cilk plus 调研报告

吕文龙
14210720082

1. cilk plus简介

Cilk Plus是一个对C和C++的语言扩展, 为C与C++增加了细粒度并行的扩展, 支持数据级与任务级的并行. 支持IA-32 架构或者Intel® 64 架构上的并行编程,同时支持Windows/Linux/OS X操作系统.

2. 编译器支持

可以在以下编译器编译的代码中, 使用Cilk_plus进行并行化加速

- Intel C++ Compiler(ICC)
- GNU Compiler Collection(GCC)
- Clang/LLVM

3. 基本语法

3.1 并行模式抽象

Cilk同时支持数据级(task level)和任务级(data level)的并行. 通过cilk_spawn/ cilk_sync/cilk_for来支持fork-join类型的任务, 尤其适合divide-and-conquer类型的算法, 通过reducer处理不同worker的共享变量, 通过array notation和#pragma SIMD来支持数据并行

3.2 cilk_spawn

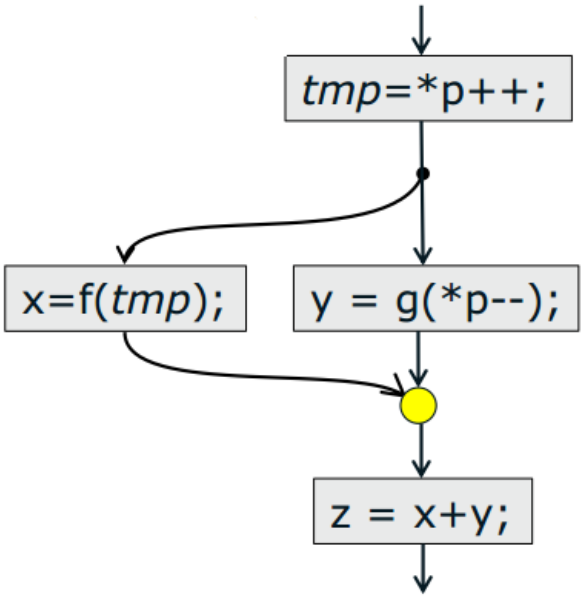
通过cilk_spawn来修饰函数调用语句, 通知编译器该语句可以(但不必须)被并行调用. cilk_spawn的调用格式如下:

```
/*
args在spawn发生前就被求值
*/
type var = cilk_spawn func(args); //func() 有返回值,type为变量类型
var = cilk_spawn func(args); // func() 有返回值
cilk_spawn func(args) ; // func() 为void
```

cilk_spawn的用法如下:

```
x = cilk_spawn f(*p++);
y = g(*p--);
cilk_sync;
z = x+y;
```

cilk_spawn告诉编译器f()与g()可以并行执行. 但是并不保证一定会被并行执行.执行过程如下图所示



不能够在函数的参数中使用cilk_spawn, 例如, 下面的用法是错误的:

```
g(cilk_spawn f()); // illegal
```

正确的方法是衍生一个函数来调用f() 和g() 。这可以很容易的用C++ lambda来实现：

```
cilk_spawn [&]{ g(f()); }();
```

注意到上面的表达方式和下面是不同的:

```
cilk_spawn g(f());
/*
在此例中, f()是在cilk_spawn之前的主线程中被求值,
而在上面的例子中, g(f())都是在被spawn出的线程中执行
*/
```

3.3 cilk_sync

cilk_sync语句表示当前函数不能继续和衍生的子任务并行执行. 在所有子任务执行完之后, 当前函数才能继续执行.
语法规则如下：

```
cilk_sync;
```

cilk_sync只同步由该函数衍生的子任务。其他函数的子任务不受影响。

3.4 cilk_for

3.4.1 cilk_for 基本语法

cilk_for语法与常规的for循环相似, 它允许循环体中的迭代并行执行.
下面为cilk_for的例子:

