Independent Study Report

Stephen W. THOMAS [1]

Department of Computer Science, University of Arizona, Tucson, AZ 85721

December 12, 2008

Contents

[1] sthomas@cs.arizona.edu

1 Introduction

This paper is intended to document and capture my activities as an independent study (599) graduate student during the Fall 2008 semester. The project advisor was Dr. Gregory Andrews and I worked closely with Joe Roback, a PhD student who was also working with Greg. I will first explain the problem statement and goals we had in mind going into the project. I will then describe Gossamer, the framework I used to create parallel applications out of sequential applications. I will then discuss the process of implementing three applications using Gossamer: knapsack, distributed grep, and the Fast Fourier Transform. I will then finish with some general observations and conclusions.

Throughout this document I will adopt the following terminology. *Application* refers to a program written in the C programming language that solves a specific problem. *Joe* refers to the UA PhD student Joe Roback. *Greg* refers to the UA professor Dr. Gregory Andrews. *FFT* refers to the Fast Fourier Transform application.

1.1 Goals

My background in parallel and distributed applications is like any other computer science student who has taken an undergraduate course on the subject: I know the basic ideas and terminology, have used standard libraries to author a few simple applications, and have a general understanding on the the design and architecture needed for parallel applications to work correctly. This made me a good candidate to learn Gossamer, since I already knew the basics and could concentrate on the goal of the application as opposed to, for example, such details as what a barrier is used for.

The overall goal for the independent study was to implement a set of applications (i) in a traditional, sequential form and (ii) using Gossamer annotations to achieve parallelism. Doing so would provide three general contributions.

1. Further the overall progress of Gossamer by gaining additional speedup measurements and suggesting new annotations and/or modifications to existing annotations.

2. Experience and document the processes and strategies for creating parallel applications from their sequential form using Gossamer annotations.

3. Further my personal education and knowledge of parallel application development, including design, implementation, and debugging.

1.2 Approach

To achieve the aforementioned goals, I worked closely with Joe to learn and use Gossamer to create a number of applications. The specific application set was dictated by the "Berkely view" paper [1] and table 1 includes the list of applications we considered in priority order. Unsure on the amount of time required to implement each application, we set a conservative goal of completing two applications: *knapsack* and *parallel grep*. These are discussed in more detail in the sections below.

Dwarf	**Application**
Dynamic Programming	Knapsack
MapReduce	Parallel grep
Spectral Methods	Fast Fourier Transform
Sparse Linear Algebra	Sparse Matrix Multiply
Unstructured Grid	Adaptive Multigrid
Graphical Models	Bayesian Networks/Hidding Markov
Combinational Logic	Checksum/RSA

Table 1: Prioritized Application List

2 Gossamer

This section briefly describes Gossamer and documents my experience learning to use Gossamer to my advantage.

2.1 Overview

Gossamer is a framework that aims to make it easy to parallelize sequential programs and easy to implement parallel programs while at the same time providing scalable performance improvements on multicore machines [5]. To date, all applications on which Gossamer has been tested get good to excellent speedup.

Gossamer provides users (i.e. application developers) a set of annotations that provide "automatic" means of transforming a conventional, sequential application into a parallel application which takes advantage of multiple processors/cores and reduces run times. Examples of annotations are: `fork method()` and `join` to create different threads to run `method()` and then `join` the results; `parallel for (...)` to turn a traditional `for(...)` loop into one that is executed by multiple threads in parallel; and `atomic{ statement }` to execute `statement` atomically. These intuitive annotations are meant to ease the pain of writing parallel applications; indeed, they significantly reduce the amount of time and effort required.

2.2 Learning and Using Gossamer

As a newcomer to the project, I was first faced with the task of learning Gossamer. This included two distinct subtasks: learning how to install/configure Gossamer (e.g. libraries, header files, linking/building) and learning how to use the annotations in my source code to create parallel applications. As it turns out, once the first task was complete, the second task was almost trivial; the annotations were extremely easy to use and simple to understand. I will expand on this statement throughout this paper.

To use Gossamer, a small set of header files are included into the main source file.

Listing 1: Including the Gossamer header files.

```
#include <gossamer.h>
#include <gossamerP.h>
```

Then, the annotations can be inserted within the source file.

Listing 2: Fibonacci calls without annotations.

```
...
x = fib(n - 1);
y = fib(n - 2);
return (x + y);
...
```

Listing 3: Fibonacci calls with annotations.

```
...
x = fork fib(n - 1);
y = fork fib(n - 2);
join;
return (x + y);
...
```

All of the annotations are very intuitive; learning what they did and how to use them was only a matter of looking at a few examples and trying them out myself. Simply looking at the list of implemented annotations [5] was the only resource I needed to get started; any further questions were just directed to Joe via a short email.

Compiling my applications was also easy, although for different reasons. Here, I obtained the entire Gossamer project (including source code, header files, makefiles, and applications) and simply emulated existing applications. Specifically, I created files in the `apps/gs` directory and edited a line into the `Makefile.gossamer.mk` file to add the names of my new applications.

Listing 4: Line from Gossamer Makefile that lists the applications to build.

```
...
fib knapsack fft mergesort nqueens quicksort simpsons wordcount
..
```

Since I was able to piggyback off of existing applications and Makefile architectures, I in effect bypassed the learning curve required to build applications with Gossamer. However, due to the high level of organziation and usability of all the other components of Gossamer, I am guessing that this step is also easy and intuitive if one had to start from scratch.

The usage of an application that employs Gossamer annotations is unmodified from the original usage; no different command line syntax, arguments, or environment variables are required. However, there are some optional settings available.

- Setting the environment variable `GOSSAMER_THREADS` to an integer number (less than or equal to the number of processors on the machine) causes an application to run on that many threads.
- Adding the flag `--stats` to the command line options when invoking your application produces useful output at the conclusion of execution.

Other environment variables are described as they are introduced in the following sections.

One issue that I did struggle with in Gossamer was debugging my applications. Since Gossamer does a translation step to transform my source code and annotations into a standard C file without saving the intermediate object files, debugging information is lost and thus standard debuggers are more difficult to use. The only way around this was for me to judiciously place `fprintf` statements throughout my code, but this sometimes made time-critical errors difficult or impossible to replicate. Also, given the amount of computation being performed, such print statements generated huge amounts of output that was difficult to parse and understand.

Once I was complete with the implementation and testing of an application, I could then execute the application on the university High Performance Computing machine called `marin`. This SGI Altix 4700 is equipped with 512-core Itanium2 clocked at 1.6 GHz and 1024 GB memory. It also includes a sophisticated batch submission system that allowed me to put my applications in a queue to be processed by a number of CPUs that I specify. More information is available online [7].

