Computer Science and Engineering 2013, 3(3): 56-66
DOI: 10.5923/j.computer.20130303.02

GOMP Profiler: A Profiler for OpenMP Task Level Parallelism

Joseph M. Arul*, Gwang-Jie Hwang, Han-Yao Ko

Department of Computer Science and Information Engineering, Fu Jen Catholic University, Hsin Chuang Dt, New Taipei City, Taiwan

Abstract The current trend in microprocessor is to go multi-core, be it in desktops, note-books or hand-held systems. Fortunately, many major software and hardware vendors specifically defined a parallel programming API model called OpenMP. Recently, OpenMP version 3.0 included a new feature, i.e., task level parallelism, for parallelizing unstructured programs. This research focuses on using a particular instrumented library, together with the OpenMP run-time library, for a specific GNU Compiler Collection (GCC) and constructs a visualization tool for presenting the performance during task level parallelism. In order to verify the capability of the GNU OpenMP (GOMP) Profiler, Barcelona OpenMP Task Suits (BOTS) were used and analysed. Four different metrics were used in order to eliminate the unnecessary overheads and understand the behavior of BOTS. Thus, by using this profiler, a programmer can understand the code better and rearrange the code in order to avoid the extra computations involved during the parallel execution. Thus, task level parallelism can be improved greatly.

Keywords Task level parallelism, OpenMP, Profiler, Multi-core, Run-time

1. Introduction

Recent computers have been built with dual, quad, and hexa cores. Hence, it is important to compile the sequential programs into parallel programs using some APIs such as OpenMP, CUDA, or MPI. In several applications, most of the parallelism is exploited in the loop level. If the loop can be naturally parallelized, then the compiler can rearrange the code to parallelize the code. This loop level parallelism is the predominant area of parallelization in the scientific programs. This can be rather straightforward and several researches have already been done.

The other form of parallelism is that, which can be found in the non-loop level, where, each activity can be grouped as a task to be executed in parallel. So the **task level parallelism** tries to find number of tasks available in a program and all the tasks are scheduled in a queue for different processors to pick these tasks from a queue to execute them. Task level parallelism, which can be stated as unstructured or thread parallelism in different granularity, such as coarse or fine granularity, is becoming more and more important in parallelizing compilers. Task level parallelism can also be considered as another form of parallelism which can be used in recursive functions that are aimed at collections of asynchronous tasks. Even though it can be used in the place of data level parallelism, it is better suited to exploit the task level parallelism when the functions are called recursively where the return values are unpredictable. It is also suitable when there is a while-loop with an unpredictable end point and there exists a task level parallelism.

However, parallelizing an existing program almost has some overheads due to extra and unnecessary computations involved during parallelization. These overheads can be caused due to various reasons such as synchronization, communication, load-imbalance, etc. Besides, it can also reduce the speedup of a parallel program. When the speedup is much less than our expectation, profiler can point to us accurately where the overheads are and what are the reasons that cause this degradation. Hence, we can try to fix the program according to the profiler that pinpoints the overhead that occurs unnecessarily.

OpenMP task level parallelism also creates a lot of overheads in many programs. There are researchers who have discussed these overheads[1]. Olivier and Prins run programs with unbalanced task graphs and evaluate the Open MP run-time overheads[2]. Duran, Corbalan and Ayguade increase the granularity of a task by a method called Adaptive Task Cut-off[3] and evaluate the efficiency of the OpenMP task scheduling strategies[4-5]. Other people focus on the tools for measuring the overheads of OpenMP task level parallelism, such as Intel Thread Profiler, Sun's performance tool and Nanos OpenMP run-time library. However, no profiler supports profiling GCC's (GNU Compiler Collection) OpenMP run-time performance.

* Corresponding author:
arul@csie.fju.edu.tw (Joseph M. Arul)
Published online at http://journal.sapub.org/computer

Hence, we build a particular OpenMP profiler for a specific compiler GCC, and it only focuses on monitoring the latest features of OpenMP task level parallelism. For this purpose, we design an instrumentation library and modify the OpenMP run-time library to collect the performance data. Thus, it is named as GNU OpenMP (GOMP) profiler. In addition, we also construct a visualization tool for presenting these performance data. Finally, we use a collection of benchmarks with this profiler and find the overheads which are pinpointed by the features of this profiler.

The rest of the paper is organized as follows: section 2 presents the background in evaluating the performance and explores the related profilers which measure the OpenMP task level parallelism. Also in section 2, the special features of this particular profiler are explained. Section 3 describes the implementation steps of this particular profiler and the special characteristics of this profiler as well as how it differs from other profilers in detail. Section 4 presents the experimental results by presenting the capability of this particular profiler. Section 5 presents the conclusion of the whole research and how it can be improved in this area of research in the future.

2. Background and Related Research

This section will present some background information about OpenMP's two different modes of profiling programs, i.e., **instrumentation** and **sampling**. Secondly, it presents some profilers those are currently in use for performance measurements of OpenMP 3.0[6]. Finally, the special features about this particular profiler as well as how it differs from other existing profilers, including some information about how this profiler was developed will be explained.

Instrumentation is to modify the binary by inserting some sort of probes while entering and exiting a function or a method which would in turn collect performance of data in every step of the way while the program is executing. It would have some overheads because the additional instructions used for the probes would have some extra computations added to the overall performance of the program. The sampling is to periodically check the states of the system or running of programs. Thus, the sampling would have less overhead, since it does not interfere with the programs. For example, by querying cache miss counter first 10ms and it would note that there were 3 cache misses, then the following 10ms the cache miss counter could note 8 cache misses and so on. Thus, the sampling can notify the user the number of cache misses encountered each 10ms and so on. So the property of sampling is that, it can not monitor all the events when the time taken is less than the period of the time expected in our system.

There are several profilers such as, profiling OpenMP performance[7] or tracing the behavior of OpenMP program, Tuning and Analysis Utilities (TAU)[8], Intel thread profiler, OpenMP Profiler (ompP)[9-10], Nanos OpenMP run-time library[11] and Sun's performance tool[12]. There are various performance tools[13-14] where each tool has its purpose and it supports a specific platform or programming language.

