



Intel[®] Technology Journal

Multi-Core Software

Parallel Software Development with Intel[®] Threading Analysis Tools

Parallel Software Development with Intel[®] Threading Analysis Tools

Lihui Wang, Intel Information Technology, Intel Corporation
Xianchao Xu, Intel Information Technology, Intel Corporation

Index words: Intel[®] Threading Analysis Tools, multi-core, parallel programming, development cycle, threading methodology

ABSTRACT

While multi-core processors are designed for greater performance with optimal power consumption, the parallel algorithm design and software development that is needed to maximize the performance potential of multi-core systems are much more complicated than those associated with serial computing. Even though parallel computing has long been studied by researchers, there is no general framework to implement parallel programming for different software applications. Programmers face three immediate challenges when applying parallelism to software development: scalability, correctness, and maintainability. Applicable parallel methodology and new software development tools are greatly needed by programmers working in this environment. To keep software applications in sync with the multi-core processors that are becoming mainstream in the marketplace, Intel provides a whole set of threading software products.

A multiple pattern matching algorithm is the core algorithm of the detection engine in the rule-based Intrusion Detection System (IDS). Most of the research on improving the performance of this algorithm is based on serial computing. Actually, the performance of the algorithm can be improved greatly through parallelization on multi-core systems. By threading and tuning a typical multiple pattern matching algorithm, we show how to apply parallel principles during each phase of a generic development cycle while utilizing Intel[®] Threading Analysis Tools to pinpoint the bottlenecks and thread-safety errors and to improve overall performance. Moreover, we specifically compare the implementation method and performance gain of the Windows* Threading API against that of Intel[®] Threading Building Blocks (Intel[®] TBB), when implementing the parallel multiple pattern matching algorithm with the experimental performance data presented.

INTRODUCTION

As multi-core processors become mainstream in the market place, software needs to be parallel to take advantage of multiple cores. However, there is no general framework available to implement parallel programming for different applications to achieve the highest performance gain. Generally implemented with multi-threading, parallel programming is notoriously difficult for developers to design, implement, and debug. In order to make life easier for developers, Intel provides a set of threading tools targeting various phases of the development cycle. In a generic development cycle, program development can be divided into four phases [1]:

- Analysis phase: Profiling the serial version of the program to determine the areas that are suitable for parallel decomposition.
- Design/implementation phase: Examining identified threading candidates, determining the changes that have to be made to the serial version, and converting them to the actual code.
- Debug phase: Ensuring the correctness of the program. Detecting and solving common threading errors such as data race and deadlocks.
- Testing/tuning phase: Validating the correctness of the program and testing its performance. Detecting performance issues and fixing them by improved design or by eliminating bottlenecks.

Intel's threading tools provide aids for developers from performance analysis to implementation and debugging:

- Intel[®] VTune™ Performance Analyzer [4]. This tool helps developers tune an application to better perform on Intel[®] architectures. It locates the performance bottlenecks and program hotspots by collecting, sampling, and displaying system-wide data down to

specific functions, modules, or instructions. It is usually used during the analysis and tuning phase of the development cycle.

- Intel® Thread Profiler [5]. This tool helps to identify performance bottlenecks in Win32* and OpenMP* threaded software. It detects threading performance issues such as thread overhead and synchronization cost. The profiler is usually used in the tuning phase.
- Intel® Thread Checker [6]. This tool helps to find bugs in Win32 and OpenMP threaded software. It locates threading issues such as race conditions, thread stalls, and potential thread deadlocks. The Intel Thread Checker is usually used during the design and debugging phases.
- Intel Threading Building Blocks (Intel TBB) [7]. This is a threading abstraction library that provides high-level generic implementation of parallel patterns and concurrent data structures [2]. Intel TBB is usually used in the design, implementation, and tuning phases.

In the sections that follow, we first introduce the principles of parallel application design; then we show how to parallelize an application with the help of threading tools during each phase of the development cycle. A multiple pattern matching algorithm is used as an example. We use the Win32 threading API and Intel TBB to implement the parallelism, and we compare the performance of the two.

PRINCIPLES OF PARALLEL APPLICATION DESIGN

Decomposition Techniques

Dividing a computation into smaller computations and assigning these to different processors for execution are two key steps in parallel design. Two of the most common decomposition techniques [3] are functional decomposition and data decomposition [2].

- Functional decomposition is used to introduce concurrency in the problems that can be solved by different independent tasks. All these tasks can run concurrently.
- Data decomposition works best on an application that has a large data structure. By partitioning the data on which the computations are performed, a task is decomposed into smaller tasks to perform computations on each data partition. The tasks performed on the data partitions are usually similar. There are different ways to perform data partitioning: partitioning input/output data or partitioning intermediate data.

