Automated Enhanced Parallelization of Sequential C to Parallel OpenMP

Dheeraj D.¹, Shruti Ramesh², Nitish B.³
Dept. of ISE, PES Institute of Technology
100-ft Ring Road, BSK III Stage, Bangalore, India

Abstract - The paper presents the work towards implementation of a technique to enhance parallel execution of auto-generated OpenMP programs by considering the architecture of on-chip cache memory, thereby achieving higher performance. It avoids false-sharing in 'for-loops' by generating OpenMP code for dynamically scheduling chunks by placing each core’s data cache line size apart. It has been found that most of the parallelization tools do not deal with significant issues associated with multicore such as false-sharing, which can degrade performance. An open-source parallelization tool called Par4All (Parallel for All), which internally makes use of PIPS (Parallelization Infrastructure for Parallel Systems) - PoCC (Polyhedral Compiler Collection) integration has been analyzed and its power has been unleashed to achieve maximum hardware utilization. The work is focused only on optimizing parallelization of for-loops, since loops are the most time consuming parts of code. The performance of the generated OpenMP programs have been analyzed on different architectures using Intel® VTune™ Performance Analyzer. Some of the computationally intensive programs from PolyBench have been tested with different data sets and the results obtained reveal that the OpenMP codes generated by the enhanced technique have resulted in considerable speedup. The deliverables include automation tool, test cases, corresponding OpenMP programs and performance analysis reports.

Keywords - OpenMP, Par4All, PIPS, PoCC, Polyhedral Model, Cache-Line Size, On-Chip Cache Memory

I. INTRODUCTION

Most of the existing parallelization techniques rely on manually identifying regions to parallelize. The hardware support for parallelism can be exploited only if the compiler has support for generating the parallel code. There are API’s like OpenMP for developing parallel applications. But the user has to manually provide annotations for it in the source code and must also ensure correctness of the code, which is a tedious process. Hence new mechanisms as in [1], [2] were developed to identify those portions of the code that can be safely executed in parallel without any human intervention.

OpenMP [3] is an API based on the simple fork-join execution model. It directs multi-threaded, shared memory parallelism in an explicit manner. It supports C/C++ and FORTRAN on multiple architectures comprising of a set of directives recognizable by the compiler, environment variables that affect the run-time behavior and a few library routines as well. We know that loops consume most of the execution time. If possible, a loop is split up by a parallelizing compiler such that separate processors can execute the iterations concurrently. This is followed by a dependency analysis pass on the code which checks whether it can be safely parallelized. Identifying the opportunities for parallelization and applying transformations automatically as in [1], [2] is a tedious task for existing compilers. A powerful mathematical model aids the compilers to perform such transformations with some restrictions specified on the input.

Geometrical representation for programs is the polyhedral model [1], [2], [4], where all analyses and high-level transformations are based on Linear Algebra and Linear Programming. After transformation, efficient code is generated, replacing original loop structure. In this model, the program is transformed into an algebraic representation which can be used to detect data dependencies [5], [6]. This representation is then converted in such a way that the degree of parallelism is improved.

The scope of polyhedral model is based on static control parts (SCoP) [1], [7]. A regular static control part is a consecutive set of statements with only 'for loops', where loop bounds, ‘if’ statement conditionals and array accesses are affine functions [8] of iterators and global parameters.

II. POLYHEDRAL TOOLS

A. PIPS

PIPS [7], [9] (Programming Integrated Parallel Systems or Parallelization Infrastructure for Parallel Systems) is a source-to-source compilation framework used to analyze and transform C and FORTRAN programs. It implements interprocedural polyhedral analyses and transformations and generates parallelized code for various architectures such as MMX, SSE, CUDA, etc. It also generates OpenMP code which can then be transformed to MPI, if desired.

B. PoCC

PoCC [7] (Polyhedral Compiler Collection) is a compiler which performs powerful parallelization and optimization based on the polyhedral model.
PoCC comprises of several GNU tools for polyhedral compilation such as:

- **Clan**: The Chunky loop analyzer is used to extract the polyhedral intermediate representation from the source code.

- **Candl**: The Chunky analyzer for dependences in loops is used to calculate the polyhedral dependences from the polyhedral intermediate representation.

- **CLooG**: The Chunky Loop Generator is the generator of the parallel code in the polyhedral model.

