Gossamer: A Lightweight Programming Framework for Multicore Machines*

Joseph A. Roback and Gregory R. Andrews
Department of Computer Science, The University of Arizona, Tucson

1 Introduction

The key to performance improvements in the multicore era is for software to utilize the available concurrency. A recent paper [3] summarizes the challenges and describes twelve so-called dwarfs—types of computing and communication patterns that occur in parallel programs. One of the key points in the paper is that a general programming model has to be able to accommodate all of the patterns defined by the dwarfs, singly or in combination. The challenge is to do so both simply and efficiently.

Parallel programming can be supported by providing a threads library [14, 15, 2]; by modifying compilers to extract parallelism or use optimistic code execution techniques [4, 6, 5]; by using concurrency features of existing languages such as Java or C#; by designing new programming languages such as Erlang [16], Fortress [24], X10 [22], and ZPL [7]; or by annotating sequential programs with directives that specify concurrency and synchronization, as in Cilk [11], Cilk++ [8], OpenMP [20], and others [25, 19, 1, 12, 17, 21].

All of these approaches are valuable and are producing useful results, but the last approach—annotating programs—has, in our opinion, the most potential to be simultaneously simple, general, and efficient. In particular, annotations are easier to use than libraries because they hide lots of bookkeeping details, and they are simpler to learn than an entire new programming language. Annotation-based approaches also have efficient implementations. However, no existing approach is general enough to support all the computational patterns (dwarfs) defined in [3].

This paper describes Gossamer, an annotation-based approach that is general as well as simple and efficient. Gossamer 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; (2) a source-to-source translator that takes an annotated sequential program and produces an optimized program that uses our threading library; and (3) a run-time system that provides efficient fine-grained threads and high-level synchronization constructs.

As will be seen, the Gossamer annotations are as simple to use as those of Cilk++, and Gossamer's performance is better than OpenMP. What sets Gossamer apart is a more extensive set of annotations that enable solving a greater variety of applications. In addition to iterative and recursive parallelism, Gossamer supports pipelined computations by a general ordering primitive, domain decomposition by means of replicated code patterns, and MapReduce [10, 26] computations by means of an efficient associative memory type.

The paper is organized as follows. Section 2 introduces our programming model and annotations by means of numerous examples. Section 3 summarizes the Gossamer translator and run-time system. Section 4 gives experimental results. Section 5 discusses related work.

2 Annotations

Gossamer provides 15 simple annotations, as listed in Table 1: ten to specify concurrency and synchronization and five to program MapReduce computations. The `fork` annotation supports task and recursive parallelism. The `parallel` annotation supports data parallelism that occurs when all iterations of a for loop are independent. The `divide/replicate` annotation supports data parallelism that occurs when shared data can be decomposed into independent regions, and the same code is to be executed on each region.

For synchronization, `join` is used to wait for forked threads to complete; `copy` is used when an array needs to be passed by value; `barrier` provides barrier synchronization within replicated code blocks; `ordered{code}` delays the execution of `code` until the predecessor sibling thread has also finished executing

*This work was supported in part by NSF Grants CNS-0410918 and CNS-0615347.

Use	Annotations
Concurrency	fork, parallel, divide/replicate
Synchronization	atomic, barrier, buffered, copy join, ordered, shared
MapReduce	mr_space, mr_list mr_put, mr_getkey, mr_getvalue

Table 1: Gossamer Annotations

```
void qsort(int *begin, int *end) {
  if (begin != end) {
    int *middle;
    end--;
    middle = partition(begin, end, *end);
    swap(end, middle);
    fork qsort(begin, middle);
    fork qsort(++middle, ++end);
    join;
  }
}
```

Figure 1: Quicksort

`code`; `buffered{write calls}` buffers the writes in a local buffer, which is flushed when the thread terminates; `buffered(ordered)` causes thread buffers to be flushed in sibling order; and `atomic{code}` causes `code` to be executed atomically.

Gossamer supports the MapReduce programming model by means of three operations: `mr_put`, which deposits a (key,value) pair into an associative memory having type `mr_space`; `mr_getkey`, which returns from an `mr_space` a key and a list (of type `mr_list`) of all values having that key; and `mr_getvalue`, which returns the next value from an `mr_list`.

Below we illustrate the use of the annotations by means of several examples, in which the annotations are highlighted in boldface.

Quicksort is a classic divide-and-conquer algorithm that divides a list into two sub-lists and then recursively sorting each sub-list. Since the sub-lists are independent, they can be sorted in parallel. An annotated version of quicksort is shown in Figure 1.

