FIBONACCI IN R, C, OPENMP AND CILK

JAN DE LEEUW

ABSTRACT. This note uses the Fibonacci sequence as an example of how to speed up R code using C. As a next step it speeds up the C code on multicore machines by loading thread-safe shared libraries that are optimized using OpenMP or Cilk.

1. COMPUTATION IN R

The Fibonacci sequence starts with $x_0 = 0$ and $x_1 = 1$, and then defines subsequent terms by

$$x_n = x_{n-1} + x_{n-2}. \tag{1}$$

Thus the first eleven terms are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$. It is simple to compute the n-th Fibonacci number in R, using recursion.

```
fibonacci<-function(n){
if (n < 2) return(n)
return(fibonacci(n-1)+fibonacci(n-2))
}
```

If we run this function for any n of reasonable size, however, it is very slow. As Table 1 shows, computing the 45th Fibonacci number in R in this way takes 2.5 hours[1]. Of course this recursion is a pretty

Date: Wednesday 25th November, 2009 — 20h 43min — Typeset in LUCIDA BRIGHT.

[1]The computer I am using for these computations is a MacPro, with two Intel 3 Ghz Quad-Core Xeon processors and 8 GB of 667 Mhz DDR2 FB-DIMM memory, running Mac OS X Leopard 10.5.6. The compiler is gcc version 4.2.1 (Apple Inc. build 5564).

dumb method to compute the Fibonacci numbers, because directly using the definition (1) is much faster.

2. Computation in C

Since the algorithm in R is so slow, there is plenty of room for improvement. One possibility, which still uses the same recursion, is to rewrite the critical parts in C. The code in C is in the chunk below.

```
int fibInt(int n);
void fibC(int *n, int *x);

void fibC(int *n, int *x) {
*x = fibInt(*n);
}

int fibInt(int n) {
if (n < 2) return n;
int x = fibInt(n - 1);
int y = fibInt(n - 2);
return x + y;
}
```

Note that we have written two functions. The function `fibC` uses the conventions needed by the `.C` interface in R. This is the function we will load into R. The other function `fibInt` is merely there to do the computations, and it will not be directly accessible from R.

We can compile our code, and link it into a shared library, with the shell command `R CMD SHLIB fibC.c`. On my system this executes the commands

```
/usr/bin/gcc-4.2 -std=gnu99
    -I/Library/Frameworks/R.framework/Resources/include
    -I/usr/local/include    -fPIC  -g -O2 -c fibC.c -o fibC.o
```

and

```
/usr/bin/gcc-4.2 -std=gnu99 -dynamiclib
    -Wl,-headerpad_max_install_names  -undefined dynamic_lookup
    -single_module -multiply_defined suppress
    -o fibC.so fibC.o   -F/Library/Frameworks/R.framework/..
    -framework R -lintl -Wl,-framework -Wl,CoreFoundation
```

We then write a calling program in R.

```
dyn.load("fibC.so")

fibonacci<-function(n) {
.C("fibC",as.integer(n),as.integer(1))[[2]]
}
```

This version is used to compute the second column of Table 1. Clearly the speedup is dramatic.

3. Computation in OpenMP

We can try to get even better results by more efficiently using the eight cores of the computer's CPU. This means programming with threads, in which multiple tasks are executed concurrently by different cores. Using multithreaded programming techniques is complicated, and there is an avalanche of books and manuals and interfaces available to try to make it more simple. Apple Computer, for example, publishes an excellent Threaded Programming Guide, which discusses POSIX threads, Cocoa threads, and Mach threads. Intel has a very well documented Threaded Building Blocks system. But generally, as a humble statistician, I currently subscribe to the The Folly Of Do-It-Yourself Multithreading credo, and I look for higher level interfaces to the native threads.