TAU aims to parallelize MPI and OpenMP programming API, and it can profile the programs written by OpenMP or MPI or both of these together. The special features of TAU are the architecture compatibility and having varied analysis tools. The first feature they implemented is the general computation model to reflect the real world parallel computing architecture. The second feature of TAU provides many tools[15] for different purposes such as, discovering parallel overheads, tracing program's call-graphs and analyzing workload of distributed parallel systems.

Intel thread profiler is a plug-in to VTune Performance analyser. It supports threaded applications written by OpenMP, POSIX threads and Windows API. You can either use VTune graphical analyser tool or the tools integrated with the Visual Studio. It can identify the limitations of parallel programs and show what type of limitations, synchronization, load imbalance or thread creation/ destroying time occuring during the execution. The only drawback of this profiler is that, it is just compatible with Intel multi-core processors, but does not support any other processors developed by other vendors.

Rest of the three profilers, ompP, Nanos OpenMP run-time library and Sun's performance tool are designed to measure the programs written by OpenMP specification version 3.0. The ompP and Nanos use instrumentation library to profile, however, the Sun's performance tool uses sampling technique to measure OpenMP performance.

2.1. Features of GOMP Profiler

This particular profiler that has been implemented has the ability to detect the run-time events and special implementation events. The run-time events are meant to monitor the run-time cut-off, accessing global queue and insufficient task events. The special implementation event is the task-wait pending overhead in GOMP. These four types of events can only be captured by instrumenting GOMP run-time library. Hence, this instrumentation method will be explained in detail in the following section. The differences between this particular profiler and the others are the following: 1). OmpP cannot detect the run-time events and special implementation events. The reason is that it instruments in source code level instead of run-time library level. 2). Nanos has the ability to measure the run-time events but is unable to capture the implementation events, because they do not instrument GOMP run-time library. 3). Sun's performance tool also does not capture the implementation events, and it is not accurate because of the sampling method.

3. Implementation Method

Four main parts of the Implementation:

There are four main portions that are involved in

implementing GOMP Profiler. The major parts of this implementation are: 1) Designing data structures to store some necessary data or representation of states of a program in those data structures; 2) Designing the functionality and the prototypes of instrumentation library calls; 3) Inserting instrumentation library calls in appropriate places for instrumentation; 4) Getting data and then visualizing those data. The following paragraphs will continue to present the detailed explanations of each of the components mentioned here.

The main purpose of this profiler is to capture the states of a program. Hence, the event types for distinguishing between different states are shown in figure 1. There are some important event types which are related to task level parallelism. How the events are instrumented is shown in figure 2 above.

```
enum instr_event_type {
    THREAD_STATE,
    TASK_ENQUE,
    TASK_EXECUTION,
    INSUF_TASK,
    MASTER_WAIT,
    TASKWAIT_PENDING,
    ACCESS_GQUEUE,
    BARRIER_WAIT,
    TASK_EXE_IMMI,
    TASK_EXE_TASKWAIT,
    ACCESS_GQUEUE_TASKWAIT,
    PARALLEL_REGION
};
```

Figure 1. Instrumented Event Types

```
struct instr_event {
/* identify each event */
    long id;
/* the region be executed */
    long region;
    enum instr_event_type type;
/*identify each GOMP task*/
    long task_id;
    long parent_id;
    double start_time;
    double end_time;
};
```

Figure 2. Instrumented Events

Two types of libraries are designed for different purposes, i.e., run-time instrumentation library and user library. The first one is for the instrumentation and the second one is for the benefit of the users. The important library calls will be presented in the following paragraph.

Instrumentation in GNU OpenMP run-time library is as follows: GNU OpenMP library calls implement most of the OpenMP's behavior for the task construct whether to delay the task construct or proceed with the execution immediately and so on.

Finally, the visualization tool is implemented by sorting the collected data of profiled OpenMP program that is stored as a list of events. Qt framework is a tool developed for high-performance cross-platform applications especially for GUI designs. So, Qt framework is used to build the visualization tool for GOMP profiler and is called a profiler viewer.

3.1. Data Structures

The details of the event types are as follows:

TASK_ENQUE: Assign a task to the global task queue.

TASK_EXECUTION: Threads get tasks from the global task queue and execute them.

INSUF_TASK (insufficient task): No task in the global task queue, so some threads have no tasks to do which results in wasting time to wait for more tasks.

TASKWAIT_PENDING (pending in task-wait region): Wait for other children tasks to finish their work.

ACCESS_GQUEUE (access global queue): The time taken by accessing the global queue.

BARRIER_WAIT (barrier waiting): Wait for other threads to arrive at the synchronization point.

TASK_EXE_IMMI (task execute immediately): Instead of enqueuing a task, execute the task directly.

TASK_EXE_TASKWAIT (task execution in task-wait region): Execute the child task.

3.2. Instrumentation Library and User Library

There are two types of libraries designed for different purposes, i.e., run-time and user library. The first one is for the instrumentation and the second one is for the users to use. The run-time library has **get_event_id**, **instr_event_start**, **instr_event_end**, **instr_set, instr_profile_all_write, set_task_id**, and **set_parent_id**. The user library call **set_profiled_filename** is used to set the name of a file which stores the profiled data and let the users specify the destination file for different profiled programs. The **set_sequential_start** and **set_sequential_end** library calls let the users specify the locations of the start point and end point in the sequential code and calculate the time cost in the sequential code. After knowing the time taken off by the sequential part and the parallel part of the code, we can understand the speedup of this parallelization of a program.

3.3. Instrumenting GNU Open MP Run-time Library

The above mentioned instrumentation library is directly inserted into the OpenMP run-time library. Directly instrumenting GOMP run-time library has the following advantages: 1). It can capture the run-time events, run-time cut-off, accessing global queue and insufficient tasks; 2). Capture special implementation events i.e., task-wait pending; 3). It adds instrumentation functionality to GCC, so users can enable the instrumentation option while compiling an OpenMP program. Any OpenMP program that is compiled by this modified GCC, the users can get the

performance data after executing the OpenMP programs.

