Documentation of KregularKompute code

Philip G. Lee

August 3, 2015

1 General Stuff

First of all, the code provided is copyrighted (or perhaps more specifically, "copylefted") and released under the GPLv3. Basically, if you modify this code to make and distribute something else, you must also release your modified source code. More information is available at `http://www.gnu.org/licenses/`.

You will notice that I haven't provided a general-purpose program to reconstruct values of the k-regular partition functions with the Chinese Remainder Theorem. However, building such a program should be relatively simple given that this suite of functions already contains functions that implement the Chinese Remainder Theorem, and I have created a program (contained in `thm4reconstruct.c`) that reconstructs specific values of the k-regular partition functions.

If you have any questions at all, please email me at `rocketman768@gmail.com`.

2 Making & Running the Executables

To compile everything, you will need `make`, a C compiler, and `mpicc` installed. You will also need the GMP library available at gmplib.org as of March 2008. Also, this code is *probably* unix-specific. If you use something other than `gcc` as a C compiler, change "`CC = gcc`" in `Makefile` to represent what compiler you have. Simply type "`make`" at the command line in the KregularKompute directory and two executables will be built: `main` & `viewer`.

The executable `main` is the parallel algorithm and must be invoked with `mpirun`. The usage is as follows:

```
main --k num --max num --prefix string [--lastmodulus num --maxmodulus num]
```

Note that `k`, `max`, and `prefix` are all required. Options `lastmodulus` and `maxmodulus` are optional, but if one is present, the other must be also. Other options:

- `--version`: prints out version to `stdout` and exits.
- `--usage`: prints out usage info to `stdout` and exits.

The executable `viewer` takes a single argument, which is a filename, reads in that file with format as seen in Figure 1, and prints out the human-readable contents to `stdout`.

3 File Formats

The first file format used within KregularKompute is the .bk file format. As seen in Figure 1, the `terms` field represents the total number of unsigned long numbers in the file other than itself. The rest of the file simply consists of these unsigned long numbers in binary format.

`terms`	unsigned long
a_1	unsigned long
a_2	unsigned long
$\vdots$	$\vdots$
$a_{\texttt{terms}}$	unsigned long

Figure 1: The .bk file format.

4 The Algorithm & its Implementation

The algorithm is as follows:

$$\begin{aligned} b_k(n) &= \frac{1}{n}\sum_{i=0}^{n-1} b_k(i) g_{n-i} \text{ , where} && (1) \\ g_i &= \sum_{\substack{e \mid i \\ k \nmid e}} e. && (2) \end{aligned}$$

However, the divisors of i come in pairs. So, we can reduce an $O(n)$ algorithm to an $O(\sqrt{n})$ algorithm by performing this calculation instead:

$$g_i = \sum_{\substack{e|i \\ 1\le e\le\sqrt{i}}} \tilde{e}$$

$$\tilde{e} = \begin{cases} e+\frac{i}{e} & \text{if } k\nmid e \text{ and } k\nmid \frac{i}{e} \text{ and } \frac{i}{e}\neq e \\ e & \text{if } k\nmid e \text{ only} \\ 0 & \textit{otherwise} \end{cases}$$

4.1 Parallelization

The method used to parallelize the algorithm is simple. If there are N processes, have the jth process store $b_k(i)$ s.t. $i \equiv j \pmod N$ and all g_i. To calculate $b_k(n)$, each process computes

$$P_j(n) = \sum_{\substack{0\le i\le n-1 \\ i\equiv j \mod N}} b_k(i)g_{n-i}, \tag{3}$$

and a group reduction takes place which computes $S := \sum_j P_j(n)$ and gives the result to the process who is $j \pmod N$. That process then computes $\frac{1}{n}S$ and stores it into memory.

4.2 Modular Arithmetic

To avoid gigantic numbers in memory, KregularKompute does all the arithmetic modulo primes. However, this presents a problem: there is a modular inversion of n required if you want to compute $b_k(n)$. So, you must do two things:

1. Pick primes $\geq p$ such that $n < p\ \forall n$,
2. Use Chinese Remainder Theorem to reconstruct the values you wish.

4.3 Chinese Remainder Theorem

Since we have to have a bound for $b_k(n)$ for Chinese Remainder Theorem to work, we need at least a crude bound. The one I chose in the code is the Hardy-Ramanujan asypmtotic formula for $p(n)$:

$$p(n) \sim \frac{1}{4n\sqrt{3}}\exp\left(\pi\sqrt{\frac{2n}{3}}\right). \tag{4}$$

Since $b_k(n) \leq p(n)$, this should be a good bound for any well-sized n.

We must calculate $b_k(n) \pmod{p_i}$ where $\prod_i p_i \geq \frac{1}{4n\sqrt{3}} \exp\left(\pi\sqrt{\frac{2n}{3}}\right)$. In the code, I chose to start picking primes at around 1 billion. Say $\#\{p_i\} = x$, then taking $x = \text{ceil}\left(\frac{\pi}{9\ln(10)}\sqrt{\frac{2n}{3}}\right)$ satisfies the inequalities. So, calculate the first x primes greater than a billion, and you're ready to go!