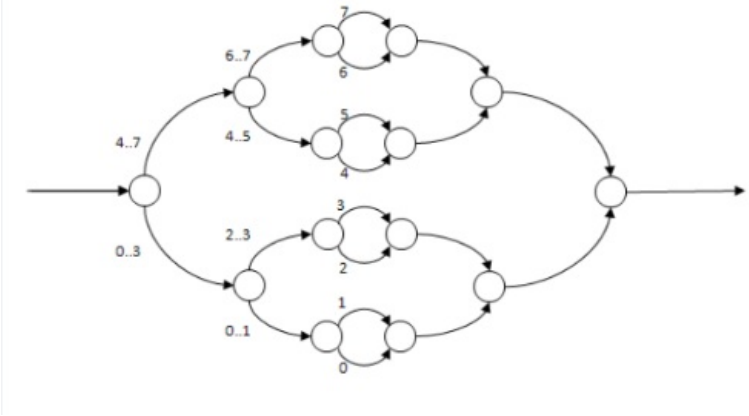
```
cilk_for (int i = begin; i < end; i += 2)
    f(i);
cilk_for (T::iterator i(vec.begin()); i != vec.end(); ++i)
    g(i);
```

cilk_for语法的基本规则如下:

```
cilk_for (declaration; conditional-expression; increment-expression)
    body
```

- declaration中必须定义和初始化某个变量,称为**控制变量**
- conditional-expression表示循环终止条件, 比较**控制变量**和**终止表达式**必须使用如下的比较操作符进行比较判断循环终止:
  1. <
  2. <=
  3. !=
  4. >=
  5. >
- 终止表达式的值不能随不同迭代而改变
- **增量表达式**increment-expression必须采用下面的操作符来修改控制变量,增加或减少的值也不能随循环体不同迭代而改变:
  1. +=
  2. -=
  3. ++
  4. --

Intel编译器会把cilk_for转化成一个函数, 然后采用二分法策略递归的调用它, 如下面有向无环图所示:



3.4.2 cilk_for循环体内的同步

cilk_for的循环体中的cilk_sync只需要等待同次迭代中spawn出的子任务, 而不需要与其他迭代进行同步.并且, 在每次迭代最后, 会有隐式的cilk_sync,任何在cilk_for循环中spawn的子任务都确保会在循环结束前同步, 相反, 任何在循环前就存在的子任务不会在循环过程中发生同步.

3.4.3 cilk_for的限制

为了能够正确的并行化循环, cilk_for的语法相对与c++中的for循环, 增加了许多限制. 运行时系统必须提前计算出循环的总次数和每次迭代中控制变量的值. 具体来讲:

- 必须只存在一个控制变量, 并且循环初始化从句必须对它赋值, 如下面的代码是不支持的:
  ```
  cilk_for (unsigned int i, j = 42; j < 1; i++, j++)
  ```
- 控制变量不能够在循环体内修改, 下面为反例:
  ```
  cilk_for (unsigned int i = 1; i < 16; ++i) i = f();
  ```
- 循环终止条件不能够在循环体内进行计算, 例如,下面的代码是不支持的:
  ```
  cilk_for (unsigned int i = 1; i < x; ++i) x = f();
  ```
- 不能够在cilk_for的循环体中使用break/return
- goto语句只能goto带循环体中的目标行, 不能用goto语句转入或者转出cilk_for的循环体

3.4.4 cilk_for的粒度

cilk_for语句将循环划分到包含一次或者多次循环迭代的区块中, 每个区块中的循环迭代串行执行. 每个区块包含的最大迭代次数称为粒度(grainsize). 当循环迭代数较多时, 大的粒度可以减少并行开销.而当迭代次数较多时, 小的粒度可以增加程序的并行度.

可以使用cilk grainsize的编译指示来指定一次cilk_for的粒度:

```
//粒度为1
#pragma cilk grainsize = 1
cilk_for (int i=0; i<IMAX; ++i) { . . . }
```

如果没有设置粒度或者设置粒度为0, 则会采用系统默认粒度. 默认值的设置等价于添加了如下编译指示:

```
#pragma cilk grainsize = min(512, N / (8*p))
```

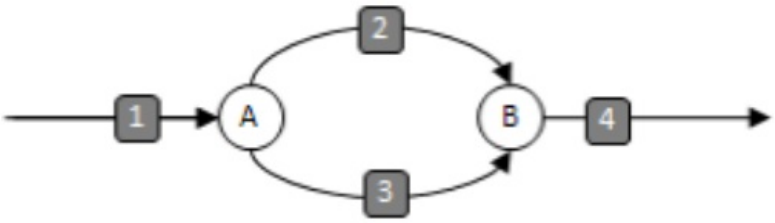
其中, N为循环迭代次数, p为当前程序创建的工作线程数量

3.5 执行模型

单词strand 用来描述程序中串行部分. cilk程序包含spawn和sync两种并行控制结构(cilk_for其实是spawn/sync的一个特例),以下面这段代码为例:

```
do_stuff_1(); //执行 strand 1
cilk_spawn func_3(); //在 A 时刻衍生 strand 3
do_stuff_2(); //执行 strand 2
cilk_sync; //在 B 时刻进行同步
do_stuff_4();
```

代码可以用下面的有向无环图(DAG)表示:



图中, 边表示strand,节点表示并行控制结构. 图中, 包含有4个strand, 一个spawn(A)和一个sync(B), 其中, strand(2)/strand(3)可以并行执行.