3.4. Visualization Tool

The collected data of profiled OpenMp programs are stored as a list of events which are sorted at the end of the program. Just listing them on the console would be hard to understand the behaviour of a program. Qt framework is a tool developed for high performance cross-platform applications especially for GUI design. It matches the characteristics of GCC which is an open source and cross-straight platform library. Figure 3 shows a visualization tool for profiled viewer.

Figure 4 shows an overview area. This area lists the following data: 1). profiled file name, 2). custom sequential time, 3) the custom parallel time, 4) speedup, 5) start time of the parallel region, 6) the end time of parallel region, 7) the elapsed time of the parallel region and 8) total task execution time. Speedup entry is calculated by the custom sequential time / custom parallel time. Total task execution time is the accumulation time of each task's execution time.

Figure 5 shows the table where each column represents different types of information of events, and each row represents an event at a specific time. The first column shows the type of event; second and third columns show the task ID and parent ID of an event respectively; fourth and fifth columns show the starting and ending time of an event; sixth column is the elapsed time of an event, it is defined by the "end time of an event – start time of the event"; the last column shows the exclusive time of an event and this time is calculated by the elapsed time minus the time of children events executed during this event. An example for exclusive time:- event B executed in event A, event B took 3 seconds and event A took 4 seconds. So the exclusive time of event A is 4-3 =1sec. By observing the parent ID, we can understand which events were executed during a particular event. Events that have the same parent ID means these events are the children of the same parent event. Looking at the exclusive time, we can note that the exact time taken by that event which does not include the time taken by the children events.

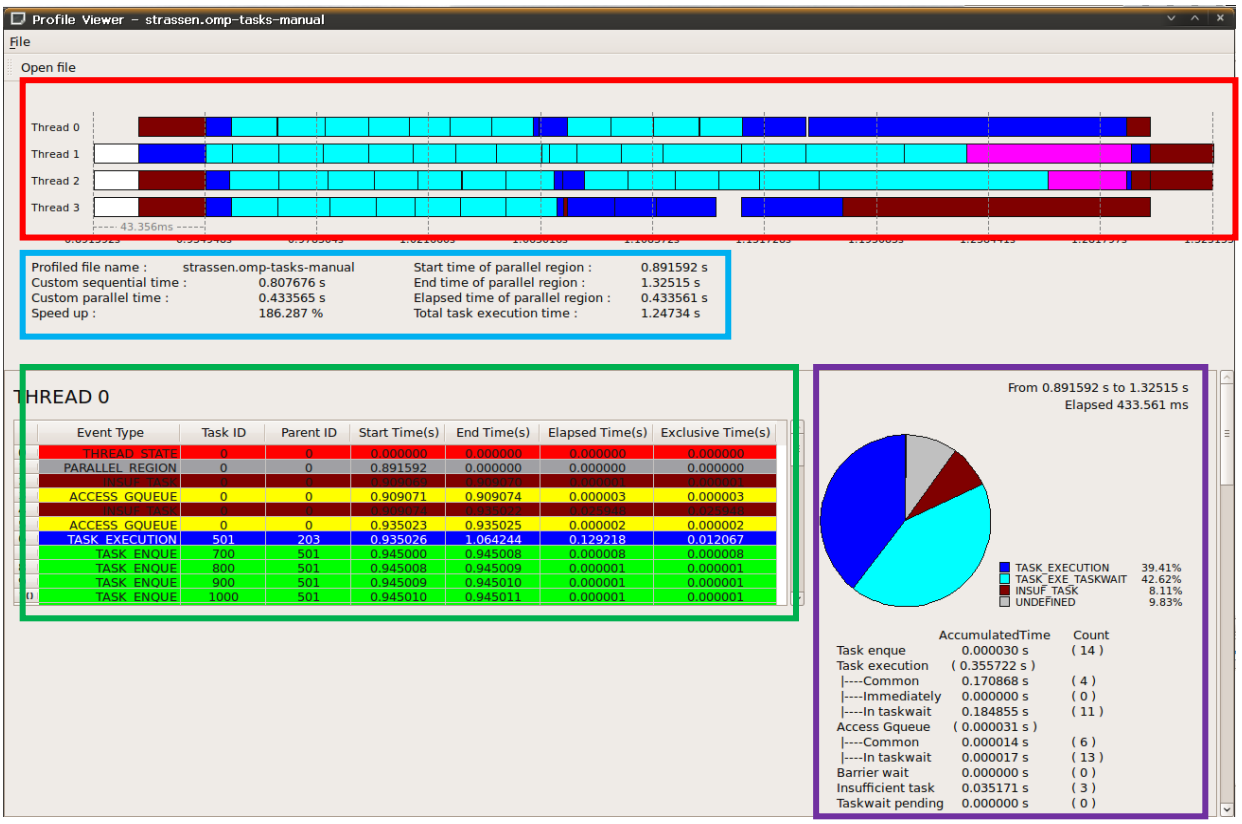


Figure 3. Visualization Tool Profile Viewer

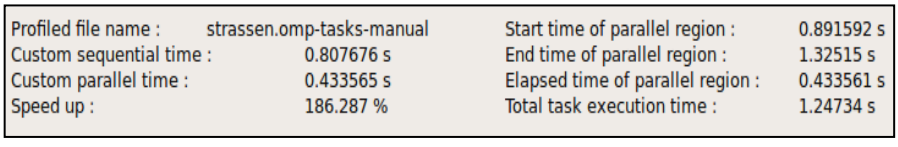
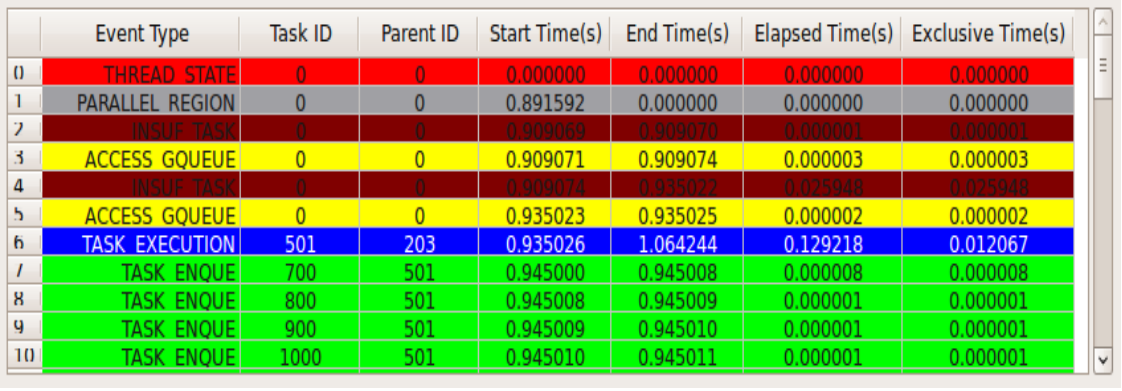