N-Queens is a depth-first backtracking algorithm [9]. It tries to solve the problem of placing `N` chess queens on an `NxN` chessboard such that no one queen can capture any other. An annotated version of a recursive N-Queens algorithm is shown in Figure 2. Each attempt at placing a queen on the board is forked and checked in parallel using the recursive `putqueen()` function.

Two issues arise from parallelizing `putqueen()`. First, the global variable `solutions` is incremented every time a solution is found, so an `atomic` annotation is added to ensure that updates are atomic. Second,

```
int solutions = 0;
void putqueen(char **board, int row) {
  int j;
  if (row == n) {
    atomic { solutions++; }
    return;
  }
  for (j = 0; j < n; j++) {
    if (isOK(board, row, j)) {
      board[row][j] = 'Q';
      fork putqueen(copy board[n][n], row+1);
      board[row][j] = '-';
    }
  }
  join;
}
```

Figure 2: N-Queens Problem

```
int main(int argc, char **argv) {
  ...
  while (!feof(infp)) {
    insize = fread(in, 1, BLKSIZE, infp);
    fork compressBlk(copy in[insize], insize);
  }
  join;
  ...
}
void compressBlk(char *in, int insize) {
  ...
  BZ2_bzCompress(in, insize, out, &outsize);
  ordered { fwrite(out, 1, outsize, outfp); }
  ...
}
```

Figure 3: Bzip2

the `board` array is passed by reference on each call to `putqueen()`, which would cause the board to be shared among the `putqueen()` threads. A `copy` annotation is used to give each thread its own copy of the `board`.

Bzip2 [23] compression uses the Burrows-Wheeler algorithm, followed by a move-to-front transform, and then Huffman coding. Compression is performed on independent blocks of data, which lends itself to block-level task parallelism. A parallel version of bzip2 using Gossamer annotations is shown in Figure 3. The `main()` function reads blocks of data and forks `compressBlock()` to compress each block in parallel; it then waits for all threads to complete. The `ordered` annotation is used within `compressBlock` to ensure that compressed blocks are output in the same order that uncompressed blocks are read from input.

Matrix Multiplication is an example of iterative data parallelism. The program in Figure 4 gives a version of matrix multiplication that uses cache efficiently by transposing the inner loops of the usual algorithm. All itera-

```
double **A, **B, **C; // initialize the arrays
parallel for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
      C[i][j] += A[i][k] * B[k][j];
```

Figure 4: Matrix Multiplication

```
double **old, **new;
int i, j, it, n, m;

old++; new++; n-=2;

divide old[n][], new[n][] replicate {
  for (it = 0; it < MAXITERS; it += 2) {
    for (i = 0; i < n; i++)
      for (j = 1; j < m-1; j++)
        new[i][j] = (old[i-1][j] + old[i+1][j] +
          old[i][j-1] + old[i][j+1]) * 0.25;
    barrier;
    for (i = 0; i < n; i++)
      for (j = 1; j < m-1; j++)
        old[i][j] = (new[i-1][j] + new[i+1][j] +
          new[i][j-1] + new[i][j+1]) * 0.25;
    barrier;
  }
}
```

Figure 5: Jacobi Iteration

tions of the outer loop are independent because they work on different rows of the result matrix, so the outer loop is prefixed by the `parallel` annotation. This results in `n` tasks.

Jacobi Iteration is a simple method for approximating the solution of a partial differential equation such as Laplace's equation in two dimensions: $\nabla^2(\Phi) = 0$. Given boundary values for a region, the solution is the steady values of interior points. These values can be approximated by discretizing the region using a grid of equally spaced points, and repeatedly updating each grid point by setting it to the average of its four neighbors from the previous iteration.

Figure 5 shows an annotated program. Here we use the `divide/replicate` annotation because we want to execute the update algorithm on sub-regions of the `old` and `new` grids. Here, `divide` partitions the shared grids along the first dimension, so each processor gets a strip of rows; Gossamer takes care of the bookkeeping required to redefine the loop bounds in each instance of the computation. Each computation first computes `new` values using the `old` grid points. A `barrier` annotation is used to ensure that all threads have completed this step before continuing. Then the computation computes `old` values using the `new` grid points. Again a `barrier` annotation is used to ensure that all threads complete this step before

```
FILE *out_fp;
char *data;
int size, run, val;
divide data[size]
  where data[divide_left] != data[divide_right]
replicate {
  while (size > 0) {
    val = *data++;
    size--;
    run = 1;
    while (val == *data && size > 0) {
      run++; data++; size--;
      if (run == RUNMAX) { break; }
    }
    buffered (ordered) {
      fwrite(&val, sizeof(int), 1, out_fp);
      fwrite(&run, sizeof(int), 1, out_fp);
    }
  }
}
```

Figure 6: Run Length Encoding

continuing. The update process is repeated `MAXITERS` times.