3 Applications

As mentioned previously, the set of applications to implement was motivated from the "Berkeley view" paper, which describes 13 "dwarfs." Each dwarf is an algorithmic method that captures a pattern of computation and communication. The idea is that by implementing a *single* application in one of the dwarfs, one is illustrating the effectiveness of that parallel method when applied to *any* application that falls under the same Dwarf. This eliminates the need to implement several similar applications that use the same patterns.

I will note for completion that the following applications were written and initially tested on an Apple machine running Mac OS 10.4.11 with a 2.2 GHz Intel Core 2 Due processor and 2 GB memory. I used Xcode 2.4.1 as my primary IDE along with the Shark profiler tool that comes as a plugin to Xcode. Once my applications were compiling and I was confident that they were producing correct output, I also performed runs on `tempus`, a machine with a single Intel Quad Core processor clocked at 2.83GHz, with a 12MB L2 Cache (6MB shared between each) and 8 GB memory. Joe also performed large run sets on `marin`.

In the following sections I describe the specific applications considered in this independent study.

3.1 Knapsack

The knapsack problem is one of combinatorial optimization and can be simply stated as follows. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible [8]. We considered the Knapsack problem in order to fulfill the Dynamic Programming dwarf.

Upon researching this problem, I quickly learned that it came in many different varieties and flavors; all with a different set of restrictions and assumptions. We concentrated on the so-called 0-1 Knapsack problem. In this form, each item can be taken either 0 or 1 times (as opposed to multiple times or a fractional number of times). It has been stated that this flavor is the most important Knapsack problem since it is a direct subproblem to many more complex problems [3].

I also quickly learned that this problem had several possible solutions, including greedy algorithms and dynamic programming algorithms, each which could be implemented iteratively or recursively. It is interesting to note that the greedy algorithm can be shown to not always produce the correct output. The dynamic programming solution is essentially a brute-force approach that computes a table of values. The following listing outlines the algorithm.

Listing 5: Basic Knapsack algorithm.

```
1  // Assume global cost[] and value[] arrays
2  knapsack(i, j){
3      if (A[i][j] != UNDEFINED){
4          return A[i][j];
5      }
6      if (cost[i] > j){
7          A[i][j] = knapsack(i-1, j);
8      }
9      else{
10         a = knapsack(i-1, j);
11         b = knapsack(i-1, j - cost[i]) + value[i];
12         A[i][j] = MAX(a, b);
13     }
14     return A[i][j];
15 }
```

Here, the 2-dimensional A array can be defined as follows. $A(j, Y)$ is the maximum value that can be attained with total weight less than or equal to Y using items i_0 up to i_j. $A(i, 0)$ and $A(0, j)$ is initially set to 0 for all i, j and the rest of the array is initially set to UNDEFINED. The dynamic programming component comes from the use of the UNDEFINED value: $knapsack(i, j)$ will immediately return $A(i, j)$ if the value is not UNDEFINED, i.e., has already been computed. This allows expensive recompilation of cells to be avoided.

3.1.1 Fork-Fork-Join Approach

My initial thoughts were to simply insert two `fork` annotations at lines 9 and 10 of Listing 5 and a `join` immediately afterwards; this would spawn work for each thread recursively, keep them busy, and the result could be computed in less than time the sequential algorithm. However, this was not immediately the case; speedup was not as good as expected. Figure 1 below lists the numbers achieved using this approach in one example scenario.

#Proc	$N = 16384$	$N = 31072$
1	1.00	1.00
2	0.98	0.96
3	1.24	1.19
4	1.23	1.19

Figure 1: Speedup achieved by adding more processors. Here, $C = 4000$ and $seed = 67$ in all cases. Much less than linear speedup is achieved, and even negative speedup occurred for 2 processors.

After some discussions with Greg and Joe, we decided that the most likely reason for the lack of linear speedup was that, due to the way the Knapsack problem recurses, threads were doing redundant work. To see why this is the case, consider the following simple scenario shown in Figure 2 and suppose we were running two worker threads, thread 1 and thread 2. Thread 1 begins in cell 1 and recursively calls Knapsack to compute cells 2 and 3; cell 2 will be computed by thread 1, and cell 3 by thread 2. In the next step, thread 1 will recursively compute cell 5, and thread 2 will also recursively compute cell 5. Since there is no communication between the two threads, they will both independently compute the value for this cell, duplicating work and thus reducing speedup. Note that this issue does not affect the correctness of the algorithm; it only affects to speedup performance.

3.1.2 AM and Table Approach

To address the above issue of work duplication, we decided to implement into Gossamer an associative memory (AM) mechanism that works similar to a hash function: given a key, a thread can "take out" a value, compute it, and "put it back." Then, at later times, other threads can use this value for their own computation. They key idea here is that when a thread has taken out a value from the AM, other threads will block waiting for it; this prevents them from duplicating the work. AM is fully described in *need cite*.

Once the AM had been implemented into Gossamer, I tested the performance. Here I used the (i, j) indices into the table as the key, and the value was the value of $A(i, j)$. Note that I used a simple pairing function to map the two integers (i, j) uniquely onto another integer that could be used as the key. With this

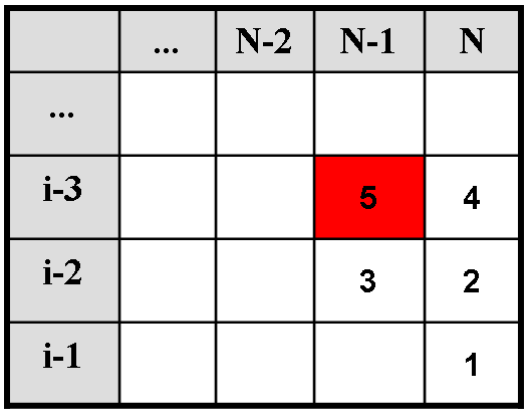

Figure 2: Example scenario that could cause duplicate work to be performed by two threads that both began on cell 1. In this case each cell would independently compute the value of cell 5.

approach, the speedup of some initial runs were again not as high as expected; we guessed that the reason had to do with the AM implementation and the hash function.

