Method of Extracting Parallelization in Very Large Applications through Automated Tool and Iterative Manual Intervention

Smita K.P, Aditi Sahasrabudhe, Vinay Vaidya
Center for Research in Engineering Sciences and Technology (CREST), KPIT Technologies, Pune, India

Abstract

Program parallelization involves multiple considerations. These include methods for data or control parallelization, target architecture, and performance scalability. Due to number of such factors, best parallelization strategy for a given sequential application often evolves iteratively. Researchers are confronted with choices of parallelization methods to achieve the best possible performance. In this paper, we share our experience in parallelizing a very large application (250K LOC) on shared memory processors. We iteratively parallelized the application by leveraging selective benefits from automatic as well as manual parallelization. We used YUCCA, an automatic parallelization tool, to generate parallelized code. Using the information generated by YUCCA, we improved the performance by modifying the parallelized code. This iterative process was continued until no further improvement was possible. We observed performance improvement of 17% compared to 5% improvement reported in the literature. The performance improvement was gained in very short time and despite the constraint of having to use only SMPs for parallelization.

Keywords: Parallelization, Optimization, Performance, Share memory processors

1 Introduction

Parallelization at software as well as hardware level is extremely important to improve performance of software significantly. Program parallelization is most fruitful when opted from start of the design phase of an application. However, in order to port legacy applications onto parallel hardware and reap its benefits, parallelization is needed after an application is designed and implemented. In order to parallelize the application code manually, high proficiency in domain of the application as well as parallelization techniques is needed. Automatic parallelization tools are a natural choice when parallelization is considered after development of application. Automatic parallelization typically focuses on select techniques and looks for specific patterns in the application that have the potential to execute in parallel. Choice of appropriate automatic parallelization tools is important to parallelize applications for specific target platforms.

In this paper, we present our work in parallelizing a very large code set consisting of more than 250K LOC (Lines of Code) for shared memory processors. In order to bring in improvement in the performance, manually parallelizing the application at algorithm level was one option. However, considering the massive size of given application, it was not feasible to use manual parallelization at algorithmic level. YUCCA (User Code Conversion Application), an automatic parallelization tool developed at our research center, supports parallelization for SMPs [1]. We used it to squeeze first level of parallelization benefits from the code. By using the intermediate results generated by YUCCA, we manually improved performance of the application on a shared memory processor. In this paper, we present the work in exploiting parallelization with respect to a given gigantic code and a thorough analysis of success and failure of all parallelization strategies that we used. We also present a unique literature review by discussing various important factors, which are considered crucial for parallelization.

2 Literature Review

In this section, we highlight some work in the area of automatic parallelization and its relevance in case of large data sets or large-scale applications.

2.1. Data Dependence Analysis

Automatic parallelization tools, which make use of static information of the code, ultimately come up with code partitions that can be dispatched on multiple processors or cores [2, 3, 4]. In order to determine partitions of the program, data dependence analysis is a must [3, 5]. Programming languages allow access to variables in various fashions including arrays, pointers, pointers of arrays, arguments to functions, parameters of functions etc. Over and above, their scope and visibility, different type qualifiers, also play an important role in program behavior. Dependency analysis also needs to understand side effects of functions called at various places in the code. Real life applications do have multiple global variables, hierarchical calls to functions, iterative and recursive functions etc. As mentioned in [6, 7], the analysis becomes crucial when the programs are irregular and unstructured. In case of irregular programs, pointers, their level of indirection and their ability to contain more
than one data points is a crucial factor [7]. Random and intuitive nature of program needs extensive data dependence analysis in unstructured programs [6]. Basis of dependency analysis is the read/update access to the variables. In [8], it is claimed that though necessary, a thorough end-to-end dependency checking may not be required for parallelization. Hence, the analysis is done on the basis of values, rather than memory.