Run Length Encoding (RLE) compresses data by converting sequences of the same value to (value,count) pairs. Figure 6 shows an RLE implementation that scans an array byte-by-byte, recording each run and writing the (value,count) pairs to output. The `divide` annotation partitions `data` into equal-sized chunks and assigns one to each processor. The original sequential code inside the `replicate` annotation is executed concurrently on each chunk. Note that here the replicated code is a while loop over the input.

The `where` annotation adjusts chunk boundaries to the right as necessary to ensure that data is not split at points that would break up runs of identical values. (Without the `where` clause, the ouput would be a legal encoding, but it might not be the same as the output of the original sequential program.) The `buffered` annotation is used to buffer `fwrite()` calls in local storage, which greatly improves efficiency. The `ordered` option on the `buffered` annotation ensures that output is written in the correct order.

Word Count calculates the number of occurrences of each word in a set of data files. This problem can be solved using the MapReduce style. The map phase finds all the words in the files; the reduce phase counts the number of instances of each word. Figure 7 gives a Gossamer implementation that uses the `fork` and `join` annotations for concurrency and the MapReduce annotations to implement intermediate storage.

The map phase forks a thread for each input file. Each thread reads the data in a file and emits words to the

```
mr_space wordcount(char *, int);
main(void) {
  char *key;
  mr_list values;
  for (i = 0; i < n; i++)
    fork map(file[i]);
  join;
  while (mr_getkey(wordcount, &key, &values))
    fork reduce(key, values);
  join;
}
void map(char *file) {
  char *word;
  while ((word = getnextword(file)) != NULL)
    mr_put(wordcount, word, 1);
}
void reduce(char *key, mr_list values) {
  int val, count = 0;
  while (mr_getvalue(wordcount, values, &val))
    count += val;
  printf("word: %s, count: %d\n", key, count);
}
```

Figure 7: Word Count using MapReduce

mr_space wordcount, which is an associative memory of (key,value) pairs. The reduce phase forks a thread for each word in wordcount. Each reduce thread extracts all of its words from wordcount, counts them, and writes its count. When the program terminates, wordcount is empty.

3 Translator and Run-time System

The Gossamer translator turns an annotated C program into an executable program. First it parses the annotated program and performs various analyses and transformations. Then the intermediate code is transformed back into C code that uses the Gossamer run-time library to implement the annotations. Finally the transformed source code is transparently piped into the GNU C Compiler for compilation and linking into an executable.

The Gossamer run-time system is implemented using the POSIX threads library (Pthreads) [14]. In particular, Gossamer uses Pthreads to create one server thread per processor on the underlying hardware platform.

Application-level threads in Gossamer are called *filaments* [18]. Each filament is a lightweight structure represented by a function pointer and its arguments. Filaments do not have private stacks; instead each filament uses the stack of the server thread on which the filament is executed. A filament cannot be preempted; once started, it executes until it terminates. This is necessary since filaments do not have private stacks, and it avoids the overhead of requiring machine-dependent context-switching code.

A fork annotation creates a new filament for the annotated function call and its arguments. The filament is enqueued with the run-time system using a round-robin scheduling algorithm to promote load balancing. Recursive function calls annotated with fork are executed sequentially if a pruning threshold is reached. (This threshold can be set by the user; it's default value is twice the number of processors.) For a parallel for loop, one filament is created for each iteration of the loop, with the loop iteration variable as its argument. These filaments are enqueued on servers threads in groups of n/P, where n is the number of iterations and P is the number of processors. This is done to ensure good memory and cache performance since most loops operate on large data sets and exhibit spatial locality. The divide/replicate annotation creates a filament for the replicate block on each processor and evenly divides the specified arrays among them. Constraints on division specified by the where annotation are evaluated concurrently by each filament. Each barrier annotation is implemented using a dissemination barrier [13] for scalability.

For an ordered annotation, code is placed before the ordered block to check if the filament's previous sibling has completed its corresponding ordered block. If so, the filament continues executing; otherwise the filament waits. Code is also placed after the ordered block to indicate completion and to signal the next filament. A buffered annotation is implemented by generating wrapper functions that buffer output, and replacing all write() and fwrite() system calls with calls to these functions. The atomic annotation wraps the code block with either mutex locks or architecture independent spin-locks; when the only statement(s) inside the block are assignments, and each statement is associative and commutative, then the assignments are performed as reduction operations instead of using locks.