Parallel Models

These are some of the commonly used parallel models [3].

- *Data parallel model.* This is one of the simplest parallel models. In this model, the same or similar computations are performed on different data repeatedly. Image processing algorithms that apply a filter to each pixel are a common example of data parallelism. OpenMP is an API that is based on compiler directives that can express a data parallel model.
- *Task parallel model.* In this model, independent works are encapsulated in functions to be mapped to individual threads, which execute asynchronously. Thread libraries (e.g., the Win32 thread API or POSIX* threads) are designed to express task-level concurrency.
- *Hybrid models.* Sometimes, more than one model may be applied to solve one problem, resulting in a hybrid algorithm model. A database is a good example of hybrid models. Tasks like inserting records, sorting, or indexing can be expressed in a task-parallel model, while a database query uses the data-parallel model to perform the same operation on different data.

Amdahl's Law

Amdahl's law provides the theoretical basis to assess the potential benefits of converting a serial application to a parallel one. It predicts the speedup limit of parallelizing an application:

$$T_{\text{parallel}} = \{(1 - P) + P / N\} * T_{\text{serial}} + O_N$$

T_{serial} : time to run an application in serial version

P: parallel portion of the process

N: number of processors

O_N : parallel overhead in using N threads

We can predict the speedup by looking at the scalability:

$$\text{Scalability} = T_{\text{serial}} / T_{\text{parallel}}$$

We can get the theoretical limit of scalability, assuming there is no parallel overhead:

$$\text{Scalability} = 1 / \{(1 - P) + P/N\}$$

When $N \rightarrow \infty$, Scalability $\rightarrow 1 / (1 - P)$

CHALLENGES OF PARALLEL PROGRAMMING

Parallel Overhead

Parallel overhead refers to the amount of time required to coordinate parallel tasks as opposed to doing useful work. Typical parallel overhead includes the time to start/terminate a task, the time to pass messages between tasks, synchronization time, and other extra computation time. When parallelizing a serial application, overhead is inevitable. Developers have to estimate the potential cost and try to avoid unnecessary overhead caused by inefficient design or operations.

Synchronization

Synchronization is necessary in multi-threading programs to prevent race conditions. Synchronization limits parallel efficiency even more than parallel overhead in that it serializes parts of the program. Improper synchronization methods may cause incorrect results from the program. Developers are responsible for pinpointing the shared resources that may cause race conditions in a multi-threaded program, and they are responsible also for adopting proper synchronization structures and methods to make sure resources are accessed in the correct order without inflicting too much of a performance penalty.

Load Balance

Load balance is important in a threaded application because poor load balance causes under utilization of processors. After one task finishes its job on a processor, the processor is idle until new tasks are assigned to it. In order to achieve the optimal performance result, developers need to find out where the imbalance of the work load lies between different threads running on the processors and fix this imbalance by spreading out the work more evenly for each thread.

Granularity

For a task that can be divided and performed concurrently by several subtasks, it is usually more efficient to introduce threads to perform some subtasks. However, there is always a tipping point where performance cannot be improved by dividing a task into smaller-sized tasks (or introducing more threads). The reasons for this are 1) multi-threading causes extra overhead; 2) the degree of concurrency is limited by the number of processors; and 3) for most of the time, one subtask's execution is dependent on another's completion. That is why developers have to decide to what extent they make their application parallel. The bottom line is that the amount of work per each independent task should be sufficient to leverage the threading cost.

MULTIPLE PATTERN MATCHING ALGORITHM

Aho-Corasick_Boyer-Moore Algorithm

The Aho-Corasick_Boyer-Moore (AC_BM) [8] algorithm is an efficient string pattern matching algorithm that is derived from the Boyer-Moore algorithm to improve efficiency when there are multiple string patterns to search against. Multiple string pattern matching is one of the core problems in the rule-based Intrusion Detection System (IDS) such as Snort [9]. The AC_BM algorithm can be used to improve the efficiency of the detection engine of the IDS. The basic idea of the algorithm is that the string patterns are first built into a pattern tree, and then the algorithm slides the pattern tree using bad character and good prefix shifts to find the matches. Figure 1 shows the pseudo code of the AC_BM algorithm. In the code, we use `acbm_init()` to construct the pattern tree, and we use `acbm_search()` to compare the given text with the pattern tree.