程序的有向无环图模型, 是不依赖于处理器数目的. 而cilk的执行模型描述了调度程序如何将strand映射到工作线程上. 仍然以上面的代码为例, 当有多个工作线程可用时, 对于strand(2)/strand(3), 程序有两种选择: 整个程序由单一线程完成, 或者调度程序选择在不同的工作线程上执行strand(2)和strand(3).

而为了保持串行语义, spawn出的函数(strand(3))总是和执行spawn的strand(strand(1))在同一个工作线程上, 此时, 如果有其他工作线程可用, 则strand(2)可以在另外的工作线程上执行, 称之为密取(stealing).

执行模型总结如下:

- 在 cilk_spawn 之后, 子任务总是和调用者在同一工作线程(即系统线程)上执行.
- 在 cilk_spawn 之后, 延续部分可以在另一工作线程上执行.
- 在 cilk_sync 之后, 程序执行可以由进入同步接点的任一工作线程继续.

3.6 reducer

从概念上讲, reducer是一个能够被多个并行执行的strand安全使用的共享变量. 每个工作线程存取一个改变两的私有拷贝, 当执行同步时, 这些变量被合并. reducer不需要加锁, 同时能够保持程序的串行语义.

3.6.1 example

下面是reducer的一个简单的例子:

```
//串行程序, 整数从1到n求和
#include <iostream>
unsigned int compute(unsigned int i)
{
    return i; // 返回通过i计算得到的值。
```

```
}
int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    unsigned int total = 0;
    // 计算整数1到n的和
    for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i) ;
    }
    // 1到n的求和结果应该是 n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;
    if (total == correct)
        std::cout << "Total ("
                  << total
                  << ") is correct"
                  << std::endl;
    else
        std::cout << "Total ("
                  << total
                  << ") is WRONG, should be "
                  <<correct
                  << std::endl;
    return 0;
}
//reducer 并行化规约
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#include <iostream>
unsigned int compute(unsigned int i)
{
    return i; // 通过i计算返回值
}
int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    cilk::reducer_opadd<unsigned int> total;
    // 计算 1..n
    cilk_for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i) ;
    }
    //1到n的求和结果应该是n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;
    if (total.get_value() == correct)
        std::cout << "Total ("
                  << total.get_value()
                  << ") is correct"
                  << std::endl;
    else
        std::cout << "Total ("
                  << total.get_value()
                  << ") is WRONG, should be "
                  <<correct
                  << std::endl;
    return 0;
}
```

3.6.2 reducer原理

一个reducer可以被简化为一个包括数值, 与规约函数的对象. 而reducer的行为描述如下:

- 一个reducer总是对应着一种规约操作, 如加法, 字符串相连等, 规约操作应当具有零元(如加法的零元是0, 乘法的零元是1, 字符串相连的零元是空字符串), 同时应当满足结合律, 才能保证结果的正确性.
- 当密取(stealing)没有发生时, reducer的行为与串行代码中的变量是一样的.
- 当密取发生时, 新的线程会获取一个reducer值为reducer零元的新的工作视图,并据此进行后续操作, 而主线程拥有spawn之前的reducer视图, 当同步发生时, 其他线程得到的reducer值与主线程的reducer值进行规约操作, 然后其他工作线程所获得的工作视图被注销, 主线程的工作视图被更新. 出于性能的考虑, cilk在这里采取了类似copy-on-write的技术, 即只有当密取发生后, 新的strand首次存取reducer的时候, reducer才会被真正创建.
- reducer的规约操作只需要满足结合律, 不需要满足交换律, reducer能够自动保持程序的串行语义.

3.7 array notation

Array notation是一种并行化的C/C++数组表示方法, 程序员使用Array notation表示数组, 编译器可以很方便的通过一些变换生成向量化的或者多线程的代码.

