Gossamer 1.0.0 (build 8513) Reference Manual

SOLAR Group
Computer Science, University of Arizona

http://www.cs.arizona.edu/solar

13 May 2008

Contents

Chapter 1

Introduction

Concurrent programming has long been considered much harder than sequential programming. This is because a programmer has to identify and create independent tasks and then to synchronize their execution in order to avoid interference in the use of shared variables (e.g., race conditions) and to generate results in the correct order. Moreover, to get good performance, the programmer also has to worry about mapping tasks to processors, balancing computational load, and organizing data structures and data reference patterns to avoid cache conflicts.

The most common way to write a concurrent program is to use a sequential language and a library such as Pthreads for shared-memory architectures or MPI for distributed-memory architectures. This approach gives the programmer detailed control but at the price of programming every aspect of concurrency and synchronization using low-level tools. Thus, there is a high barrier to entry (lots to learn) and the resulting programs are hard to debug and modify.

At the other extreme, one could use a high-level model such as OpenMP[1], ZPL , MapReduce , Multilisp , or Cilk . These give the programmer an abstract view of concurrencydata parallelism in the case of OpenMP, ZPL, functional (recursive) parallelism in the case of MapReduce, Multilisp, and Cilkand leave the low-level details of managing processes and implementing synchronization to a compiler and run-time system. This lowers the barrier to entry and facilitates experimentation, but each of the above models supports only a single type of concurrency and hence does not provide a general solution.

In between these two approaches, one can write concurrent programs in languages such as C# and Java that contain explicit mechanisms for forking threads and synchronizing their execution. The resulting programs are easier to understand and modify than those written using libraries, and they are more portable. However, the programmer has to manage most aspects of concurrency and synchronization.

The goal of this project is to ease the transition from sequential to concurrent programming by making it easy to write parallel programs for multicore machines while at the same time enabling one to produce efficient programs. Our approach has three components:

1. A set of high-level annotations that one adds to a sequential program (C in our case) in order to specify concurrency and synchronization. The annotations can be used to experiment with ways to parallelize a sequential program, or they can be used directly to implement a parallel algorithm. A key attribute of the annotations is that if they are elided (removed) from the program, the result is simply the underlying sequential program.

2. A threading framework that provides fine-grained threads and high-level synchronization constructs (e.g., barriers, atomic actions, reductions) and that implements these efficiently on top of an underlying threads package (Pthreads in our case).

3. A source-to-source translator that takes an annotated sequential program and produces a program that uses our threading framework. The resulting program is (transparently) compiled, linked, and executed as if it had been written directly. The translator will provide feedback to warn the programmer about suspect annotations (when possible), such as loop-carried dependencies in parallel loops and unprotected shared variables, as well as suggestions to speed the learning process of new users.

The net result if the project is successful will be a software system that makes it easy to experiment with ways to parallelize existing sequential programs as well as to write efficient concurrent programs.

Three pieces of prior work served as the inspiration for this project. First, a large group of faculty at the University of California at Berkeley recently wrote a seminal paper that summarizes the challenges resulting from the architectural trend to double the number of cores on a chip with each new microprocessor generation. That paper summarizes 13 so-called dwarfsclasses of applications that have common computing and communication patterns (see the table on page 4). The key point is that a general programming model has to be able to accommodate all of these patterns, and the challenge is to do so both simply and efficiently. Second, the Cilk project at MIT showed that it is possible to support recursive (fork/join) parallelism easily by adding five simple keywords as annotations to sequential C programs; their run-time system is then able to implement the resulting parallelism efficiently on shared-memory machines. Third, the Filaments project of a decade ago at the University of Arizona showed that it is possible efficiently to support a fine-grain programming model for iterative (data parallel) and recursive parallelism on both shared- and distributed-memory machines (the latter by means of a customized implementation of distributed shared memory).

This project, called Gossamer, aims to support programming applications that use any of the 13 Berkeley dwarfs, solely or in combination, by means of simple annotations to C programs that are similar in spirit to those introduced by Cilk but that go far beyond what Cilk supports. We are using the Filaments implementation for shared-memory machines as the starting point for the Gossamer run-time library. The translator for programs written in C plus Gossamer annotations will be similar in spirit to the translator that is part of Cilk, but again it will go far beyond what Cilk provides.