Serial Implementation

```

pattern_tree *acbm_init(pattern_data
*patterns, int npattern);
int acbm_search(pattern_tree *ptree, unsigned
char *text, int text_len, unsigned int
matched_indexes[], int nmax_index);

int _tmain(int argc, char* argv[])
{
    pattern_data *patterns = NULL;
    int npattern = argc - 2;
    ...
    /* read app args, file length, and other
    initialization operations. */

    ptree = acbm_init(patterns, npattern);

    matched = acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexes,
nmax_index);
    printf("total match is %d\n", gMatched);
    return 0;
}

```

Figure 1: Serial implementation of AC_BM algorithm

Before moving on to parallelize the application and tuning the performance, it is better to baseline the performance of the serial version. We test the performance against different workloads based on different file sizes.

System environment:

- OS: Windows XP* SP2;
- CPU: Intel® Core™2 Duo processor 6300¹ @ 1.86 GHz; Memory: 1.00 GB

Figure 2 shows the test result of matching patterns against different file sizes:

Text size	1M	2M	4M	8M	16M	32M
Time (ms)	12.1 67	24.8 26	49.3 27	96.3 04	191. 728	383.3 53

Figure 2: Performance of different file sizes

Analysis

Identifying performance bottlenecks is the first step in improving the performance of an application. For simple applications, it's possible to find the bottleneck by looking at the algorithm and source code, and by analyzing the nature of an application. However, for large and complicated applications, it's difficult to predict performance issues when the bottlenecks are not only dependent on the algorithm, but also on the execution environment such as the processor or memory system. That is why we need a performance analyzing tool.

The Intel VTune Performance Analyzer finds performance bottlenecks by automating the process of data collection with three types of data collectors: sampling, call graph, and counter monitor. These data collectors help find the hotspots and the bottlenecks in the system, application, and micro-architecture levels. We use the VTune Performance Analyzer to profile the serial version of the AC_BM application and uncover the parallel opportunities.

In counter monitor view (as shown in Figure 3), we find that the average processor time is only 39.453%, which means that the processors are not fully utilized; therefore, we can improve the performance by introducing more threads.

Counter Name	Graph Scale	Average
Processor (_Total) : % Processor Time	1.0	39.453
Processor (_Total) : % Privileged Time	1.0	19.531

Figure 3: Counter monitor view

The call graph view (as shown in Figure 4) shows the critical path of the application. It helps to determine which function in the call tree to thread that will result in the best performance gain. Threading a function higher up in the tree results in a coarser-grain level of parallelism, while threading a child function lower in the tree results in a finer-grain level of parallelism. In the graph, we can see that the only function we can thread on the critical path is acbm_search, and acbm_search doesn't have any child function.

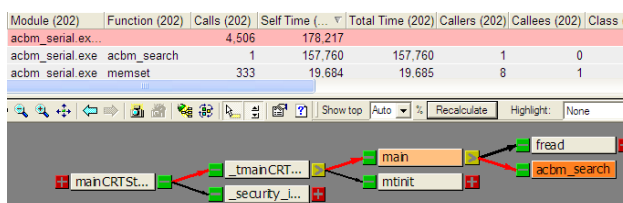


Figure 4: Call graph view

Threading Methods

Currently there are several threading strategies: OS threading libraries such as the Win32 API; compiler directives OpenMP; Message passing API (MPI); and the newly introduced Intel TBB. Each method has its own features and there is no simple way to determine which threading method is the best. It mainly depends on the specific application to be threaded.

In the following sections, we parallelize the AC_BM algorithm using the Win32 threading library and Intel TBB, and we compare the performance of the two.

Estimate the Performance Speedup

Before threading the application it is better to estimate the performance speedup of the parallelization: on the one hand, you may know about the upper bound of the performance limit and set the optimization goal for threading and tuning; on the other hand, estimating the performance speedup helps you to determine when to stop and accept the performance gain without trying for more tuning iterations.

In our system, there are two processors, and in the VTune Performance Analyzer sampling view, we find that the acbm_search function takes up 92.90% of CPU time. According to Amdahl's law

$$T_{parallel} = \{(1-0.929) + 0.929/2\} * T_{serial} + O_N;$$

Suppose $O_N \rightarrow 0$, Scalability $\rightarrow 1.867$

So the upper bound of the speedup is 1.867.

¹ Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families.

See www.intel.com/products/processor_number for details.

