并行程序设计与算法 第四次作业

刘森元, 21307289

中山大学计算机学院

1 简答题

1.1 习题 1

假定在 Bleeblon 计算机上，浮点型变量能够存储小数点后 3 位数字，他的浮点寄存器可以存储小数点后 4 位，并且在任意的浮点操作后，结果存储在前被四舍五入为小数点后 3 位。现在假设一个 C 程序声明了一个数组：

```
float a[] = {4.0, 3.0, 3.0, 1000.0};
```

考虑如下代码：

```
int i;
float sum = 0.0;
#pragma omp parallel for num_threads(2) \
        reduction(+:sum)
    for (i = 0; i < 4; ++i)
        sum += a[i];
    printf("sum = %4.1f\n", sum);
```

假设系统将迭代 `i = 0, 1` 分配给线程 0，将迭代 `i = 2, 3` 分配给线程 1，那在该 Bleeblon 计算机上，程序的输出是什么?

在 Bleeblon 计算机上，浮点型变量能够存储小数点后 3 位数字，浮点寄存器可以存储小数点后 4 位，并且在任意的浮点操作后，结果存储在前被四舍五入为小数点后 3 位。这意味着每次浮点数的加法操作都会进行四舍五入。

在这个例子中，线程 0 负责处理 `a[0]` 和 `a[1]`，线程 1 负责处理 `a[2]` 和 `a[3]`。因此，线程 0 的 sum 会是 `4.0 + 3.0 = 7.0`，线程 1 的 sum 会是 `3.0 + 1000.0 = 1003.0`。

然后，这两个 sum 会被加在一起，得到 `7.0 + 1003.0 = 1010.0`。但是，由于 Bleeblon 计算机的特性，这个结果会被四舍五入为小数点后 3 位，所以最终的结果是 `1010.0`。

所以，程序的输出应该是 `sum = 1010.0`。

1.2 习题 2

考虑循环

```
a[0] = 0;
for (i = 1; i < n; ++i)
    a[i] = a[i - 1] + 1;
```

在该程序中存在循环依赖。

(1) 分析该程序中存在的循环依赖，并设计改写程序消除此依赖

在这个循环中，每次迭代都依赖于前一次迭代的结果，因为 `a[i]` 的值依赖于 `a[i - 1]`。这就是循环依赖。为了消除这种依赖，我们可以直接将 `i` 赋值给 `a[i]`，因为 `a[i]` 实际上就是 `i`。

```
for (i = 0; i < n; ++i)
    a[i] = i;
```

(2) 加入 OpenMP 指令，对改写后的程序并行化

使用 OpenMP 并行化这个循环：

```
#pragma omp parallel for
for (i = 0; i < n; ++i)
    a[i] = i;
```

这样，每个线程都会处理一部分迭代，而这些迭代是独立的，因此不会有任何依赖问题。

1.3 习题 3

我们考察 8000x8000 作为之前的矩阵-向量乘法程序的输入时该程序的性能，如果一个缓存行包含 64 字节或者 8 个双精度数，将输入向量表示为 y，那么：

(1) 假定线程 0 和线程 2 被分配给了不同的处理器，在线程 0 和线程 2 之间的伪共享 (false-sharing) 可不可能在向量 y 上发生？为什么？

不可能发生。

如果一个缓存行可以存储 8 个双精度数，那么向量 y 将被分成 1000 个缓存行。

如果我们按照线程编号的模数 4 来分配这些缓存行（因为我们有 4 个线程），那么线程 0 和线程 2 将分别处理那些模数为 0 和 2 的缓存行。

这意味着它们将处理不同的缓存行，因此不会发生伪共享。

(2) 如果线程 0 和线程 3 被分配给了不同的处理器，那么伪共享 (false-sharing) 可不可能发上在向量 y 的任何地方？

可能发生。原因是，如果我们按照线程编号的模数 4 来分配缓存行，那么线程 0 和线程 3 将分别处理那些模数为 0 和 3 的缓存行。

这意味着它们可能会处理相邻的缓存行。

如果一个线程写入其处理的缓存行，而另一个线程正在读取或写入相邻的缓存行，那么就可能发生伪共享。因为当一个线程写入一个缓存行时，整个缓存行（包括相邻的数据）都会被标记为无效，从而导致其他线程必须从主内存中重新加载数据。

1.4 习题 4

使用一维数组和 OpenMP 指令来实现并行的矩阵-向量乘法，其中矩阵为 `float A[m*n]`，输入向量为 `float x[n]`，结果向量为 `float y[m]`。(手写代码给出关键的循环部分即可)

```
#pragma omp parallel for
for (int i = 0; i < m; ++i) {
    y[i] = 0.0;
    for (int j = 0; j < n; ++j) {
        y[i] += A[i*n + j] * x[j];
    }
}
```