Below we summarize work to date on annotations, the Gossamer threads package, and the translator.

1.1 Program Annotations

Our annotations allow one to specify concurrency and synchronization in a high-level, intuitive way. As a first example, following is the core of a parallel program for multiplying two n by n matrices:

```
parallel for (i = 0; i<n; i++) {
  parallel for (j = 0; j<n; j++) {
    for (k = 0; k<n; k++)
```

```
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
  }
}
```

The annotation parallel indicates that all iterations of the for loop may be executed in parallel. This program thus specifies n^2 lightweight tasks, one for each combination of the values of i and j. (Note that if the two instances of parallel are deleted, what remains is the familiar sequential program.) If the programmer were to be overzealous above and also to have used parallel on the innermost loop, the Gossamer translator will issue a warning about a loop-carried dependency.

A slightly more complex example is illustrated by the following, which computes the number of pixels in an n by n image that are lit (i.e., non-zero):

```
int sum = 0;
parallel for (i = 0; i<n; i++) {
  parallel for (j = 0; j<n; j++) {
    if (image[i][j])
      reduce {sum++;}
  }
}
```

Again the program specifies n^2 lightweight tasks. In addition, the reduce annotation specifies that the code inside the braces accumulates a value in a shared variable and is to be executed atomically. (Again, if the annotations are removed, what is left is the sequential algorithm.) Gossamer implements this reduction operation by having each processor use a private value of sum to accumulate results from the tasks executed by that processor, and then atomically updating the shared value sum after all tasks finish. Gossamer also provides an annotation atomic to specify arbitrary critical sections (the translator will implement these as reductions when it can do so).

Gossamer also supports parallel execution of while loops, as in the following program fragment, which computes a run-length encoding of the values in an array. (A run-length encoding represents an array by a sequence of ordered pairs (value, count), where each ordered pair indicates that value occurs count consecutive times.)

```
parallel[NCPU](divide[data],private[size,val,run,results])
while (size>0) {
  val = *data++;
  size--;
  run = 1;
  // find run length of current value
  while (val -- *data && size>0) {
    run++; data++; size--;
  }
  copy (val,run)into results;
}
wait-for-predecessor;
if (vals same) combine last pair of predecessor with my first pair;
write out results;
end-parallel
```

This example is more complex because there is no natural degree of parallelism in the while loop, results have to be written out in order, and the extent of the parallel code extends past the end of the while loop. The annotation on the first line says to use NCPU instances of the while loop, divide the data array among the processors, and give each task a private copy of variables size, val, run, and results. The divide and private annotations are optional parameters of the parallel annotation; they are needed here to indicate how the variables used in the body of the loop are to be treated. (The default is that they are shared, which will obviously not work.) The annotation wait-for-predecessor specifies that the current task is to delay at that point until its predecessor has terminated. The annotation on the last line specifies the end of the code that is to execute in parallel.

Gossamer also supports recursive and task parallelism by means of fork and join. For example, the body of a recursive quicksort algorithm could be parallelized as follows:

```
if (array section small) { sort it and return; }
partition array into left and right halves;
fork quicksort(left half);
fork quicksort(right half);
join;
```

Here fork is prepended to each recursive call to indicate that it should be executed concurrently. The join waits for both calls to complete. (An alternative form, `join[c]`, waits for c children to complete.)

Gossamer also provides a barrier annotation to indicate a barrier synchronization point in a parallel iterative algorithm, and an abort annotation to terminate execution of children of a parent task in a parallel recursive computation such as branch and bound or tree search. A few additional annotations may be required as we program examples of all 13 of the dwarfs identified by the Berkeley group.

The table below lists the 13 dwarfs, the applications of each that we have programmed to date or are working on, and the annotations that are used in applications that we have programmed.