Figure 4. Overview of Profiled Program



	Event Type	Task ID	Parent ID	Start Time(s)	End Time(s)	Elapsed Time(s)	Exclusive Time(s)
0	THREAD_STATE	0	0	0.000000	0.000000	0.000000	0.000000
1	PARALLEL_REGION	0	0	0.891592	0.000000	0.000000	0.000000
2	INSUF_TASK	0	0	0.909069	0.909070	0.000001	0.000001
3	ACCESS_GQUEUE	0	0	0.909071	0.909074	0.000003	0.000003
4	INSUF_TASK	0	0	0.909074	0.935022	0.025948	0.025948
5	ACCESS_GQUEUE	0	0	0.935023	0.935025	0.000002	0.000002
6	TASK_EXECUTION	501	203	0.935026	1.064244	0.129218	0.012067
7	TASK_ENQUE	700	501	0.945000	0.945008	0.000008	0.000008
8	TASK_ENQUE	800	501	0.945008	0.945009	0.000001	0.000001
9	TASK_ENQUE	900	501	0.945009	0.945010	0.000001	0.000001
10	TASK_ENQUE	1000	501	0.945010	0.945011	0.000001	0.000001

Figure 5. Table Lists Events which Were Executed by Threads

4. Experiments and Analysis of GOMP Profiler

In order to evaluate the efficiency of this profiler and how well this profiler can help the programmer to improve performance, we have chosen a set of benchmarks which use OpenMP tasks to achieve parallelism. These benchmarks are different from general loop-level parallelism, thus they are commonly used to parallelize the recursive programs or while-loop structures. There are many benchmarks available for analysing loop level parallelism[16]; however, few are available for task level parallelism.

Barcelona OpenMP Tasks Suite (BOTS)[17] is a collection of applications that exploit task level parallelism in particular, and it aims to evaluate the OpenMP implementation. For the experiment purpose, we use BOTS version 1.0 as our benchmark tool. The summary of 8 benchmarks is listed in table 1. Before describing each application, we will elaborate the detailed characteristics of each of these applications as shown in table 1.

4.1. Barcelona OpenMP Tasks Suite

Table 1. Summary of Barcelona Task Suite Benchmarks

Application	Origin	Domain	Com. Str	Nested	CutOff
Alignment	AKM	Dynamic	Iterative	No	None
FFT	Cilk	Special	At leafs	Yes	None
Fibonacci	-	Integer	At each N	Yes	Dep B
Floorplan	AKM	Optimization	At each N	Yes	Dep B
Health	Oden	Simulation	At each N	Yes	Dep B
Nqueen	Cilk	Search	At each N	Yes	Dep B
SparseLU	-	Search Lin. Alg	Iterative	No	None
Strassen	Cilk	Dense Lin. Alg.	At each N	Yes	Dep B

Dep.B = Depth based At each N = At each Node

4.2. Using GOMP to analyse BOTS Benchmarks

The experimental environment was as follows: Operating system was Linux distribution Ubuntu 10.04 with kernel 2.6.32-22; benchmark programs were as mentioned above, Barcelona OpenMP Task Suite (BOTS); OpenMP compiler was GCC compiler 4.4.1 modified version that was the GOMP with additional instrumentation library implemented for this research; CPU used for this research was Intel Core 2 Quad Q6600 2.4GHz; memory was DDR2 and the size was 2GB. Since this platform used quad-core, the ideal speedup comparison with the original sequential version would be 4 times. In addition, we implemented these benchmarks with all 4 cores, not 2 or 3, and found the behaviour of each benchmark via GOMP Profiler. In the last part of this section, the relationship between experimental results and the program's algorithm will be presented.

4.3. Visualization Tool

The special features of the GOMP Profiler from various aspects are as follows; **speedup**, **run-time cut-off**, **task overhead**, **extra parallel computation** and **synchronization overheads**.

First, the **speedup** of the parallel program is a basic metric to analyse how fast the parallel version of the program is compared to the sequential version. When the speedup is significantly lower than the number of processors, it would mean that the parallelization has not achieved its maximum goal and has some problems in the parallelization part which needs to be improved. This metric informs the user as to how one must carefully analyse the detailed aspects of parallelization in relation to task level parallelism, if the metric does not come close to linear speedup as expected by the user.

Secondly, **run-time cut-off** which indicates a region of a program that is marked with "#pragma omp task" should be treated as an OpenMP task, but it is executed directly by a thread. This can be judged by the results of run-time scheduler or scheduled directly by the user. If the tasks are inserted into a global/local task queue, it has a tradeoff. Hence, without inserting tasks to a global/local queue and scheduling tasks, tasks take less time during run-time cut-off phase.

Third, **average** task time indicates the average time cost by a task. Less time means that, it spends more time in creating and scheduling which can be considered as overhead time. Generally, time taken by creating and scheduling a task is constant and it is the time taken by each

task. If creating and scheduling a task takes 100 ns, then the average task time of a program is 900 ns, the overhead is 10% that it is shown below as to how to calculate the creating and scheduling overhead of a task.

$$\frac{\text{creating and scheduling time (overhead of a task)}}{\text{Average task time + creating and scheduling time (overhead)}}$$

If the average task time is 100 ns, then the overhead would be 50% of the program. Thus, when the average task time is too small, the creating and scheduling overhead would be large.