4.4 Running Time

If n is the amount of terms of b_k you wish to calculate, the original algorithm is $O(n^2)$. The Chinese Remainder Theorem version is $O\left(n^{5/2}\right)$ since x is order $\sqrt{n}$. By doing tests, I saw that the running time decreases in direct proportion to the number of processes if each was run on a physically real and separate CPU.

4.5 General Comments

I was absolutely sure this algorithm would be communication-bound, but on Clemson's cluster, each processor was nearly 100% active with the job. So, when you decide you want to modify the code, take a look at optimizing some loops or something.

5 Notes About the Source Files

5.1 main.c

`FIRSTPRIME` is defined to be a billion. The algorithm starts out with primes just greater than `FIRSTPRIME`, so if you change this definition, you should also change the estimate on how many primes you need. You will obviously need less if `FIRSTPRIME` is larger. This would be the "`d_numprimes...`" line, and you would change the "`9 * log(10)`" to be the natural log of `FIRSTPRIME`.

5.2 chinesert.c

```
void crt( mpz_t x,  unsigned long *a,  unsigned long *m,  unsigned long elts )
```

Computes the solution to `x` $=$ `a[i]` $\pmod{\texttt{m[i]}}$, $0 \leq i \leq$ `elts` using Chinese Remainder Theorem. You need to make sure the `m[i]` are all relatively prime, else this won't work. The algorithm must calculate the product of all elements of the `m[]` array, so it saves computing power by only doing this once. If you supply a different `m[]` array after the first call, you must call `crt_reset()` beforehand. If you want to use this function without first calling `misc.c:init()`, then you should define `CRT_STANDALONE` at compile time.

5.3 dump.c

```
void dump(  char *file )
```

Dumps the contents of a binary file, whose contents are of the type described in Figure 1, to `stdout`.

5.4 gcompute.c

```
void gcompute(  unsigned long *ret,  unsigned long k,  unsigned long high,  unsigned long modulus )
```

Computes the sequence g_i (mod `modulus`), $0 \leq i \leq$ `high`, from the `k`-regular algorithm and stores it in `ret`. Note that `ret` must be allocated and contain enough space for the list of numbers.

5.5 inverse.c

```
 unsigned long inverse(  unsigned long number,  unsigned long modulus )
```

Uses Euler's method to calculate $\texttt{number}^{-1}$ (mod `modulus`). If you want to use this function without first calling `misc.c:init()`, then you should define `INV_STANDALONE` at compile time. This function also has a built-in check to insure the answer is really the inverse (it returns 0 if the check fails). If you want to skip this check to speed up the algorithm, define `INV_NOCHECK` at compile time.

5.6 kregular.c

```
void kregular( unsigned long k,  unsigned long terms,  unsigned long modulus,  char *outfile)
```

Computes $b_k(i)$ (mod `modulus`), $0 \leq i \leq$ `terms`, and writes the result to outfile. If you want to get regular updates via `stdout` about how many terms are left to compute, define `KREG_UPDATES` at compile time, and define `ALARM_INTERVAL` to be the amount of seconds between updates. I think this *could* cause issues since we are interrupting processes in a parallel algorithm. I haven't tried it though. Email me what happens when you try it.

5.7 makefilenames.c

```
void makeFilename(  char *name,  unsigned long k,  unsigned long modulus,  unsigned long high,  char *prefix )
```

Puts a string into the **already allocated** `name` starting with `prefix` and ending with a meaninful filename.

5.8 misc.c

```
int init()
```

Does some set-up stuff. Initializes global variables and stuff. Need to call this early on in the main program.

```
void teardown()
```

Opposite of `init()`.

```
void die()
```

Calls `MPI_Finalize()` and exits with value -1.

5.9 prime.c

```
int isprime( unsigned long n )
```

Returns 0 if `n` is not prime and 1 if it is. This shouldn't be used too heavily because I didn't do anything fancy here.

```
unsigned long nextprime( unsigned long n )
```

Returns the next prime greater than `n`. This shouldn't be used too heavily because I didn't do anything fancy here.

5.10 sum_mod_op.c

```
void sumMod( unsigned long *in, unsigned long *inout, int *len, MPI_Datatype *dptr )
```

An MPI reduction function that adds numbers together and reduces via `_sum_modulus` defined in `algorithms.h`.

```
void sumModOpInit( unsigned long modulus )
```

Sets `_sum_modulus` to `modulus` and creates the MPI operator `SUM_MOD_OP` defined in `algorithms.h`.

5.11 viewer.c

Makes an executable that dumps the contents of a file described by 1 to `stdout`.

5.12 theorem4.pbs

A PBS script I made to run `main` and help me prove cases of Theorem 4 in "The parity of the 5-regular and 13-regular partition functions."