Dwarf	**Applications**	**Annotations**
Dense linear algebra	Matrix multiply, jzip	parallel, fork, wait-for-predecessor
Sparse linear algebra	Sparse matrix multiply	
Spectral methods	Fast Fourier transform	
N-body	Barnes-Hut parallel	
Structured grids	Multigrid, Jacobi	parallel, barrier
Unstructured grids	Adaptive multigrid	
Map/Reduce (Monte Carlo)	Parallel grep, Monte Carlo pi	
Combinational logic	Checksum, RSA encryption	
Graph Traversal	Quicksort, jzip	fork, join
Dynamic programming	Shortest path, knapsack	fork, join
Back-track and branch/bound	N-queens, TSP, knapsack, nearest neighbor	fork, join
Graphical methods	Bayesian networks, hidden Markov	
Finite state machine	Run-length encoding	parallel, wait-for-predecessor

1.2 Gossamer Threads Package

The Gossamer library provides a high-level interface that supports implementation of the programming annotations. It is implemented using the Pthreads library. In particular, Gossamer uses Pthreads to create NCPU server threads, one per core on the underlying hardware platform. Each server thread has an execution stack and a task list. The tasks result from the Gossamer annotations; each task is represented by a function pointer and the arguments to that function.

For a parallel for loop with one iteration variable, the function consists of the loop body and the argument is an instance of the iteration variable. For a parallel while loop, the function consists of an instance of the loop that is synthesized from the original loop based on the specifications in the divide and private annotations (if any) that are arguments to the parallel annotation. The number of tasks resulting from a parallel annotation is fixed; Gossamer assigns these tasks evenly to server threads so as to balance computational load.

The other way in which the programmer can specify tasks is by means of the fork annotation. This annotation directly specifies the function to execute and its arguments. Forked tasks are assigned to server threads in a round-robin fashion. The Gossamer implementation uses an adjustable pruning threshold to switch to sequential (potentially recursive) execution when the number of dynamically forked tasks gets too large to be executed efficiently in parallel.

Although we have here only scratched the surface of describing how the Gossamer library operates, suffice it to say that our initial prototype implementation is able to achieve reasonable speed-up on our current experimental platform, which is a 4-processor, shared-memory machine with 64-bit x86 processors. The following table gives performance numbers for applications we have constructed to date.

Application	**Arguments**	**1 CPU**	**2 CPU**	**2 Speedup**	**4 CPU**	**4 Speedup**
Fibonacci	48	46.88	27.31	1.72	12.58	3.73
Jacobi	8192, 16	16.78	8.73	1.92	4.55	3.68
Matrix multiply	2048	148.06	75.76	1.95	40.72	3.63
Multigrid	16384, 4	63.42	42.27	1.50	21.41	2.96
N-body	16384, 8, 512	5.80	3.69	1.57	2.36	2.46
N-queens	15	108.76	54.74	1.99	35.76	3.04
Run-length encoding	1Gb data file	5.83	3.98	1.46	3.09	1.88
Adaptive quadrature	1, 8192, 10-12	7.45	3.96	1.88	2.10	3.55
jzip with iterative blocks	10Mb data file	10.00	6.00	1.67	4.98	2.04
jzip with recursive sort	5Mb data file	7.66	6.35	1.21	6.15	1.25

As can be seen, the applications that inherently have good parallelism get relatively good speedup. The basically sequential algorithmsrun-length encoding and jzip still get noticeable speedup. In the case of jzip, we only have 4 processors so could not effectively use both forms of parallelism at once. Moreover, there is more to be gained from compressing blocks in parallel than from parallel sorting, which is a step in one of the three phases and accounts only 10% of the overall execution time in the sequential program. However, if we had more processors in our test platform and thus could have employed both forms of parallelism at once, we would have seen quite reasonable speed improvement. Note that all of these timing results are for minimally annotated programs and our first prototype implementation of the Gossamer library.

1.3 Gossamer Translator

We have only just begun to design the software that will translate programs that are written in C with our annotations to ones that are written in C together with the Gossamer library. The translator will have the functionality summarized earlier on page 2. Initially we will focus on simply converting code to use the Gossamer library; later we will incorporate feedback mechanisms such as dependency checking, race-condition checking, and generating performance suggestions.

Chapter 2

Programming in Gossamer

2.1 Gossamer Compiler

The Gossamer compiler is... Various warning levels of `gsc` can detect loop carried dependencies, unsafe uses of global variables in parallel code and other parallel conundrums.

Figure 2.1 illustrates the process by which a Gossamer program is compiled. Gossamer program files, which end with `.gc` by convention, are first transformed to ordinary C code by the Gossamer type-checking preprocessor `gsc`, producing `.c` files. The C code is then compiled using the `gcc` compiler and linked with the Gossamer runtime system. You do not need to worry about this compilation process, though, since the `gsc` command takes care of everything.