PARALLELIZATION WITH THE WIN32 THREADING LIBRARY

Decomposition Method

In call graph view, we can see that the `acbm_search` function is called only once, but it takes up a significant amount of CPU time. The operation of `acbm_search` is to match the pattern tree against the given file to count the total matches between the patterns and the file. Since different text portions of the file are independent, each portion can be matched against the pattern tree separately. Data decomposition is used to logically divide the file into several parts and to use one thread to match against the patterns and return the matches of each part. Thus for the `acbm_search` function, these threads only have two points of serialization, i.e., loading file and combining the final result of total matches after all threads are finished.

Implementation

Figure 5 and Figure 6 show the source codes that illustrate the basic operations of threading the algorithm with the Windows threading API.

When splitting the file into several parts, some matched strings may be split into different text portions. So in each search thread, after getting the matched pattern number from `acbm_search()` for their assigned portion of text, we use `acbm_connection()` to calculate the matched number of strings on the text boundary.

```
#define THREAD_NUM 4
CRITICAL_SECTION rsCriticalSection;
int gMatched;
int _tmain(int argc, char* argv[])
{
    pattern_data *patterns = NULL;
    int npattern = argc - 2;
    HANDLE hThread[THREAD_NUM];
    DWORD dwThread[THREAD_NUM];
    file_info *finfo[THREAD_NUM];
    ...
    /* read app args, file length, read
file, and other initialization operations.
*/

    ptree = acbm_init(patterns, npattern);

    int step = textLength / THREAD_NUM;

    for(i = 0; i < THREAD_NUM; i++)
    {
        ... // initialize finfo[i]
        hThread[i] = CreateThread(
            NULL, 0, ACBMThreadProc, finfo[i],
            0, &dwThread[i]);
    }

    WaitForMultipleObjects(THREAD_NUM,
hThread, true, INFINITE);
    printf("total match is %d\n", gMatched);
    return 0;
}
```

Figure 5: Main() function

```

DWORD WINAPI ACBMThreadProc(LPVOID pParam)
{
    unsigned int matched_indexes[0xffff];
    int nmax_index = 0xffff;

    file_info *finfo = (file_info*)pParam;
    if(finfo == NULL) return 1;

    textLength = finfo->file_len;
    text = finfo->text;

    gMatched += acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexes,
nmax_index);

    ...
    /* Adjust the value for the adjacent parts
of the divided text portions */
    gMatched += acbm_connection(...);

    ... // clean up code
    return 0;
}

```

Figure 6: AC_BM Search thread

Debugging

When introducing the threads, it is necessary to ensure their correctness. There are some notorious thread bugs, such as data race, stall, and deadlock, and they are usually difficult to detect. In our application, we use the Intel Thread Checker to check whether there are any bugs in the threads.




Short Description	Severity	Description	Count
Read -> Write data-race		Memory write at "acbm_win.cpp":693 conflicts with a prior memory read at "acbm_win.cpp":693 (anti dependence)	2
Write -> Read data-race		Memory read at "acbm_win.cpp":693 conflicts with a prior memory write at "acbm_win.cpp":693 (flow dependence)	2
Write -> Write data-race		Memory write at "acbm_win.cpp":693 conflicts with a prior memory write at "acbm_win.cpp":693 (output dependence)	2

Figure 7: Data race detected by Thread Checker

From the results shown in Figure 7, we can see that there is data race on the result calculating part. In order to avoid incorrect access to gMatched, we use a CRITICAL_SECTION to protect gMatched between different threads.

```

CRITICAL_SECTION rsCriticalSection;
DWORD WINAPI ACBMThreadProc(LPVOID pParam)
{
    ...

    EnterCriticalSection(&rsCriticalSection);
    gMatched += acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexes,
nmax_index);
    gMatched += acbm_connection(...);
    ...
    LeaveCriticalSection(&rsCriticalSection);

    ...
    return 0;
}

```

Figure 8: Fix the result calculating part

Test and Performance Tuning

After multi-threading the application, we test the performance with a 32M file: the result is 339.553ms. The scaling performance to the serial version, 383.353ms, is not very good, which means we need to tune the performance.

Performance tuning involves several steps:

1. Gather performance data.
2. Analyze data and identify issue.
3. Generate alternatives to resolve the issue.
4. Implement enhancements.
5. Test results.

All these steps are performed in one iteration of the performance tuning process. Usually, several iterations are required to get the best performance. Before any tuning process, we need to ensure our program is correct and ready for tuning. Another thing that should be kept in mind is to only make one change for each tuning iteration.

In order to find tuning opportunities for a multi-threaded program, we use the Intel Thread Profiler to gather thread-related data.

