Effective Parallelization of Loops
in the Presence of I/O Operations

Min Feng   Rajiv Gupta   Iulian Neamtiu

Department of Computer Science and Engineering
University of California, Riverside
{mfeng,gupta,neamtiu}@cs.ucr.edu

Abstract

Software-based thread-level parallelization has been studied for exploiting data parallelism in purely computational loops to improve program performance on multiprocessors. However, none of the previous efforts deal with efficient parallelization of *hybrid loops*, i.e., loops that contain a mix of computation and I/O operations. In this paper, we propose a set of techniques for efficiently parallelizing hybrid loops. Our techniques apply DOALL parallelism to hybrid loops by breaking the cross-iteration dependences caused by I/O operations. We also support speculative execution of I/O operations to enable speculative parallelization of hybrid loops. Helper threading is used to reduce the I/O bus contention caused by the improved parallelism. We provide an easy-to-use programming model for exploiting parallelism in loops with I/O operations. Parallelizing hybrid loops using our model requires few modifications to the code. We have developed a prototype implementation of our programming model. We have evaluated our implementation on a 24-core machine using eight applications, including a widely-used genomic sequence assembler and a multi-player game server, and others from PARSEC and SPEC CPU2000 benchmark suites. The hybrid loops in these applications take 23%–99% of the total execution time on our 24-core machine. The parallelized applications achieve speedups of 3.0x–12.8x with hybrid loop parallelization over the sequential versions of the same applications. Compared to the versions of applications where only computation loops are parallelized, hybrid loop parallelization improves the application performance by 68% on average.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures

General Terms Languages, Performance

Keywords DOALL Parallelization, Speculative Parallelization, Helper Threading, I/O contention

1. Introduction

Uncovering parallelism is crucial for improving an application's performance on shared memory multiprocessors. Software-based thread-level parallelization has been widely studied for exploiting parallelism on multiprocessors. Many parallel programming models have been proposed to facilitate the development of parallel applications on shared memory multiprocessors. The shared-memory paradigm used by these programming models makes them easy-to-use for the programmer.

A majority of the parallel programming models (e.g., Threading Building Blocks [20], OpenMP [8], Galois [14], SpiceC [10]) focus on exploiting data parallelism in loops including DOALL, DOACROSS, pipelined, and speculatively parallelized loops. However, these programming models only target loops that contain pure computations, i.e., they are free of I/O operations. Since many applications contain loops with I/O operations, they fail to yield much speedup due to these loops still being sequential. Therefore, it is highly desirable to support parallel programming models which allow parallel execution of *hybrid loops*, i.e., loops with both computation and I/O operations. For example, Velvet [30] is a popular bioinformatics application. In its version 1.1 (i.e., the latest version when we conducted this work), 18 pure computation loops have been parallelized using OpenMP. However, since OpenMP lacks support for parallelizing loops with I/O operations, none of the hybrid loops in Velvet have been parallelized. Actually, Velvet has 39 hybrid loops, out of which 26 loops can be parallelized by the approach presented in this paper. These hybrid loops take a significant portion (27%–53%) of the total execution time in our experiments on a 24-core machine. Therefore, to further improve the performance of Velvet, developers must exploit the parallelism in hybrid loops. Interestingly, while we were writing this paper, the developers of Velvet were working independently on manually parallelizing the hybrid loops, which only confirms our observation of the need to parallelize hybrid loops.

In a few of the previous efforts [10, 26] where hybrid loops are parallelized, DOACROSS parallelism is used (i.e., synchronization is imposed to deal with cross-iteration dependences) even if the computation part of the loop can be performed using DOALL (i.e., no cross-iteration dependence in the computation part). However, DOACROSS execution is not as efficient as DOALL. Because DOACROSS execution assigns only one iteration per scheduling step, it results in significant synchronization and scheduling overhead. In addition, the speculation-based loop parallelization techniques proposed in these works are not applicable when the loop contains I/O operations. While the work on parallel I/O [22, 24, 25] provides a set of low-level techniques that change the I/O subsystem to improve the performance of multiple simultaneous I/O operations, it does not provide any means for programmers to parallelize hybrid loops.

In this paper, we propose compiler techniques for efficiently parallelizing *hybrid loops*, i.e., loops that contain I/O operations in addition to computation. Our techniques break the cross-iteration

dependences involving I/O operations. Therefore, we are able to employ DOALL parallelism whenever there is no cross-iteration dependence in the computation part of the loop. Our techniques also support speculative execution of I/O operations to enable speculative parallelization of hybrid loops. We observe increased I/O bus contention with aggressive parallelization of hybrid loops. For performance scalability, we propose the use of helper threading to reduce the I/O bus contention. The helper thread enables the working threads to refill their input buffers and flush their output buffers so that the I/O traffic is spread out and not bursty. Unlike Parallel I/O work, our techniques do not require changes in the I/O subsystem.

We provide an easy-to-use programming model for exploiting parallelism in hybrid loops. The programming model contains two I/O related pragmas. Programmers can parallelize hybrid loops by merely inserting the pragmas preceding the loops in sequential code. Similarly, to enable speculative parallelization of hybrid loops, programmers just need to use one of the proposed pragmas at the beginning of the loop. Employing helper threading in parallel loops simply requires calling helper threading APIs preceding the loops.

We have developed a prototype implementation of our programming model. The core components of the implementation consist of a source-to-source translator and a user-level runtime library. We evaluate our implementation on a 24-core Dell PowerEdge R905 server using eight benchmarks from PARSEC [3] and SPEC CPU2000 benchmark suites, and two real applications. These benchmarks are parallelized via DOALL and speculative parallelism. Our implementation achieves 3.0x–12.8x speedup in these benchmarks. In comparison to the parallelized versions of benchmarks without hybrid loop parallelization, our technique improves the performance by 30% to 272%.

The rest of the paper is organized as follows. Section 2 illustrates our approach to parallelizing hybrid loops. Section 3 presents our programming model and its implementation. Section 4 presents the evaluation. Section 5 discusses related work and Section 6 concludes this paper.

2. Parallelizing Hybrid Loops

We begin by discussing the challenges in parallelizing hybrid loops and then describe our approach to overcoming these challenges. First, we discuss why existing techniques for DOALL parallelization and speculative loop parallelization cannot be directly applied if a loop also contains I/O operations. Our goal is to generalize these techniques so that they can be applied to hybrid loops. Second, we show that parallelization of hybrid loops can lead to I/O contention which must be effectively handled to realize the full benefits of parallelization.

Enabling DOALL Parallelization of Hybrid Loops. Figure 1(a) shows a typical loop with I/O operations that can be found in many applications (e.g., bzip2, parser [12], and stream encoder/decoder). Each loop iteration first checks if the end of the input file has been reached. If not, data is read from the file, computation on the data is performed, and finally results are written to the output file. When the computation within a loop does not involve cross-iteration dependences, maximum parallelism can be exploited via DOALL parallelization where all loop iterations can be executed in parallel. However, in a hybrid loop, even when the computation does not involve cross-iteration dependences, DOALL parallelization is not possible because the file read/write operations introduce cross-iteration dependences due to the movement (in our example, advancement) of the file pointer. In prior work, e.g., [10], such loops are parallelized using DOACROSS parallelism which

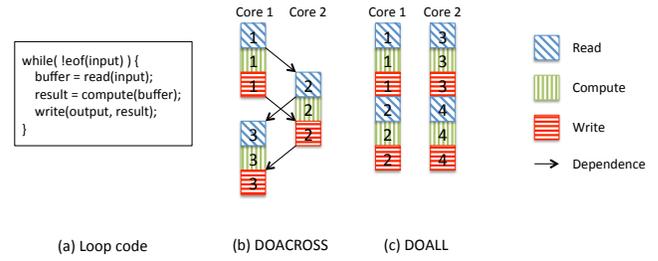incurs high scheduling and synchronization overhead—see Figure 1(b) for DOACROSS loop execution.


```
while( !eof(input) ) {
    buffer = read(input);
    result = compute(buffer);
    write(output, result);
}
```

| (a) Loop code | (b) DOACROSS | (c) DOALL |

Figure 1. Execution of the loop example.

