1　引言

计算机系统结构以功能和性能为目标，紧密围绕计算、存储、网络三方面来提高系统性能，以满足社会不断增长的需求。指令执行速度（IPS，Instructions Per Second）作为衡量计算机性能的一种方式，有多种技术措施可以对其进行改善提升。本文将从硬件、软件两方面对提升计算机的指令执行速度的多种技术措施进行论述。

2　技术措施

2.1 硬件

2.1.1　提升CPU的主频

摩尔定律表明，集成电路上的晶体管数目也就是CPU的主频每过18个月就会翻一番。但到了 21 世纪，单核 CPU 的主频已经很难保持这样的速度继续提升了。原因很容易理解，因为计算机硬件是不可能超出制作材料本身的限制无限发展的，再考虑到高性能CPU的高功耗问题，所以CPU主频的提升虽然能对计算机系统的指令执行速度有所帮助，并且能通过增加晶体管的密度与降低电压来改善问题，但还是受到一定程度的限制。

2.1.2　优化指令

CPI是衡量计算机在相同主频下的性能的一种指标。通过对指令集进行优化，减少每条指令的平均时钟周期数CPI。在此基础上产生的精简指令集对指令的执行效率也带来了提升，即指令系统尽可能只包含高使用频率的少量指令。

2.1.3　指令级并行

指令级并行可以让CPU上的多个功能单元同时执行指令，以此提高处理器的性能。其中包括流水线——分阶段安排使用CPU的功能单元；多发射——同时启动多条指令两种方式。

流水线技术将复杂的指令功能进行了分解，若干个子操作组成一个主操作，且任一子操作在处理器中拥有独立的硬件，则可以通过让所有子操作同步执行来获得加速：每个操作将其结果交给下一个操作，并接受前一个操作的输入。

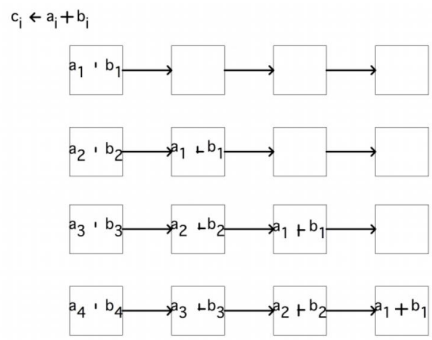

图1 流水线技术的演示图

而多发射技术可以视作超标量技术的实现基础：超标量处理器分析多个指令以找到数据相关性，并行执行彼此不依赖的指令。

2.1.4　从单核处理器到多核处理器

上文提到单核CPU的性能提升受到制作材料与功耗的限制，那么就需要思考、采取其他的硬件技术，即多核CPU的设计与应用。这种方式我认为可以在一定程度上可以视作用“数量”换取“质量”，用多个相对简单的处理器去替代过于复杂、昂贵的单处理器。通过提高芯片内的并行处理能力，引入数据的并行性，来提高整体的性能。

但是普通的程序并不能用并行处理器去运行，因此我们需要将串行的程序转化为并行的程序，并通过改善其中的细节，来利用并行处理器去高效的运行程序。

多核处理器可以通过利用并行性来加快单个代码的执行速度的方式来提高计算机性能，这可以通过两种不同的方式实现。MPI库通常用于通过网络连接的处理器之间的通信。另一种方式是使用共享内存和共享缓存，并使用线程系统如OpenMP进行编程。这种模式的优点并行性可以更加动态，因为运行时系统可以在程序运行过程中设置和改变线程和内核之间的对应关系。