The fourth aspect is the **extra parallel computation**. In general, parallel version of a program must do more work than the sequential version. So the extra parallel computation just refers to such works, but does not include the types of synchronization overheads we defined. Hence, it would indicate to us the need to decrease it when there is a large amount of extra parallel computation. Thus, the profiler helps the users to analyse and to decrease the extra parallel computation involved in the parallel version of the application to achieve maximum parallelism.

Finally, the **synchronization overheads** would indicate what percentage of the total execution time is taken by synchronization. In this research, synchronization overheads are categorized by four different types of synchronization; **accessing global task queue**, **barrier waiting**, **insufficient task** and **task-wait pending**.

Accessing_global_task_queue overhead is normally caused by task scheduler, while OpenMP task enqueue and thread dequeue task must do synchronized operations and thus it will be serialized rather than the parallelization of this portion. **Barrier waiting** will happen at the beginning of each parallel region or at the end of parallel region. At the beginning of a parallel region, OpenMP will create a team of threads and set a barrier to synchronize these threads, waiting until the last thread is finished by the creating process. After the parallel region, each thread except the master thread will wait for the barrier until the next parallel region is reached.

Insufficient task directive is that, there are no more tasks to be executed in the queue during a particular time, so it notifies that the task required by the processor is greater than the tasks executed in the queue. Task-wait pending is an implementation overhead. GOMP uses a simple method to implement the task-wait directive, so it can not do the load-balance of tasks in the task-wait region. However, by using a simple method, this research proposes a solution and discusses them at the later part of the experiment results.

In order to precisely measure the speedup of each application for accuracy, each program was run 10 times and the average sequential and parallel time for speedup was taken. The other types of characteristics such as run-time cut-off, average task time, extra parallel computation and synchronization overhead, were also observed 10 times, and the best observation was chosen to represent that particular program when all the observations were similar. Otherwise, we present the differences between various observations. The following paragraphs present the characteristics of each application that resulted from using GOMP Profiler.

4.4. Alignment Program

This benchmark aligned 100 protein sequences that were run in sequential as well as parallel version. The sequential version took 52.234803 secs and the parallel version took 13.1954 secs as shown in table 2. Thus, the speedup achieved for this application was 3.95 by using four processors. Even though it shows an excellent parallelism, it is important to note why this program achieves parallelism close to maximum number of processors. Since the profiler can clearly indicate to us the details of how the time spent in different parts of the program, we can note the special characteristics of this program using the task level parallelism. By using this profiler, one can note that the average task time was 52.6186secs and the total time of enqueuing and scheduling time was only 0.789ms. Hence, one can point out that the task overhead is nearly zero. Run-time cut-off rate has reached the maximum of 94.77% (Table 3), which is due to the program's iterative structure.

Table 2. Basic Information of Alignment Program

Program Name	Alignment-omp-task
Input file	Prot 100.aa
Sequential time	52.234803 sec
Parallel Time	13.1954 sec

Table 3. Profiled Information of Aligned Program

Speedup	395%
Run-time cutoff	94.77%
Average task time	10.63 ms
Extra Computation	~0%
Synchronization	~07%

At the beginning of the execution of the alignment application, a large number of tasks had been generated and OpenMP run-time scheduler had limited maximum number of tasks, thus the tasks were executed immediately when the number of tasks exceeded the limited number. Hence, there was nearly no synchronization overhead at all during these programs' execution. Finally, by using this profiler, we can analyse, why the result of this parallel version was an excellent one and it needed no further improvement: high run-time cut-off rate was 94.77% that resulted from generating tasks rate which was greater than the tasks consumed rate; fewer tasks to enqueue (258 tasks) means no task overhead; no synchronization overhead which comes close to zero and the synchronization operation was dependent on the high run-time cut-off rate; finally, there was no extra parallel computation. All of these aspects contributed to the nature of achieving high level parallelism.

4.5. FFT Program

This particular application processes a vector size of 33554432 by FFT which can be seen from table 4. It was an example of a very fine grain program, where the average task time was only 8.64 μsec as shown in table 5, which resulted

from the workload of each recursive function that was too small. Each recursive function was scheduled by an OpenMP task. However, it had a very high run-time cut-off rate, i.e., 99.26% (Table 5), which helped to reduce the overall overhead, and had a significant improvement. However, there was a slight performance degradation that was mainly due to extra parallel computation, i.e., 3.2557secs; this parallel computation time was about 22.85% that exceeded the sequential code time. As a result, the speedup of FFT benchmark which can be calculated as 100% /[(100% + 22.85%) / 4 cores] = 325% only, it was mainly due to extra parallel computation overhead. The speedup was 311%, however, it could have achieved the speedup of 325% with only little bit extra parallel computation overheads.

Table 4. Basic Information of FFT Program

Program Name	fft-omp-task
Problem Size	33554432
Sequential time	14.249425 sec
Parallel Time	4.57206 sec

Table 5. Profiled Information of FFT Program

Speedup	311%
Run-time cutoff	99.26%
Average task time	8.64 µs
Extra Computation	22.85%
Synchronization	1.54%

While observing the results generated by the profiler, the users can not just understand why there was 22% extra parallel computation and what was the reason for this extra parallel computation? However, it can point out to the users of the need to analyse the source code. So, by tracing the source code of FFT program and observing the OpenMP task and task-wait directives process, it is obvious that OpenMP task and task-wait directives were the extra work. The extra parallel overhead was due to OpenMP run-time library. Here, there was a need to take a closer look at the run-time library overhead and tried to reduce the run-time library overhead. In order to solve this overhead involved in such applications, the users must re-implement the OpenMP task and task-wait directives with lower workloads.

4.6. Fibonacci Program

This is a common application that is used for calculating Fibonacci number. For our experiment, the Fibonacci number 40 was chosen. The timeline which can be noted from figure 6, generated by the profiler for Fibonacci program that clearly showed that thread1 had almost 100% task-wait pending overhead (pink color-second row) and the other thread shown on the fourth row had roughly one third overhead (pink color). There was 34.37% synchronization overhead which can be seen from table 7, was due to the total task-wait pending time, i.e., 0.456257sec. and the total parallel computation time was 1.327692secs. (parallel time * number of processors). It can be noted that the high level of synchronization operation took large amount of time in the total parallel computation. Hence the speedup for this benchmark was only 262%.