MapReduce functions are replaced by type-specific implementations determined by the types specified in the mr_space on which they operate. MapReduce keys and values can be any C primitive type or null-terminated strings. For efficiency, keys are hashed in per processor linked-lists of key/value arrays; the list is used to chain key collisions on the same processor. Insertion thus does not require locking; only key (but not value) removal requires locking.

4 Experimental Results

The Gossamer package has been tested on the seven applications shown earlier plus n-body and multigrid. These cover a range of programming styles and computation/communication models. Timing tests were run on an Intel Xeon 8-core multiprocessor system having two quad-core processors running at 2.0 GHz with 12Mbyte of L2 cache and 8Gbyte of shared memory; the operating

Application	Parameters	Execution Time (seconds)					Speed Up (relative to sequential)			
		Sequential	1-CPU	2-CPU	4-CPU	8-CPU	1-CPU	2-CPU	4-CPU	8-CPU
Quicksort	n = 100,000,000	17.86	17.41	9.04	6.45	5.65	1.026	1.98	2.77	3.16
N-Queens	n = 14	9.29	10.41	5.01	2.42	1.31	0.89	1.86	3.85	7.11
Bzip2	file = 256.0 MB	46.06	46.37	23.88	12.58	6.41	0.99	1.93	3.66	7.19
Matrix Multiplication	n = 4096	198.51	193.06	96.18	47.42	23.74	1.03	2.06	4.19	8.36
Jacobi Iteration	grid = 1024x1024, iterations = 65536	277.99	278.04	124.89	67.99	38.94	1.00	2.23	4.09	7.14
Run Length Encoding	file = 4.0 GB	6.23	6.48	3.23	1.62	0.84	0.96	1.93	3.85	7.44
Word Count	files = 1024, file size = 2.0 MB	124.06	125.08	61.32	35.48	19.05	0.99	2.02	3.50	6.51
Multigrid	grid = 1024x1024, iterations = 512	0.15	0.16	0.08	0.04	0.02	0.97	1.99	3.78	6.29
N-Body	bodies = 32768, iterations = 16384, tree builds = 16	193.10	193.05	97.07	49.15	25.36	1.00	1.99	3.93	7.61

Table 2: Performance Results

system was Ubuntu Linux 9.10. The applications were compiled with the GNU C compiler version 4.4.1 using -O3 optimization. No hand-tuning was done, so the applications do not get the best possible performance.

Table 2 gives execution times and speedups for the algorithmic portion of each of the 9 applications tested. All timing tests were run in single-user mode. The execution times are the averages of ten test runs, as reported by `gettimeofday()`, rounded to the nearest hundredth of a second. The reported speedups are the ratios of the *sequential* program times to the other times.

Gossamer demonstrates excellent speedups on most tests. The super-linear speed achieved for Jacobi iteration and matrix multiplication is due to cache effects, because both algorithms can make effective use of the doubling of shared L2 cache that is available to the run-time when using both processor sockets. The performance of multigrid falls off somewhat on 8 processors because it uses smaller grids and has more barrier synchronization points. Quicksort has the worst performance on 4 and 8 processors because array partitioning is a sequential bottleneck, and the function has to be forked several times before all processors have work.

Measurements of the total execution time yield very similar speedup numbers. This is because most programs do relatively little initialization—or the initialization is part of the timed portion of the algorithm. Run length encoding is the one program that has poor speedup numbers for overall execution time—1.35 on 4 cores and 1.42 on 8 cores—because the entire 4.0 GB file is read before parallel execution begins.

5 Related Work

Gossamer is an annotation-based approach, so the most closely related work is Cilk++ [8] and OpenMP [20]. Cilk++ provides three keywords to express parallelism and synchronization: `cilk_spawn` and `cilk_sync` for task and recursive parallelism, and `cilk_for` loop parallelism. Cilk++ also provides library support for generic mutex locks and C++ templates called *hyperobjects* that allow Cilk strands to coordinate when updating a shared variable or performing a reduction operation. An output stream *hyperobject* can be used to to achieve the same behavior as Gossamer's `buffered` annotation. Cilk++ cannot (at all directly) support all computational dwarfs, because it lacks annotations equivalent to Gossamer's `divide/replicate`, `barrier`, `copy`, and MapReduce.

OpenMP supports shared-memory programming in C, C++ and Fortran on many architectures. OpenMP consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The OpenMP execution model consists of a master thread that forks worker threads and divides tasks among them. A parallel code section is marked with a preprocessor directive that causes threads to be created before the section is executed. Task parallelism and data parallelism can be achieved using work-sharing constructs to divide a task among the threads. OpenMP also supports reductions on scalar variables. OpenMP lacks annotations equivalent to Gossamer's `buffered` and MapReduce. Also, OpenMP's `ordered` construct is restricted to use only inside parallel for loops. Timing tests on recursive programs like quicksort and n-queens show that Gossamer is orders of magnitude faster due to the overhead created by OpenMP's inability to automatically prune parallel recursive calls.

Other annotation-based approaches include UPC [25], Intel Ct [12], ATI Stream SDK [1], Nvidia CUDA [19], OpenMP to GPGPU [17], and RapidMind [21]. Each of these targets specific applications and hardware, most commonly stream computations executed on general-purpose graphics processing units. This yields excellent performance, but over a limited range of applications.

Acknowledgement

The referees provided helpful feedback that improved the paper.

References

[1] AMD CORPORATION. ATI Stream SDK v2.0. http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx, 2009.

[2] APPLE INC. Grand Central Dispatch (GCD) reference, 2009.

[3] ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J., MORGAN, N., PATTERSON, D., SEN, K., WAWRZYNEK, J., WESSEL, D., AND YELICK, K. A view of the parallel computing landscape. *Commun. ACM 52*, 10 (2009), 56–67.

[4] BENKNER, S. Vfc: the vienna fortran compiler. *Sci. Program. 7*, 1 (1999), 67–81.

[5] BIK, A., GIRKAR, M., GREY, P., AND TIAN, X. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems, 2001.

[6] BLUME, W., EIGENMANN, R., FAIGIN, K., GROUT, J., HOEFLINGER, J., PADUA, D., PETERSEN, P., POTTENGER, W., RAUCHWERGER, L., TU, P., AND WEATHERFORD, S. Polaris: Improving the effectiveness of parallelizing compilers. In *In Languages and Compilers for Parallel Computing* (1994), Springer-Verlag, pp. 141–154.

[7] CHAMBERLAIN, B. L., CHOI, S.-E., LEWIS, E. C., SNYDER, L., WEATHERSBY, W. D., AND LIN, C. The case for high-level parallel programming in zpl. *IEEE Comput. Sci. Eng. 5*, 3 (1998), 76–86.

[8] CILK ARTS INC. Cilk++ technical specifications and data sheet. Data Sheet, Cilk Arts Inc., September 2009. Available online, http://software.intel.com/en-us/articles/intel-cilk/.

[9] DAHL, O. J., DIJKSTRA, E. W., AND HOARE, C. A. R. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.

[10] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[11] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (New York, NY, USA, 1998), ACM, pp. 212–223.

[12] GHULOUM, A., SPRANGLE, E., FANG, J., WU, G., AND ZHOU, X. Ct: A flexible parallel programming model for tera-scale architectures. http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf, October 2007.

[13] HENSGEN, D., FINKEL, R., AND MANBER, U. Two algorithms for barrier synchronization. *International Journal of Parallel Programming 17*, 1 (1988), 1–17.

[14] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. pub-IEEE-STD, pub-IEEE-STD:adr, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[15] INTEL CORPORATION. Intel thread building blocks. http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm, 2006.

[16] LARSON, J. Erlang for concurrent programming. *Commun. ACM 52*, 3 (2009), 48–56.

[17] LEE, S., MIN, S.-J., AND EIGENMANN, R. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), ACM, pp. 101–110.

[18] LOWENTHAL, D. K., FREEH, V. W., AND ANDREWS, G. R. Efficient support for fine-grain parallelism on shared-memory machines. *Concurrency: Practice and Experience 10*, 3 (1998), 157–173. http://citeseer.ist.psu.edu/lowenthal99efficient.html.

[19] NVIDIA CORPORATION. Nvidia Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda_home.html, 2009.

[20] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP specifications, v3.0. http://openmp.org/wp/openmp-specifications/, May 2008.

[21] RAPIDMIND INC. RapidMind Multicore Development Platform. http://www.rapidmind.net/, 2009.

[22] SARASWAT, V. A., SARKAR, V., AND VON PRAUN, C. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2007), ACM, pp. 271–271.

[23] SEWARD, J. Bzip2, 2008. Available online, http://www.bzip.org/.

[24] SUN MICROSYSTEMS INC. The Fortress language specfication, v1.0. Tech. rep., Sun Microsystems Inc., March 2008.

[25] UPC CONSORTIUM. UPC language specifications, v1.2. Tech. rep., UPC Consortium, 2005.

[26] YOO, R., ROMANO, A., AND KOZYRAKIS, C. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (Oct. 2009), pp. 198–207.