在共享内存中，所有处理器都访问相同的内存；我们也说它们有一个共享地址空间。因此，如果两个处理器都引用一个变量 x，它们就会访问同一个内存位置

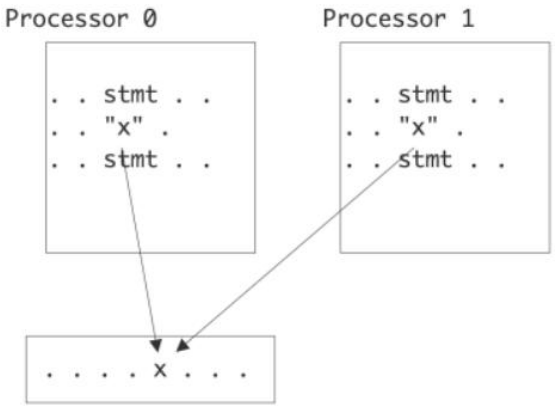

图2 共享内存系统访问同一变量x的位置

2.1.5　乱序

在支持动态多发射技术的系统中可以乱序的执行指令。乱序就是一条指令在从内存中加载数据的同时，先执行后面的指令。乱序技术在多条指令不相关的前提下，能够节约指令等待的时间，提高计算机的CPI。

2.1.6　缓存Cache

缓存技术的诞生是对于传统的冯诺依曼结构计算机，CPU和主存的分离会导致冯诺依曼瓶颈，限制指令执行的效率。缓存是CPU与主存的中间存储器，利用了局部性原理，可以显著的降低访问主存的延迟。在通过其他技术提升了计算机系统的计算能力的情况下，使用缓存可以提升计算机的存储性能，以此来综合提升计算机的指令执行效率。

2.1.7　集成主存控制器

随着CPU频率的提高，低速的内存成为CPU提升频率的阻碍，因此出现了前端总线。CPU通过高速的前端总线和主板的北桥芯片连接，北桥上有内存控制器用来访问内存。CPU集成内存控制器可以显著减少访问内存的延迟。

2.1.8　单指令多数据流系统SIMD

传统的冯诺依曼结构计算机是单指令流单数据流系统SISD，而单指令多数据流系统SIMD能够对多条数据项进行相同的指令操作，达成数据并行，比较适合处理大型的数据并行问题。

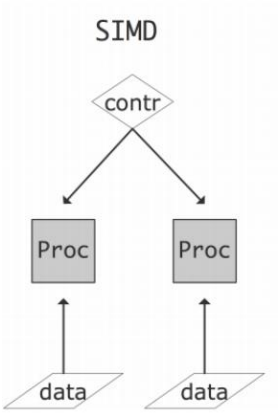

图3 单指令多数据流系统SIMD

2.1.9　多指令多数据流系统MIMD

多指令多数据流系统MIMD能够对多条数据项进行不同的指令操作，能够实现异步，即每个处理器可以看作是自主的，按照自己的节奏进行计算处理，能够整体提升指令的执行效率。

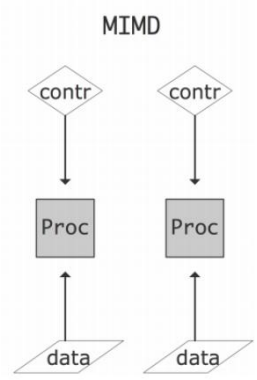

图4 多指令多数据流系统MIMD

2.1.10　对称多处理器SMP和单芯片多处理器CMP

对称多处理器SMP上的多个CPU可以共享内存和总线结构，可以提升并行度。而CMP将SMP集成到同一芯片上，处理器一对一的负责进程，实现共享缓存、提高缓存利用率，进一步提高并行度。

2.1.11　超线程

超线程是英特尔的一项技术，让多个线程真正同时使用处理器，这样处理器的一部分将得到最佳利用。如果一个处理器在执行一个线程和另一个线程之间切换，它将保存一个线程的本地信息，并加载另一个线程的信息。

GPU有对多线程的支持。这意味着硬件实际上对多个线程的本地信息有明确的存储，其每个内核可以支持多达四个线程。

2.1.12　并行计算集群

多个单独的计算机通过网络连接起来形成计算集群。每个节点都是一个多处理器并行机；多个计算节点通过 Infiniband 网络连接。

2.2 软件

现在已处于并行硬件的时代。绝大多数计算机使用多核处理器，大部分指令都可以并行执行，但我们要考虑并行后的代价：并行后程序是否变得简单，以及并行后加速效率是否明显等问题。