Table 6. Basic Information of Fibonacci Program

Program Name	fib-omp-task manual
Input file	40
Sequential time	0.872276 sec
Parallel Time	0.331923 sec

Table 7. Profiled Information of Fibonacci Program

Speedup	262%
Run-time cutoff	0%%
Average task time	28.914 ms
Extra Computation	~0%
Synchronization	34.37%

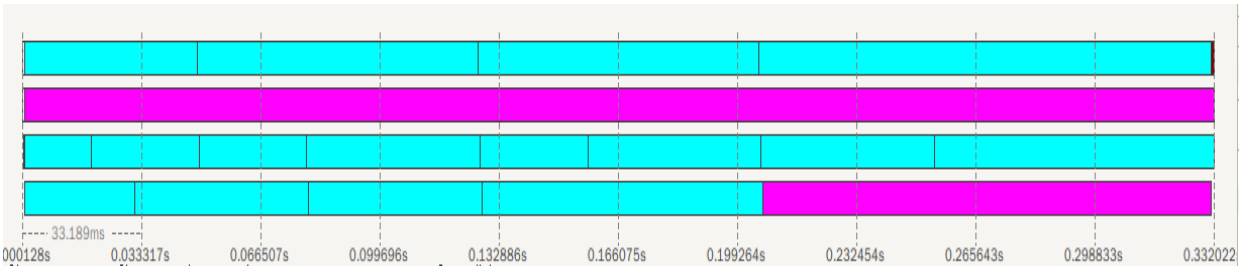


Figure 6. Timeline of Fibonacci Program with 4 Threads

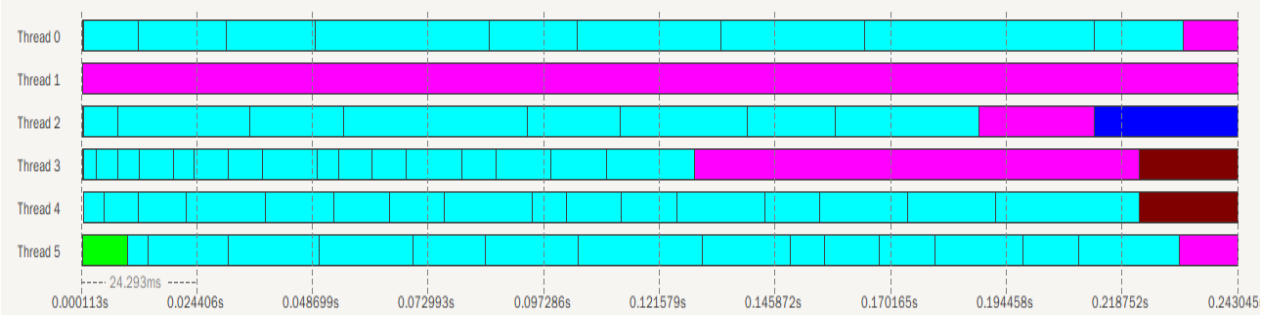


Figure 7. Timeline of Fibonacci Program with 6 Threads

Before trying to eliminate this task-wait pending overhead, one must understand why it happened? It was mainly caused by GOMP that incompletely supports OpenMP specification. In the definition of OpenMP specification, the current task region is suspended until the children tasks which are defined before task-wait directives are finished. GOMP implements this behavior of task-wait directive, but it waits for the child task instead of suspending. This behavior would contribute to the wasted time for awaiting children tasks. Figure 7 shows one particular situation of Fibonacci (14) waiting for the children tasks to complete Fib (12) and Fib (13). Fibonacci (14) enqueues the suspending task, and executes tasks which are generated from other tasks until Fib (12) and Fib (13) have been finished. Because the GOMP does not support task switching, programmer can improve the code by increasing the number of threads used. The programmer must find a way to solve this problem.

For this example, we increased the number of threads from 4 to 6, the additional threads which can execute tasks while any other threads were pending on the task-wait directive. Even though the timeline chart can not present the behavior of thread switching between the cores of the processor, we can understand the improvement in performance through speedup metric, which was about 262% with 4 threads improved to 359% with 6 threads. Thus, it had 37% improvement. Besides, the impact on the speedup was due to a little bit of insufficient task overhead at the end of execution and it can be solved by decreasing the execution time of tasks.

4.7. Floorplan Program

Table 8. Basic information of Floorplan Program

Program Name	floorplan-omp-task
Input file	Input.20
Sequential time	25.043471 sec
Parallel Time	11.5 sec

Table 9. Profiled information of Floorplan Program

Speedup	217%
Run-time cutoff	0%
Average task time	1.5102 ms
Extra Computation	9.29%
Synchronization	39.74%

The floorplan program arranged 20 pre-defined cells and returned the minimum area. The floorplan also had large percentage of task-wait pending overhead as shown in figure 8. Here too, similar to the previous program, we used the same method to deal with the implementation problem, increased the number of threads to 8 and got the final speedup of 363%. Besides, the extra parallel computation was 9.29%, which was caused by the task creation overhead. The ideal parallel computation time was about 109.29% / 4 = 27.32% of the sequential code time and the maximum speedup should be 366%. Consequently, it gave an excellent parallelism of 3.63 which was nearly close to the maximum speedup of 3.66 with extra parallel computation.

4.8. Health Program

Table 10. Basic information of Health Program

Program Name	Health.omp-task Manual
Input file	large.input
Sequential time	10.469463 sec
Parallel Time	6.025481 sec

Table 11. Profiled information of Health Program

Speedup	173%
Run-time cut-off	0%
Average task time	0.047 ms
Extra Computation	116%
Synchronization	2.05%

Health program simulates de Columbian Health Care system with large input as shown in table 10, which had 4 different levels of hospitals, population of 160 persons and simulating 365 times. The biggest performance problem for this application was that extra parallel computation as shown in table 11 which was about twice the sequential version. We can note that the maximum speedup gained could not be greater than 2 with a quad-core processor. We needed to directly trace the source code and to look for the reasons of extra computation. Thus, we found out that there was one critical section for synchronization which generated extremely large amount of overhead. In addition, Health program had slightly large enqueue and accessing global queue overheads. Combining these two types of overheads were about 4% of the total parallel time, but others were less than 1%. Solving the critical section synchronization overhead, has two different methods: one is to reduce the frequency of using critical section; and the other is to reduce the workload of critical section. Reducing the frequency of critical section represents that we should change the program's logic or algorithm for this purpose. The workload of critical section is rather short and so reducing the workload is less important.