From there, we decided to make an optimization to the AM methodology. The idea was as follows. In the context of the Knapsack application, we (as developers and users of Gossamer) have *a priori* knowledge of two important pieces of information: (i) the AM datastructure we are trying to create is will always be 2-dimensional (i.e., a table) and (2) the size of the table will be known at runtime and will not change throughout the execution. The AM paradigm could not take either of these facts into consideration, and thus missed oppertunities for optimization. For example, we didn't actually need to hash (i, j); we knew that each (i, j) pair was already unique and thus could be used directly as an index into the table. This led us to create a new AM-like concept in Gossamer, which we called a *table*. A table is very similar to the AM in that users could "get" and "put" values into and out of the table; the main difference is that no hashing, and thus no chaining, is required.

As an aside, one subtle yet important issue that faced both the AM and the table in Gossamer was that of datatypes. Since the goal was to not restrict users to using specific datatypes for the keys and values in the table, functions needed to be written to support every combination of datatypes that the user might use. There is no easy way to do this in C since there is no template-like mechanisms and types must be explicitly declared at compile time. The brute-force solution would be to implement the functions with every possible combination of datatypes, perhaps by copy-and-paste method in the source code and manually changing the datatypes. However, this is a software maintaince nightmare since if a bug was found in one flavor of the function, it had to be manually propogated to all the other functions. Also, the sheer number of combinations would be too large for practical purposes. The elegant solution that was found involved a clever use of macros to make each of the core functions independent of datatypes; then just one additional line in the code was needed for each possible pair of datatypes. This provided a best-of-all-worlds solution at the cost of slightly increased complexity of the functions, since they are wrapped within a macro syntax.

After I had completed the table implementation, I once again tested my Knapsack application and once again ran into issues, only this time the issues were different and slightly more troubling. For larger input sizes and for large number of processors, I was seeing my application deadlock at random points during the execution. Upon further investigation and several discussions with Joe and Greg, some light was shed on the problem. Briefly, the subtrees of work that knapsack produces are not independent and disjoint from each other. That is, it is possible that both recursive calls in Listing 5 could eventually need a common cell. Further, it is possible for threads to perform work "higher up" in the tree before completing (and thus releasing) the cell it is currently working on. This happens due to the task queuing mechanism currently implemented in Gossamer. These two facts combine for possible deadlocks in execution, since it is possible for thread 1 to be waiting in cell A for (a locked) cell B, and thread 2 to be waiting in cell B for (a locked) cell A. When this occurs, no forward progress is ever made and the execution must be manually killed.

Since there is no general mechanism to detect or avoid this deadlock given the nature of Knapsack and the scheduling mechanisms employed by Gossamer, it was decided that this table approach could not

provide a general solution.

3.1.3 Unrolling Approach

Given the lack of success using AM and tables in Knapsack, we decided to revisit the fork and join approach with an added twist: to "unroll" the recursive calls a number of times in the main function to start off the threads, then make the recursive calls, and then combine the results again in the main function. The idea was that this might ensure that each thread gets an independent unit of work and avoid duplicating work that is high up in the tree, and thus more expensive.

The following Listing shows how to unroll knapsack to ensure 4 threads each initially get an independent unit of work.

```
1
2  main() {
3    fork knapsack1(...);
4    fork knapsack1(...);
5    join;
6  }
7
8  knapsack1() {
9    ...
10   fork knapsack(...);
11   fork knapsack(...);
12   join;
13   ...
14  }
15
16 knapsack() {
17  // No forks and joins; regular sequential recursive algorithm
18 }
```

At the top of the $knapsack$ method I printed the ID of the thread performing the computation. This confirmed that, indeed, each thread's work was initially independent from all other threads. However, this technique still suffered from small performance gains due to work being duplicated further down the tree.

3.1.4 Conclusions

As we have seen, the dynamic programming solution to knapsack is not congruent to parallel algorithms. On the one hand, duplicate work (and thus marginal gains in performance) occurs when each cell is not locked. On the other hand, deadlock can occur when each cell is locked. Thus, a different dynamic programming solution that doesn't exhibit these characteristics should be chosen as the representative algorithm for this dwarf.

The full source code for the locking and non-locking knapsack approaches are listed in Appendix A.

3.2 Parallel Grep

`grep` is a useful and extremely well-known command-line utility which has been bundled with every Unix-like operating system for decades. Briefly, it prints lines in a file that match a given pattern. The idea for this parallel application was to split up the work (i.e., chunks of text to search) between each thread and then combine the results when all threads were finished executing. To achieve this, we used the so-called MapReduce paradigm currently used by Google and Stanford [2, 4]. In the MapReduce model, users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all the intermediate values associated with the same intermediate keys. In our case, the *map* function took a filename to parse and accumulated lines that match the given pattern and then the *reduce* function emitted all accumulated lines.

I initially implemented a sequential grep application without any bells and whistles; it simply looked for a given string in a given set of files, and output all the lines that matched according to a simple C library `strstr()` function call. The next step was to rewrite the application so that it contained a map and reduce function as described above, and then insert Gossamer annotations as required. However, since this was a new programming paradigm being considered by Gossamer, the annotation functionality was not yet complete and thus I was unable to move forward and test my implementation.

The full source code for the (untested and currently not compiling) parallel grep is listed in Appendix B.

3.3 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a special class of implementations for the Discrete Fourier Transform (DFT) problem that has a runtime of $O(n\lg n)$. The DFT is a well known and highly documented problem that has numerous applications in signal and image processing and can be briefly described as follows: given a function in the time domain, transform it into a function in the frequency domain.

My tasks were to research this problem and become familiar enough with it to implement it with Gossamer. Of all the sequential implementations I discovered from various resources, there seemed to be a common mechanism used: divide and conquer. The implementations would recursively divide the original function into two halves and make a recursive call on each half; the base case was executed when only two elements were left in the array.

Listing 6: Sequential recursive FFT implementation.

```
1  void fft(int N, int offset, int delta, double (*x)[2], double (*X)[2], double (*XX)[2])
2  {
3    int N2 = N/2;             /* half the number of points in FFT */
4
5    if(N != 2){  /* Perform recursive step. */
6        /* Calculate two (N/2)-point DFT's. */
7        fft(N2, offset,       2*delta, x, XX, X);
8        fft(N2, offset+delta, 2*delta, x, XX, X);
9
10       /* Combine the two (N/2)-point DFT's into one N-point DFT. */
11       for(k=0; k<N2; k++){
12          // Combination omitted for brevity
13       }
14     }
15   else{  /* Perform 2-point DFT. */
16       // 2-point DFT omitted for brevity
17     }
18 }
```

Since at each step the array was divided cleanly in half, my initial instinct was to simply add a pair of `forks` and a `join` on lines 7, 8, and 9. Placing these annotations indeed provided speedup when the application was executed in parallel, but not the near-linear speedup I was expecting. To investigate this issue, I performed a number of tasks.