To fully exploit the parallelism in the loop, we need to find a way to break the cross-iteration dependences due to I/O operations. In this paper we develop solutions to enable DOALL parallelization which leads to the execution shown in Figure 1(c). With DOALL parallelization we can eliminate the synchronization between consecutive iterations and employ more efficient scheduling polices such as Guided Self Scheduling [17]. Figure 2 compares the performance of DOACROSS with the performance of DOALL on a microbenchmark constructed based on the example loop type. In the microbenchmark, the *compute* function is composed of a loop. We can adjust the ratio of I/O workload vs. computation workload by varying the loop size. The figure shows the performance comparison for both I/O-dominant workload and computation-dominant workload. In the I/O-dominant workload, the I/O calls take around 75% of the execution time of the sequential loop while in the computation-dominant workload the I/O calls take only 25% of the loop execution time. In both cases, DOALL performs better than DOACROSS, especially with large number of parallel threads. *Therefore, the first challenge addressed in this work is to efficiently parallelize hybrid loops by enabling DOALL parallelization.*
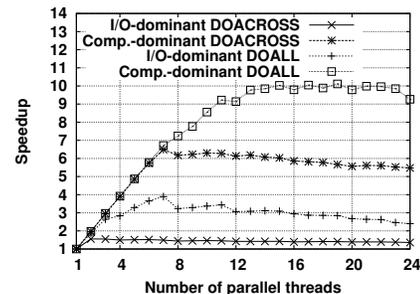

Figure 2. Performance comparison of DOACROSS and DOALL on the example loop.

Performing Speculative Parallelization of Hybrid Loops. Now let us consider the situation in which cross-iteration dependences exist in the computation part of the loop. We cannot perform DOALL parallelization of such loops even if we break the dependences introduced by I/O operations. Recent works have shown that an effective approach to handling cross-iteration dependences in the computation part is to employ speculative parallelization of loops. This approach is very effective when the cross-iteration dependences manifest themselves infrequently. Previous works [9, 26] have shown that speculative parallelization works better than non-speculative DOACROSS parallelization; however, these works also assume that the loops do not contain I/O operations. To apply speculative parallelization to the loops with I/O operations, we need to enable the speculative execution of the I/O
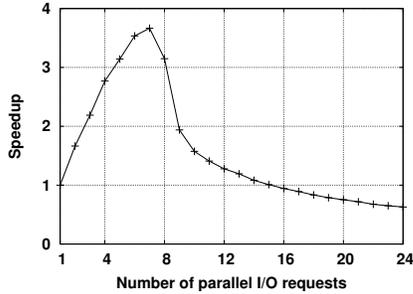
Figure 3. Performance of reading a 1 GB file using different numbers of parallel I/O requests.

operations. *Therefore, the second challenge of efficiently parallelizing hybrid loops is to develop solutions for speculative execution of I/O operations.*

I/O Contention due to Parallelization. Once DOALL and speculative parallelization of hybrid loops have been achieved, we are faced with yet another challenge. By increasing parallelism we also increase I/O bus contention. Figure 3 shows the performance of a microbenchmark that reads a 1 GB file from the disk. The microbenchmark creates multiple parallel threads, each of which then sends an I/O request to read a portion of the file. We varied the number of parallel threads (i.e., number of parallel I/O requests) to examine the impact of I/O bus contention. We can see that the I/O performance degrades quickly with more than 7 parallel I/O requests due to the I/O bus contention. With more than 16 parallel I/O requests, we actually get a slowdown compared to the performance with just one I/O request. Figure 2 also shows that the DOALL performance of the example loop degrades slightly with large number of parallel threads. *Therefore, to effectively parallelize hybrid loops, we must develop techniques for reducing I/O bus contention.*

2.1 DOALL Parallelization of Hybrid Loops

To apply DOALL parallelization to a hybrid loop we need to break the cross-iteration dependences introduced by the I/O operations in the loop. Our dependence-breaking strategies for input operations differ from those for output operations. Thus, we discuss them separately below.

Cross-iteration dependences caused by the input operations. These dependences arise because the starting file position for an iteration depends on the input operations performed in the previous iteration. The DOALL parallelization assigns each parallel thread a chunk of consecutive iterations for execution. Therefore to break the dependences we identify the starting file position corresponding to each parallel thread by directly calculating it before the execution of the loop. The method for computing the starting file positions depends upon the file access pattern used in the loop. Our examination of `Velvet` [30] and the programs in two benchmark suites, SPEC CPU 2000 and PARSEC [3], has identified three commonly-used file access patterns which are described next.

FSB: Fixed Size Blocks. The first access pattern, shown in Figure 4(a), is called the fixed-size blocks pattern as in this pattern each loop iteration reads a fixed-size block of data. The stride of the file pointer is equal to the size of the data block read by each iteration. Since we know the stride of the file pointer before entering the loop, we can easily calculate the size of data to be read by each parallel thread. Thus the starting file position of a parallel thread can be calculated by summing up the size of data to be read by previous parallel threads. Given the starting file position of each parallel thread, the time complexity of moving the file pointer to the starting file position via a *seek* operation is O(1).

FLS: Fixed Loop Size. In the second pattern, as shown in Figure 4(b), the loop first reads the total number of blocks from the file and then accesses one delimited block during each loop iteration. Since the blocks are of variable size, delimiters are used to separate them (e.g., in a text file, the delimiter is '\n'). From the total number of blocks, we can calculate the number of blocks, n, to be read by each parallel thread. Using a *scan* operation, we can locate the starting file position of parallel thread i by skipping $n * i$ occurrences of the delimiter. The time complexity is O(N), where N is the file size. This strategy is slow because it requires a *scan* as opposed to a *seek*.