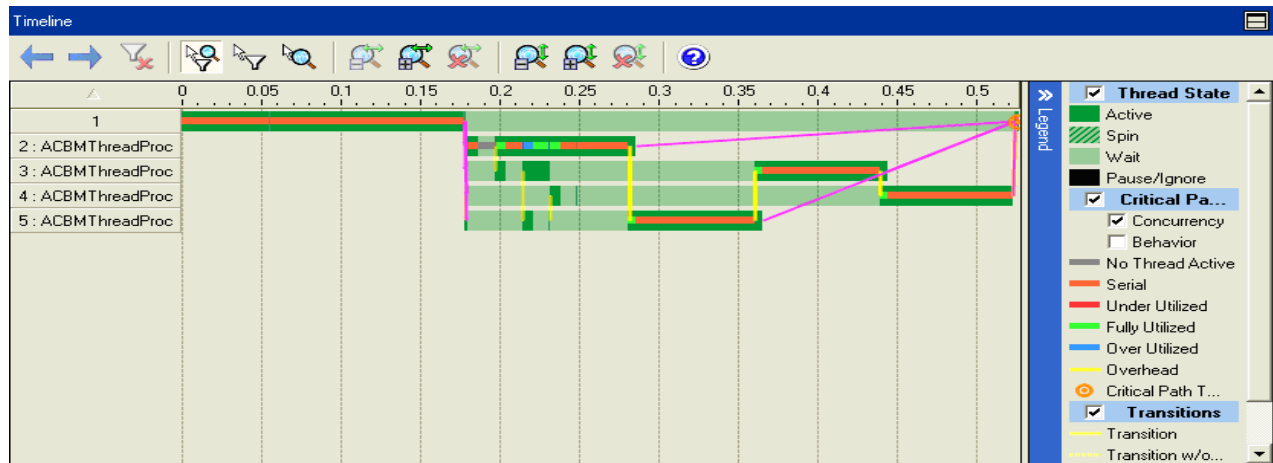


Figure 9: Thread Profiler Timeline view

In the Timeline view (shown in Figure 9), we can see that a large majority of the threads are executed in serial (the orange line indicates serial execution) and there are overhead and transitions between different threads (represented by yellow lines). We can double click one of the yellow lines and it will drill down to the source code where the problem occurs.

```

EnterCriticalSection(&rsCriticalSection);
gMatched += acbm_search(...);
...
LeaveCriticalSection(&rsCriticalSection);
    
```

Figure 10: Computing the search result

The transition and overhead are caused by acquiring access to the critical section. And, since only one thread can get access to the critical section at one time, the code essentially allows only one thread to do the search, which causes the serial problem. We modify the code like this (in Figure 11):

```

int matched = acbm_search(...);
matched += acbm_connection(...);
EnterCriticalSection(&rsCriticalSection);
gMatched += matched;
...
LeaveCriticalSection(&rsCriticalSection);
    
```

Figure 11: Improved approach to computing the search result

In this way, each thread performs the search independently and only needs to access the critical section when it adds the result to global variable gMatched.

After the modification, we tested the application and found that the time it spent on a 32M file was reduced to 255.500ms.

Now we have completed one iteration of tuning. We achieved improved performance after this iteration. Next we run the Thread Profiler again to see whether there are any other issues.

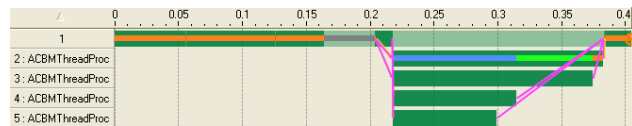


Figure 12: Thread Profiler Timeline view

In Figure 12, we can see that the serial execution of different threads has been solved. Notice that in the timeline view, the thread that ends last still has a small portion of serial execution, and the main thread needs to wait for all threads to exit and join the result. Instead of allowing the main thread to idle, we may let the main thread share a portion of data and do the search too. After the modification, the time spent is reduced to 252.643ms. See the timeline view in Figure 13.

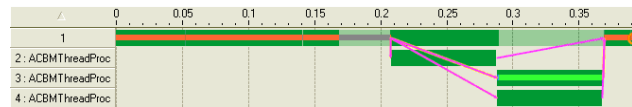


Figure 13: Thread Profiler Timeline view

We can see that there are no more overhead and transitions between different threads; Thread No. 3 is fully utilized during its life cycle. However, there are actually only two threads running at a time; the other two threads are just waiting. Remember, the execution environment is a dual-core system so there won't be more than two threads running at one time in a CPU. Delegating more than two threads may only incur extra overhead, due to thread creation and scheduling. We therefore change the THREAD_NUM to 2. The time spent is now reduced to 248.702ms.