2.2. Large Applications

Large program code sets typically have code distributed across multiple source and header files. Probabilistically, large number of LOC tends to cover variety of complex program syntaxes. The automatic parallelizing tools should be capable of handling and mapping of all these syntactical patterns accurately. The efficient use of system resources by the parallelization tools becomes an important factor especially in case of large applications [9, 10]. Applications having very large number of lines are difficult to parallelize, as they require a thorough understanding of the application as well as the domain [11, 12, 13].

Irrespective of the code size, performance benefits of parallelization depend on the inherent degree of parallelization the code supports. If the chunks into which the application is partitioned for parallelization are of small size, then switching time between these partitions increases. This in turn leads to less performance benefit.

2.3. Overhead of Parallelization

All parallelization techniques including use of APIs (like Pthreads, OpenMP, MPI, Intel TBB, etc.) add overhead on parallelized application. In all of these cases, the overhead arises from the synchronization and context switching time of these APIs. The absolute overhead time does not depend on the data size or the execution time. It depends on the number of parallel threads or processes created and how often the control switches from one to another [14, 15]. In [8], since the execution is broadly split into two paths, one is speculative execution and other is actual execution, most of the overheads are placed on the speculative path. However, as compared to the multithreaded/multiprocessing application, behavior oriented parallelization incurs additional overhead for protecting the data from unauthorized accesses by speculative paths. In order to keep the overhead minimum, the parallelization techniques believe on parallelizing the calls to the most time consuming functions, rather than parallelizing the body of that function [16].

2.4. Avenues of Parallelization

When parallelizing the source code of an application, it can be looked at from many perspectives. The most promising way is to find parallelization at the algorithm level. For example, an mp3 decoder applies same steps on left and right channel of the audio stream. These steps can be executed in parallel. Similarly, for image processing applications, if the same video stream is being used for two different applications, the applications can be run in parallel after preprocessing on the common video is done. The parallelization efforts at this level, though time consuming and difficult, typically derive more benefits than the parallelization efforts local to application code [17]. Automatic parallelization techniques focus on code sections which consume most of the application time, loops [18, 3], control paths [19, 20], etc.

2.5. Performance Improvement Depends on Size of Input Data

Parallelization tools, which make use of static analysis for dependency checking, cannot determine the size of input data. Moreover, parallelization tools focus only some part of the code / selected control paths in the code for parallelization. If the parallelized code and the input data are tightly coupled and size of input is large, then performance improvement resulting from parallelization of the code is more.

2.6. Target Architecture of Hardware on which Parallelized Code is Executed

The performance improvement because of parallelization, needless to say, is dependent on the hardware platform on which the parallelized code is executed. Number of cores/processors, processing speed, memory architecture play important roles in deciding performance gain of parallelized applications. As described in [6, 10], the shared memory architectures typically have multilevel caches, and typically have one or two levels of global memory accessible to all processors. When multiple applications execute on such hardware, with only one thread dedicated to each application, processor spends more time on memory access and gives less throughput. Data parallel applications tend to give more performance on shared memory architectures, such as RISC, GPGPUs etc. [21].

2.7. Static and Dynamic Analysis Methods

For parallelization of an application, behavior of program and the data needs to be modeled. This can be done in by analyzing the application at compile time or run time. Static analysis methods [2, 22, 14] collect information about the behavior of data and program at compile time. In order to complete the analysis for all possible behaviors of data and code, these methods tend to be more conservative and time consuming [22]. Such tools need to follow safer approach if the worst-case conditions or if the behavior of the program cannot be predicted [8]. In case of dynamic methods, the analysis is done speculatively based on the partial execution or based on the partial visibility to the input data [22]. Few analysis methods combine benefits of static and dynamic analysis, and analyze the applications in hybrid manner.
2.8. Handling I/O

In most of the algorithms, I/O operations need to be executed in the same order as they were in the sequential version. Hence, their presence poses a big challenge for parallelizing tools [7].

2.9. Library Code

Many applications use third party libraries in the applications. Absence of their source code can be a bottleneck for source code parallelizing tools. In such cases, depending on the input and output of the library APIs, worst cases need to be considered by parallelization tools. This problem gets aggravated if the libraries are using difference software languages for their implementation [7, 23].