2.2.1　OpenMP

OpenMP 是对编程语言 C 和 Fortran 的一个扩展。它的主要并行方法是循环的并行执行：基于编译器指令，预处理器可以安排循环迭代的并行执行。

由于 OpenMP 是基于线程的，它的特点是动态并行：在代码的一个部分和另一个部分之间，并行运行的执行流的数量可以变化。并行性是通过创建并行区域来声明的。

与 MPI 相比，OpenMP 的一个重要优势在于它的可编程性：可以从一个串行代码开始，通过增量并行化来改造它。

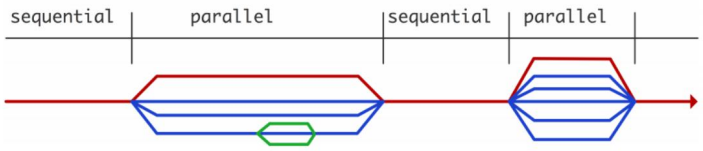

图5 OpenMP线程的创建和删除

并行编程远比串行编程更为复杂。在共享内存系统中的一种并行方式是：用户编写普通程序，并借助 OpenMP 向编译器发布何时并行或如何并行的指令。因此我们需要注意的哪些代码块可以被并行化，如果不能，怎么修改能让它可以被并行化，并且要考虑数据依赖、循环依赖、数据竞争等问题，保证运算结果的正确性、准确性。

OpenMP的简单使用——以critical指令和reduction子句为例：

critical指令表明该指令包裹的代码块只能由一个线程来执行，不能被多个线程同时执行。如果多个线程试图执行同一个critical代码块的时候，其他线程会被堵塞，直到上一线程完成了代码块的操作，下一个线程才能继续执行这一代码块。

reduction通过规约，避免数据竞争，表明某个变量私有，每个线程创建变量的副本，在线程结束后加到全局变量上去。

以下代码实现了从0到9999999的求和：

```
#include <omp.h>
#include <iostream>

int main(int argc, char *argv[])
{
    double spend_time;
    long i;
    unsigned long long sum = 0;
    omp_set_num_threads(8);

// 串行
    double ts = omp_get_wtime();    //计时开始
    for (i = 0; i < 10000000; i++)
        sum += i;

    double te = omp_get_wtime();    //计时结束
    spend_time = te - ts;
    printf("sum=%lld serial computing count=%f\n", sum, spend_time);

// critical 并行
    sum = 0;
    ts = omp_get_wtime();   //计时开始
#pragma omp parallel for
    for (i = 0; i < 10000000; i++)
#pragma omp critical
        sum += i;
    te = omp_get_wtime();   //计时结束
    spend_time = te - ts;
    printf("sum=%lld parallel computing with critical count=%f\n", sum,
spend_time);

// reduction 并行
    sum = 0;
    ts = omp_get_wtime();   //计时开始
#pragma omp parallel for reduction(+: sum)
    for (i = 0; i < 10000000; i++)
        sum += i;
    te = omp_get_wtime();   //计时结束
    spend_time = te - ts;
    printf("sum=%lld parallel computing with reduction count=%f\n", sum,
spend_time);

    system("pause");
}
```

运行结果：

```
sum=49999995000000 serial computing count=0.032000
sum=49999995000000 parallel computing with critical count=50.342000
sum=49999995000000 parallel computing with reduction count=0.007000
```

分析：

Serial computing是串行执行的循环；parallel computing with critical基于串行循环添加了并行化，为避免数据冲突将 sum += 1 设为临界区，执行时增加了加解锁的操作。虽然结果一致，但是过多的加解锁带来了沉重的开销，导致运行效率低下，一定程度上可以视为串行；parallel computing with reduction是正确的OpenMP编程，既利用了多线程提高了并行的效率，又避免了同步操作产生的开销。