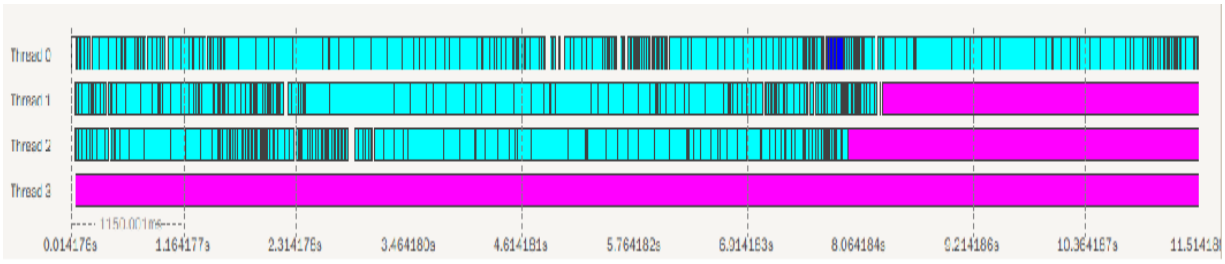


Figure 8. Timeline of Floorplan Program with 4 Threads

4.9. N-Queen Program

For this N-Queens' problem as shown in table 12, we try to solve 14 N-Queens. As described by N-Queens, the extra computation is from the operation of the parent task which copies its chessboard to the children tasks. It is also considered the fine grain task problem and uses a manually cut-off method to increase the task time. However, it did not consider reducing the memory allocation together. Consequently, it had little synchronization overhead and 11.98 % extra computation as can be seen in table 13.

Table 12. Basic Information of N-queen Program

Program Name	nqueens.omp-task Manual
Problem size	14x14 board
Sequential time	53.706184 sec
Parallel Time	15.4563 sec

Table 13. Profiled Information of N-queen Program

Speedup	347%
Run-time cutoff	0%
Average task time	25.216 ms
Extra Computation	11.98%
Synchronization	2.67%

We modified the N-Queens' benchmark in order to have the ability to eliminate memory allocation while having manually cut-off. The extra computation was still present, but it was reduced from 11.98% to 6.72%. So, the speedup had greatly improved from 347% to 375%.

4.10. SparseLU Program

Table 14. Basic Information of SparseLU Program

Program Name	Sparselu.single-omp-tasks
Problem size	S1=50x50, S2=100x100
Sequential time	16.55991 sec
Parallel Time	4.30962 sec

Table 15. Profiled Information of SparseLU Program

Speedup	384%
Run-time cutoff	6.55%
Average task time	1.436 ms
Extra Computation	1.26%
Synchronization	~0%

SparseLU factorized the S2=50*50 entries in the S1=100*100 square matrix as shown in table 14. There were four properties present in this program in order to achieve a good parallelism: not fine grain task, 1.436 ms taken by each task; little extra parallel computation overhead; and nearly zero synchronization overhead (All three properties can be noted from table 15). In order to achieve good parallelism, good load balancing and using dynamic scheduling are very important. Even though there was only 6.55% tasks that was run-time cut-off, and since it was coarse grained tasks which helped during the run-time cut-off. So, combining all the previous properties, the speedup of Sparse LU was 384% using a quad-core processor.

4.11. Strassen Program

Table 16. Basic Information of Strassen Program

Program Name	Strassen.-omp-tasks
Problem size	1024
Sequential time	0.794679 sec
Parallel Time	0.264685 sec

Table 17. Profiled Information of Strassen Program

Speedup	300%
Run-time cutoff	0%
Average task time	16.361 ms
Extra Computation	17.36%
Synchronization	12.76%

Strassen program used 1024*1024 matrix multiplication as shown in table 16. We can note that it had an extra computation which was 17.36% and synchronization overhead which was 12.76% as presented in table 17. Our findings can not show any reasons for extra computation, because of that there was no task creation overhead while having coarse grain tasks, no user-defined critical section and no extra memory allocation. Thus, we assume that the extra parallel computation must have come from events which we could not detect, such as cache coherence, false sharing or cache conflicts, etc. In addition, during the starting and ending stages, it had insufficient tasks (can be noted from figure 9 shown in brown color); In the beginning, master thread executed a small portion of the work (the first dark blue section in the fourth row) and then generated other tasks, so other threads must wait (the first brown part of first three rows) until that part of the work was finished; At the end, each thread executed one of the last 4 tasks. If the workload of each task was not the same or tasks had different start times, the faster the threads, it would have to wait for the execution of slower threads. Finally, we could see that this program had only 300% speedup which was mainly due to extra parallel computation and insufficient tasks.

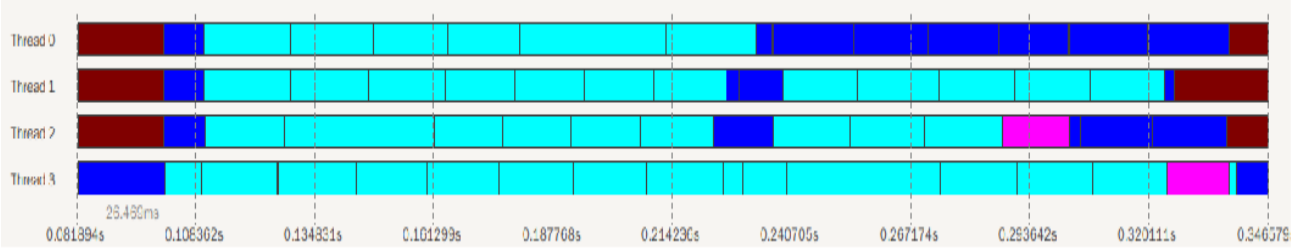


Figure 9. Timeline for Strassen Program with 4 Threads

According to the improved results of these experiments, we can present the capabilities of GOMP Profiler. The main features that focus on speedup, run-time cut-off, task overhead, extra parallel computation and synchronization overheads which can help us to discover the problem of task level parallelism. In addition, it helps us to understand which part of the program needs to be modified for better improvement, and where does the problem come from (run-time library or user level implementation).