FCS: Fixed Cumulative Size. In the third pattern, shown in Figure 4(c), each loop iteration accesses one delimited block that is encountered after skipping data blocks whose cumulative size reaches a given number. Because the data blocks are of variable size, delimiters are used to separate them. Since we know the total size of data to be read by the loop and the total number of parallel threads (T), we can locate the starting file position of parallel thread i by first skipping the i/T fraction of the data and then looking for the first occurrence of the delimiter. In this case, we actually assign equal amount of data instead of equal number of iterations to each parallel thread, which is different from typical DOALL parallelization. This strategy requires two operations—a *seek* and a *scan*. This *scan* is faster than the scan performed in *FLS* since it only scans to the first occurrence of the delimiter. The time complexity of this strategy is O(L), where L is the maximum length of a data block. This time complexity is much lower than that of assigning equal number of iterations to every thread—O(N), where N is the file size.
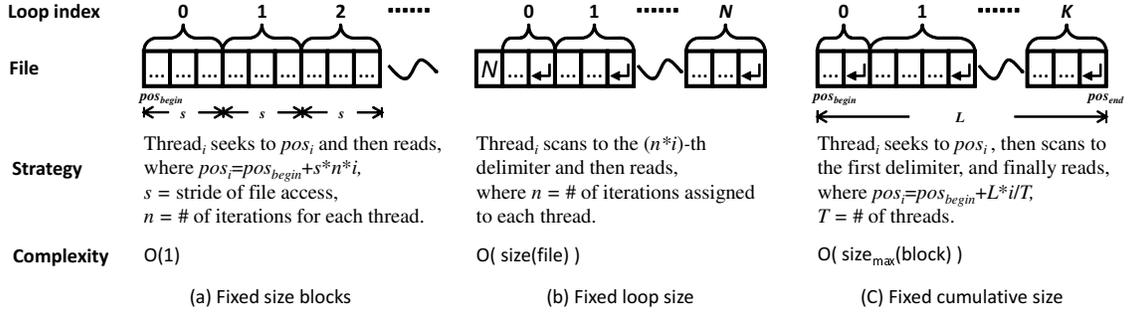
Cross-iteration dependences caused by the output operations. Simply computing the starting file position cannot help break cross-iteration dependences caused by output operations. This is because the calculated file position does not exist until all previous output operations have completed. Therefore, to break these dependences, we propose a different approach. We create an output buffer for each parallel thread. Each thread writes the outputs into its output buffer during the parallel execution of the loop. As a result, the output operations in one thread no longer depend on those in other threads. Flushing these buffers can be performed in parallel with the sequential code following the loop, as shown in Figure 5(a).

2.2 Speculative Parallelization of Hybrid Loops

Speculative parallelization is a way to efficiently exploit potential, but not guaranteed, parallelism in loops. Software speculative execution of non-I/O code has been studied in previous work [9, 26]. However, I/O operations cannot be executed speculatively in the same way because they use system calls. The code executed by the system calls is hidden from the compiler and runtime library, which thus cannot monitor the execution of system calls. Moreover, the results of I/O operations cannot be simply reversed once they are done. Therefore, to efficiently parallelize loops with I/O operations using speculative parallelism, we need support for speculative execution of I/O operations.

Speculative execution of input operations. Speculative execution of the input operations in each iteration is enabled by creating a copy of the file pointer at the start of each iteration and then using the copy to perform all input operations in the iteration. If speculation succeeds, the original file pointer is discarded and the copy is used in the subsequent iterations. If speculation fails, we simply discard the copy. One way of creating the copy is to instantiate a new file pointer and then *seek* to the current file position.
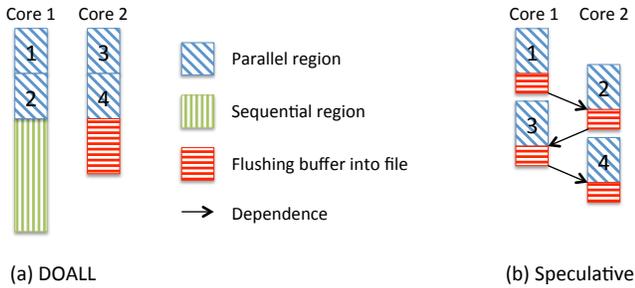
Speculative execution of output operations. Speculative execution of a loop iteration that contains output operations should satisfy the atomicity semantics, i.e., either all output operations occur or none occur. Therefore, the output operations in the loop iteration should not actually write to the file since that cannot be

Loop index	0　　1　　2　……	0　　1　……　N	0　　1　……　K

Strategy	Thread$_i$ seeks to pos_i and then reads, where $pos_i=pos_{begin}+s*n*i$, s = stride of file access, n = # of iterations for each thread.	Thread$_i$ scans to the $(n*i)$-th delimiter and then reads, where n = # of iterations assigned to each thread.	Thread$_i$ seeks to pos_i, then scans to the first delimiter, and finally reads, where $pos_i=pos_{begin}+L*i/T$, T = # of threads.
Complexity	O(1)	O(size(file))	O($size_{max}$(block))
	(a) Fixed size blocks	(b) Fixed loop size	(C) Fixed cumulative size

Figure 4. Commonly used file access patterns of loops and strategies for locating the starting file position for each parallel thread.

Programming constructs	Description
#pragma SpiceC parallel doall|doacross|pipelining	specify a parallel loop and the form of parallelism in which the loop is to be executed
#pragma SpiceC subregion [regionname] [after(iteration, region)]	specify a subregion in the parallel loop that is executed under the order specification
#pragma SpiceC commit [atomicity] [after(iteration, region)]	perform a commit operation with specified atomicity check and execution order

Table 1. SpiceC programming constructs.

Figure 5. Strategies for flushing the output buffer.

reversed. Speculative execution of the output operations in an iteration can be realized by using the output buffering described in the previous section. If speculation succeeds, we keep the buffered output. If speculation fails, we simply discard the buffered output. We flush the output buffer at the end of each iteration as shown in Figure 5(b). Flushing the buffer at the end of each iteration requires small amount of memory since the buffer does not need to hold the outputs from the entire loop, but it incurs synchronization overhead because the flush operations need to be performed sequentially. The flush operation in an iteration needs to wait for the completion of the flush operation in the previous iteration as shown in Figure 5(b).

2.3   I/O Contention Reduction via Helper Threading

Parallelizing hybrid loops can lead to increased contention on the I/O bus. We propose the use of helper threading to reduce this I/O contention. Before entering a hybrid loop, we create a helper thread which monitors the file buffers of the associated file pointers and performs the following tasks. For an input file pointer, the helper thread refills the file buffer when the size of remaining data in the buffer is less than a predefined threshold. For an output file pointer, it flushes the file buffer when the size of buffered output is larger than a predefined threshold. The use of a helper thread actually causes the I/O requests sent to the I/O bus to be serialized. Therefore, it eliminates the slowdown caused by the bursty nature

of I/O requests. Moreover, since the parallel threads only access the buffer in the memory instead of the file on the disk, the I/O latency is also hidden by the helper thread. The helper threading can also be used in sequential loops to reduce I/O latency.

3.   Our System for Parallelizing Hybrid Loops

The strategies described to enable parallelization of hybrid loops have been incorporated into the SpiceC [10] system. We chose SpiceC because it already supports programming constructs, in the form of compiler directives, that can be used to express DOALL, DOACROSS, pipelining, and speculative parallelism. We have extended the SpiceC programming model to allow parallelization of hybrid loops. This section first presents our approach to programming parallel loops in the presence of I/O and illustrate it using several examples from real applications. Next we describe our implementation of the programming model.

3.1   Programming Parallel Hybrid Loops

Let us first consider our programming model for parallelizing hybrid loops. We show how parallel hybrid loops are programmed and describe the use of I/O helper threads to boost the I/O performance on multicores. We use the C standard I/O library (`stdio`) for illustrating our programming model. All the proposed APIs can be used for other I/O libraries, e.g., the C++ I/O library (`iostream`). The SpiceC [10] programming constructs used to express parallelism in this section are summarized in Table 1.

3.1.1   Parallelizing Loops with Input Operations

To enable DOALL parallelization of loops with input operations, we introduce the *pinput* clause:

```
#pragma SpiceC parallel doall \
        pinput(file, stride, start, end)
```

The *pinput* clause is designed to be used with the SpiceC DOALL construct. To parallelize a loop with input operations, programmers just need to insert the DOALL construct combined with the *pinput* clause. The *pinput* clause has four parameters: *file*, *stride*, *start*,

and *end*. Parameter *file* is the input file pointer that causes cross-iteration dependences. Parameter *stride* gives the stride of *file* in the loop; it can be either the size of data block read by each iteration or the delimiter between data blocks. We distinguish between them based on the parameter's type. Parameter *start* is the initial position of *file* at the beginning of the loop. By default (i.e., when this parameter is empty), the current position pointed by *file* is used as the initial position. Parameter *end* is the ending position that *file* will reach at the end of the loop. The *end* should be left empty if the ending position of *file* is unknown before entering the loop (e.g., the FLS file access pattern shown in Figure 4(b)). Given these parameters, the compiler can then calculate the starting file position for each parallel thread using the strategies described in Section 2.1.

Pattern	Pragma Example
FSB	pinput(file, size, 0, EOF)
FCS	pinput(file, delimiter, 0, EOF)
FSB	pinput(file, size)
FLS	pinput(file, delimiter)

Table 2. Examples of the *pinput* clause.

Table 2 shows four examples of the *pinput* clause covering four different file input patterns. In the first two examples, the loop reads the whole file. We use 0 as the initial file position so that the compiler knows that the file pointer starts from the beginning of the file. The use of *EOF* as the ending file position tells the compiler that the file pointer will reach the end of the file. In the last two examples, the loop reads a fixed number of data blocks from the current file position. The initial file position is not given in the *pinput* clause since the current file position is used as the initial position. The ending file position is left empty because it is unknown. The first and third examples exhibit the FSB pattern (as shown in Figure 4(a)) since they use a constant block size as the stride of the file pointer. In the second example, a delimiter is used as the stride and the total data size read by the loop can be calculated by the initial and ending file position. Therefore, it exhibits the FCS pattern (as shown in Figure 4(c)). The last example exhibits the FLS pattern (as shown in Figure 4(b)) since a delimiter is used as the stride and the loop size is fixed.

```
file=fopen("input", "r");
fscanf(file, "%d", &ntuples);
#pragma SpiceC parallel doall pinput(file, "\n") {
    for( i=0; i<ntuples; i++) {
        fgets(line, maxsize, file);
        read_line(line, &index, &x, &y);
        tuples[index] = create_tuple(x,y,0);
    }
}
```

Figure 6. An input loop of benchmark *DelaunayRefinement*.

Figure 6 shows a real example of DOALL input loop that is similar to the fourth case given in Table 2. The original input loop is from the *DelaunayRefinement* benchmark [16]. Although the computation part of this benchmark has been parallelized in various ways [14, 21], its input loop, which contains I/O, has never been parallelized. The input loop reads an array of *ntuples* tuples from the file. Each iteration reads a line from the file and then creates a tuple structure from the input. In the example, the pragmas inserted to parallelize the loop are highlighted in bold. The SpiceC DOALL construct is used to identify the parallel region and type of parallelism. The *pinput* clause is used to specify the file input pattern. Since each iteration reads one line from the file, we give "\n" as the delimiter in the *pinput* clause.

3.1.2 Parallelizing Loops with Output Operations

To parallelize a loop with output operations using DOALL parallelism, we need to buffer the output of each iteration. Programmers can achieve this by using the *boutput* clause with the SpiceC DOALL construct as follows.

```
#pragma SpiceC parallel doall \
        boutput(file, isparallel)
```

The *boutput* clause tells the compiler that the output to *file* in the loop is written into its buffer. The buffer will not be flushed until the end of the loop, as shown in Figure 5(a). The *boutput* clause has two parameters: *file* and *isparallel*. Parameter *file* is the file pointer whose output needs to be buffered. Parameter *isparallel* specifies whether buffer flushing is performed in parallel with the computation threads or in a sequential fashion.

```
#pragma SpiceC parallel doall boutput(outfile, true) {
    for (index = 0; index<length; index++) {
        nucleotide = getNucleotide(descriptor, index);
        switch (nucleotide) {
            case ADENINE:
                fprintf(outfile, "A");
                break;
            case CYTOSINE:
                fprintf(outfile, "C");
                break;
            . . .
        }
    }
}
```

Figure 7. An output loop of bioinformatics application `Velvet`.

Figure 7 shows an output loop of `Velvet` [30], a widely-used bioinformatics application (genomic sequence assembler). Although the computationally-intensive part of `Velvet` has recently been parallelized with OpenMP, its hybrid loops have not been parallelized. In the original output loop, each iteration of the loop gets a nucleotide from the descriptor and outputs a character based on the type of the nucleotide. The pragmas used to parallelize the loop are highlighted in bold. The SpiceC pragma is used to mark the parallel region and the *boutput* clause is used to buffer all outputs of *fprintf* so that the output operations do not cause any cross-iteration dependence on the file pointer. Buffer flushing is programmed to be performed in parallel with the sequential code after the loop.

The *boutput* clause can also be used to program DOACROSS loops with output operations. For DOACROSS parallelism, the buffer is flushed at the end of each iteration, as shown in Figure 5(b). Figure 8 shows the kernel of the benchmark *Parser*. Each iteration first reads a line from *stdin* and then parses the line. Because of the cross-iteration dependences in the *parse* portion of the loop, the loop cannot be parallelized using DOALL parallelism. However, since these dependences rarely manifest themselves at run-time, the loop can be parallelized using speculative DOACROSS parallelism. Because the *parse* portion of the loop calls *printf* to output the results, programmers need the *boutput* clause when applying speculative execution to the *parse* portion. In the figure, the pragmas inserted to speculatively parallelize the loop are highlighted in bold. The first SpiceC pragma is used to identify the parallel region and type of parallelism. The loop is divided into two subregions by the rest of the SpiceC pragmas. The first subregion, *READ*, performs the input operations. The *pinput* clause is used at the beginning of the loop to tell the compiler the file input pattern.

```
#pragma SpiceC parallel doacross \
    pinput(infile, "\n", 0, EOF) boutput(stdout, false) {
    for(index=0; !feof(infile); index++) {
        #pragma SpiceC subregion READ {
            fgets(line, max_line, infile); }
        #pragma SpiceC subregion PARSE {
            if ( special_command(line) ) continue;
            first_prepare_to_parse(1);
            while ( !success ) {
                /* parser code here */
                printf(" Linkage %d", index+1);
                /* parser code here */
            }
            #pragma SpiceC commit atomicity \
                after(ITER-1, PARSE)
        }
    }
}
```

Figure 8. Speculative parallelization of a kernel from *Parser*.

The compiler can then calculate the starting position of *infile* for each iteration and break the cross-iteration dependences introduced by *fgets*. The second subregion, *PARSE*, parses the input; it is executed speculatively as specified by the *commit* pragma at the end of the *PARSE* subregion. Since *printf* cannot be executed speculatively, the *boutput* clause is used with the DOACROSS construct to buffer the outputs to stdout in the loop. This enables speculative execution. Buffer flushing is performed at the end of each iteration in sequential order, as specified by parameter *isparallel* in the *boutput* clause.

3.1.3 Programming I/O Helper Threads

We use helper threading to reduce the I/O contention caused by the hybrid loop parallelization. Table 3 summarizes our API for programming I/O helper threads. Function *inithelper* is used to create a new I/O helper thread; it returns the handle of the created helper thread which can then be bound to a file pointer using the function *sethelper*. Once bound to a file pointer, the helper thread continues to monitor the buffer corresponding to that file pointer.

API	Description
inithelper()	initialize a I/O helper thread
sethelper(file, helper)	bind a helper thread to a file pointer

Table 3. APIs for programming I/O helper thread.

Figure 9 shows an example of using I/O helper threading in a parallel loop taken from *DelaunayRefinement* as shown in Figure 6. We create a helper thread and bind it to the input file pointer before entering the loop. Upon entering the parallel thread, the helper thread will automatically monitor the buffer corresponding to the file pointer copy in each parallel thread. Programmers do not need to code the binding of the helper thread to each copy of the file pointer.

I/O helper threading can also be used in sequential loops to reduce the I/O latency. Similar to the example of parallel loop, use of I/O helper thread in sequential loops is straightforward: programmers just need to call *inithelper* and *sethelper* before entering the loop.

3.2 Implementation

Figure 10 presents the overview of our implementation of the programming model. The core components of the implementation

```
file=fopen("input", "r");
fscanf(file, "%d", &ntuples);
helper = inithelper();
sethelper(file, helper);
#pragma SpiceC parallel doall pinput(file, "\n") {
    for( i=0; i<ntuples; i++) {
        fgets(line, maxsize, file);
        read_line(line, &index, &x, &y);
        tuples[index] = create_tuple(x,y,0);
    }
}
```

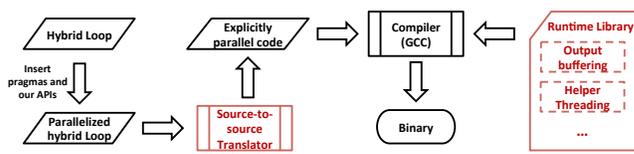**Figure 9.** Example of using I/O helper threads.

Figure 10. Implementation overview.

consist of a source-to-source translator and a user-level runtime library. The translator analyzes the hybrid loops parallelized with SpiceC directives and our APIs and translates them into explicitly parallel C/C++ code. We implemented the analysis by extending ROSE [18], a compiler infrastructure to build source-to-source code translators. The explicitly parallel code is compiled by the GCC compiler and linked with our runtime library. The runtime library implements output buffering and helper threading. Next, we describe the code transformation performed by the source-to-source translator and then elaborate how output buffering and helper threading are implemented.

3.2.1 Loop Transformation

API	Description
bwrite(index, data, size, file)	write *data* of *size* into the buffer of *file* in iteration *index*
bputs(index, string, file)	write *string* into the buffer of *file* in iteration *index*
bprintf(index, file, format, ...)	write formatted data into the buffer of *file* in iteration *index*
bflush(file, ALL/index, ispar)	flush the buffer of *file* using a separate thread or not

Table 4. Low-level functions for buffering outputs.

The code transformation from DOALL hybrid loops to C/C++ code is done automatically in our implementation. Figure 11 shows an example of the code transformation. Figure 11(a) shows a DOALL hybrid loop parallelized by our extended SpiceC directives. Each iteration of the loop reads 100 bytes from the input file, then processes them, and finally outputs them into the output file. Figure 11(b) shows the transformed main program. We insert initialization of the parallel threads at the beginning of the program and close the parallel threads at the end of the program. The DOALL loop is outlined into function *wrapper*. All variables used in the DOALL loop are wrapped into a structure which is then passed as a parameter to the outlined loop. We call function *start_doall* to execute the outlined loop in the parallel threads. Function *join_doall* is a synchronization method that waits until all parallel threads finish their work. Function *bflush* is called after the
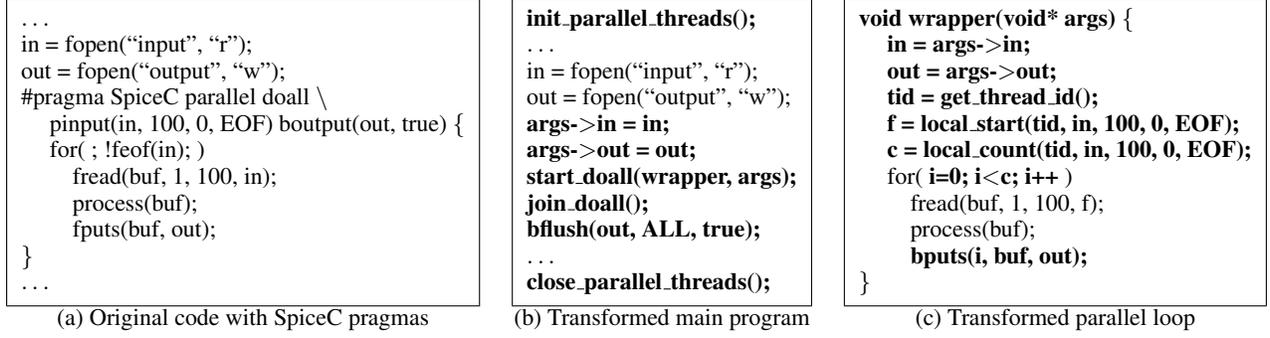
```
...
in = fopen("input", "r");
out = fopen("output", "w");
#pragma SpiceC parallel doall \
    pinput(in, 100, 0, EOF) boutput(out, true) {
    for( ; !feof(in); )
        fread(buf, 1, 100, in);
        process(buf);
        fputs(buf, out);
}
...
```
(a) Original code with SpiceC pragmas

```
init_parallel_threads();
...
in = fopen("input", "r");
out = fopen("output", "w");
args->in = in;
args->out = out;
start_doall(wrapper, args);
join_doall();
bflush(out, ALL, true);
...
close_parallel_threads();
```
(b) Transformed main program

```
void wrapper(void* args) {
    in = args->in;
    out = args->out;
    tid = get_thread_id();
    f = local_start(tid, in, 100, 0, EOF);
    c = local_count(tid, in, 100, 0, EOF);
    for( i=0; i<c; i++ )
        fread(buf, 1, 100, f);
        process(buf);
        bputs(i, buf, out);
}
```
(c) Transformed parallel loop

Figure 11. Example of code transformation.

loop to flush the buffer of file pointer *out*. Figure 11(c) shows the outlined loop. Before the loop, three functions are called to prepare the workload for the current thread. Function *get_thread_id* is used to get the ID of the current thread. Function *local_start* is called to calculate the starting file position of the current thread. Function *local_count* is called to calculate the number of iterations to be performed in the current thread. Function *fputs* is replaced with our function *bputs* for buffering the outputs. Table 4 lists our substitute for the C standard I/O functions. They are designed to buffer and flush the outputs for breaking the cross-iteration dependences introduced by the output operations.

3.2.2 Output Buffering

To enable output buffering (e.g., *bputs*, *bflush*), we create an output buffer for each parallel thread. The structure of each output buffer is a linked list, as shown in Figure 12. Each node in the linked list is a buffer of predefined length which can typically hold the output from several iterations. Once a node is full, a new node is created and appended to the linked list.

Flushing the output buffers takes $O(n)$ time, where n is the total size of all output buffers. Figure 12(a) shows the data layout of the output buffers for a DOALL loop with two parallel threads. The output buffer of thread 1 stores the output from iteration 1 to 6 and the output buffer of thread 2 stores the rest. It is straightforward to flush the output buffers with this data layout. We can flush the output buffers starting from the first thread and ending with the last thread, which takes $O(n)$ time. Figure 12(b) shows the data layout of the output buffers for a DOACROSS loop. The output buffer of thread 1 stores the output from odd iterations and the output buffer of thread 1 stores the output from even iterations. In this case, we can flush the output buffers in a round-robin manner, which also takes $O(n)$ time. The output of the current iteration in a DOACROSS loop can be flushed efficiently (as shown in Figure 8) since the output of the current iteration is always pointed to by the tail pointer of the output buffer.
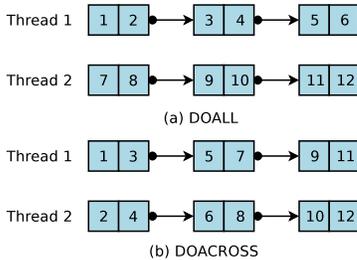

Figure 12. Buffer layout.

