核” 表示两个线程被分别指派到 C1 核和 C4 核上运行所得到的结果，“C1 核和 C8 核” 表示两个线程被分别指派到 C1 核和 C8 核上运行所得到的结果。分析图 2-4 我们可以知道 C1，C2 之间是共享 cache，C1，C4 之间是片内总线相连，C1，C8 之间是片外总线相连。一般情况下在 C1 和 C2 核上运行程序将得到最好的性能，在 C1 与 C4 核上次之，在 C1 和 C8 核上执行由于片外总线最慢，因为性能应该相对最差。但是从图 2-4 我们得出在 C1 核和 C2 核上运行的性能在很多情况下比在 C1 核和 C4 核上运行性能差，甚至平均性能上也有这样的情况。经过分析，我们认为问题在于 C1 与 C2 核间，正是因为是 Cache 共享，它的情况要比共享总线复杂的多。共享总线方式我们只需要考虑总线带宽，冲突解决方式等，但在 Cache 共享中，我们需要考虑 cache 行大小、cache 一致性和 cache 替换策略，甚至于程序中共享数据的多个线程的操作特点等。因为这些都可能影响 Cache 访问失效率，极端情况下还会引起 cache 颠簸，使性能严重下降。比较在 C1 核与 C4 核以及 C1 核和 C8 核上的运行情况我们可以得到基于 PLCN 模型的线程绑定核技术针对总线互联的多核多处理器是有效的。

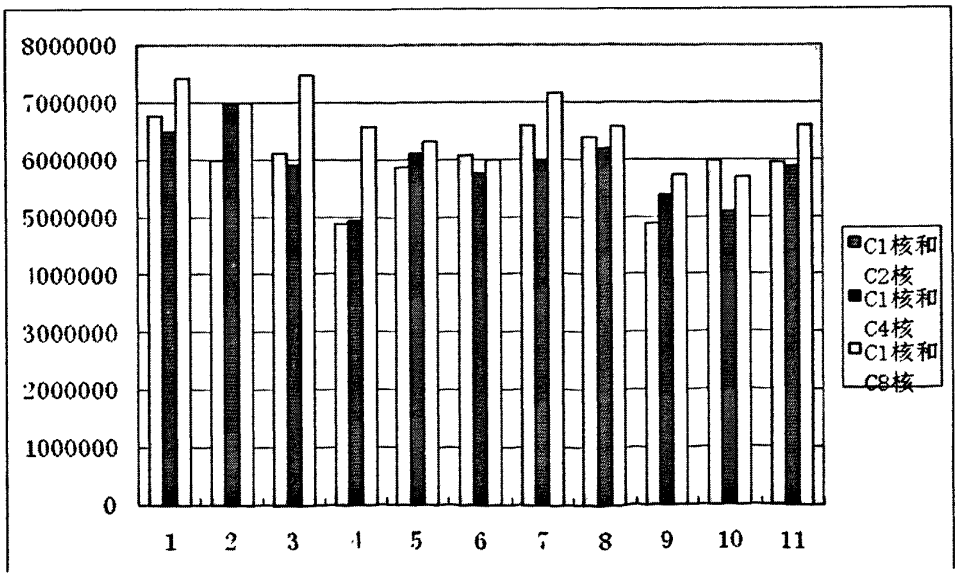

图 5-1 两线程在不同核上的运行情况

5.2 PLCN 模型下不同并行技术对并行算法性能改善程度的分析

为了进一步分析线程绑定核并行技术与传统的 OpenMP 这种不定核并行技术之间的差别，我们分别用 k-means 算法[21]、博弈树搜索算法和遗传算法在两种方式下进行并行优化，并进行比较。博弈树搜索算法和遗传算法前面已经做了详细介绍，下面我们重点介绍 K-means 算法的并行化[21]。

5.2.1 现有的 K-means 算法介绍

K-means 算法是目前在科学和工业应用中应用比较多也易于实现而且时间和空间复杂度相对较小的聚类算法之一。目前对 K-means 串行算法的研究已经非常深入，其中已做出的主要研究成果有：基于模拟退火的动态聚类算法[39]；聚类分析的遗传算法方法[40]；分组遗传算法（Grouping Genetic Algorithm, GGA）[41]；用 Tabu 搜索算法求解 K-means 聚类问题[42]。当然还有很多 K-means 算法的串行实现方法这里不一一列出来。