5. Conclusions and Future Work

GOMP Profiler focuses on the OpenMP task level parallelism and aims to achieve a specific platform GNU OpenMP. We have shown the capability of this particular profiler in section 4. It provides several metrics to pinpoint the problems of Barcelona OpenMP Task Suite. The simplest way to figure out the parallelism is the speedup metric. If the speedup is significantly less than the number of processors, it indicates that there could be some extra overheads in the program. Among all the benchmarks used, the alignment and SparseLU benchmarks had rather promising speedups. The rest of the benchmarks, the speedup metric showed that they had some problems in parallelizing the benchmarks. Other metrics such as run-time cut-off metric using GOMP showed that it had good cut-off rates in the following benchmarks: alignment and FFT. It reflected that those benchmarks had reduced the creating and scheduling overheads by run-time cut-off, vice versa. Extra parallel computation metric showed almost every benchmark had 0~20% extra parallel computation except the health benchmark which had 216%. All this information helped us to find out the synchronization overhead that occurred in health benchmark. These types of synchronizations are the following: accessing global queue, barrier waiting, insufficient tasks and task-wait pending. Every program normally has 2% of global access queue and barrier waiting overheads. However, insufficient task and task-wait pending overheads varied for different benchmarks. Fibonacci and floorplan had large amount of task waiting overhead, and Strassen had many insufficient tasks. All of these metrics represent that the GOMP Profiler has the ability to pinpoint the extra overheads in task level parallelism of any OpenMP task programs. From the results of GOMP Profiler, we can analyse and optimize these programs. In the future, these four overheads can be individually measured in all the benchmarks run on parallel version.

ACKNOWLEDGEMENTS

Our Research group greatly appreciates Fu Jen Catholic University's office of Research and Development for their constant support and help in accomplishing our goal to complete this research. Our sincere thanks, also to the Ministry of Education (MOE) in Taiwan for their support and assistance.

REFERENCES

[1] Mohr, E. Kranz, D.A. Halstead and R.H., Jr., "Lazy Task Creation A Technique for Increasing the Granularity of Parallel Programs", IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No.3, pp: 264 – 280, (1991).

[2] Stephen L. Olivier and Jan F. Prins, "Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs", Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an age of Extreme Parallelism, pp: 63 – 78, (2009).

[3] Alejandro Duran, Julita Corbalán and Eduard Ayguadé, "An Adaptive Cut-off for Task Parallelism", Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, No.36, (2008).

[4] Alejandro Duran, Julita Corbalán and Eduard Ayguadé, "Evaluation of OpenMP Task Scheduling Strategies", Proceedings of the 4th International Workshop on OpenMP, (IWOMP-2008) In Lecture Notes in Computer Science, Vol.5004, pp: 100-110, (2008).

[5] OpenMP Application Program Interface, Version 3.0, OpenMP Architecture Review Board, (May 2008).

[6] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan and Guansong Zhang , "The Design of OpenMP Tasks", IEEE Transactions on Parallel and Distributed Systems, Vol. 20, No. 3, pp: 404-418, (2009).

[7] Diego Novillo, "OpenMP and Automatic Parallelization in GCC," in Proceedings of the GCC Developers' Summit, pp: 1 - 10, (June 2006).

[8] Sameer S. Shende and Allen D. Malony, "The TAU Parallel Performance System", International Journal of High Performance Computing Applications, Vol. 20, No. 2, pp: 287 – 311, (2006).

[9] Karl Fürlinger, Michael Gerndt and Technische Universität München, "ompP: A Profiling Tool for OpenMP", Proceedings of the First International Workshop on OpenMP, (IWOMP) (2005).

[10] Eduard Ayguade, Alejandro Duran, Jay Hoeflinger, Federico Massaioli and Xavier Teruel, "An Experimental Evaluation of the New OpenMP Tasking Model", Languages and Compilers for Parallel Computing: 20th International Workshop, pp: 63–77, (2007).

[11] Xavier Teruel, Xavier Martorell, Alex Duran, Roger Ferrer and Eduard Ayguade, "Support for OpenMP Tasks in Nanos v4", CASCON 2007, Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research, pp: 256-259, (2007).

[12] Karl Fürlinger and David Skinner, "Performance Profiling for OpenMP Tasks", Proceedings of the 5th International Workshop on OpenMP, (IWOMP), pp: 132 – 139, (2009).

[13] Bernd Mohr, Allen D. Malony, Sameer Shende and Felix Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP", The Journal of Supercomputing, Vol. 23, No.1, pp: 105 – 128, (2002).

[14] Mubrak S. Mohsen, Rosni Abdullah and Yong M. Teo, "A Survey on Performance Tool for OpenMP,"World Academy of Science, Engineering and Technology, Issue 49, pp: 754 – 765, (2009).

[15] Luiz DeRose, Ted Hoover Jr. and Jeffrey K. Hollingsworth, "The Dynamic Probe Class Library: An Infrastructure for Developing Instrumentation for Performance Tools", Proceedings of the 15th International Parallel and Distributed Processing Symposium, Vol. 1, pp: 10066.2, (2001).

[16] Michael Gerndt, Technische Universität München , Bernd Mohr , Forschungszentrum Jülich Gmbh and Jesper Larsson Träff, "Evaluating OpenMP Performance Analysis Tools with the APART Test Suite", Test Suite, Fifth European Workshop on OpenMP, pp: 147 – 156, (2003).

[17] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell and Eduard Ayguade, "Barcelona OpenMP Task Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP", Proceedings of the 38th International Conference on Parallel Processing, pp: 124-131, (2009).