3.2.3 I/O Helper Threading

To enable I/O helper threading, our runtime library implements an extended version of the standard file pointer by dividing the file buffer into two parts of equal size: the f-buffer and the h-buffer, where the f-buffer is used as the file buffer directly accessed by the I/O operations and the h-buffer is the helper buffer used by the helper thread.

For input file pointers, all input operations read data from the f-buffer. Once the f-buffer is empty, it is switched with the h-buffer. The helper thread keeps monitoring the h-buffer. If the h-buffer is empty, it refills it by calling I/O system calls. Output file pointers work in a similar way.

In our implementation, we typically do not put the helper thread to sleep to minimize the refilling latency. However, when the number of parallel threads is equal to the number of processor cores in the system, the helper thread will compete with the parallel threads for CPU resources. Therefore in this case, instead of busy idling the helper thread, we put the helper thread to sleep when it does not find any buffer that needs refilling. The helper thread is then woken when an f-buffer is switched with the h-buffer in a buffer pair.

4. Evaluation

This section evaluates the prototype implementation of our programming model. The experiments were conducted on a 24-core DELL PowerEdge R905 machine. Table 5 lists the machine details.

Processors	4×6-core 64-bit AMD Opteron 8431 Processor (2.4GHz)
L1 cache	Private, 128KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 6 cores, 6144KB
Memory	32GB RAM
OS	Ubuntu server, Linux kernel version 2.6.32

Table 5. Dell PowerEdge R905 machine details.

4.1 Benchmarks

Our programming model was applied to eight applications. Two are real-world applications, while the others are from the PARSEC [3] and SPEC CPU2000 suites. We selected these applications using the following criteria: (1) the applications must have at least one hybrid loop that can be efficiently parallelized (i.e., there is no frequent cross-iteration dependence in the computation part); and (2) the hybrid loop(s) must take a significant portion of execution time. Applying our techniques on applications that do not satisfy these criteria would diminish our capacity to measure and evaluate our approach. We applied DOALL parallelism to the hybrid loops

Name	Source	Loops	Input	Output?	Speculation?	Helper?	% runtime	# stmts
velveth	real application	8	FCS	Yes	No	Yes	53%	18
velvetg	real application	18	FCS	Yes	No	Yes	27%	46
spacetyrant	real application	4	–	Yes	No	No	95%	8
DelaunayRefinement	lonestar	3	FLS	No	No	Yes	23%	7
bzip2	SPEC CPU2000	1	FSB	Yes	No	Yes	99%	13
parser	SPEC CPU2000	1	FCS	Yes	Yes	No	99%	8
blackscholes	PARSEC	1	FLS	No	No	Yes	45%	6
fluidanimate	PARSEC	3	FLS	Yes	No	Yes	36%	10

Table 6. Benchmark summary. From left to right: benchmark name, source of the benchmark, number of parallelized hybrid loops, input file access pattern, whether output buffering is used, whether speculative parallelization is used, whether helper threading is used, percentage of total execution time taken by the hybrid loops, number of statements added or modified for parallelization.

in seven applications except `parser` (speculative parallelism was required to parallelize `parser`). Table 6 shows the details of the benchmarks.

Velvet [30] is a popular genomic sequence assembler. It contains two applications—`velveth` and `velvetg`. Velveth constructs the dataset and calculates what each input sequence represents. Velvetg manipulates the de Bruijn graph that is built on the dataset. 18 computation loops in `velveth` and `velvetg` have already been parallelized using OpenMP. In the experiment, we parallelized the hybrid loops in them. All parallelized input loops have the FCS pattern. We use output buffering to parallelize the loops that contain output operations. We used nucleotide sequence *SRR027005* [1] as input. SpaceTyrant [2] is an online multi-player game server. We parallelized its *backup* thread which executes the *backupdata* function to backup game data. The *backupdata* function has 4 output loops for storing different types of data. Output buffering was used to parallelize these loops. In the experiments, we assume that every data block is dirty and needs to be written to the file. DelaunayRefinement [16] is a meshing algorithm for two-dimensional quality mesh generation, originally written in JAVA; we ported it to C++. Its computation loop has been parallelized in previous work [14]. We parallelized the three input loops in the *read* function. The three loops read different aspects of the input graph. They all have the FLS pattern. We applied DOALL parallelism to them by breaking the I/O dependences. Bzip2 is a tool used for data compression and decompression. In the experiments, we parallelized its compression loop using DOALL. There are many superfluous cross-iteration dependences on global variables in `bzip2`. To remove these dependences, we replicated buffers for each iteration and made many global variables local to each iteration. Some of the local variables are summarized into global variables after the loop. Parser is a syntactic parser for English. We used speculative parallelism to parallelize its *batch_process* function which reads and parses the sentences in the input filed. The function contains a FCS loop. We broke the I/O dependences by calculating the starting file position for each iteration and using output buffering. We need to speculate on dependences for control variables which may be altered by the special commands in the input file. Blackscholes is a computational finance application. We parallelized the input loop in the *main* function. The loop is a FLS loop that contains only input operations. Fluidanimate is designed to simulate an incompressible fluid in parallel. In its original *Parsec* version, the number of threads supplied by users must be a power of 2. We modified the workload partitioning to enable an arbitrary number of threads. For the PARSEC benchmarks used in the experiments, we use their pthread-based parallel versions.

4.2 Performance

Figure 13 shows the absolute speedup of the parallelized applications over their sequential versions for varying number of parallel threads. Figure 13(a) shows the speedup when applying our hybrid loop parallelization techniques—we achieve 3.0x–12.8x speedup. On average, we improve the performance of these applications by 6.6x on the 24-core machine. For some benchmarks, the performance degrades with 24 parallel threads. This is caused by the contention between the helper thread and parallel threads. Fluidanimate has unstable performance across varying the number of parallel threads because its workload cannot be evenly partitioned with certain numbers of threads. The performance of SpaceTyrant goes down with larger number of threads. This is caused by the dynamic memory allocation for output buffering. For comparison, Figure 13(b) shows the speedup of these parallelized applications without hybrid loop parallelization. The speedup of SpaceTyrant is always 1 since its backup thread cannot be parallelized without hybrid loop parallelization. The speedup of these applications without hybrid loop parallelization is between 2.3x–8.8x which is significantly lower than the speedups with hybrid loop parallelization.

Figure 14(a) shows the relative speedup of hybrid loops with parallelization vs. without parallelization. For seven applications, hybrid loop parallelization improves the hybrid loop performance by factors greater than 5x. On average, hybrid loop parallelization improves the loop performance by a factor of 7.54x. Figure 14(b) shows the relative parallelized full application speedups with hybrid loop parallelization vs. without hybrid loop parallelization. On average, hybrid loop parallelization improves the application performance by 68%.

4.3 Impact of Helper Threading

Figure 15 shows the impact of I/O helper threading. On average, I/O helper threading improves the performance of parallelized hybrid loops by 11.9%. I/O helper threading usually provides more benefit with larger number of threads except 24 parallel threads where the I/O parallel thread competes with the parallel threads for processing resources. Figure 16 shows the impact of the buffer size of the helper thread in two applications—`velveth` and `DelaunayRefinement`. I/O helper threading achieves higher speedup with larger buffer sizes.

The buffer size is a critical factor that determines whether a helper thread can efficiently load data for multiple threads. We use the following example to show how we set the proper buffer size for a helper thread. Figure 17 compares the computation time with the data load time for different numbers of iterations in *DelaunayRefinement*. The trend of the curves is similar for hybrid loops in other applications. We define tc_k as the computation time of k iterations and tl_k as the time of loading data for k iterations. From the figure, tc_k increases much more quickly than tl_k with the increase of k.
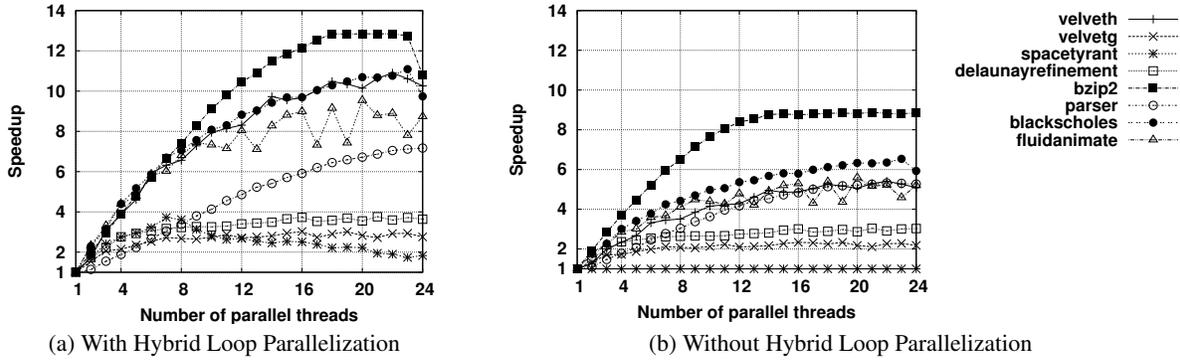
(a) With Hybrid Loop Parallelization   (b) Without Hybrid Loop Parallelization

Figure 13. Absolute parallelized application speedup over sequential programs.

(a) Parallelized hybrid loops speedup   (b) Parallelized full application speedup
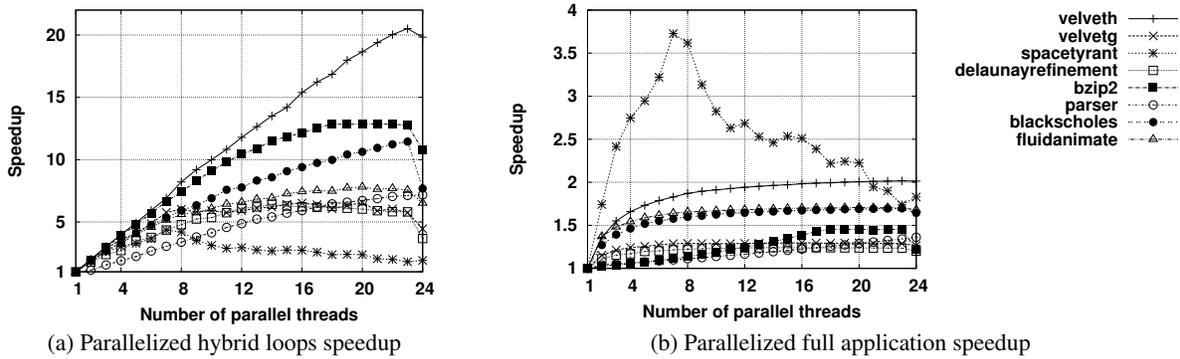
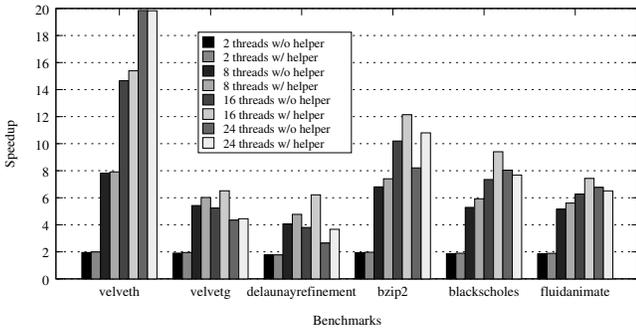**Figure 14.** Relative speedup: with hybrid loop parallelization vs. without hybrid loop parallelization.

Figure 15. Impact of helper threading.

For a helper thread that loads data for p parallel threads, its buffer for each parallel thread should be able to hold data for n iterations where $tc_n > p * tl_n$ since loading data should be finished before the data in the buffer is used up. Since tc_k increases much more quickly than tl_k, there always exists n satisfying $tc_n > p * tl_n$.

4.4 Overhead

Figure 18 shows the breakdown of the hybrid loop execution time for the parallelized applications. The time is divided into four categories: computation, I/O, speculation, and synchronization. For 5 out of 8 applications, most time is spent on the computation. `SpaceTyrant` and `Blackscholes` spend most time on I/O operations since their hybrid loops contain very little computation. Since most loops are parallelized using DOALL parallelism, very little

synchronization overhead was introduced. `Parser` has the highest synchronization overhead since it is parallelized speculatively. I/O operations take a higher percentage of execution time with larger number of parallel threads due to I/O bus contention. Figure 19 shows the memory overhead incurred by hybrid loop parallelization. For 5 out of 8 benchmarks, the memory overhead is smaller than 10MB. `Velveth` and `velvetg` have high memory overhead since their output, which needs to be buffered in the memory during loop execution, is large.

5. Related Work

Parallel programming models. Many programming models have been proposed to enable exploitation of data parallelism in sequential programs on shared memory multiprocessors. OpenMP [8] is a widely used programming model that provides a set of compiler directives for parallelizing sequential programs on shared-memory systems. Threading Building Blocks (TBB) [20] is a programming model that provides a set of thread-safe containers and algorithms for expressing parallelism on shared-memory systems. Single Program Multiple Data (SPMD) is another category of programming models and the message passing interface (MPI) [11] is currently the de facto standard for SPMD. Partitioned global address space (PGAS) [6, 7] is a set of parallel programming models which aim to combine the performance advantage of MPI with the programmability of a shared-memory model. Galois [14, 15] introduces a programming model to exploit the data parallelism in irregular applications. SpiceC [10] is a recently proposed parallel programming model for both multicores and manycores. SpiceC can be used to express multiple forms of parallelisms, including DOALL, DOACROSS, pipelining and speculative parallelism. Fi-
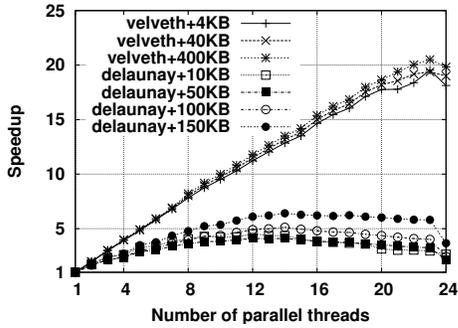
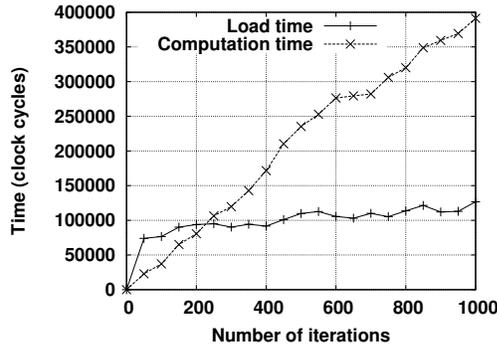**Figure 16.** Speedup by varying buffer size.

Figure 17. Computation time vs. data load time of benchmark *DelaunayRefinement*.
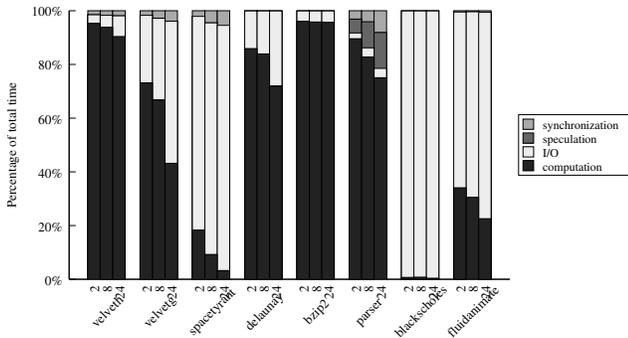

Figure 18. Breakdown of hybrid loop execution time.

nally, software-based thread level speculation (TLS) techniques have been proposed for automatically parallelizing sequential programs [9, 13, 26–28]. They are all based on state separation, i.e., the results of speculative computations are stored in a separate space from the non-speculative state space. Speculative Decoupled Software Pipelining (Spec-DSWP) [29] is another series of TLS works. Software multithreaded transactional memory system [19] has been developed to optimize the performance of Spec-DSWP.

None of the above programming models and techniques provide support for efficiently parallelizing hybrid loops. They cannot break the cross-iteration dependences caused by I/O operations. Therefore, they cannot perform the I/O part of loops in parallel. In addition, the TLS techniques that focus on speculative execution of data computation do not provide any rollback mechanism for I/O operations in case of misspeculation.
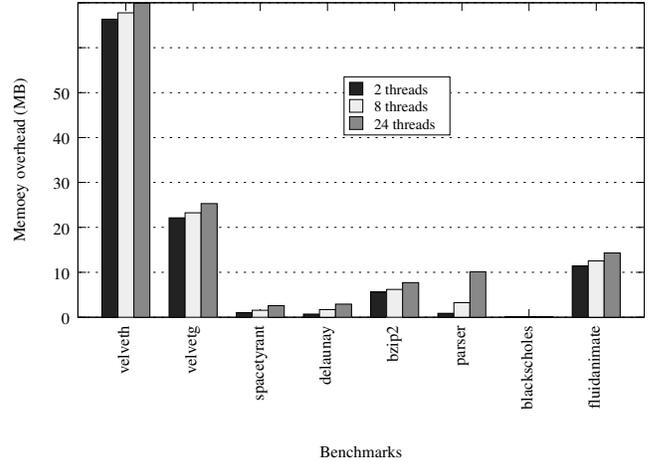

Figure 19. Memory overhead.

Parallel I/O. Parallel I/O has been proposed to improve the performance of multiple I/O operations at the same time. It is mostly designed to deal with massive amounts of data on distributed systems. Research work in parallel I/O can be mainly divided into two different groups: parallel file systems and parallel I/O libraries. Parallel file systems [22] usually spread data over multiple servers for high performance. They allow shared accesses to files from multiple processes. Parallel I/O libraries such as ROMIO [24] are APIs designed to access parallel file systems. Collective I/O [25] has been proposed to optimize non-contiguous I/O requests from multiple processes; it coordinates accesses to files by a group of processes in which collective I/O functions are called.

Our techniques are orthogonal to parallel I/O. Parallel I/O provides lower-level programming constructs designed to improve the I/O throughput of large-scale systems. However, when using parallel I/O, programmers must be highly skilled in order to express, and make efficient use of, parallel I/O operations. We make it easy for programmers to parallelize hybrid loops by providing a higher-level programming model. Our compiler techniques are designed to support our programming model and optimize the performance of hybrid loops written in our model.

More specifically, our approach differs in two ways. First, to parallelize a loop with operations using MPI I/O, programmers must write code to calculate the starting and ending offset for each thread and explicitly set the offset using the MPI I/O APIs. Programmers also need to take care of synchronization, scheduling, load balancing, etc. Using our programming model, programmers just need to insert a few pragmas. Second, Parallel I/O, such as MPI I/O, does not provide any support for speculative parallelization, while our techniques do.

I/O support for transactional memory. Unrestricted transactional memory [4] is a hardware transactional memory technique that has been proposed to support I/O calls in transactions. Transactions usually cannot contain I/O calls because these operations cannot easily be rolled back. Unrestricted transactional memory gives up some concurrency in exchange for gaining the ability to perform I/O calls within transactions by allowing only a single overflowed transaction per application.

I/O prefetching. Helper threading has been used in software-guided prefetching to hide I/O latency [5, 23]. To minimize I/O latency, these techniques require timely prefetching. They rely on the profiler or operating system to insert prefetching calls. Our helper threading technique is designed to reduce contention on the I/O bus instead of hiding the I/O latency. Therefore, we only require

data residing in main memories (i.e., off-chip memories) instead of caches (i.e., on-chip memories) when they are read. Since main memories have very large capacity nowadays, we do not require very timely prefetching. Moreover, our helper threading is specially designed for loops with contiguous I/O accesses. Therefore, we do not require any support from a profiler or the operating system.

6. Conclusions

In this paper, we identified the opportunity to parallelize hybrid loops, i.e., loops with computation and I/O operations. We presented several techniques for efficiently parallelizing hybrid loops. We proposed an easy-to-use programming model for exploiting parallelism in hybrid loops. Parallelizing hybrid loops using our model requires few modifications to the code. We developed a prototype implementation of our programming model. The implementation was evaluated on a 24-core machine using eight applications, from PARSEC and SPEC CPU2000 benchmark suites, and real-world applications. The applications with hybrid loop parallelization achieve 3.0x–12.8x speedup while in comparison 2.3x–8.8x speedup was observed without hybrid loop parallelization.

Acknowledgments

References

[1] DDBJ sequence read archive. http://trace.ddbj.nig.ac.jp/dra/index_e.shtml.

[2] Space tyrant. http://spacetyrant.com/st.c.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, 2006.

[5] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19:111–170, May 2001.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 519–538, 2005.

[7] U. Consortium. UPC language specifications, v1.2. *Berkeley Lab Technical Report LBNL-59208*, 2005.

[8] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE computational science & engineering*, 5(1):46–55, 1998.

[9] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 223–234, 2007.

[10] M. Feng, R. Gupta, and Y. Hu. SpiceC: scalable parallelism via implicit copying and explicit commit. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 69–80, 2011.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1994.

[12] J. L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *Computer*, 33:28–35, July 2000.

[13] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 157–168, 2009.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 211–222, 2007.

[15] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 233–243, 2008.

[16] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, 2009.

[17] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36:1425–1439, 1987.

[18] D. Quinlan. Rose: Compiler support for object-oriented framework. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC)*, 2000.

[19] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–76, 2010.

[20] J. Reinders. *Intel threading building blocks*. O'Reilly Media, 2007.

[21] M. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2007.

[22] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[23] S. W. Son, S. P. Muralidhara, O. Ozturk, M. Kandemir, I. Kolcu, and M. Karakoy. Profiler and compiler assisted adaptive I/O prefetching for shared storage caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 112–121, 2008.

[24] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS)*, pages 180–187, 1996.

[25] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS)*, pages 182–191, 1999.

[26] C. Tian, M. Feng, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–341, 2008.

[27] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 62–73, 2010.

[28] C. Tian, C. Lin, M. Feng, and R. Gupta. Enhanced speculative parallelization via incremental recovery. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 189–200, 2011.

[29] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 49–59, 2007.

[30] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18:821–829, 2008.