2.10. Automatic Generation of Parallelized Code

APIs generated by the parallelization tool has to support target architecture [24]. APIs like OpenMP, Pthreads are used for parallel application executing on shared memory architectures, MPI are used for distributed memory architectures. OpenCL and CUDA provide APIs for GPGPUs. Inserting parallelization constructs at proper places in the code without altering its semantics and functional behavior is a tough task for parallelization tools [25].

3 Overview of Automatic Parallelization Tool

YUCCA, ‘User Code Conversion Application,’ is an automatic parallelization tool, which parallelizes complete projects/code sets, written in C language. YUCCA tool (earlier named as S2P tool [1]) is a source-to-source conversion tool; i.e. when a C application is given as input, YUCCA generates parallelized C application as output. The parallel code is a multithreaded code with Pthreads and OpenMP constructs inserted at relevant places. Throughout this text, words ‘code’, ‘program’ and ‘application’ are used interchangeably and they refer to the inputs and outputs of the YUCCA tool. YUCCA inserts Pthreads APIs in case of task parallelization and OpenMP APIs in case of loop parallelization. YUCCA tool consists of a compiler-like front-end that can preprocess, scan and parse application code and an intelligent back-end that performs static dependency analysis to identify parallelizable sections of code [1, 26]. YUCCA’s front end synthesizes information about application in an XML schema. YUCCA’s back end performs rigorous dependency analysis on this information. Results of these analysis methods are released to the user. The end result of dependency analysis includes a task dependency matrix (TDM) similar to static task graph in [2]. TDM is nothing but a matrix in which every code section is checked against each other for control as well as data dependencies [27]. Empty values in the matrix denote that there is no dependency in between two code sections. Non-empty values denote the dependency in the form of line number(s) on which the dependency exists between two code sections. Every such code section is called as a ‘task’. According to [1], criteria for defining boundaries of tasks is ‘first level programming constructs’ in ‘main’ function of the C application. Expressions, selection statements, control statements, iterative statements, function call sites that are immediately contained by ‘main’ function form tasks. Nested programming constructs are analyzed for dependencies; however TDM reports these dependencies only at the ‘first hierarchical level’ of main function. The information presented in TDM is further used for partitioning the code. The code insertion module in YUCCA then inserts parallelization constructs around and inside these partitions. Scheduler executing on a multicores processor [28] schedules the multithreaded application generated by YUCCA.

4 OpenMX - Application for Nano-Scale Material Simulations

The code set, which we have considered as case study is OpenMX version 3.3. The software package OpenMX (Open source package for Material eXplorer) is designed for large number of nano-scale material simulations [29]. The algorithms used in OpenMX enable researchers to study electronic, magnetic, and geometrical structures of variety of materials. Hence, the package finds applications in areas of biomaterials, magnetic materials, nano-scale conductors, carbon nanotubes etc. Using this package, researchers working in these areas can have deep understanding of various complicated and useful materials [29]. Since the package is computationally intensive, improving its performance by parallelization helps to quickly simulate properties of above mentioned materials. The application has earlier been parallelized using MPI (Message Passing Interface) on distributed memory systems using three methods. Results of parallelization can be found on [29].

The OpenMX source code consists of 250 LOC spread across 192 C files and 10 header files. Work described in [30], denotes cost for developing parallel programs in terms of Person Minutes per LOC. According to the metric specified, mean cost of developing an OpenMP application is 24.8 person minutes/LOC. To develop an application of size 250K, it would take more than 12,916 person days (assuming 8 hours per person day). The numbers indicated denote the time to develop the application rather than parallelize it. Assuming that 25% of the effort would be spent in parallelizing the application, the effort reduces to 3,229 person days. Since the metrics mentioned in the literature mentions code development by novice programmers, we would still like to reduce it by 50% to consider parallelization efforts by parallelization experts. This consideration further brings down the efforts to 1614 days. This number gives a sense of huge efforts required for manually parallelizing an application of size of 250LOC.
5 Parallelization of OpenMX using YUCCA Tool

OpenMX code contained varied syntaxes, including six level of pointer, macro definitions containing approximately 10000 words on a single statement, numerous conditional operators etc. As per YUCCA’s architecture, a binary expression is formed for each expression in the code and hence conditional operators are not handled. In order to overcome this, we changed the conditional statements to selection statements and completed the parsing of the OpenMX application through YUCCA. We could successfully parse the entire application through YUCCA with all the code complexities mentioned above.