The first investigatory task was to play with the prune value (specified by the `GOSSAMER_PRUNE` environment variable) to determine whether or not threads were getting a fair amount of work. On `tempus` I created a script that iteratively set the number of threads to 1, 2, 3, and 4 and for each number of threads, set the prune value to 1, 2, 4, 8, ..., 512. For each combination of threads and prune, I ran the FFT application 30 times on a large, randomly generated input set. By taking the average of these runs and comparing the different prune values, it became clear that the value of the prune parameter was relatively unimportant to the execution time[2] and was not the cause of smaller speedup.

The second task was to run profiling tools to determine where the application was spending its time during execution. The tools indicated that each thread was using an equal amount of time and spending the majority of their CPU cycles performing floating point calculations. Both of these characteristics indicate that near linear speedup should be occuring.

The next investigatory step was to compile the application in Release mode[3]. Doing so immediately increased the speedup results; Joe conjectures that the reason is because Release mode turns off debugging symbols and enables the -O2 compilation flag. It also activates the SSE and MMX units for floating point operations. Figure 3 below shows the speedup achieved.

#Proc	$N = 2^{20}$	$N = 2^{24}$
1	1.00	1.00
2	1.90	1.97
4	3.52	3.80

Figure 3: Speedup achieved in FFT. Experiments were run on `tempus`, described above.

The full source code for FFT is listed in Appendix C.

[2] For all prune values greater than or equal to 8.

[3] In hindsight, this trivial step should have been performed first.

4 Conclusion

During this semester the following was achieved.

1. I learned to use Gossamer within various applications to parallelize their execution.
2. I was a "guinea pig" for the project to provide feedback on various aspects of Gossamer, such as the learning curve, usage, and syntax of the annotations. I also collaborated with Joe using SVN for version control to minimize the pain involved with multiple developers.
3. I researched and implemented three applications that each represented one of the 13 Dwarfs, and critically evaluated each application to assess performance increases and venues for possible improvement. This involved collecting all relevant details about the problem being solved, comparing the different implementation strategies, implementing it in C, finding large input data sets, verifying the correctness of the output, using Gossamer annotations to parallelize, and running test cases to determine performance gains.
4. I provided an initial implementation of the table AM scheme, which is a specific variant of the general AM scheme.

Through all of these activities I gained a deeper understanding of several topics. First, by turning sequential algorithms into parallel algorithms, I gained insight into the overall processes and theory of parallel application design, debugging, and profiling. This will contribute to my success as a student and as a application developer. Second, by being involved in all aspects of Gossamer (a PhD research project) I was heavily exposed to the methods and activities of the PhD research process, which makes me better prepared to enter the next phase of my academic career. Finally, by implementing several applications and modifying a mature and complex framework (Gossamer), I gained technical programming skills that make me an overall more complete programmer.

In addition, my activities were helpful in moving Gossamer forward in progress. The applications I implemented help test and validate Gossamer, and my experiences will hopefully give insight into how the typical user will use Gossamer. In addition, Joe was forced to explain several basic assumptions and ideas to me, which causes old thoughts to be challenged and perhaps improved upon. Finally, my fresh outlook and new ideas hopefully added some useful content to the overall project.

References

[1] K. Asanovic, et al, "The landscape of parallel computing research: a view from Berkeley," Technical Report UCB/EECS-2006-183, December 2006.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.

[3] H. Kellerer et al., "Knapsack Problems," (Springer, 2004).

[4] C. Ranger et al., "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[5] J. A. Roback and G. R. Andrews, "Gossamer: A Lightweight Approach to Using Multicore Machines," 2008

[6] Apple Developer Connection, "Optimizing with Shark: Big Payoff, Small Effort", January 19, 2004. URL `http://developer.apple.com/tools/shark_optimize.html`. Viewed December 6, 2008.

[7] University of Arizona Learning Technology Services, "Research Computing," `http://anise.u.arizona.edu/HPC2/02_hpc/hpc_1.shtml`. Viewed August 2008.

[8] Wikipedia, "Knapsack Problem," `http://en.wikipedia.org/wiki/Knapsack_problem`. Viewed August 2008.