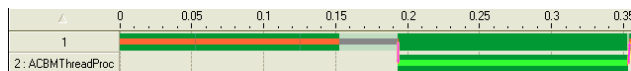


Figure 14: Thread Profiler Timeline view

Now the threads are all well utilized, and Figure 14 shows that there are no transitions and imbalances any more. The performance scalability compared to serial processing is $383.353ms/248.702ms = 1.54$ (the ideal is 1.867). Since the performance for the recent tuning iterations is not improved dramatically, we decide to accept the improvement.

From the tuning process presented above, we can see that the number of threads impacts the overall performance. In our example, the application works best when there are two threads, and the application happens to run on a dual core system. However, it doesn't mean that all applications running on a dual core system have the best performance when they have two threads. In the AC_BM algorithm, ACBMThreadProc only does some searches and calculations. However, for some other applications, the thread may need to wait for some resources, such as network connection or data loading. If this is the case, such a thread can relinquish the CPU and let another thread that is not waiting for resources to have the CPU. So in the case of this application, it may perform better if it has more threads than the number of CPU cores. There is no easy way to determine the number of threads for an application especially when the application is large and complex. Further, it is not easy to determine which task should be assigned to each thread to get the best performance. It is better to use the Thread Profiler to get the execution information about each thread and discover the tuning opportunities.

PARALLEL COMPUTING WITH INTEL® TBB

Challenges of Parallelizing with the Win32 Threading Library

While the Win32 threading library gives programmers great flexibility by giving them detailed control over threads, the library brings challenges for them too: programmers must rewrite the common parallel programming utilities; troubleshoot threading bugs, and try to avoid thread-safe issues. There is no way to maintain load balance and achieve good scalability unless the programmers perform these tasks manually, and these tasks require a comprehensive understanding of the threading method, application, and the features of the system.

Intel Threading Building Blocks

To make it easier for programmers to realize parallelism, Intel TBB provides a high-level generic implementation of parallel patterns and concurrent data structures. With TBB, you specify task patterns instead of threads. Logical tasks are mapped automatically onto physical threads, thus hiding the complexity of operating system threads.

Implementation

According to the analysis detailed in the “Parallel with the Win32 Threading Library” section, the AC_BM algorithm can be parallelized by using data decomposition. We can separate the text into several blocks, calculate the matched number in each block, and sum the total matched number from all the blocks. Intel TBB provides the template `parallel_reduce` to do this kind of decomposition.

Figure 15 and Figure 16 show the source code of the implementation with Intel TBB. Figure 15 shows the definition of *Search* task that performs the pattern matching operations. Figure 16 shows the `main()` function.

```

struct Search {
    int value;
    Search() : value(0) {}
    Search( Search& s, split ) {value = 0;}
    void operator()( const blocked_range<int>&
range ) {
        int textLength = range.end() -
range.begin();
        unsigned int matched_indexes[0xffff],
connection_matched_indexes[0xffff];
        int matched, connection_matched;
        int nmax_index = 0xffff;

        matched = acbm_search(ptree, (unsigned
char *) (text+range.begin()), textLength,
matched_indexes, nmax_index);
        value += matched;
        /* Adjust the value for the adjacent part
of the divided text portions */
        if(range.begin() > 0){
            textLength = (ptree->max_depth - 1) * 2;
            connection_matched = acbm_connection(...);
            value += connection_matched;
        }
    }
    void join( Search& rhs ) {
        value += rhs.value;
    }
};

```

Figure 15: AC_BM search task

```

int main(int argc, char* argv[])
{
    // Initialization and read file...
    ...
    task_scheduler_init init;
    Search total;
    int grainsize;
    //Set the grainsize
    ...

    parallel_reduce( blocked_range<int>( 0,
textLength, grainsize), total );
    matched = total.value;
    printf("total match is %d\n", matched);
    return 0;
}

```

Figure 16: Main() function

```

Template<typename Range, typename Body>
void parallel_reduce(const Range& range,
Body& body);

```

Figure 17: Declaration of parallel_reduce()

Figure 17 shows the declaration of the parallel_reduce() function. A parallel_reduce performs parallel reduction of body over each value in range. Grain size specifies the biggest range considered indivisible, i.e., Intel TBB splits the range into two subranges, if the size of the range exceeds the grain size. A too-small grain size may cause scheduling overhead within the loop templates and counteract the speedup gained from parallelism. A too large grain size, on the other hand, may unnecessarily limit parallelism. For example, if the grain size is so large that the range cannot be split, then the application will not be parallelized. Therefore, it is important to set the proper grain size. Normally, setting grain size to 10,000 is high enough to amortize the scheduler overhead; the correct size also depends on the ratio between the task number and processor number.