使用正确的并行程序得出的加速比为：4.571

以一个更复杂的OpenMP程序为例（为节省篇幅，此处不附上代码）：

程序主要功能：对一个2000阶的方阵实现LU分解——LU分解是线性代数领域中矩阵分解的一种，将一个矩阵分解为一个下三角矩阵和一个上三角矩阵的乘积。

实验结果：

表1 利用OpenMP编程实现LU分解

线程数		1	2	4	8	16
耗时（单位:s）	第1轮	23.677	15.937	11.327	10.352	12.445
	第2轮	23.475	14.921	11.352	10.061	11.180
	第3轮	18.813	15.335	10.120	9.957	10.676
平均耗时		21.99	15.40	10.93	10.12	11.43
加速比		1	1.43	2.01	2.17	1.92

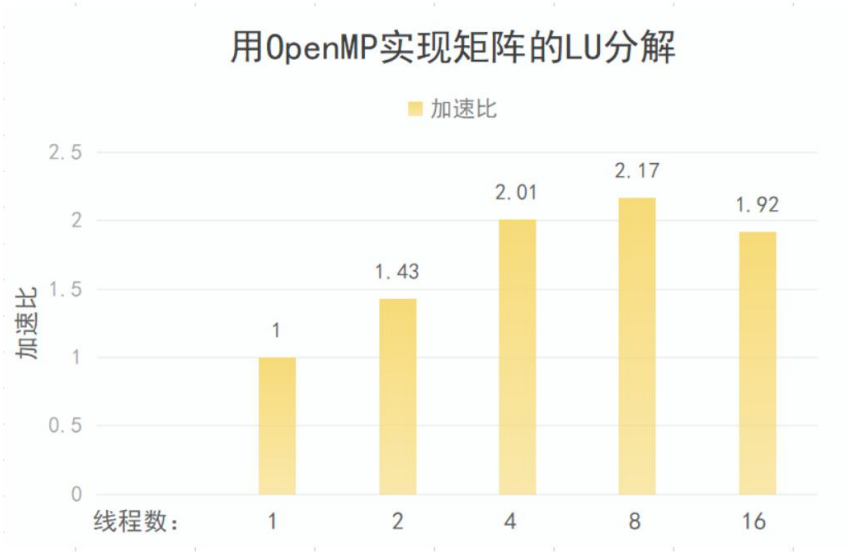

图6 运行结果的柱形图

分析：

随着线程数的增加，加速比先逐渐增大至最高，再产生略微的下降。当线程数增加，越来越充分利用物理 CPU 核的并行资源。由于物理CPU核有限的，当线程数增加到一定的数额后再增加线程数也无法从物理上增加程序运行的并行程度，反而因为线程数的增加，带来切换线程、线程通信和管理等的额外开销，所以最终运行时间增加，加速比下降。

2.2.2 MPI

MPI是一个跨语言的通讯协议，可以用于编写并行计算机程序，其实现由一些指定惯例集组成。MPI的追求目标是高性能，大规模和可移植性。MPI在今天仍是高性能计算的主要模型。

MPI示例程序：

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc,char *argv[])
{
    int myid,numprocs;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf("Process %d of %d\n", myid, numprocs);

    MPI_Finalize();
}

/*
编译命令:mpicc -o pi.exe pi.c
运行命令:mpirun -np 6 pi.exe

运行结果:
Process 1 of 6
Process 2 of 6
Process 3 of 6
Process 4 of 6
Process 0 of 6
Process 5 of 6
*/
```

2.2.3　CUDA

CUDA是英伟达发明的并行计算平台和编程模型。它通过利用图形处理器独特的处理能力，大幅提高了计算性能。CUDA并行计算架构可以使GPU解决复杂的计算问题。

CUDA示例程序：

```c
#include <stdio.h>
__global__ void hello(){
    printf("Hello, threadIdx is:%d\n",threadIdx.x);
}

int main(){
    hello<<<1,8>>>();
    cudaDeviceReset();
}

/*
编译命令:nvcc -o main main.cu
运行命令:./main

运行结果:
Hello, threadIdx is:0
Hello, threadIdx is:1
Hello, threadIdx is:2
Hello, threadIdx is:3
Hello, threadIdx is:4
Hello, threadIdx is:5
Hello, threadIdx is:6
Hello, threadIdx is:7
*/
```