Our first attempt relies on the GOMP implementation of OpenMP that comes with the gcc-4.2 compiler. OpenMP puts compiler directives in regular C or C++ code, and consequently requires only fairly minimal modifications to an existing project. It is described in numerous books, and on numerous websites. Note that gcc-4.2 implements the OpenMP 2.5 standard, and not the recent OpenMP 3.0, which will be in gcc-4.4.

We modify our C code by putting in the OpenMP directives. This gives

```
1
2  int fibCInt(int n);
3  void fibOMP(int *n, int *x);
4
5  void fibOMP(int *n, int *x) {
6  #pragma omp parallel
7  *x = fibCInt(*n);
8  #pragma omp end parallel
9  }
10
11 int fibCInt(int n)
12 {
13 int x, y;
14 if (n < 2) return (n);
15 #pragma omp task shared(x)
16 x = fibCInt(n - 1);
17 #pragma omp task shared(y)
18 y = fibCInt(n - 2);
19 #pragma omp taskwait
20 return x + y;
21 }
```

For the compilation we add the `-fopenmp` compiler/linker flag, but otherwise nothing needs to be changed. The results, in the third column of Table 1, are disappointing. This is probably due to my clumsy use of `OpenMP`, and I am looking for improvements.

4. Computation in `Cilk`

`Cilk` is an elegant extension of `C` designed to handle computations on shared memory multiprocessor machines. It was developed at MIT and it adds a limited number of keywords to the `C` language. The `Cilk` system, the manual, and various presentations and papers can be downloaded from the `Cilk` website. A commercial spin-off, called `Cilk++`, is in preparation and has its own website.

The Cilk implementation of the Fibonacci recursion is in the next chunk. Note that it consists of a regular C procedure, calling a Cilk procedure. It seemed necessary to set things up this way, because you cannot call Cilk procedures directly from R. Combining C and Cilk can be done in various ways (section 2.3 of the Cilk manual), but it introduces some complications and an additional layer of glue.

```
#include <cilk.h>

extern int EXPORT(fibCInt)(CilkContext* context, int x
    );

cilk int fibCInt(int n)
{
if (n < 2) return n;
else
{
int a,b;
a = spawn fibCInt(n - 1);
b = spawn fibCInt(n - 2);
sync;
return(a + b);
}
}

void fibCilk(int *n, int *x) {
int argc = 3;
char* argv[]= {"fibCilk","--nproc","8"};
CilkContext* context;
context = Cilk_init(&argc,argv);
*x = EXPORT(fibCInt)(context,*n);
Cilk_terminate(context);
}
```

The source file is compiled using

```
cilkc -I/Library/Frameworks/R.framework/Resources/include
        -I/usr/local/include -I/usr/local/include/cilk
        -fPIC  -g -O2 -c fibCilk.cilk -o fibCilk.o
```

and then linked by

```
gcc-4.2 -std=gnu99 -dynamiclib -Wl,-headerpad_max_install_names
 -undefined dynamic_lookup -single_module -multiply_defined suppress
 -o fibCilk.so fibCilk.o   -F/Library/Frameworks/R.framework/..
 -framework R -lintl -lcilk -Wl,-framework -Wl,CoreFoundation
```

As the last column of Table 1 shows, `Cilk` managing eight threads gives a 30-50% speedup for the larger values of n. We plan to do some additional experimentation with `Cilk`.

Appendix A. Timings

Table 1. Timings

n	R	C	pthreads	OpenMP(4)	OpenMP(8)	Cilk(8)	Cilk(16)
25	0.612	0.001	0.001	0.001	0.002	0.007	0.010
30	6.768	0.007	0.003	0.008	0.010	0.015	0.018
35	75.476	0.078	0.029	0.078	0.091	0.118	0.119
40	840.769	0.856	0.311	0.865	1.017	1.271	1.180
45	9348.170	9.482	3.437	9.567	10.718	14.651	13.206

Department of Statistics, University of California, Los Angeles, CA 90095-1554

E-mail address, Jan de Leeuw: deleeuw@stat.ucla.edu

URL, Jan de Leeuw: http://gifi.stat.ucla.edu