From the result of the Intel Thread Checker, there are no data races and thread-safety errors. Figure 18 shows that there are no transitions or imbalances between different threads either.

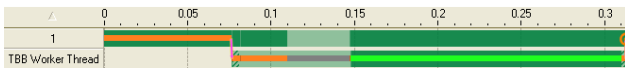


Figure 18: Thread Profiler Timeline view

Figure 19 shows the performance for different grain sizes when the test file is 32M. The application achieves the best performance when the grain size equals 8M.

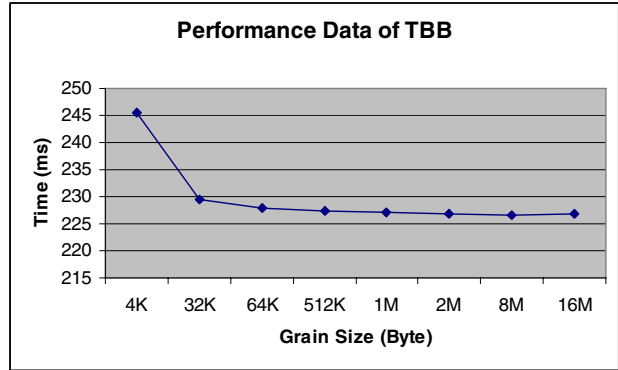


Figure 19: TBB performance of different grain sizes

RESULTS

Figure 20 shows the performance of Serial, Win32, and Intel TBB implementation of the ACBM search algorithm when the test file is 32M. Both Win32 and Intel TBB achieve better performance than the serial version.

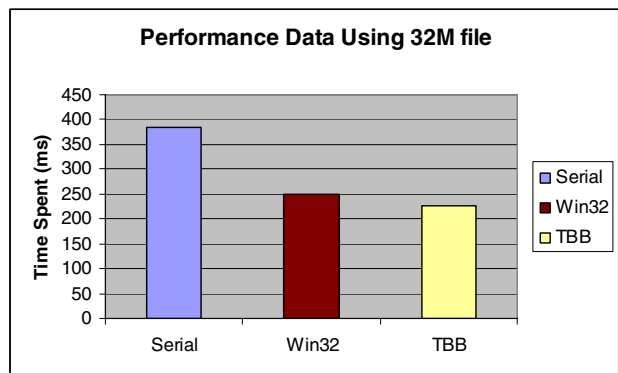


Figure 20: Performance data for different implementations

In order to evaluate the performance gain of parallelizing with Win32 and Intel TBB, we test the two applications with different file sizes and calculate the corresponding scalability. Figure 20 shows that both Win32 and Intel TBB attain better scalability as the file size grows. Intel TBB has better performance than Win32 for all the different file sizes. When the file size is 256M, $Scalability_{tbb} = 1.714$ and $Scalability_{win32} = 1.580$. Also, the average $Scalability_{tbb} = 1.655$, and the average $Scalability_{win32} = 1.549$. Considering the ideal scalability is 1.867, Intel TBB demonstrates good scaling performance on a dual core system. Since the AC_BM algorithm can be used in the IDS detection engine, which is the core module of an IDS, parallelizing the algorithm can significantly improve the performance of the IDS.

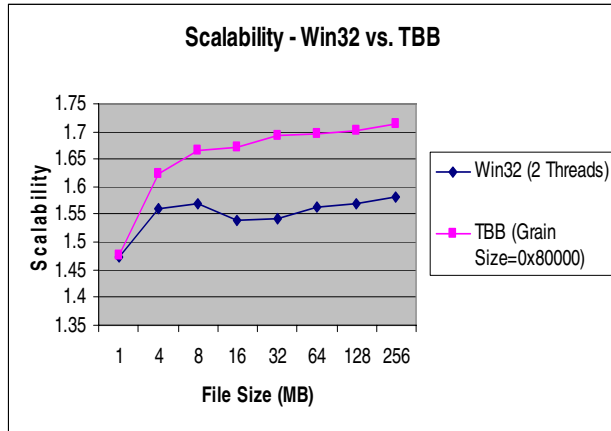


Figure 21: Scalability comparison between Win32 and Intel TBB

From the test results, we can see that Intel TBB demonstrates better performance than Win32 for the AC_BM algorithm. The reason is that with Intel TBB, we specify tasks instead of threads. A task can be assigned to a thread dynamically; Intel TBB selects the best thread for a task by using the task scheduler. If one thread runs faster, it is assigned to perform more tasks. However, with the Win32 Threading Library, a thread is assigned to a fixed task, and it can not be reassigned to other tasks even though it is idle.