现有 K-means 并行算法主要是基于集群系统实现的。主要的研究成果有：聚类 K-means 算法及并行化研究[43]；基于并行遗传算法的 K-means 聚类研究[44]；基于集群环境的 K-means 聚类算法的并行化[45]。这些并行算法的实现都是基于集群系统，本文提出的并行算法是基于当前流行的多核多处理器计算机。

5.2.2 K-means 算法描述

K-means 算法又称 k-均值算法，该算法以 k 为参数，把 n 个对象分为 k 个簇，以使类内具有较高的相似度，而类间的相似度的计算根据一个簇中对象的平均值（被看作簇的重心）来进行[46]。

K-means 算法描述：

1）首先从 n 个数据对象任意选择 k 个对象作为初始聚类中心；

2）对于剩下每个对象，则根据它们与这些聚类中心的相似度（欧几里德距离），分别将它们分配给与其最相似的（聚类中心所代表的）聚类；

3）再计算每个所获新聚类的聚类中心（该聚类中所有对象的均值）；

4）不断重复这一过程直到准则函数开始收敛为止。

5.2.3 多核多处理器下的 K-means 并行算法

K-means 算法中的 n 个数据可以是二维的也可以是多维的，由于多维的实现方法与二维的实现方法类似，本文就采用二维的数据形式来实现 K-means 并行算法。K-means 算法的核心就是计算每个点与 k 个类中心点的欧几里德距离，这将占去 K-means 算法的大部分运行时间，根据分析可以得出计算每个点与 k 个类中心点的欧几里德距离并无数据之间的依赖关系，因而可以把距离之间的计算分为多个线程同时进行，从而实现在多核多处理器上并行算法，这样可以在很大程度上提高算法的运行效率。

算法的主体部分实现：

```
for(i=0;i<K;i++)
{
        随机得到k个类中心点;
}
DWORD WINAPI ThreadProcess1(LPVOID p)//定义线程 1 所要完成的操作
{
    int i,j;
        for(i=0;i<N/2;i++)
```

```
        {
          for(j=0;j<K;j++)
            {
计算每个点与k个类中心点的欧几里德距离;
找出与ni距离最近的中心点ki;
            }
}
    return 0;
}
DWORD WINAPI ThreadProcess2(LPVOID p)
{
    int i,j;
        for(i=N/2;i<N;i++)
        {
for(j=0;j<K;j++)
            {
计算每个点与k个类中心点的欧几里德距离;
找出与ni距离最近的中心点ki;
            }
}
    return 0;
}
```

5.2.4 多核多处理器下的并行算法测试结果

图 5-2 为 K-means 算法并行化后在双核双处理器上的测试结果。图 5-3 所示为博弈树搜索算法并行化后在双核双处理器上的测试结果。图 5-4 为遗传算法并行化后在双核双处理器上的测试结果。从图 5-2、图 5-3 和图 5-4 分析得知定核后的性能提升有限，与不定核的 OpenMP 并行技术相比并没有表现出太大优势。分析其主要原因应该是定核在总线互连的多核多处理器层次架构上得到的性能提升会由于 cache 共享环境下出现的诸如 Cache 颠簸的现象而有所损失。

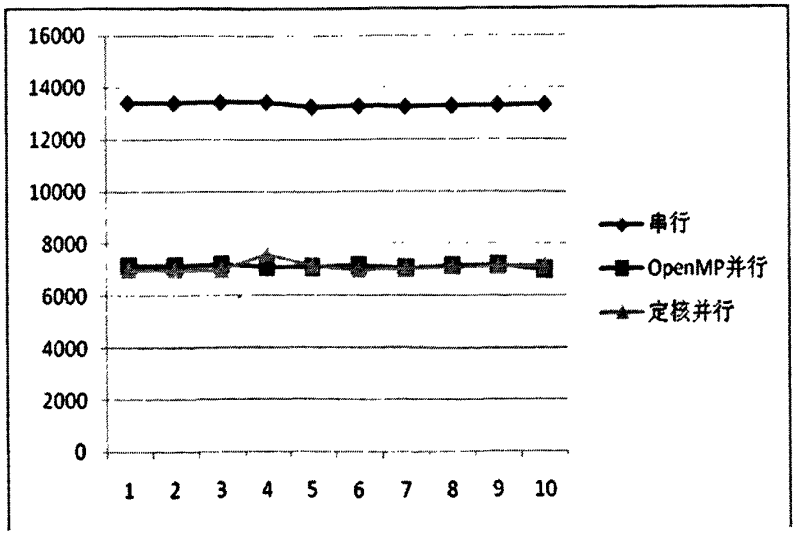

图 5-2 k-means 算法的定核并行与 OpenMP 并行

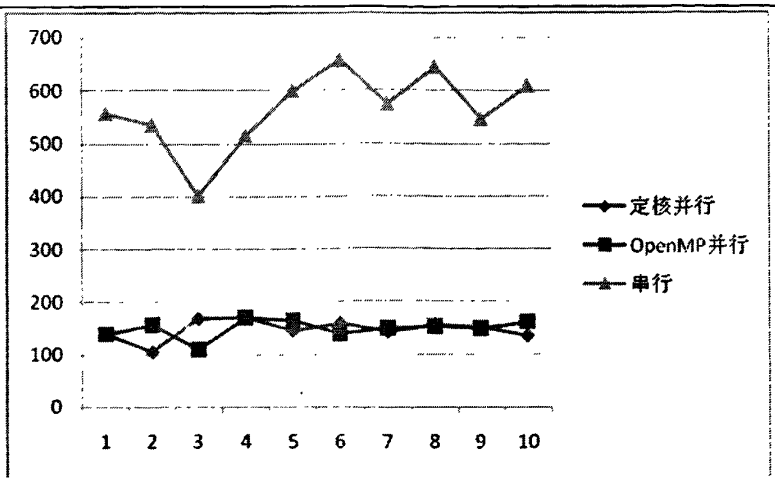

图 5-3 博弈树搜索算法的定核并行与 OpenMP 并行

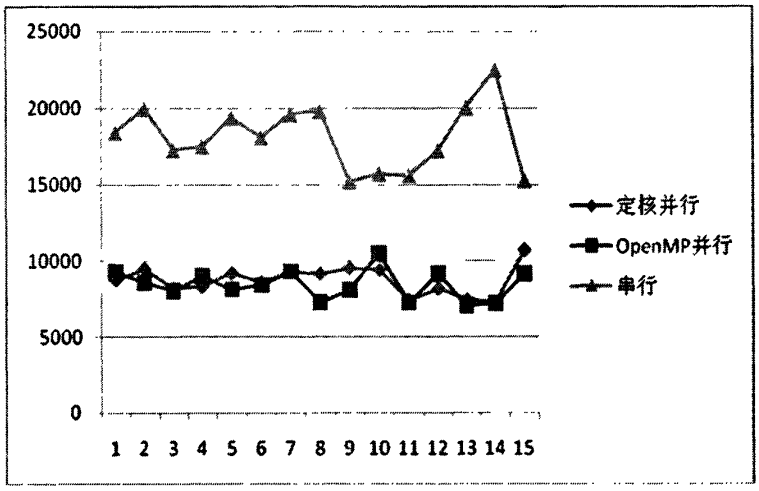

图 5-4 遗传算法的定核并行与 OpenMP 并行

更进一步，由 3.5 节和 4.5 节可以看到搜索树算法在并行定核以后，并行加速比在双核上 1.90.，在四核上 3.84。而遗传算法并行后双核上加速比为 1.22，四核上为 2.15，显然遗传算法的并行加速比在定核的情况下要比搜索树算法的加速比低。分析两个并行算法，我们发现搜索树线程间无数据关联，但并行后的遗传算法多个线程间有一定数据关联，而具有数据关联的各个线程在 cache 共享的多核间执行则有可能引起 cache 颠簸等情况使其性能下降。这也进一步说明定核并行的方法并不不太适合 cache 共享的多核架构。

通过以上测试与分析我们初步认为线程定核执行的并行优化方案更适合总线架构的多核多处理系统。尽管目前的多核处理器，尤其是 Intel 多核处理器大多采用 cache 共享的方式。但按照计算机体系结构的发展趋势，众核将是今后的主要研究方向，而 cache 共享的连接方式并不能适应多达几十核甚至上百核的众核系统。众核大多会采用片上网络的互联结构，该互联结构是对于层次化的总线结构的扩展。多线程定核模型的编程方式会比较适应这样的架构。由于时间的关系，对于 cache 共享对 PLCN 模型的影响没有做更深入的研究，该研究有待后续的工作中进行。

5.3 本章小结

本章对 PLCN 模型进行进一步的分析，同时，在 PLCN 模型下不同并行技术对并行算法性能改善的程度进行了比较和分析，提出了该模型适应的范围，和该模型今后的研究方向。

第六章 总结与展望

6.1 论文工作的总结

多核技术是近几年乃至今后一段时间主要发展的计算机技术之一，也是计算机体系结构方向重点研究的问题之一。它将过去仅靠 ILP 方法提高 CPU 性能延伸到了使用 ILP 与 TLP 技术，共同提高处理器性能。这不仅解决了 ILP 技术目前面临的高功耗和发展空间有限的问题，也是的著名的摩尔定律得以延续。但与 ILP 不同的是 TLP 技术必须需要软件的密切配合才能发挥最大效力。因此，将过去单线程串行的程序进行合理的并行化，并根据体系结构的不同分配线程到核，是一个很值得研究的课题。

本文分析了基于多核多处理架构的系统特点，针对多线程绑定到核的技术对两个经典游戏 AI 算法进行了并行化研究。并对多核多处理器架构的 PLCN 模型进行了进一步的探讨，确定了其适应范围。

论文主要工作包括：

第一. 基于多核多处理器架构的 PLCN 模型提出了线程绑定到每个核的算法。

第二. 基于多核多处理器架构对 Alpha-Beta 剪枝搜索算法进行并行化研究，提出了 BC-Alpha-Beta 剪枝搜索算法。通过验证与分析，得出了并行后的算法不仅速度上明显优于串行程序，而且搜索深度上也可以得到加深，并将该并行算法应用于五子棋程序中进行了实现。

第三. 基于多核多处理器架构对遗传算法进行并行化研究，提出了 BC-Genetic 并行遗传算法，将种群划分成多个子种群，进行并行计算从而提高效率。论文将该算法应用于简单足球游戏中进行了实现。

第四. 对 PLCN 模型进行了进一步的测试与分析，得出了该模型的适应范围，并预测该模型对层次型总线互连与片上网络互连的众核系统具有一定的适应性。

6.2 进一步的研究展望

由第五章分析我们可以在以下两点对 PLCN 模型和线程定核并行技术进行进一步研究。

1） 深入研究共享 cache 下对线程执行带来的性能影响，由于静态分析比较困难，建议在操作系统内核中采用动态性能检测与分析以及动态线程迁移的方法，解决 cache 对性能的影响问题。

2） 进一步研究片上网络的模型，完善在片上网络结构上的 PLCN 模型。可重点考虑网络结构和路由算法对多线程定核技术的影响。

致谢

在论文完成之际，首先，衷心感谢我的导师杨全胜副教授。这篇论文从选题、构思、拟定提纲，开题、撰写、修改直到定稿，杨老师都付出了大量的时间和精力。正是在他的悉心指导、鼓励和严格要求下，我才对多核并行化有了更深入的认识，这篇论文才得以完成。在这两年多的学习和科研中，杨老师严谨的治学态度、敏锐的科研洞察力和锐意进取的工作作风都深深地影响了我。生活中，杨老师和蔼可亲、风趣幽默、体贴入微，他不仅为我提供了便利和宽松的学习环境，而且以其高尚的人格魅力，为我树立了人生的榜样，所有这些都是我一生的宝贵财富。

衷心感谢东南大学的各级领导和老师。感谢他们为我提供的和谐的学习和科研环境，感谢他们在学习和生活中给予的帮助。

感谢实验室 Intel 多核技术小组的王晓蔚老师、阎升、成炼和李正兴同学，他们共同为我营造了一个沟通和交流的平台。并给予我很多帮助和启发，对我的研究有很大的帮助。

感谢实验室所有的兄弟姐妹，他们为我营造了一个和谐进步的小环境，使我快乐的学习、研究和生活。

最后，感谢我的父母、家人和朋友多年来对我一如既往的支持、包容和鼓励。

参考文献

[1] 黄国,睿张平,魏广博.多核处理器的关键技术及其发展趋势.计算机工程与设计,2009.

[2] David N.L.Levy,eds.Computer Games.New York:Springer New York Inc,1988.335-365

[3] 许舜钦. 电脑西洋棋和电脑象棋的回顾与前瞻.电脑学刊台湾 1993 2:1-8

[4] 张玉志.计算机围棋博弈系统[硕士论文].北京:中国科学院计算技术研究,1991.

[5] Kierulf,Anders.Smart Game Board:A Workbench for Game-Playing Programs,with Go and Othello as Case Studies:[Ph.D.Thesis No.9135].Switzerland:Swiss Federal Institute of Technology(ETH) Z"urich,1990.

[6] 蔡自兴,徐光祐.人工智能及应用.北京：清华大学初版社,1996.

[7] 陈国良等.并行计算机体系结构.北京:高等教育出版社,2002.

[8] 王小平，曹立明等.遗传算法一理论、应用与软件实现.西安:西安交通大学出版社,2002.

[9] 陈国良等.并行算法一结构·算法·编程.北京:高等教育出版社,2003(第二版).

[10] 陈国良等.并行算法实践.北京:高等教育出版社,2004.

[11] 于涛. 基于 Intel 多核架构的循环语句自动并行化关键技术的研究[硕士论文].南京:东南大学,2009.

[12] Intel 64 and IA-32 Architectures Optimization Reference Manual[EB/OL]. http://www.intel.com/products/processor/manuals.html, 2005

[13] Intel.Intel CoreTM Microarchitecture[EB/OL]. http://www.intel.com/products/processor/manuals.html, 2006

[14] Michael J.Quinn 著.陈文光,武永卫等译.MPI 与 OpenMP 并行程序设计.北京:清华大学出版社,2004.

[15] Zou Feng,Yang Quan Sheng. The Research of the Multithreaded Allocation Algorithm based on Greedy Algorithm in Multi-core and Multi-processor System, 1st International Conference on Information Science and Engineering (ICISE2009),2009.

[16] Ferrante, J., Ottenstein, K.L., Warren, J.D. The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems, 1987, 9(3): 319-349.

[17] 王小春编著.PC 游戏编程(人机博弈).重庆:重庆大学出版社,2002.

[18] 王骐,博弈树搜索算法的研究及改进.浙江:浙江大学,2006.

[19] 王骄,李思仲,徐长明等.中国象棋计算机博弈问题研究(三)基本博弈搜索引擎,2003.

[20] 戴锦馄,计算机象棋,微计算机应用,1994,15(3):5-11.

[21] 邹峰,杨全胜. 基于多核多处理器的 k-means 算法并行优化研究.东南大学校庆报告优秀论文,2009.

[22] J.Holland,Adaptation in Natural and Artificial Systems,1975.

[23] Lucas A. Wilson, Michelle D. Moore, Jason P. Picarazzi, Simon D. San MiquelParallel Genetic Algorithm for Search and Constrained Multi-objective Optimization, Proceedings of the 18th International Parallel and Distributed Processing Symposium,2004.

[24] 周远晖,陆玉昌,石纯一.基于克服过早收敛的自适应并行遗传算法[J].清华大学学报，1998, 38(3):93-95.

[25] Pedro Lima, Robotic Soccer Austria.Itech Education and Publishing,2007.

[26] 郑志军,郑守淇.粗粒度并行遗传算法分析.小型微机计算机系统,2006,27(6):1002-1006.

[27] Lawrenec L.Larmore.Parallel construction of optimal alphabetic trees. Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures,1993.

[28] Plamenka Borovska, Milena Lazarova.Efficiency of Parallel Minimax Algorithm for Game Tree Search. International Conference on Computer Systems and Technologies - CompSysTech'07,2007.

[29] Jonathan Schaeffer. Improved parallel alpha-beta search. Proceedings of 1986 ACM Fall joint computer conference,1986.

[30] 徐磊,徐莹,张丹丹.多核构架下 OpenMP 多线程应用运行性能的研究计算机工程与科

学,2009,31(11)50-57.

[31] 刘轶,张昕,李鹤等.多核处理器大规模并行系统中的任务分配问题及算法[J].沈阳:小型微型计算机系统,2008,5,972-975.

[32] 张锦熊,韦化.基于 OpenMP 的对称矩阵 LDL_T 分解并行算法实现[J].桂林:广西科学院学报. 2008,24 248-250.

[33] 孙安香,张理论,宋君强.并行计算的数据重分配[J]. 国防科技大学学报 2002.43-47.

[34] 雁玉忠 串行程序并行化技术研究与一种新实现构想[D]:[硕士学位论文]成都:西南交通大学计算机科学与技术学院,2003.

[35]C.McNairy and R.Bhatia,Montecito:A dual-core,dual-thread Itanium processor,IEEE Micro ,2005,25.

[36]C.Seungryul Hill-climbing SMT processor resource distribution ,Ph.D.thesis,University of Matyland,2006.

[37]McGraw, James R., and Timothy S. Axelrod. "Exploiting multiprocessors: Issues and options." In Robert G. Babb II, (ed.), Programming Parallel Processors, Pages 7-25.Reading, MA: Addison-Wesley, 1988.

[38]Kenjiro T, Kenji K, Toshio E, et al. A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources[C]//Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego, USA: [s. n.], 2003.

[39] 杨忠明，黄道，王行愚. 基于模拟退火的动态聚类算法[J].控制与决策，1997，12(suppl):520-523.

[40] 刘健庄，谢维信，黄建军等.聚类分析的遗传算法方法[J].电子学报. 1995. 23(11):81-83.

[41] Fralkenauer E. The Grouping Genetic Algorithms[J]. Widening the scope of the Gas.Belgian Journal of Operation Research Statistics and Computer Science, 1993, 33:79-102.

[42] AL-Sultan K S, sclim S Z.A Simulated annealing algorithm for the clustering problem[j]. Pattern Recognition, 1991, 24(10):1003-1008.

[43] 毛嘉莉. 聚类 K-means 算法及并行化研究[D].重庆大学硕士论文. 2003.

[44] 戴文华，焦翠珍，何婷婷.基于并行遗传算法的 K-means 聚类研究[J].计算机科学 2008. 35(6):171-174.

[45] 王辉，张望，范明.基于集群环境的 K-means 聚类算法的并行化，河南科技大学学报[J]. 2008. 29(4):42-45.

[46] [加] Jiawei Han，[加] Micheline Kamber 著. 范明，孟小峰等译. 数据挖掘概念与技术[m]. 机械工业出版社，2001.

作者简介

个人信息：

邹峰，男，1985.10，籍贯江苏，硕士在读。专业：计算机系统结构。研究方向：多核程序设计。

教育背景：

2007.9 – 2010.6	东南大学计算机科学与工程学院	计算机系统结构专业	硕士
2003.9 – 2007.6	扬州大学信息工程学院	计算机科学与技术专业	学士

读研期间发表的论文：

[1] 邹峰，杨全胜等.基于多核多处理器的k-means算法并行优化研究，东南大学校庆报告优秀论文，2009.4.

[2] Zou Feng, Quan Sheng Yang,Sheng Yan, Xiao Wei Wang. The Research of the Multithreaded Allocation Algorithm based on Greedy Algorithm in Multi-core and Multi-processor System,The International Conference on Information Science and Engineering, ICISE2009,2009.7.

读研期间参加的项目：

[1] 教育部-英特尔信息技术专项科研基金项目。项目编号：MOE-INTEL-08-12。项目名称：基于 Intel 多核架构的程序并行优化技术研究