- **LetSee**: The Legal transformation Space explorator is mainly used for computing and exploring the legal affine scheduling of a program.

- **PLuTo**: Pluto automatically parallelizes and also performs the locality optimization for multicores. It uses the polyhedral model internally.

**C. PIPS-PoCC Interface**

Firstly PIPS parses and analyzes the applications, it then extracts static control parts and outlines them. The new SCoP modules are generated with prefix ‘SCoP_’, which are then locally optimized and parallelized by PoCC compiler. Finally, PIPS inlines ScoPs and gives back the original structure to user’s application. Thus the integration of PIPS and PoCC as in [7] automates the parallelization and local optimizations in the polyhedral model. One major application is found in the two popular automatic compilers for the C language, terapyps for an FPGA based embedded processor and p4a for NVIDIA GPU, which internally make use of the PIPS-PoCC integration as an optimization phase as depicted in figure 1 [7].

**D. Par4All**

Par4All [10] is a source-to-source compiler for serial C and FORTRAN programs, developed by HPC Project, Mines ParisTech and Institut Telecom, which automatically parallelizes and also optimizes them. It is a platform which merges various open source developments and eases the migration of sequential software not only to multicores and high end PC’s, but also to GPU’s. It creates an entirely new source code, thus allowing the original code to remain unmodified. The generated sources only need to be processed through usual compilers. It is the source-to-source aspect that makes Par4All defacto interoperable with other tools such as front-end or back-end compilers to build complete tool chains. The global architecture of Par4All is presented in figure 2 [10]. From the user’s high level point of view, the source code gets processed as follows:

- The source files pass through the preprocessor of pips. C source files also pass through p4a_recover_includes to instrument the #include processing for later recovering;

- The preprocessed files pass through a splitter that creates one file per function and a compilation unit file that keeps track of all the file-global declarations;

- Each function file or compilation-unit file can be parsed on demand according to the pyps script;

- A predefined pyps program applies many different pips phases on the code and regenerates the transformed sources;
p4a_recover_includes is applied again as a post-processor to recover most of the #include work;

The sources are post processed by p4a_post_processor.py to cope with some of the special syntax (cuda, OpenCL, etc.) that can not be directly represented in pips;

The generated final files are copied to the final directory;

If requested, the final target back-end compilers are called to produce parallelized executable program.

Figure 2 Intestinal View of P4A

III. PROBLEMS ASSOCIATED WITH MULTICORE

This section briefly describes the concept of cache line size which has significant impact on the performance, followed by the problems associated with multicore such as cache-coherency and false-sharing.

A. Cache Line Size

Cache [11] is a small segment of very fast memory which is used to hold recently used data and instructions and can greatly accelerate the calculations which repeatedly access the same memory regions over a small time interval. Cache-lines [11] are aligned chunks of caches which facilitate movement of data into and out of the cache. They may be 64/128 bytes wide. Some principles of good cache use [12] are:

- A program must utilize all words present in each cache line that it visits, thus avoiding unnecessary wasting of time in loading unused parts of the line.
- A program must use a cache line extensively and should not return to it later. This is because whenever a program returns to a cache line for the second time after a delay, the cache line might have been displaced by other data, resulting in the program being delayed twice while waiting for the same data.

B. Cache Coherence

A value written to memory by the program first enters the cache of the core which executes the code. If other caches contain a copy of that cache line, their copy is invalidated indicating they cannot be used. The new value is then written to main memory, and other caches will have to read it again from there whenever they need it. This kind of synchronization is already implemented at the hardware level, however it is time costly. Modern processors implement it through MESI[13] protocol, which indicates cache line can be in one of these states: Modified, Exclusive, Shared and Invalid. "Modified" state is entered whenever any data in a cache line is changed by a core and other caches that contain a copy of same cache line are made to transit to "Invalid". Whenever the other cores need the data, they have to read it from main memory.

C. False Sharing

False sharing [13]-[18] i.e. cache-line ping-ponging is mainly caused due to repeated invalidations of the caches of other cores by one or more cores, even when accessing isolated state i.e. in a shared cache line there occur writes, but at different locations. Due to this, other cores cannot read data from their local caches and are forced to fetch data from main memory, which considerably slows them down. Let us assume different variables are being used by two different threads operating on two different cores, which appears to be embarrassingly parallel since isolated data is being used by different threads. If the same cache line contains the two variables with at least one variable being modified, then it would result in contention for that cache line. This situation leads to false sharing and is named so because unintentionally different threads are sharing a cache line although they are not sharing any data.
0 and a small portion is altered. Thread 1 in core 1 also wants to modify the same cache line, but a different portion of it. The modification of cache line by core 0 is detected by the hardware coherency logic, because of which the core 0’s L1 cache line must be sent to L2 cache, and before the update gets completed it must be loaded into the L1 cache of core 1. In a multiprocessor system, there could even be false sharing between cores which are not sharing the cache at all. E.g., in Figure 4 [15], the core 0 threads may suffer from false sharing problem with the core 2 threads.

Figure 4 Shared Cache - A Dual-Core, Dual-Processor system

False-sharing can be avoided by allocating different cache lines with non-shared data as in [15]. Another technique is that a local function variable is made to hold a copy of the global variable, and then before the function terminates, the data is copied back. During the process, the data structures need to be grouped together and size of the data elements must be kept less than or equal to some multiple of cache-line size, thus avoiding wastage of the cache resource. Also by moving the threads to the same core, false sharing can be avoided.

IV. PERFORMANCE TUNING FOR MULTICORE

Effective loop scheduling and partitioning as in [16], [17] are required to achieve good load balancing and optimal performance in a multithreaded application. To balance workload among cores, OpenMP provides 4 scheduling schemes: static, dynamic, runtime and guided. The OpenMP ‘fork’ construct conveys loop scheduling and partitioning information to compiler and runtime library via schedule (kind [, chunksize]) clause, which causes loop iterations to be better partitioned and distributed across the threads, and hence the cores; thus resulting in optimal load balancing.

One way to avoid false sharing is via cache-line alignment [16], which is splitting up of loops into larger chunks which are at least cache-line size apart. This will reduce overlapping memory and improve cache efficiency. The frequency of a thread retrieving work from the work queue and the overhead of accessing work queue increases as the chunk size shrinks which affects performance. The cache-line alignment can be achieved using the schedule clause in OpenMP.

Syntax: schedule (static | dynamic | guided | auto [, chunk])

- static [, chunk] - The iterations are distributed among the threads in a round-robin fashion, in blocks of size called "chunk". If "chunk" is not specified, each thread would execute N/P chunks approximately, where N is the loop length and P is the number of threads. Sometimes in the same parallel region, the iterations would be assigned to threads equally spread over multiple loops. However it is only valid under certain conditions. This method has the lowest overhead and least optimal workload distribution.

- dynamic [, chunk] - Here, the chunks are handled using the first-come, first-serve scheme with the default chunk size set to 1. This dynamically assigns chunks to threads so that once a thread completes, it begins with the next portion of work. This technique incurs highest overhead with optimal workload distribution.

- guided [, chunk] - This behaves just like "dynamic" scheduling, with the size of the portion of work exponentially getting reduced. The technique attempts to seek balance between overhead and workload optimization.

- auto - In this case, the compiler or runtime system decides the best scheme. The choice could also be based on the implementation.

- runtime - This involves setting of the scheduling scheme at runtime, which is done via the OMP_SCHEDULE environment variable. This method provides flexibility to the end-user to dynamically select the type of scheduling, which by default is set to static.

By default, static-even scheduling scheme as in [16] is used by an OpenMP worksharing or parallel for loop. This minimizes the chances of memory conflicts which arise when the same memory is being accessed by more than one processor. This technique is advantageous since loops generally touch memory sequentially. Also splitting up the loop into large chunks would reduce memory overlapping, thus providing a reasonable chance for good processor cache efficiency.

In the absence of schedule clause, iterations are divided into number of chunks of similar size which is equal to the number of threads present within the OpenMP team, which
provide more work to first chunk than subsequent chunks. In order to tune the application to handle the situations where variable amounts of work is assigned to each iteration or situations where certain cores are faster than others, dynamic and guided scheduling mechanisms are used.

Chunk-size is an optional parameter, which when specified, has to be a positive integer constant that would be invariant inside the loop or it could be an integer expression. It has to be chosen w.r.t. cache-line size whenever updates are needed. This allows a single thread to use the entire cache-line, if possible. The false sharing problem is avoided once threads are operating at least cache-line size apart as in [17]. Let us consider the following example:

```c
float a[1000], b[1000];
#pragma omp parallel for schedule(dynamic,8)
for(i=0; i<1000; i++) {
    a[i] = a[i] * b[i] * expf(i);
}
```

Let us assume there is a dual-core processor system with 64 bytes cache line size. In the above example, two array sections (also called as chunks) could be located in the same cache line since the size of the chunk has been specified as 8 in the schedule clause. This means that 32 bytes per cache line are taken by each chunk of the array, thus leading to two chunks being placed in the same cache line. This would cause many cache-line invalidations because two threads can read/write two chunks simultaneously, though they are not operating on the same chunk. This situation leads to false-sharing since it is not necessary for the two threads to actually share the same cache line.

One simple tuning method to eliminate the false-sharing is to use schedule(dynamic,16), so that one chunk takes entire cache line. Thus false-sharing can be eliminated by setting the chunk size so that it is aware of cache line size, thereby significantly improving application performance.

V. OPTIMIZING PARALLELIZATION

A. Algorithm

Step 1: Input Source C file.

Step 2: Identify Static Control Parts (SCoP’s) i.e. ‘for’ loops in the source file

Step 3: Conduct Data Dependency Analysis on the ‘for’ loops.

Step 4: Omit the ‘for’ loops which are Data Dependent or which involve file I/O.

Step 5: Create a separate SCoP Module for each identified SCoP.

Step 6: Parallelize each SCoP Module as follows:

Step 6.1: Add the parallel control structure ‘#pragma omp parallel for’

Step 6.2: If nested loop present, then add the ‘private’ clause.

Step 6.3: If Data Race detected, then add ‘reduction’ clause, which is a synchronization construct.

Step 7: Optimize the Parallelized SCoP Modules

Step 7.1: Fetch the Cache Line Size from the location –/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size.

Step 7.2: Determine the Data-Type of the array.

Step 7.3: Set chunk size as Cache_Line_Size / Size_of(data-type), so that one chunk takes atleast one entire cache line. This avoids false sharing.

Step 7.4: Divide the iterations among the threads by adding the worksharing construct i.e. ‘schedule(dynamic,chunk_size)’ clause where chunk_size is calculated from step 7.3.

Step 8: Inline the SCoP Modules and the required parallel OpenMP program is obtained.

B. Data Flow Diagrams

In the optimization phase, auto-generated parallel OpenMP programs are considered as input, the cache-line size specific to the processor architecture is fetched, the common problems associated with multicore are identified and the performance of the application is tuned by considering the on-chip cache which finally yields optimized parallel OpenMP code, with better performance. Considering the architectural constraints, an attempt has been made to avoid false-sharing in ‘for-loops’ by generating OpenMP code for dynamically scheduling chunks by placing each core’s data cache line size apart. This is achieved using schedule clause by setting chunk size with respect to the cache-line-size, thus resulting in each core’s data being cache line size apart.
VI. EXPERIMENTAL RESULTS

The OpenMP code auto-generated by the parallelization tool has been further optimized using the new technique, which avoids false sharing and results in considerable performance enhancement. The parallel OpenMP programs have been analyzed using the VTune Performance Analyzer on Intel Core 2 Duo and Intel i5 architectures, with the Polybench programs considered as the benchmark, which is a set of computationally intensive programs often used in the polyhedral community. They are benchmarks from linear algebra, data mining, and stencil computation and solver and manipulation algorithms operating on matrices. The results of performance comparison have been tabulated in tables below.