The next step to automatic parallelization was the execution of YUCCA back-end, which actually does the conversion of sequential code to parallel code using Pthreads, based on thorough analysis of information present in the XML schema. YUCCA tool is capable of parallelizing loops and tasks selectively as well as simultaneously. In order to generate parallelized C code, YUCCA first generated Task Dependency Matrix (TDM) [1] showing information about the created tasks and the dependencies between all these tasks. After TDM generation, YUCCA released parallelized code for the entire OpenMX package. Table 1 shows comparison of execution time of sequential and YUCCA parallelized version of OpenMX application for various input sizes.

<table>
<thead>
<tr>
<th>Input File</th>
<th>Size (KB)</th>
<th>Execution Time of OpenMX application (mm:ss)</th>
<th>Percentage Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methane.dat</td>
<td>4</td>
<td>00:12</td>
<td>0%</td>
</tr>
<tr>
<td>C60.dat</td>
<td>6</td>
<td>01:00</td>
<td>0%</td>
</tr>
<tr>
<td>DIA216_D C.dat</td>
<td>17.8</td>
<td>09:57</td>
<td>-4.7%</td>
</tr>
<tr>
<td>DIA512_D C.dat</td>
<td>36.9</td>
<td>36:24</td>
<td>Memory allocation error</td>
</tr>
</tbody>
</table>

The results mentioned in this section as well as subsequent sections are recorded on experimental set up mentioned in Table 2.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel i3 processor-dual core (HT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>32-bit Ubuntu Linux 10.04</td>
</tr>
<tr>
<td>Speed</td>
<td>3.2 GHz</td>
</tr>
<tr>
<td>RAM Size</td>
<td>4 GB</td>
</tr>
</tbody>
</table>

The first and second column of table 1 shows the name and size of the files. The third column shows the execution time of the input file with OpenMX sequential version. Fourth column shows execution time of OpenMX parallel version generated by YUCCA. Fifth column shows the percentage of performance improvement of parallel version over sequential version.

By comparing results of execution of parallelized version of OpenMX with the sequential one, we could verify that both the versions are functionally equivalent. By looking at result table 1, we can see that there is no performance improvement after the parallelization. However, there is performance degradation for the input file ‘DIA216_DC.dat’ with large size among the first three files. A thorough inspection of the parallelized code leads to following crucial observations:

1. The core computation of the application is handled by only one thread, and the computation varied for different input files according to the settings we are specifying the input.dat file.
2. By default, YUCCA partitions tasks in the main function and inserts parallelization constructs around the boundaries of these tasks. In case of OpenMX code, the core computation happens at inner code level. Hence the code section which performed this core time-consuming computation is not parallelized.
3. There was an overhead because of large number of threads created.
4. For one of the input files (DIA512_DC.dat), we were not able to execute the parallelized version successfully, because the system ran out of memory. As the file size is large, memory required for the computation is more. In case of parallelized code, as there was more number of parallel tasks, there is a limitation on memory available to each thread. This limited thread-stack memory did not suffice for the computations in case of large files.

5.1 Customization of Parallelized OpenMX Code

5.1.1 Parallelization of DFT Function

To get an insight of performance of parallelized application, we profiled the OpenMX sequential code using Valgrind and found that the function ‘DFT ()’ takes around 85% of the total execution time. Since, ‘DFT’ function was at 2 levels inside main function, YUCCA created only one thread for the whole ‘DFT ()’ function. Hence, the core time consuming computation did not get parallelized. All the threads, except the thread executing DFT function, were idle most of the time. Therefore, we realized that parallelizing ‘main’ function will not give any performance benefit. Hence, we modified the YUCCA tool in such a way that the tool will parallelize the function according to user inputs. With a modified version of YUCCA tool, we got another parallelized version of OpenMX code.

5.1.2 Cosmetic Code Changes to Reduce the Size of TDM

The parallelized code was once again checked for correctness by comparing its results with the results
of serial version. Once the functional equivalence was tested, we observed that 45 tasks are created, in the parallelized code. YUCCA creates one thread per task. Because of 45 threads, we started facing the issue related to memory exhaustion. In order to fix the memory related issues resulting from more number of tasks (and thus threads), we made some cosmetic changes to the code. As shown in figure 1, we modified the application code, such that only limited numbers of tasks were created. As per YUCCA design, 5 tasks (and hence 5 threads) will be created for code in figure 1.a, and only 3 tasks (and hence 3 threads) would be created by making some superficial changes as shown in figure 1.b.

```c
{
    Selection statement#1;
    Loop#2;
    Loop#3;
    Control statement#4;
    FunctionCall#5;
}
```

Figure 1.a. Sample code snippet before changes

```c
{
    {
        Selection statement#1;
        Loop#2;
    }
    {
        Loop#3;
        Control statement#4;
    }
    {
        FunctionCall#5;
    }
}
```

Figure 1.b. Sample code snippet after superficial changes

By making similar change in the OpenMX code, less number of tasks, and hence threads, were created by YUCCA. After making these changes, the parallel version executed without any memory allocation errors. However, it did not lead to any performance improvement as shown in table 3.

### Table 3: Benchmarking of parallel execution time after DFT parallelization and cosmetic code changes

<table>
<thead>
<tr>
<th>Input File</th>
<th>Size (KB)</th>
<th>Execution Time (mm:ss)</th>
<th>Percentage Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methane.dat</td>
<td>4</td>
<td>00:12</td>
<td>0%</td>
</tr>
<tr>
<td>C60.dat</td>
<td>6</td>
<td>01:00</td>
<td>0%</td>
</tr>
<tr>
<td>DIA216_DC.dat</td>
<td>17.8</td>
<td>09:57</td>
<td>0%</td>
</tr>
<tr>
<td>DIA512_DC.dat</td>
<td>36.9</td>
<td>36:24</td>
<td>0%</td>
</tr>
</tbody>
</table>

**5.1.3. Merging Data Dependency Analysis with Value Analysis**

A deeper look at the TDM and the side effect analysis report generated by YUCCA gave us the list of variables, which were causing the dependency. In one case, only one variable was creating interdependencies between two tasks. In the dependent task, a function was invoked four times. Out of four call sites, first call site used a particular literal value of the argument and the remaining three function call sites passed another value of the argument. Inside the body of function definition, a selection statement bifurcated use of these literals to different computations. We used value analysis within the function definition and gathered that only the first call site would pose dependency problems instead of all the four call sites. Hence, in the parallelized code, we removed the synchronization constructs that were placed due to last three call sites. Removal of these constructs opened fewer opportunities for parallelization.

**5.1.4. Data Privatization**

In another case, the dependency was there due to update of an array variable. Even if two different memory locations/indices of array are accessed or updated by two tasks, YUCCA would treat it like dependency pertaining to the entire array variable, when executed in task parallelization mode. The array update was happening in all parallel tasks on different indices. Moreover, the values were not being utilized for any functional calculation. Since the array update occurred in the core computation loop of all tasks, each following task needed to wait until the completion of previous task. To avoid this situation, we applied data privatization for the array variable in all the tasks, and added code to transfer the values from temporary array variable to original array variable after the loop iteration. Also the wait and post synchronization signals were placed accordingly. Because of this modification, the computations in the parallel tasks were happening until the point where the data transfer from temporary variables to actual array variables happened. Only the tasks were waiting for some fraction of time. Therefore, we could achieve performance improvement for the parallelized version.

By doing manual optimization in YUCCA parallelized code, we achieved a performance benefit of 11 – 17%. The increment factor depends on the size of the input file. From the result table, we could see that the performance improvement is less for DIA512_DC.dat file (size: 36.9KB). This is because of two reasons. Based on the contents of the input file, different computations are performed in the application. Other reason is the size of the file. If we select files with same input settings, then the percentage of performance benefit decreased for files with larger size. In the graph shown in figure 2, performance benefit obtained for DIA512_DC.dat is less than the benefit obtained for DIA216_DC.dat. But the benefit obtained for C60.dat is more than what we obtained for DIA512_DC.dat. Here the files, DIA216_DC.dat and
DIA512_DC.dat have same settings in the input file. Hence, when there are files with same settings, then performance benefit is less for file with larger size.

![Graph showing size of input files versus percentage of performance improvement](image)

**Figure 2: Size of OpenMX input files versus percentage of performance improvement**

### 5.2. Role of Automatic Parallelization Tool for Obtaining Parallelized OpenMX Application with Improved Performance

Even if we did some manual modifications in the code for optimization, all the modifications were based on the results produced by YUCCA tool. There were no manual interventions in dependency analysis or task creation. As far as task synchronization is considered, we changed positions for some of the wait/post signals to resolve some dependencies. However, we did not add/remove any synchronization signals added by YUCCA tool. In addition, there were no dead locks/data races in the parallelized code produced by YUCCA tool. We reached this conclusion based on testing done on the parallelized code regarding functional integrity.

<table>
<thead>
<tr>
<th>Input File</th>
<th>Size (KB)</th>
<th>Execution Time of OpenMX (mm:ss)</th>
<th>Percentage Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methane.dat</td>
<td>4</td>
<td>00:12</td>
<td>0%</td>
</tr>
<tr>
<td>C60.dat</td>
<td>6</td>
<td>01:00</td>
<td>25%</td>
</tr>
<tr>
<td>DIA216_DC.dat</td>
<td>17.8</td>
<td>09:57</td>
<td>15%</td>
</tr>
<tr>
<td>DIA512_DC.dat</td>
<td>36.9</td>
<td>36:24</td>
<td>11.13%</td>
</tr>
</tbody>
</table>

Table 4: Benchmarking of parallel execution time after DFT parallelization, cosmetic code changes, value analysis, and data privatization

As mentioned in this paper, there had been past efforts of parallelization of OpenMX code. Parallelization of the code using MPI was carried out by 3 different ways [29]. For 2 processors, the performance improvement was always less than or close to 5%. By leveraging the strength of automatic parallelization tool and manual comprehensive efforts, we could get performance gain up to 17% as shown in table 4. Apart from the gain in performance improvement, we would also like to highlight the speed of parallelization. As mentioned in section II, it takes almost 12,916 person days to develop the parallel application of the size of OpenMX. As against this number, first round of parallelization using YUCCA merely took 48 hours. Further superficial changes to the application and customization of YUCCA code was completed in less than 50 days.

### 6. Conclusion and Future Work

In this paper, we have shown how to combine usage of automated tool along with selective changes to the code iteratively to achieve best possible parallelization. For manual customization, we made use of information generated by the tool itself. These efforts involved reducing numbers of threads, data privatization, and exploiting parallelization at multiple first level blocks in the code. By making such changes to the code and without even understanding the functionality of the code, we could fetch 17% improvement in the performance of the code, when executed on shared memory processor. We completed parallelization of 250 KLOC code in about 2 person months. This is estimated to 3% of the manual effort required for parallelizing such a large code.

The next challenge is to convert the lessons learnt from manual parallelization into an algorithm. This algorithm can further be used to enhance the automatic tool.

### 7. References

6. Lumsdaine, Andrew, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry, "Challenges in parallel

graph processing" Parallel Processing Letters 17, no. 1, 2007, pp. 5-20.