A Knapsack

Listing 7: Fork-version of Knapsack.

```
/*
 * Integer Knapsack using Gossamer (Dynamic Programming).
 *
 * Copyright (c) 2008. All Rights Reserved.
 * Joseph A. Roback <robackja@cs.arizona.edu>
 * Stephen W. Thomas <sthomas@cs.arizona.edu>
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include <gossamer.h>
#include <gossamerP.h>
#include <gs_hash.h>
#include "common.h"

/*************************************************************************
 * The following determines which implemenation is used...
 * Gossamer Table, Sequential (no Gossamer), or Gossamer Hash Table
 *************************************************************************/

//#define GS_TABLE
//#define SEQUENTIAL


#define TYPE_MIN   (1U)
#define TYPE_MAX   (UINT8_MAX-1U)
//#define TYPE_MAX   (10U)
#define TYPE_UNDEF (UINT8_MAX)



/* The arrays that hold the parameters for the knapsack problem */
int *cost  = NULL;
int *value = NULL;


/* Define Ah[i][j] to be the max value that can be attained w/ cost <= j using items up to i. */

/* Annotation: AMTable Ah */
#ifdef GS_TABLE
gs_table_int_t *Ah = NULL;
#elif defined SEQUENTIAL
int **A;
#else
struct gs_hashtable_int_int *Ah = NULL;
#endif

// Global variable that holds the number of columns in the array.
static int gCols;
static int debug;
static int count;

// Debug array- to see how much "rework" was done.
int **Adone;


/* Returns the index into the linear array, given an (i,j) index into a 2-D array. */
// Should this be moved into Gossamer code?
static __inline__ int linear(int a, int b)
{
        return (int) ((a * (gCols+1)) + b);
}

/* Returns a unique number, given "a" and "b". Algorithm comes from
    the pairing function as defined in Homer and Selman's "Computatability and
    Complexity Theory" textbook (page 46, section 3.3)*/
static __inline__ int unique(int a, int b)
{
        return linear(a, b);
//    return (int) ((1.0/2.0)*((a+b)*(a+b+1)) + b);
}


// Allocates and populates the global cost, value, and A arrays.
```

```
void createArrays(int n, int C, int seed)
{
        int i, j;

        // Allocate memory for local tables
        cost  = (int *) malloc(sizeof(int) * n);
    value = (int *) malloc(sizeof(int) * n);

        /* Create hash table */
        /* Gossamer Annotation: AMinit(ah, n*c, int); */
        #ifdef GS_TABLE
        Ah = gsTableMakeTable_int((n * (C+1)));
        #elif defined SEQUENTIAL
        A = (int **)malloc(sizeof(int *) * n);
    for (i = 0; i < n; ++i){
        A[i] = (int *)malloc(sizeof(int) * (C+1));
    }
        #else
        Ah = gs_hash_createtable_int_int((n*(C+1)));
        #endif

        // Debug only- rework array
        Adone = (int **)malloc(sizeof(int *) * n);
    for (i = 0; i < n; ++i){
        Adone[i] = (int *)malloc(sizeof(int) * (C+1));
    }
        for (i = 0; i < n; ++i) {
                for (j = 0; j <= C; ++j) {
                        Adone[i][j] = 0;
                }
        }


        /* Initialize values of cost, value, and Ah. */
        srandom(seed);

        /* Note that A[i][j] = 0 iff (i ==0 or j==0), TYPE_UNDEF otherwise.
                This is from the specification of the algorithm. */
        for (i = 0; i < n; ++i) {
                cost[i]  = UNIFORM_RANDOM(TYPE_MIN, TYPE_MAX);
                value[i] = UNIFORM_RANDOM(TYPE_MIN, TYPE_MAX);

                if (debug){
            printf("%03d, %03d, %03d\n", i, cost[i], value[i]);
        }

                // Initialize Ah
                for (j = 0; j <= C; ++j) {

                        #ifdef GS_TABLE
                        int x = linear(i, j);
                        #elif defined SEQUENTIAL

                        #else
                        int x = unique(i, j);
                        #endif

                        if (i == 0 || j == 0) {
                                /* Annotation: AMwrite(Ah, i, j, val); */
                                #ifdef GS_TABLE
                                gsTableOut_int(Ah, x, 0);
                                #elif defined SEQUENTIAL
                                A[i][j] = 0;
                                #else
                                gs_hash_put_int_int(Ah, x, 0);
                                #endif

                        }
                        else {
                                /* Annotation: AMwrite(Ah, i, j, val); */
                                #ifdef GS_TABLE
                                gsTableOut_int(Ah, x, TYPE_UNDEF);
                                #elif defined SEQUENTIAL
                                A[i][j] = TYPE_UNDEF;
                                #else
                                gs_hash_put_int_int(Ah, x, TYPE_UNDEF);
                                #endif
                        }
                }
        }
}


/* De-allocates memory from global arrays. */
```

```
163 void destroyArrays(int n)
164 {
165     free(cost);
166     free(value);
167 
168         // Annotation: AMfree(Ah);
169         //TODO: fix this call
170         //gsTableFreeTable_int(Ah);
171 }
172 
173 
174 /* Returns the optimal value possible using items from 0...i with a max cost of j */
175 int knapsack(int i, int j)
176 {
177         int a, b, val = 0;
178         int idx;
179 
180 
181 
182         #ifdef GS_TABLE
183         idx = linear(i, j);
184         #elif defined SEQUENTIAL
185         idx = 0;
186         #else
187         idx = unique(i, j);
188         #endif
189 
190         // Annotation: in(Ah, key, &val)
191         #ifdef GS_TABLE
192         //gsTableRead_int(Ah, idx, &val);
193         #elif defined SEQUENTIAL
194         val = A[i][j];
195         #else
196         gs_hash_read_int_int(Ah, idx, &val);
197         #endif
198 
199 
200         // Dynamic programming - check to see if we've already calculated this value.
201         /*if (val != TYPE_UNDEF) {
202                 return val;
203         }*/
204 
205 
206 
207         // Annotation: in(Ah, key, &val) */
208         #ifdef GS_TABLE
209         printf("%lu: getting     %3d %3d\n", TID, i, j);
210         fflush(stdout);
211         gsTableIn_int(Ah, idx, &val);
212         #elif defined SEQUENTIAL
213 
214         #else
215         printf("%lu: getting     %3d %3d\n", TID, i, j);
216         fflush(stdout);
217         gs_hash_get_int_int(Ah, idx, &val);
218         #endif
219 
220         // Dynamic programming - check to see if we've already calculated this value. */
221         if (val != TYPE_UNDEF) {
222                 #ifdef GS_TABLE
223             printf("%lu: putting old %3d %3d\n", TID, i, j);
224             fflush(stdout);
225                 gsTableOut_int(Ah, idx, val);
226                 #elif defined SEQUENTIAL
227                 #else
228                 printf("%lu: putting old %3d %3d\n", TID, i, j);
229             fflush(stdout);
230                 gs_hash_put_int_int(Ah, idx, val);
231                 #endif
232                 return val;
233         }
234 
235         Adone[i][j]++;
236 
237         // If we're this far, then a recursive step is needed.
238         // (Algorithm taken from Wikipedia)
239         if (cost[i] > j) {
240                 val = knapsack(i-1, j);
241         }
242         else {
243                 #ifdef SEQUENTIAL
244                 a       = knapsack(i-1, j);
245                 b       = knapsack(i-1, j - cost[i]);
246                 #else
```

```
247                 a       = fork knapsack(i-1, j);
248                 b       = fork knapsack(i-1, j - cost[i]);
249     printf("%lu: rtj          %3d %3d\n", TID, i, j);
250         fflush(stdout);
251                 join;
252                 #endif
253
254                 b       = b + value[i];
255                 val = MAX(a, b);
256         }
257
258         // Annotation: AMwrite(Ah, key, val)
259         #ifdef GS_TABLE
260         printf("%lu: putting new %3d %3d\n", TID, i, j);
261         fflush(stdout);
262         gsTableOut_int(Ah, idx, val);
263         #elif defined SEQUENTIAL
264         #else
265         printf("%lu: putting new %3d %3d\n", TID, i, j);
266         fflush(stdout);
267         gs_hash_put_int_int(Ah, idx, val);
268         #endif
269
270         return val;
271 }
272
273 int main(int argc, char **argv)
274 {
275         int n, C, seed;
276         int result;
277         gstimer_t ftime;
278         char *impType;
279         int i, j;
280
281         debug = 0;
282         count = 0;
283
284     /* Check arguments */
285         if (argc < 4) {
286                 fprintf(stderr, "Usage: knapsack <n> <C> <seed> [debug_output]\n");
287                 exit(EXIT_FAILURE);
288         }
289
290         /* Read number of items, Cost constraint, and seed */
291         n               = strtoul(argv[1], NULL, 10);
292     C           = strtoul(argv[2], NULL, 10);
293         seed    = strtoul(argv[3], NULL, 10);
294         if (argc > 4){
295                 debug = strtoul(argv[4], NULL, 10);
296         }
297
298         gCols = C;
299         createArrays(n, C, seed);
300
301         /* Compute Knapsack */
302         gsStartWatch(&ftime, true);
303         result = knapsack(n-1, C);
304         gsStopWatch(&ftime);
305
306         // Debug only - how much work was redone?
307         for (i = 0; i < n; ++i) {
308                 for (j = 0; j <= C; ++j) {
309                         printf("Adone[%d][%d] = %d.\n", i, j, Adone[i][j]);
310                 }
311         }
312
313         #ifdef GS_TABLE
314         impType = "table";
315         #elif defined SEQUENTIAL
316         impType = "seque";
317         #else
318         impType = "hash";
319         #endif
320
321         /* Print result to stdout */
322         /*printf("Knapsack(%" PRIu64 ", %" PRIu64 ") = %" PRId32 ", algorithm exec time: %.8f seconds\n",
323                     n, C, result, gsWatchValue(&ftime));*/
324         printf("Knapsack_%s(%d, %d, %d) = %d algorithm exec time: %.8f seconds\n", impType,
325                     n, C, seed, result, gsWatchValue(&ftime));
326
327         destroyArrays(n);
328
329         return 0;
330 }
```