A. Performance Comparison Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Set Complexity</th>
<th>Serial Execution (s)</th>
<th>Parallel Execution (s)</th>
<th>Optimized Execution (s)</th>
<th>Parallel Speed-up</th>
<th>Optimized Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>Optimal</td>
<td>90.283</td>
<td>94.115</td>
<td>10.734</td>
<td>-</td>
<td>1.014</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>237.328</td>
<td>237.529</td>
<td>237.529</td>
<td>1.297</td>
<td>1.297</td>
</tr>
<tr>
<td>Sysm-Exp</td>
<td>Optimal</td>
<td>68.599</td>
<td>68.599</td>
<td>68.599</td>
<td>-</td>
<td>1.313</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>210.247</td>
<td>210.247</td>
<td>210.247</td>
<td>1.313</td>
<td>1.313</td>
</tr>
<tr>
<td>Ada</td>
<td>Optimal</td>
<td>44.089</td>
<td>44.089</td>
<td>44.089</td>
<td>-</td>
<td>1.344</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>160.363</td>
<td>160.363</td>
<td>160.363</td>
<td>1.344</td>
<td>1.344</td>
</tr>
<tr>
<td>Gemm</td>
<td>Optimal</td>
<td>72.031</td>
<td>72.031</td>
<td>72.031</td>
<td>-</td>
<td>1.344</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>190.145</td>
<td>190.145</td>
<td>190.145</td>
<td>1.344</td>
<td>1.344</td>
</tr>
<tr>
<td>Gtriu87</td>
<td>Optimal</td>
<td>2.364</td>
<td>2.364</td>
<td>2.364</td>
<td>-</td>
<td>1.335</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>6.662</td>
<td>6.662</td>
<td>6.662</td>
<td>1.335</td>
<td>1.335</td>
</tr>
<tr>
<td>Gtriu01</td>
<td>Optimal</td>
<td>8.174</td>
<td>8.174</td>
<td>8.174</td>
<td>-</td>
<td>1.290</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>18.978</td>
<td>18.978</td>
<td>18.978</td>
<td>1.290</td>
<td>1.290</td>
</tr>
</tbody>
</table>

From the above two tables, it can be noticed that there is performance enhancement for most of the benchmarks after optimization. However there are some failure cases, where the optimization failed to improve the speed-up. The dashed entry corresponds to a rare case where the parallel execution itself has taken more time than the serial execution.

B. Performance Comparison Graphs

Graphs below show the performance results and speed-up attained for the above considered benchmarks upon avoiding false sharing, which have been tested on the Intel i5 architecture for large problem size. The timings are all measured in seconds.
C. Performance Analysis Screenshots

The serial, parallel and optimized programs were profiled using intel® VTune™ Performance Analyzer and the snapshots have been presented. They show the execution times for the serial, parallel and optimized programs along with the CPU usage. These results are that of symm-exp PolyBench benchmark on intel® i5 machine.
It can be observed that the parallel execution time is slightly less than the serial execution, in terms of milliseconds; but the execution time after optimization is significantly less than the execution time for parallelization, which is in terms of a few seconds. This clearly indicates the impact of false sharing on the program performance.

From the system monitor snapshots below, it is evident that during serial execution, only one core is executing, while all other cores are idle which indicates that there is load-imbalance due to uneven distribution of work-load among the cores. However during the parallel and optimized parallel executions, all the cores are executing the program in parallel and there is good load-balancing among the cores.
VII. CONCLUSION

An attempt has been made to implement a technique to improvise the parallel execution of auto-generated OpenMP programs by considering the architecture of on-chip cache memory, thereby achieving better optimization. It was found that existing parallelization tools do not deal with some of the problems associated with multicore such as false-sharing, which can significantly degrade performance of the auto-generated OpenMP code. The new technique avoids false-sharing in ‘for-loops’ by generating OpenMP code for dynamically scheduling chunks by placing each core’s data cache line size apart. An open-source parallelization tool was analyzed and its power has been unleashed to achieve maximum hardware utilization. Some of the computationally intensive programs from PolyBench have been tested on different architectures, with different data sets and the results obtained reveal that the OpenMP codes generated by the enhanced technique have resulted in considerable speedup.

VIII. FUTURE ENHANCEMENTS

Whenever adjacent parallel loops exist, the application performance can be improved by further reducing threading overhead by entering a parallel region only one time and then the work can be divided within the parallel region. Additional constructs to handle non-loop code e.g. a work-sharing section can be considered. Current techniques only focus on parallelizing for-loops and support only OpenMP 2.5. However by implementing the sections construct, parallelization of independent basic blocks can be achieved. Also the task construct introduced newly in OpenMP 3.0 helps parallelize recursive functions, supports while-loop parallelization and deals with pointers as well. Current auto-parallelization tools are incapable of supporting the task directive, which if taken into account could prove to be a major add-on to the proposed optimization technique.

REFERENCES