3.7.1 数组表示

cilk plus向标准的c/c++中引入了数组子部操作符:

```
section_operator := [<lower_bound>:<length>:<stride>]
//lower_bound,length,stride均为整数
```

使用方法如下:

```
arr[lb:len:stride];
```

表示数组arr[lb],a[lb+stride],a[lb+2*stride]...a[lb+(len-1)*stride]. 注意, 这里用长度取代了数组上界, 更容易确定数组大小, 当stride没有指定时, 默认为1.当stride小于一时, 行为**未定义**, 当数组长度在声明时可以确定, 可以用arr[:]表示整个数组, lower_bound和length必须同时指定或者同时省略, 不能只指定其中一个.

3.7.2 操作符映射

大部分的c/c++操作符都对数组子部有效. 使用这些操作符对数组进行运算时, 它们被隐式的映射到子部的各个成员. 例如:

```
//数组操作符对应的操作数必须具有相同的维数和大小,
a[:] * b[*] //数组对应元素相乘.
a[3:2][3:2] + b[5:2][5:2] //从a[3][3]和b[5][5]开始的两个2*2矩阵相加
```

赋值操作符也可以用于数组子部的每个元素

```
a[:][:] = b[:][2][:] + c;
e[:] = d;
e[:] = b[:] [1][:]; // 错误, 维数不同
a[:][:] = e[:]; // 错误, 维数不同
```

在任一左侧元素被赋值前, 赋值操作符右侧先被计算. 编译器会引入临时数组来确保语义正确性. 例如, 下面的代码因为会创建临时数组, 编译器仍然可以进行向量化:

```
a[1:s] = a[0:s] + 1; // 使用 a[1:s-1] 的原值
```

3.7.3 条件表达式

在分支语句(if)中也可以使用 Array Notation 表达式. 使条件测试为真的数组下标所对应的 if 语句体才会被执行. 可以使用 else-if 和 else 语句. 例如:

```
// 若 a[n] < b[n] , 则 mask[n] 值为-1;  否则值为 1
if (a[:] < b[:])
    mask[:] = -1;
else
    mask[:] = 1;
mask[:] = a[:] < b[:] ? -1 : 1;
```

在一个if-else(条件表达式和分支执行语句)中的所有数组子部的维数大小必须相同, 同时, 语句只能有一个出入口, 不可以包含goto/break/return这样的语句. 也不能从外部跳转到if-else块内.

3.7.4 函数映射

把数组子部作为参数传递给标量函数时, 函数映射将会隐式地把数组子部的每个元素作为参数来调用标量函数. 如果有多个数组子部参数传递给同一个标量函数, 它们维数必须相同. 例如:

```
a[:] = sin(b[:]);
a[:] = pow(b[:], c); // b[:]**c
a[:] = pow(c, b[:]); // c**b[:]
a[:] = foo(b[:]); // 用户自定义函数
a[:] = bar(b[:], c[:][:]); // 错误，维数不同
```

3.8 pragma SIMD

编译指示#pragma SIMD是cilk plus提供的另一种向量化的方式,使用方法如下:

```
void saxpy( float a, float x[], float y[], size_t n )
{
#pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```

SIMD编译指示有5个可选子句, 分别是:

- vectorlength(num1,num2...): 指导编译器从若干个向量长度中进行选择进行向量化
- private(expr1,expr2...): 指导向量优化单元使得这些左值(L-value)表达式 expr1, exper2, ..., exprN 对于每一次循环都是私有的.
- linear(var1:step1, var2:step2 ... varN:stepN): 每次循环,varN的值增加stepN
- [no]assert: 在向量化失败时是否报错

4. 总结

cilk plus对c++的并行编程提供了一种较为简洁而优雅的抽象, 语法上比较清晰, 并且能够同时支持多种并行方式, 不过只能支持Intel处理器上的代码, 降低了一些可移植性.

参考文献

1. Wikipedia contributors, "Cilk Plus," Wikipedia, The Free Encyclopedia, (accessed November 18, 2014)
2. ISC2012_Tutorial_9_CilkPlus_Robison_final
3. Cilk Plus: Language Support for Thread and Vector Parallelism
4. 英特尔® C++编译器Cilk 语言扩展
5. 数组表示法的 C/C++扩展简介