2.2 Compiling Gossamer Programs

In order to compile Gossamer programs, Gossamer 1.0.0 (build 8513) installs the `gsc` command, which is, roughly speaking, a source to source compiler. The `gsc` command understands that files with the `.gc` extension are Gossamer programs and acts accordingly.

`gsc` accepts many of the same arguments as the `gcc` compiler. For example, you can type the command

```
$ gsc -O2 <program name>.gc -o <program name>
```

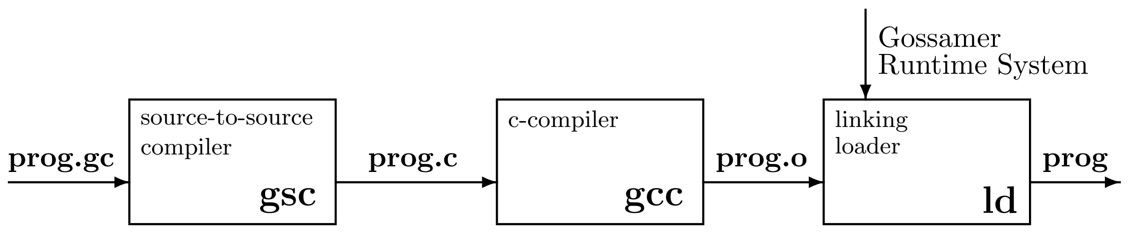

Figure 2.1: Compiling Gossamer program. Gossamer code is passed through the `gsc` compiler, a type-checking preprocessor which generates C output. This output is then compiled with `gcc` and linked with the Gossamer runtime library.

to produce the `<program name>` executable. You can also use many of your favorite `gcc` options, such as `-g`, `-Wall`, and so on.

The `gsc` compiler accepts the following arguments:

`-E` Stop after producing `.i` files (fully preprocessed files).

`-c` Stop after producing `.o` files (binary files to be linked).

`-v<n>` Set verbosity level to `<n>` to show what commands are being executed. If `<n>` is omitted, the default level is 2.

`-gsp` Instruct `gsc` to insert code to instrument for profiling.

`-MMD` Produce dependency rules (see the option for `-MMD` to gcc).

`-I<directory>` Add the directory to the head of the list of directories to be searched for header files.

`-D<argument>` Pass the `-D<argument>` to the preprocessor.

`-A<argument>` Pass to the preprocessor.

`-include<argument>` Pass to the preprocessor.

`-Wp,<argument>` Pass to the preprocessor.

`-g<argument>` Pass to gcc (produce symbol table information).

`-O<argument>` Pass to gcc (optimize, e.g., `-O2`).

`-Wl,<argument>` Pass to gcc, which will pass it to the linker.

`-Wc,<argument>` Strip the `-Wc,` and pass to the C compiler. Useful if your C compiler accepts a command the `gsc` doesn't understand.

`-Wall` Pass to gcc, and pass `-Wgs,all` to `gsc`.

`-Wgs,<argument>` Strip the `-Wgs,` and pass to `gsc`.

`-W<argument>` **(not matching the above)** Pass to gcc, such as `-Wextra`.

`-l<argument>` Pass to gcc, to specify a library to link.

`-L<argument>` Pass to gcc, to specify a directory to look for libraries in.

`-static` Pass to gcc, to specify that static linking should be used.

`-n` Do not actually do any compilation. Useful when `-v` option is specified.

`-m*<argument>` machine-specific options, passed to gcc. (If you specify this argument and are not using the right kind of machine, you may get unspecified behavior.) The `-m` argument is passed to the preprocessor and to gcc.

`-f*<argument>` options, passed to gcc. The `-f` argument is passed to the preprocessor and to gcc.

`-o <file>` Specify where to write the output.

`-help` Print help and quit.

`-version` Print version and quit.

Like `gcc`, you may not run arguments together. Thus `-Ec` is not the same as `-E -c`.
Example: You can write things like

```
$ gsc -O2 -g3 -DFOO=1 -DBAR foo.c pars.gc hello.S obj.o -o prog -lm
```

which will cause

- `foo.c` to be preprocessed and compiled, and assembled,
- `hello.s` to be preprocessed and assembled,
- `pars.gc` to be preprocessed, transformed, and compiled, and assembled, and
- then all of the resulting compiled files (including `obj.o`) to be linked together with the math library (`-lm`) and written to a program called `prog`.
- The preprocessing will be performed with the `FOO` macro defined, and the compilation will performed with level 2 optimization and symbol tables.

During program development, it is useful to collect performance data and profile a Gossamer program. The Gossamer runtime system will profile a program for performance when it is compiled with the flags `-gsp`.

```
$ gsc -gsp -O2 <program name>.gc -o <program name>
```

The flag `-gsp` instructs the Gossamer compiler to instrument the program to collect data about how much time each processor spends working, how many thread migrations occur, how much memory is allocated, etc.

2.3 Running Gossamer Programs

The Gossamer runtime system accepts the following standard options, which must be specified before a Gossamer program's standard command line arguments:

`--help` List available options.

`--ncpus` n Use n processors in the computation. If $n = 0$, use as many processors as are available. The parameter n must be at least 0. The default is to use as many processors as are available. The parameter may also be set via the `GOSSAMER_NCPUS` environmental variable.

`--stats` l Set statistic level to l. The higher the level, the more detailed the information. The default level is 0, which disables any printing of statistics.

`--no-stats` Equivalent to `--stats 0`.

`--pthread-stacksize` *size* Set the size of each worker thread's stack (in bytes). The pthread stacksize may also be set via the `GOSSAMER_PTHREAD_STACKSIZE` environmental variable.

`--` Stop parsing options and pass subsequent options to the Gossamer program unaltered.

These options must be specified before all user options, since they are processed by the Gossamer runtime system and stripped away before the rest of the user options are passed to `main`.

Gossamer program compiled with profiling support can be instructed to print performance information by using the `--stats` option. The command line

```
$ <program name> --stats <args>
```

yields an output similar to the following:

```
<program output>

PROFILING STATISTICS:

4 CPUS: %100.00 %99.99 %98.99 %100.00
Memory Allocated: 26.45 MB
...
```

After the output of the program itself, the runtime system reports several useful statistics collected during execution. Additional statistics can be obtained by setting the statistics level higher than 1.

2.4 Example Application – Fibonacci

In this section, we will go through the steps of compiling, running, and debugging a gossamer application. We use Fibonacci because it is simple and easy to understand the program, allowing the programmer to focus on additions from Gossamer.

We use the example Fibonacci program from Appendix **??**, Figure **??**. To compile Fibonacci without debugging or profiling

```
$ gsc -O2 fib.gc -o fib
```

This will produce an executable called `fib` and can be run just as a normal executable. The following code will calculate Fibonacci of 40

```
$ ./fib 40
fib: 102334155
```

By default, Gossamer compiled programs will attempt to use all available CPUs on the system. To limit the number of CPUs to 8, one of the following two methods can be used

```
$ export GOSSAMER_NCPUS=8
$ ./fib 40
fib: 102334155

or

$ ./fib --ncpus 8 40
fib: 102334155
```

Gossamer will first look for the environment variable called `GOSSAMER_NCPUS`. The user can also pass the number of CPUs to use as the command line parameter **`--ncpus`**. If both `GOSSAMER_NCPUS` and **`--ncpus`** are specified, **`--ncpus`** takes precedence and overwrites any value from the environment.

Chapter 3

Associative Memory

3.1 Using Associative Memory in Gossamer

3.2 Profiling Associative Memory Performance

3.3 Associative Memory Implementation

Chapter 4

Performance Tuning and Profiling

Appendix A

Keyword Transformations

A.1 Fork/Join

A.1.1 Parallel Quicksort

figref of (appendix-a-fj-simple-fig), we will go through the steps of the Gossamer compiler, `gsc`. First, any recursive procedures that are called with `fork` are cloned and Gossamer keywords are elided. This transformation is showed in Figure ??. Next, a structure which contains a parallel procedure's return type and arguments is created. Figure ?? shows the structure for `fib()`. Next, a parallel wrapper procedure is created for `fib()`. This wrapper procedure is invoked by the Gossamer runtime system and handles the unmarshaling(spelling?) of `fib()`'s arguments and pruning of recursive calls when the pruning threshold is exceeded. The wrapper procedure is shown in Figure ??. Finally, the `fib()` procedure is transformed to invoke the Gossamer runtime library and the transformation is shown in Figure ??.

```
#include <gossamer.h>

int fib(int n) {
    int x, y;

    if (n <= 2) {
        return 1;
    } else {
        x = fork fib(n-1); // Fibonacci of (n-1)
        y = fork fib(n-2); // Fibonacci of (n-2)
        join;
        return x + y;
    }
}

int main(int argc, char **argv) {
    printf("fib: %d\n", fib(atoi(argv[1])));
    return 0;
}

int fib(int n) {
    int x, y;

    if (n <= 2) {
        return 1;
    } else {
        struct _gossamer_fib_args *tmp1 = malloc(...);
        tmp1->retval = &x;
        tmp1->n = n - 1;
        gsCreateFJFilament(mt_fib, tmp1);

        struct _gossamer_fib_args *tmp2 = malloc(...);
        tmp1->retval = &y;
        tmp1->n = n - 2;
        gsCreateFJFilament(mt_fib, tmp2);

        gsFJJoin();

        return x + y;
    }
}

struct _gossamer_fib_args {
    void *retval;
    int n;
};

void __gs_run_fib(void *args) {
    struct _gossamer_fib_args *fibargs = (struct _gossamer_fib_args *) args;
    *((int *) fibargs->retval) = fib(fibargs->n);
    free(args);
}

static int solutions = 0;

void putqueen(char **board, int n) {
    int i, j;

    if (row == n) {
        atomic { solutions++; }  // atomically increment global variable
        return;
    }

    for (j = 0; j < n; j++) {
        if (upleftOK(board, row, j) &&
            aboveOK(board, row, j) && uprightOK(board, row, j)) {
            board[row][j] = 'Q';
            fork putqueen(copy board, row+1); // fork new filament
                                              // copying char ** parameter
            board[row][j] = '-';
        }
    }
    join;  // wait for <n> children to finish
}

int main(int argc, char **argv) {
    char **board = board_init();
    put_queen(board, 0);
    return 0;
}

gs_safe static int *data;

void quicksort(int l, int r) {
    register int i, j;

    if (r > l) {
        i = l - 1;
        j = r;

        for (;;) {
            while (i < r && data[++i] < data[r]) ;
            while (j > 0 && data[--j] > data[r]) ;
            if (i >= j) { break; }
            swap(i, j);
        }
        swap(i, r);

        quicksort(l, i-1);
        quicksort(i+1, r);
     }
}
```

A.1.2 More Complex Recursive Fork/Join

Solving the N-Queens problem with Gossamer and Fork/Join parallelism is shown in Figure **??**. The procedure `putqueen()` is recursively called `n` (size of the board) times, each time requiring a copy of the board. Gossamer will then create a copy of `board` using heap memory for each fork call. The allocated memory is then freed before just before `putqueen()` call returns.

In this particular N-Queens implementation, the number of found solutions is kept in a global variable and incremented in the recursive procedure `putqueen()` (see Figure **??**, line `08`). Using global data structures in parallel procedure requires the `atomic` block to provide mutual exclusion and prevent two or more threads from simultaneously updating the same memory location. In Gossamer, an `atomic` block can implemented in one of two ways

- Code block is surrounded by global locks, either spin locks or mutexes

- Code block is implemented as reduction ... when is the implicit ”reduce” called ?? each time variable (solutions) is used inside a parallel context ??

The Gossamer compiler, `gsc`, will decide which synchronization method is appropriate for a given situation by analyzing the `atomic` code block.

Parallel Quicksort Example!

A.2 Parallel

Parallel Examples Go Here...

Appendix B

Bugs and Known Issues

B.1 Bugs

The Gossamer is bug free!

B.2 Known Issues

There are no known issues with such a perfect system!!

Appendix C

FAQ

C.1 Frequently Asked Questions

- **My recursive program has a segmentation fault and I can't figure out why.**
  Possible causes for strange segmentation faults with recursive programs is running out of stack space. Worker threads are allocated 10MB by default, but can overwritten by the environment variable `GOSSAMER_PTHREAD_STACK=<hex>`.

Bibliography

[1] OpenMP. The OpenMP API specification for parallel programming. http://www.openmp.org.