Listing 8: Table-version of Knapsack.

```
/*
 * Integer Knapsack using Gossamer (Dynamic Programming).
 *
 * Copyright (c) 2008. All Rights Reserved.
 * Joseph A. Roback <robackja@cs.arizona.edu>
 * Stephen W. Thomas <sthomas@cs.arizona.edu>
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include <gossamer.h>
#include <gossamerP.h>
#include <gs_hash.h>
#include "common.h"

/*************************************************************************
 * The following determines which implemenation is used...
 * Gossamer Table, Sequential (no Gossamer), or Gossamer Hash Table
 *************************************************************************/

//#define GS_TABLE
//#define SEQUENTIAL


#define TYPE_MIN   (1U)
#define TYPE_MAX   (UINT8_MAX-1U)
//#define TYPE_MAX   (10U)
#define TYPE_UNDEF (UINT8_MAX)



/* The arrays that hold the parameters for the knapsack problem */
int *cost  = NULL;
int *value = NULL;


/* Define Ah[i][j] to be the max value that can be attained w/ cost <= j using items up to i. */

/* Annotation: AMTable Ah */
#ifdef GS_TABLE
gs_table_int_t *Ah = NULL;
#elif defined SEQUENTIAL
int **A;
#else
struct gs_hashtable_int_int *Ah = NULL;
#endif

// Global variable that holds the number of columns in the array.
static int gCols;
static int debug;
static int count;

// Debug array- to see how much "rework" was done.
int **Adone;


/* Returns the index into the linear array, given an (i,j) index into a 2-D array. */
// Should this be moved into Gossamer code?
static __inline__ int linear(int a, int b)
{
        return (int) ((a * (gCols+1)) + b);
}

/* Returns a unique number, given "a" and "b". Algorithm comes from
    the pairing function as defined in Homer and Selman's "Computatability and
    Complexity Theory" textbook (page 46, section 3.3)*/
static __inline__ int unique(int a, int b)
{
        return linear(a, b);
//    return (int) ((1.0/2.0)*((a+b)*(a+b+1)) + b);
}


// Allocates and populates the global cost, value, and A arrays.
void createArrays(int n, int C, int seed)
```

```
{
        int i, j;

        // Allocate memory for local tables
        cost  = (int *) malloc(sizeof(int) * n);
    value = (int *) malloc(sizeof(int) * n);

        /* Create hash table */
        /* Gossamer Annotation: AMinit(ah, n*c, int); */
        #ifdef GS_TABLE
        Ah = gsTableMakeTable_int((n * (C+1)));
        #elif defined SEQUENTIAL
        A = (int **)malloc(sizeof(int *) * n);
    for (i = 0; i < n; ++i){
        A[i] = (int *)malloc(sizeof(int) * (C+1));
    }
        #else
        Ah = gs_hash_createtable_int_int((n*(C+1)));
        #endif

        // Debug only- rework array
        Adone = (int **)malloc(sizeof(int *) * n);
    for (i = 0; i < n; ++i){
        Adone[i] = (int *)malloc(sizeof(int) * (C+1));
    }
        for (i = 0; i < n; ++i) {
                for (j = 0; j <= C; ++j) {
                        Adone[i][j] = 0;
                }
        }


        /* Initialize values of cost, value, and Ah. */
        srandom(seed);

        /* Note that A[i][j] = 0 iff (i ==0 or j==0), TYPE_UNDEF otherwise.
                This is from the specification of the algorithm. */
        for (i = 0; i < n; ++i) {
                cost[i]  = UNIFORM_RANDOM(TYPE_MIN, TYPE_MAX);
                value[i] = UNIFORM_RANDOM(TYPE_MIN, TYPE_MAX);

                if (debug){
            printf("%03d, %03d, %03d\n", i, cost[i], value[i]);
        }

                // Initialize Ah
                for (j = 0; j <= C; ++j) {

                        #ifdef GS_TABLE
                        int x = linear(i, j);
                        #elif defined SEQUENTIAL

                        #else
                        int x = unique(i, j);
                        #endif

                        if (i == 0 || j == 0) {
                                /* Annotation: AMwrite(Ah, i, j, val); */
                                #ifdef GS_TABLE
                                gsTableOut_int(Ah, x, 0);
                                #elif defined SEQUENTIAL
                                A[i][j] = 0;
                                #else
                                gs_hash_put_int_int(Ah, x, 0);
                                #endif

                        }
                        else {
                                /* Annotation: AMwrite(Ah, i, j, val); */
                                #ifdef GS_TABLE
                                gsTableOut_int(Ah, x, TYPE_UNDEF);
                                #elif defined SEQUENTIAL
                                A[i][j] = TYPE_UNDEF;
                                #else
                                gs_hash_put_int_int(Ah, x, TYPE_UNDEF);
                                #endif
                        }
                }
        }
}


/* De-allocates memory from global arrays. */
void destroyArrays(int n)
```

```
{
    free(cost);
    free(value);

        // Annotation: AMfree(Ah);
        //TODO: fix this call
        //gsTableFreeTable_int(Ah);
}


/* Returns the optimal value possible using items from 0...i with a max cost of j */
int knapsack(int i, int j)
{
        int a, b, val = 0;
        int idx;



        #ifdef GS_TABLE
        idx = linear(i, j);
        #elif defined SEQUENTIAL
        idx = 0;
        #else
        idx = unique(i, j);
        #endif

        // Annotation: in(Ah, key, &val)
        #ifdef GS_TABLE
        //gsTableRead_int(Ah, idx, &val);
        #elif defined SEQUENTIAL
        val = A[i][j];
        #else
        gs_hash_read_int_int(Ah, idx, &val);
        #endif


        // Dynamic programming - check to see if we've already calculated this value.
        /*if (val != TYPE_UNDEF) {
                return val;
        }*/



        // Annotation: in(Ah, key, &val) */
        #ifdef GS_TABLE
        printf("%lu: getting     %3d %3d\n", TID, i, j);
        fflush(stdout);
        gsTableIn_int(Ah, idx, &val);
        #elif defined SEQUENTIAL

        #else
        printf("%lu: getting     %3d %3d\n", TID, i, j);
        fflush(stdout);
        gs_hash_get_int_int(Ah, idx, &val);
        #endif

        // Dynamic programming - check to see if we've already calculated this value. */
        if (val != TYPE_UNDEF) {
                #ifdef GS_TABLE
            printf("%lu: putting old %3d %3d\n", TID, i, j);
            fflush(stdout);
                gsTableOut_int(Ah, idx, val);
                #elif defined SEQUENTIAL
                #else
                printf("%lu: putting old %3d %3d\n", TID, i, j);
            fflush(stdout);
                gs_hash_put_int_int(Ah, idx, val);
                #endif
                return val;
        }

        Adone[i][j]++;

        // If we're this far, then a recursive step is needed.
        // (Algorithm taken from Wikipedia)
        if (cost[i] > j) {
                val = knapsack(i-1, j);
        }
        else {
                #ifdef SEQUENTIAL
                a       = knapsack(i-1, j);
                b       = knapsack(i-1, j - cost[i]);
                #else
                a       = fork knapsack(i-1, j);
```

```
248                 b       = fork knapsack(i-1, j - cost[i]);
249     printf("%lu: rtj         %3d %3d\n", TID, i, j);
250         fflush(stdout);
251                 join;
252                 #endif
253
254                 b       = b + value[i];
255                 val = MAX(a, b);
256         }
257
258         // Annotation: AMwrite(Ah, key, val)
259         #ifdef GS_TABLE
260         printf("%lu: putting new %3d %3d\n", TID, i, j);
261         fflush(stdout);
262         gsTableOut_int(Ah, idx, val);
263         #elif defined SEQUENTIAL
264         #else
265         printf("%lu: putting new %3d %3d\n", TID, i, j);
266         fflush(stdout);
267         gs_hash_put_int_int(Ah, idx, val);
268         #endif
269
270         return val;
271 }
272
273 int main(int argc, char **argv)
274 {
275         int n, C, seed;
276         int result;
277         gstimer_t ftime;
278         char *impType;
279         int i, j;
280
281         debug = 0;
282         count = 0;
283
284     /* Check arguments */
285         if (argc < 4) {
286                 fprintf(stderr, "Usage: knapsack <n> <C> <seed> [debug_output]\n");
287                 exit(EXIT_FAILURE);
288         }
289
290         /* Read number of items, Cost constraint, and seed */
291         n               = strtoul(argv[1], NULL, 10);
292     C           = strtoul(argv[2], NULL, 10);
293         seed    = strtoul(argv[3], NULL, 10);
294         if (argc > 4){
295                 debug = strtoul(argv[4], NULL, 10);
296         }
297
298         gCols = C;
299         createArrays(n, C, seed);
300
301         /* Compute Knapsack */
302         gsStartWatch(&ftime, true);
303         result = knapsack(n-1, C);
304         gsStopWatch(&ftime);
305
306         // Debug only - how much work was redone?
307         for (i = 0; i < n; ++i) {
308                 for (j = 0; j <= C; ++j) {
309                         printf("Adone[%d][%d] = %d.\n", i, j, Adone[i][j]);
310                 }
311         }
312
313         #ifdef GS_TABLE
314         impType = "table";
315         #elif defined SEQUENTIAL
316         impType = "seque";
317         #else
318         impType = "hash";
319         #endif
320
321         /* Print result to stdout */
322         /*printf("Knapsack(%" PRIu64 ", %" PRIu64 ") = %" PRId32 ", algorithm exec time: %.8f seconds\n",
323                     n, C, result, gsWatchValue(&ftime));*/
324         printf("Knapsack_%s(%d, %d, %d) = %d algorithm exec time: %.8f seconds\n", impType,
325                     n, C, seed, result, gsWatchValue(&ftime));
326
327         destroyArrays(n);
328
329         return 0;
330 }
```

B Parallel Grep

Listing 9: Untested Parallel Grep.

```
/*
 * Simple string matcher (grep-lite) using Gossamer.
 * No command-line options.
 *
 * Copyright (c) 2008. All Rights Reserved.
 * Joseph A. Roback <robackja@cs.arizona.edu>
 * Stephen W. Thomas <sthomas@cs.arizona.edu>
 *
 *
 * TODO
 * - Resolve type issues for key and values (have int, int; need char *, char *).
 * - Test.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include <gossamer.h>
#include <gossamerP.h>

#include "common.h"

#define MAX_LINE_LENGTH 8192

typedef uint32_t result_t;


// TODO: Annotation: AM Ah;
gs_hashtable_int_t *amlines = NULL;

/* The pattern to match. Make it global in case we want to use RegEx library in future */
char *gPattern;



/* The Map function.
 * For each line in file that matches gPattern, emit the line.
 */
void map(char *file)
{
char curLine[MAX_LINE_LENGTH], *result;
        FILE *f;

        // Open the file, check for error
        if ( (f = fopen(file, "r")) == NULL){
                fprintf(stderr, "Warning: error with file %s.\n", file);
                exit(EXIT_FAILURE);
        }

        // For each line, run strstr() and emit results
        while(fgets(curLine, MAX_LINE_LENGTH, f) != NULL) {
                result = strstr(curLine, gPattern );
                if (result){
                        // TODO: Annotation: AMput(amlines, file, curLine);
                        // TODO: Key and value type?
                        gsHashOut_int(amlines, file, curLine);
                }
        }

        fclose(f);
}


/*  The Reduce function. For each file, emit all lines accumulated. */
void reduce(char *file)
{
        char *line;
        // TODO: Annotation: while (AMgetNB(amlines, file, line)) {
        // TODO: Key and value types?
        while (gsHashReadNB_int(amlines, file, line)){
                        printf("%s: %s\n", file, line);
        }
        // Nothing else to do here.
}

```

```
79
80
81 int main(int argc, char **argv)
82 {
83         gstimer_t ftime;
84         int i, numfiles = argc-2, numwords = -1;
85         char **files = NULL;
86
87     // Check arguments
88         if (argc < 3) {
89                 fprintf(stderr, "Usage: ggrep <pattern> <file>... \n");
90                 exit(EXIT_FAILURE);
91         }
92
93         // Read in pattern
94         gPattern = (char *)malloc(sizeof(char) * (strlen(argv[1]) + 1));
95         gPattern = strcpy(gPattern, argv[1]);
96
97         // SETUP: Initialize AM for <char *> key and <char *> values
98         // TODO: Annotation: amlines = AMinit(char *, char *);
99         // TODO: What will this look like?
100        amlines = gsHashMakeTable_int((numfiles));
101
102        // Start timer
103        gsStartWatch(&ftime, true);
104
105        // MAP: map() over each file from cmd line input
106        for (i = 2; i < numfiles; i++) {
107                fork map(argv[i]);
108        }
109        join;
110
111        // REDUCE: reduce() over key set from associate memory
112        // TODO: Annotation: numwords = AMkeyset(amlines, &files);
113        // TODO: What is this call going to actually look like now?
114
115        for (i = 0; i < numwords; i++) {
116                fork reduce(files[i]);
117        }
118        join;
119
120        // Stop timer and output results.
121        gsStopWatch(&ftime);
122        printf("Grep %s, algorithm exec time: %.8f seconds\n",
123                    gPattern, gsWatchValue(&ftime));
124
125        return 0;
126 }
```

C Fast Fourier Transform

Listing 10: Fast Fourier Transform.

```
/*
 * Recursive Fast Fourier Transform using Gossamer.
 *
 * Copyright (c) 2008. All Rights Reserved.
 * Joseph A. Roback <robackja@cs.arizona.edu>
 * Stephen W. Thomas <sthomas@cs.arizona.edu>
 *
 *
 * Original sequential implementation motivated by the works in:
 * http://courseware.ee.calpoly.edu/~jbreiten/C/
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include <gossamer.h>
#include <gossamerP.h>
#include <gs_hash.h>
#include "common.h"


#define TWO_PI (6.28318530717958647692528676655900057683943L)

#define PI (3.14159265)
#define PISQ (9.86960440109)

double sine(double x)
{
    const double B = 4/PI;
    const double C = -4/(PISQ);

    double y = B * x + C * x * abs(x);
    return y;
}


/* Prints out an Nx2 array to stdout */
void printArray(double (*x)[2], long int N){
        int i;
        for (i=0; i<N; ++i){
                printf("%12.4f %12.4f\n", x[i][0], x[i][1]);
        }
}



/* FFT recursion */
void fft_rec(int N, int offset, int delta,
             double (*x)[2], double (*X)[2], double (*XX)[2])
{
  int N2 = N/2;            /* half the number of points in FFT */
  int k;                   /* generic index */
  double cs, sn;           /* cosine and sine */
  int k00, k01, k10, k11;  /* indices for butterflies */
  double tmp0, tmp1;       /* temporary storage */

  if(N != 2){  /* Perform recursive step. */
      /* Calculate two (N/2)-point DFT's. */
      fork fft_rec(N2, offset,       2*delta, x, XX, X);
      fork fft_rec(N2, offset+delta, 2*delta, x, XX, X);
          join;

      /* Combine the two (N/2)-point DFT's into one N-point DFT. */
      for(k=0; k<N2; k++){
          k00 = offset + k*delta;    k01 = k00 + N2*delta;
          k10 = offset + 2*k*delta;  k11 = k10 + delta;
          cs = cos(TWO_PI*k/(double)N);
          //sn = sin(TWO_PI*k/(double)N);
          sn = sine(TWO_PI*k/(double)N);
          tmp0 = cs * XX[k11][0] + sn * XX[k11][1];
          tmp1 = cs * XX[k11][1] - sn * XX[k11][0];
          X[k01][0] = XX[k10][0] - tmp0;
          X[k01][1] = XX[k10][1] - tmp1;
          X[k00][0] = XX[k10][0] + tmp0;
          X[k00][1] = XX[k10][1] + tmp1;
```

```
79            }
80        }
81      else{  /* Perform 2-point DFT. */
82            k00 = offset; k01 = k00 + delta;
83            X[k01][0] = x[k00][0] - x[k01][0];
84            X[k01][1] = x[k00][1] - x[k01][1];
85            X[k00][0] = x[k00][0] + x[k01][0];
86            X[k00][1] = x[k00][1] + x[k01][1];
87        }
88    }
89
90
91    /* FFT */
92    void fft(int N, double (*x)[2], double (*X)[2]){
93      /* Declare a pointer to scratch space. */
94      double (*XX)[2] = malloc(2 * N * sizeof(double));
95
96      /* Calculate FFT by a recursion. */
97      fft_rec(N, 0, 1, x, X, XX);
98
99      /* Free memory. */
100     free(XX);
101   }
102
103
104
105   int main(int argc, char **argv)
106   {
107           gstimer_t ftime;
108           char *inFileName, *outFileName;
109           long int N;
110           int i;
111           FILE *f;
112           double (*x)[2];
113       double (*y)[2];
114
115       /* Check arguments */
116           if (argc < 4) {
117                   fprintf(stderr, "Usage: fft <N> <input_file> <output_file> \n");
118                   exit(EXIT_FAILURE);
119           }
120
121           N = strtoul(argv[1], NULL, 10);
122           inFileName  = argv[2];
123           outFileName = argv[3];
124
125           /* Allocate arrays */
126           x = malloc(2 * N * sizeof(double));
127           y = malloc(2 * N * sizeof(double));
128
129           /* Read file */
130           f = fopen(inFileName, "r");
131           if (f){
132                   for (i=0; i<N; ++i){
133                           fscanf(f, "%lg%lg", &x[i][0], &x[i][1]);
134                   }
135           } else {
136                   perror(inFileName);
137                   exit(EXIT_FAILURE);
138           }
139           fclose(f);
140
141           /* Compute FFT */
142           gsStartWatch(&ftime, true);
143           fft(N, x, y);
144           gsStopWatch(&ftime);
145
146           /* Output file */
147           f = fopen(outFileName, "w");
148           for (i=0; i<N; ++i){
149                   fprintf(f, "%12.4f %12.4f\n", y[i][0], y[i][1]);
150           }
151           fclose(f);
152
153           printf("FFT(%s) algorithm exec time: %.8f seconds\n", inFileName, gsWatchValue(&ftime));
154
155           return 0;
156   }
```