From a developer's point of view, Intel TBB is easier to use. Intel TBB can dynamically decide the most suitable number of threads according to the platform and workload. Developers don't need to consider the granularity of the task by themselves. Further, when the application is migrated to another platform, Intel TBB automatically selects the best number of threads and assigns the tasks to the right threads to gain the best performance. Another advantage of using Intel TBB is that developers don't need to consider threading issues such as data race and inappropriate synchronization when they use the templates provided by Intel TBB, such as `parallel_reduce`, `parallel_while`, etc.

CONCLUSION

As multi-core processors become mainstream in the marketplace, software development needs to go parallel to fully exploit the performance potential of multi-core systems. For the rule-based IDS such as Snort, the parallel pattern matching algorithm in its detection engine can greatly improve the performance of intrusion detection, which in turn improves the performance of the overall IDS. As the IDS becomes more and more important in network security, and the efficiency of its detection engine contributes more to the overall performance of the IDS, it

is worth it to parallelize the detection engine to attain the greatest performance gain in a multi-core environment.

Intel Threading Analysis Tools help developers write high-performance parallel software in all of the phases of the development cycle. By parallelizing the Aho-Corasick_Boyer-Moore algorithm using both the Win32 threading API and Intel TBB, we show how to parallelize and tune applications with Intel threading tools. We also show how to apply the parallel methodology to the real application during the analysis, design, and tuning phases. After performance tuning, both threading methods achieve much better performance than their serial counterparts. The Intel TBB attains a greater performance gain compared to the Win32 threading API, and it provides a more generic threading model to ease the implementation of parallelism.

ACKNOWLEDGMENTS

Thanks to Albert Hu for providing great advice on the paper content and organization. Thanks to Fane Li also for providing technical support on the parallel methodology.

REFERENCES

- [1] Shwetha Doss and John O'Neill, Ph.D., "Best Practices for Developing and Optimizing Threaded Applications," *Go Parallel*, at <http://www.devx.com/go-parallel/Article/33534>.
- [2] Michael Voss, I., "Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms," *DevX.com*, at <http://www.devx.com/cplus/Article/32935>.
- [3] Ananth Grama and Anshul Gupta, *Introduction to Parallel Computing*, Addison Wesley, Chapter 3, Boston, 2003.
- [4] "Getting Started with the VTune™ Performance Analyzer," Intel Corporation, at <http://fm1cedar.cps.intel.com/softwarecollege/CourseDetails.asp?courseID=17>.
- [5] "Intel® Threading Profiler Getting Started Guide," Intel Corporation, at <http://fm1cedar.cps.intel.com/softwarecollege/CourseDetails.asp?courseID=179>.
- [6] "Intel® Threading Checker Getting Started Guide," Intel Corporation, at <https://fm1cedar.cps.intel.com/SoftwareCollege/CourseDetails.asp?courseID=178>.
- [7] "Intel® Threading Building Blocks Reference Manual," Intel Corporation, at <http://www3.intel.com/cd/software/products/asm-na/eng/294797.htm>.

[8] C Jason Coit, Stuart Staniford, Joseph McAlerney, "Towards faster pattern matching for intrusion detection or exceeding the speed of snort," *DARPA Information Survivability Conference and Exposition*, 2001.

[9] Brian Caswell and Jeremy Hewlett, "Snort User Manual," at <http://www.snort.org>.

AUTHORS' BIOGRAPHIES

Lihui Wang joined Intel's IT Flexible Services Group in 2006. She works on system and application software development for Intel product groups. Her current work is on Service Delivery Operation. She received B.S. and M.S. degrees in Computer Science from Sichuan University. Her e-mail is lihui.wang at intel.com.

Xianchao Xu joined Intel's IT Flexible Services Group in 2005. He is part of the Research Engineering team and works on the development of wireless and TXT research prototypes. He also works on enabling Active Management Technology (AMT) technology and has taken part in the development of AMT configuring software. He received his B.S. degree in Computer Science from the University of Science and Technology of China and his M.S. degree in Computer Architecture from the Institute of Computing Technology, the Chinese Academy of Sciences. His e-mail is james.xu at intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS® Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from <http://www.intel.com>.

Additional legal notices at:

<http://www.intel.com/sites/corporate/tradmarx.htm>.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm