Performance Analysis of Java Concurrent Programming: A Case Study of Video Mining System

Wenlong Li, Eric Li, Ran Meng, Tao Wang, Carole Dulong
Intel China Research Center
Intel Corporation
{wenlong.li,eric.q.li,ran.meng, tao.wang,carole.dulong}@intel.com

Abstract

As multi/many core processors become prevalent, programming language is important in constructing efficient parallel applications. In this work, we build a multithreaded video mining application with Java, examine the thread profiling information and micro-architecture metrics to identify the factors limiting the scalability, and employ a number of ways to improve performance. Besides, we conduct some thread scheduling experiments. According to the experiments and detailed analysis, we conclude that for this video mining application: (1) Java is a good parallel language candidate for many core processors in terms of performance, scalability, and ease of programming; (2) Thread affinity mechanism is effective in improving data locality, but brings little benefit to multithreaded Java application due to its conservative memory model in JVM.

1. Introduction

As Java emerges as one of the major programming languages for software development and parallel Java applications are widely adopted as standard benchmarks for commercial multiprocessor server, characterizing and tuning Java parallel application on SMP machine becomes increasingly important. Besides, using built-in thread support in language specification to parallelize applications is of great interest, [1] illustrates the necessary of thread support in language design. Java, a popular programming language on various platforms, provides a high performance and scalable concurrent utility classes for creating concurrent applications with great convenience in latest JDK1.5.

Rapid advances in the technology of media capture and storage has contributed to an amazing growth of digital video content. While content generation and dissemination grows explosively, how to help users efficiently search, browse and manage multimedia contents becomes more and more important, such as mining digital home photos and videos which will be huge volume in the near future. Both technology push (e.g., video analysis and data mining) and application pull (e.g., various digital library, video on-demand, interactive TV) have contributed to the proliferation of video mining, e.g., video shot detection, surveillance, and highlight event detection in real-time [2][3].

In this paper, we parallelize the shot boundary detection application in video mining system with Java concurrent programming, and perform characterization and performance tuning on SMP machine. Besides, we also investigate the effect of thread affinity mechanism in this study.

2. Related work

There are a lot of studies in behavior evaluation of single threaded Java application since its first introduction in 1995 [4]. Luo et al. evaluated the characterization of multithreaded Java server applications on Pentium III [5]. Karlsson et al. studied the memory system behavior of Java middleware applications on SMP system. Wei et al. characterized the performance of Java multithreaded applications on SMT processors [6]. Luiz et al. evaluated the memory system behavior of Database Management Systems running the TPC benchmarks [7].

Most of the multithreaded workloads used in these work are developed in JDK below version 1.5, in which the Java platform provides a set of basic primitives for writing concurrent programs. However, the built-in primitives, such as synchronized blocks, Object.wait(), and Object.notify(), are insufficient to develop many sophisticated programming tasks, which in return lead developers to implement their own high-level synchronization facilities, but given the difficulty of concurrency issues, their implementations may not be correct, efficient, or high quality.

In this paper, we extend previous work by examining the Java concurrent programming provided in latest Java 2 Platform on SMP machine and evaluating the effect of thread affinity mechanism in multithreaded Java

1-4244-0054-6/06/$20.00 ©2006 IEEE

application. The latest Java 2 Platform, Standard Edition release 5.0 (J2SE 5.0 or JDK1.5), which is also known as Tiger, provides a new multithreading method in Java programming language [8]. The original mechanisms for coordinating threads with wait() and notify() are now enhanced with new sophisticated mechanisms. It plays as a part of the java.util.concurrent package, which consists of thread-safe connections, semaphores, mutexes, thread pools, locks, and barriers.

3. Application construction

3.1. Introduction to Java

As an emerging language, there are many advantages of Java compared to the commonly used C/C++ language, such as automatic memory management, bound checking at compile time, and bytecode to enable porting across different platforms. With these advanced features and the well-known "write once run anywhere" property, Java has gained widespread popularity in many areas. In order to provide high performance for Java application, many researches are studied in literature to improve the efficiency of Java Virtual Machine and Just-In-Time (JIT) compiler. With the continuing improvement in JVM technology and JIT compiler, Java performance is now very competitive with that of C/C++. The JIT compilers, present in most JVMs, convert Java byte codes to native code with amazing efficiency, and in the latest generation (represented by Sun's HotSpot and IBM's JVM) they demonstrate the potential to start beating C/C++ performance for computation intensive applications [9].

By providing built-in threading language support, Java significantly eases concurrent programming. The java.util.concurrent package in JDK1.5 provides classes and interfaces aiming to simplify the development of concurrent applications by providing high quality implementations of common building blocks used in the concurrent applications and the parallel garbage collector keeps the overhead of automatic memory management within an acceptable level. Besides, the sufficient thread primitives provided in Java enable programmer to manage threads easier and more flexible than OpenMP, such as thread synchronization. Consider the common scenario in video/audio decoding: where two threads simultaneous decode frames from video and audio tracks, and they must wait for a universal synchronization event before moving on to process the next frame at a specified time interval. Following shows the code example.

```
Thread A                    Thread B
while (…)                   while (…)
{                           {
   Wait for Event C;           Wait for Event C;
   Execute TaskA;              Execute TaskB;
}                           }
```

If we try to parallelize the above code in OpenMP, we can use the if-then-else structure or the sections directive of OpenMP to assign the two different jobs to two threads, but have the difficulty of synchronizing these two threads, besides, the program is hard to read and maintain. In contrast, we can create two threads responsible for executing these two tasks, and use the wait()/notify() or CyclicBarrier utility in Java to synchronize them. The parallel version in Java is shown as follows.

```
Thread A                                  Thread B
Class A implements Runnable               Class B implements Runnable
{                                         {
  public void run()                         public void run()
  {                                         {
    while (...)                               while (…)
    {                                         {
      Wait (); // Wait for Event C             Wait();//Wait for Event C
       Execute TaskA;                          Execute TaskB;
    }                                         }
  }                                         }
}                                         }
```

3.2. Application construction

The background of our research is a video mining project intending to extract the highlights from a sport video in an automated, real-time and accurate way. As the foundation for the other high level modules such as scene/store segmentation and video structural summary, the shot boundary detection system should meet the high performance and the scalability requirements. We construct the shot boundary detection system in Java translated from the C++ code shared by Tsinghua University, who achieves the best results in the competition of TRECVID 2004 and 2005. Fig 1 depicts the framework of shot boundary detection system.

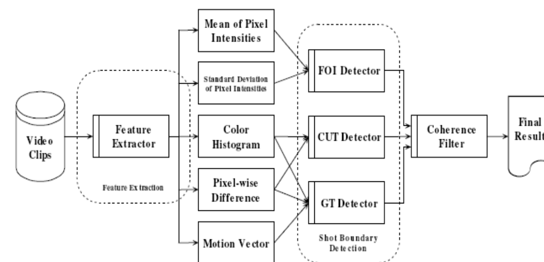

Figure 1. Overview of shot boundary detection system

In this framework, Java Media Framework (JMF) [10] serves as the video decoder to decode video frames sequentially, where the ***Demultiplexer*** class is used to split the video stream into audio and video tracks and the ***Codec*** class decodes the input video into consecutive frames with specific input and output format from the video track. After that, the low-level features are extracted from the decoded frames. This process repeats until all

frames are processed. In the end, all the features are fed into the shot detection module to output the final shots.

There are two different working scenarios to apply this application, i.e., the online and offline mode, where the online mode will capture the video stream either from the TV or the web broadcasting, and the offline mode will operate on existing raw media files.

3.3. Parallel study

Apparently, the shot detection system can be partitioned into three phases, i.e., video decoding period, feature extraction upon all the decoded frames, and the final shot boundary detection. The execution time breakdown of these there modules indicates the former two modules are most time-consuming, which constitutes around 5%-25% (in video decoding) and 75%-95% (in feature extraction) of total time respectively depending on the features used. In current implementation, the decoding phase is about 8% and feature extraction is about 92%. The shot boundary detection module is extremely fast, which is not worth parallelization.

There are many opportunities to exploit thread level parallelism at different granularities in shot detection application. In order to achieve the optimal speedup over its well-tuned sequential code on multiple processor system, we present our parallel considerations in this section.

In general, multimedia application tends to use data rather than functional parallelism to fully take advantage of its natural data independencies. In the shot detection system, both video decoder and the feature extraction module have abundant data parallel opportunities. In video decoder, the straightforward way is to exploit the parallelism at the Group of Picture (GOP) or slice level [11], while for feature extraction, the decoded frames are independent with each other, and hereby, can be processed simultaneously. In contrast, though functional level parallelism is also interesting, e.g., different functions, like IDCT, MC, VLD etc. in video decoding, and different features working as the basic data processing unit, it suffers a lot from the load imbalance among different threads and cannot provide enough scaling performance with a large processor number.

3.3.1. Data splitting scheme

As aforementioned, it is much easier to parallelize shot detection through data level parallelism. Naturally, splitting the input raw video data into a number of smaller chucks is a straightforward way to express this data level parallelism. In this scheme, each thread uses the same routine as the serial version, decodes one chuck of data, and extracts features upon the decoded frames without little interference with each other. Since the raw data stream is split manually and each thread has to find the new semantic picture start code, we must pay attention to the neighbor threads to guarantee the consistence of decoded frames.

Generally, we set the segmented data file number to the amount of processors in the multiprocessor system, to follow a simple static assignment policy. For example, if there are totally four threads and the video file time length is ten minutes, each thread will be assigned with a 2.5 minutes data stream respectively. Though introducing more small data files may solve the load imbalance in the static scheme, it suffers a lot from the additional video stream parsing overhead, and more synchronizations incurred with smaller task granularity.

Apparently, this scheme can only be applicable in the offline mode, which assumes the raw video data is already obtained, and can be accessed from any position of the file. Therefore, the online real-time mode will get definitely no benefit through this parallelization scheme, which motivates us to find some appropriate schemes to handle both the online as well as the offline mode, and keep the equivalent scaling performance.

3.3.2. Task level parallelism

In contrast to the direct data splitting scheme, we look into the modules of the shot detection system and analyze the working pattern among the two modules. The video decoder works very similar to a task producer, generating a sequence of video frames, while feature extraction, on the other hand, reads in the decoded frames and looks like a task consumer, and the frames are considered to be a number of tasks accordingly. Obviously, it complies well with the well-known producer-consumer threading model. The master thread puts the decoded video frames into a shared buffer, and the feature extraction worker thread fetches the decoded frame from this buffer and extracts necessary features for the following shot boundary detection module. When the frame buffer is full, the master thread is suspended, and will be waked up when there are available free slots in the shared buffer. Typically, the worker threads will be set equal to the number of processors to avoid the under-utilizing the computing power, and overcome the potential load imbalance between the decoder thread and worker threads.

In our Java implementation, we use a ***taskQ*** model to parallelize the shot detection application. Video decoding is processed in a master thread and the feature extraction of each decoded frame is encapsulated in a ***task*** and pushed back into the ***taskQ***. These ***tasks*** are dynamically executed concurrently in worker threads. In this work, we use the utilities in Java.util.concurrent package to implement this scheme, where ***task*** is defined as a class that implements the Runnable interface, and

LinkedBlockingQueue is used as the ***taskQ*** to hold all the ***tasks***. The access to ***taskQ*** is protected by the "lock" primitive. Fig 2 illustrates this execution model, where one thread executes the ***taskQ*** block in single-thread manner, conceptually enqueuing each ***task*** it encounters, while all worker threads dequeue the ***tasks*** and execute them from this queue. The ***task/taskQ*** model in Java is very like taskQ OpenMP extensions provided by Intel library [12], but encapsulated in the language support, which brings a lot of convenience for the parallel application development.

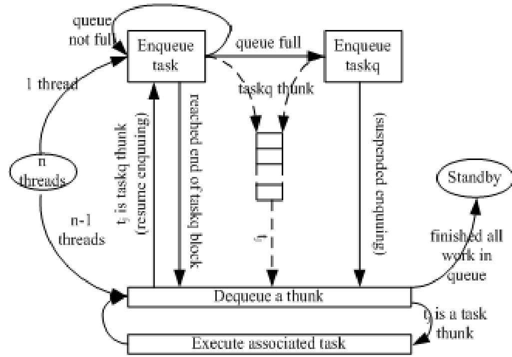

Figure 2. Execution model of TaskQ scheme

In addition to the master-slave threading model, another interesting scheme - pipeline based parallel scheme is also discussed here. In this scheme, all the working threads are treated equally without distinguishing the decoding and feature extraction threads explicitly. They all follow the same executing pattern, where each thread will be responsible for one frame video decoding and feature extraction, and strictly observe the timing dependency, e.g., as displayed in Fig 3, thread *k* has to wait until previous thread *k*-1 finishes one frame decoding, and initializes its own decoding procedure to get the next frame sequentially.

When comparing these two task level parallel schemes, we find they almost exhibit the same performance, though their underlying parallel mechanisms are quite different. They both have to keep some shared resources, and must respect the video decoder timing dependencies. Finally, we choose the ***taskQ*** model, to fully utilize the new parallel primitives provided by JDK 1.5.

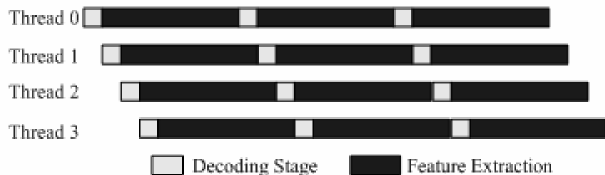

Figure 3. Execution model of pipeline based scheme

After analyzing the possible parallel schemes, we would like to study their scalability performance. In theory, due to the two schemes highly depend on the video decoder, which essentially runs in the critical path during the whole parallel period. Therefore, the theoretical speedup depends on the time ratio of feature extraction and video decoding module, e.g., the ratio for the MPEG-1 video stream is 98:12, indicating that the maximal speedup is 8.17 according to Amdahl's law. However, since the scaling data is estimated upon the serial video decoder, accelerating the video decoder will directly solve this problem. As discussed earlier, slice level parallelism can balance the granularity and the parallel efficiency, and provide enough parallelism to the shot detection system [13].

4. Experimental results

4.1. Experimental setup

We use latest JDK1.5 and JMF2.1 provided by Sun Corp. to construct the shot detection application. All experiments are conducted on a 4-way 2.8G Intel Xeon Hyper-Threading enabled shared memory system. Each processor is equipped with 8KB L1 data cache, 12KB trace cache, 512KB L2 unified cache, and 2MB L3 unified cache. The operating system is Windows 2003 Server Data Center, and the JVM is the Sun ***Java Runtime Environment*** (JRE) 1.5.0_04 enabled with server runtime compiler. The input data are all MPEG-1 video stream from the TRECVID data suite [14].

In order to compare the performance of Java and C++ for this video mining application, we also implement a C++ shot detection application, where we use IPP [15], a highly optimized routine for video and audio processing, to construct this system. Experimental result shows Java achieves a comparable performance as C++ in this application.

For performance characterization and thread profiling, we use Intel VTune performance analyzer [16] to measure different micro-architectural metrics and JProfiler to monitor the thread running and interaction [17], respectively.

4.2. Detailed characterization

4.2.1. Impact of multithreading

Java is a multithreaded application (Besides the application thread running on top of JVM, there are some helper threads existing inside JVM, such as ***garbage collection*** thread responsible for recycling the un-used heap space), we vary the number of application threads through adjusting the number of worker threads to study the impact of multithreading on application performance and cache system performance. Fig 4 shows the speedup with increased number of application threads to qualify

the application's scalability performance. All the scaling data are normalized with respect to the serial application running on the same system. The "Base" column in gray represents the parallel implementation without any performance tuning.

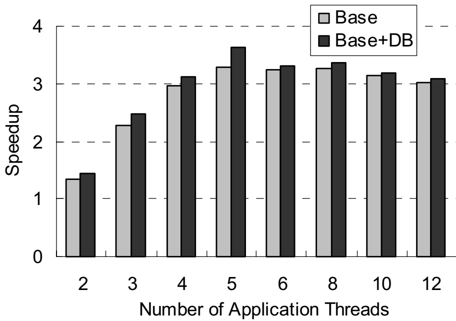

Figure 4. Speedup with increased number of application threads

It can be easily observed that, with the increase of application threads, the performance first goes up and saturates at 5 threads, where all the threads work concurrently to fully utilize the system resource with very little contention. Particularly, in the case of 5 threads, the CPU utilization rate achieves near 100%, which indicates most of the execution time is consumed by the application. However, when the thread number exceeds 5, we can notice a steady performance degradation, which is caused by the increased overhead in thread synchronization and scheduling. Fig 5 depicts the execution time breakdown between system and user activities, all the data are normalized to the baseline - 2 application threads. The behavior is consistent with the speedup performance, where the application spent the lowest OS time on 5 threads.

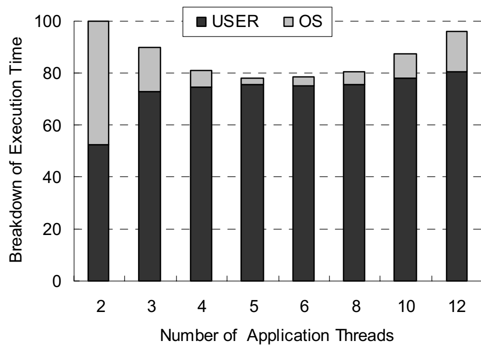

Figure 5. the Execution time breakdown with increased number of application threads

Besides the scaling performance, we also study the cache behavior on 4-way SMP system. Fig 6 presents the L1 cache misses with different processors, where all the data are normalized to 2 application threads. L1 cache misses increase a little with the thread number, while L2 and L3 cache misses remain flat, revealing that the application is not very sensitive to the thread number due to the regular data access pattern and the relative large capacity the L2 and L3 cache.

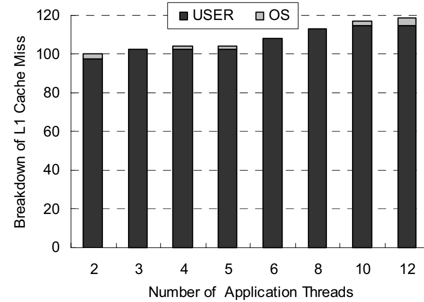

Figure 6. Changes of first level data cache miss with increased number of application threads

When we further investigate how the L3 miss is distributed, we find a large majority of the cache misses come from the feature extraction module. It works on different sets of image and therefore, the data cannot fully fit in the LLC cache and violate the premise of locality of reference. We use data blocking to improve the cache locality by segmenting the whole image into a sequence of strips. Each subset of the large data is processed before moving on to the next one. Upon the completion of the feature extraction of these strips, the final results will be merged together. Fig 4 shows the performance comparison between the shot detection with and without using the data blocking technique, generally we achieve around 5~10% performance improvement over the non-data blocking version. Particularly, we notice the parallel application gets even more benefit with the increase of processor number. The reason can be contributed to the alleviation of the memory bandwidth contention through reducing the LLC cache misses. In the rest of the paper, we will only use the data blocking enabled application for further study and analysis.

The previous analysis on the application performance and memory system indicates that the number of threads affects the scalability of the system, processor resources utilization, and memory system performance. Besides this, we also study the impact of multithreading on instruction cache and branch prediction performance. Due to contention from multiple threads, instruction cache miss rate increases as increasing application threads, but its effect to performance is insignificant. L1 instruction miss is smaller, below 2 trace cache misses per 1,000 instructions. We also collect ***Instruction TLB*** (ITLB) misses for this multithreaded application. ITLB is responsible for translating instruction address into physical address to access L2 cache when the trace cache miss occurs. The ITLB misses are very small, about 0.07 per 1,000 instructions. For branch prediction

performance, the mispredicted branches per 1,000 instructions are 1.01, which is not a dominant factor to performance.

4.2.2. Scalability analysis

To understand the scalability limiting factors, we characterize the parallel performance from the high level general parallel overheads, e.g., synchronizations penalties, load imbalance, and sequential area, to the detailed memory hierarchy behavior, e.g., cache miss rates and FSB (front side bus) bandwidth.

JProfiler [17] is a useful tool to analyze the Java performance bottlenecks by providing the thread intervention information in timeline, where the critical path of the application and all the threads status can be visualized through its GUI interface. We use JProfiler to collect some general parallel factors, such as the time spent in the sequential area, load imbalance among the working threads, to roughly capture the application's scaling performance. Fig 7 shows the sample of thread historical view with 5 application threads. The sequential portion and load imbalance are relatively small; however, we can noticeably observe the thread synchronization and scheduling overhead (marked as waiting state). According to the statistical data, the percentage of parallel, sequential and thread synchronization/scheduling region are 86%, 3% and 11% respectively. Therefore, the theoretical maximum speedup is expected to be 3.9 according to Amdahl's law [18].

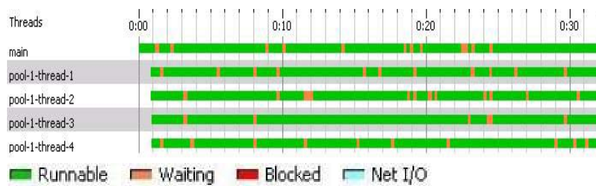

Figure 7. Thread running history view with 5 application threads

However, due to the thread synchronization and scheduling overhead, the real speedup we obtain is 3.62. These overhead is expected to possibly increase when available threads increase, especially when thread number is more than available processor count. Fig 8 shows the speedup of our parallel system in different number of processors. Our parallel implementation achieves a near linear speedup.

Besides the general scalability performance factors, memory subsystem also plays an important role in identifying the scaling performance bottlenecks. We profile the application with VTune, and performance metrics are chosen to be different level cache misses and system memory bandwidth. Fig 9 shows the L1 cache misses, L2 cache misses, and L3 cache misses per 1,000 instructions, respectively. It is interesting to see that the cache miss rates vary little with the number of processors. Though data structures in our system are large in size, not to fit in L3 cache, data blocking optimization, together with the regular data access pattern deliver very good cache performance in our application.

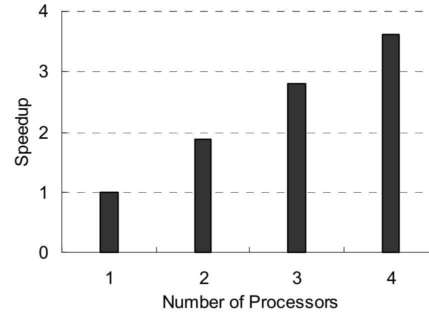

Figure 8. Speedup

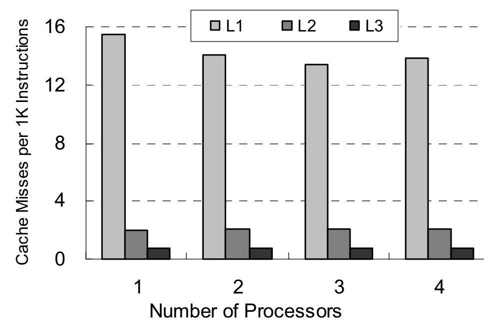

Figure 9. Cache misses per 1,000 instructions

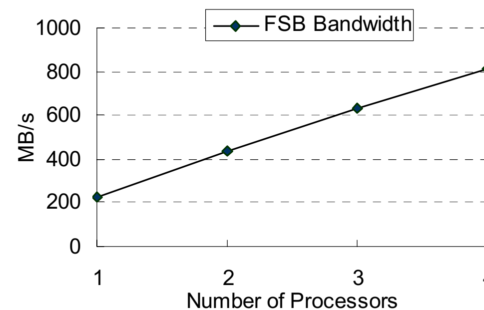

Figure 10. FSB bandwidth

Generally speaking, memory bandwidth is also a key factor which may potentially limit the speedup on shared memory system. However, due to the intrinsic data independencies among the worker threads as well as very low data sharing between the master thread and worker threads, the cache coherency traffics are tremendously reduced; coupled with the high cache locality performance on the L2 and L3 cache, it has a much lower memory bandwidth requirement. Fig 10 shows how the bus bandwidth utilization varies with the number of

processors. For all data inputs, the bus bandwidth increases linearly with the number of processors, but far from the saturation (3.2GB/s) even with 4 processors.

4.2.3. Thread scheduling

Since Java itself is a multithreaded application, thread scheduling, an essential component in JVM, plays an important role in driving the application to a high performance [19]. There are already a lot of studies on thread scheduling, for instance, Robert [19] gives a comprehensive study on different scheduling techniques and concludes that OS decision has a significant impact on the performance with a relatively large processor count.

On a multiprocessor system, when threads migrate from one processor to another, it will incur thread context switch penalty, data cache coherency traffic, and even false sharing where one cache line appears to be shared by multiple processors. In this work, we use thread affinity mechanism by binding threads on the same processor to take advantage of the cache locality. However, experiment shows this application benefits from this scheduling technique by less than 1%.

The inefficiency of hardware thread affinity in Java lies in one possible reason. In order to port applications in various platforms, Java assumes a conservative memory model [20]. When synchronizing on a monitor (lock), the JMM requires that the cache of current processor be invalidated immediately after the lock is acquired, and flushed (writing any modified memory locations back to main memory) before it is released. Although the number of synchronization requests in our application is small, the cache flushing to memory triggered by JVM counteracts the improved data locality benefit from thread affinity. To overcome the limitation, several proposals have been presented to support as many possible memory models to accommodate practical architectures [21].

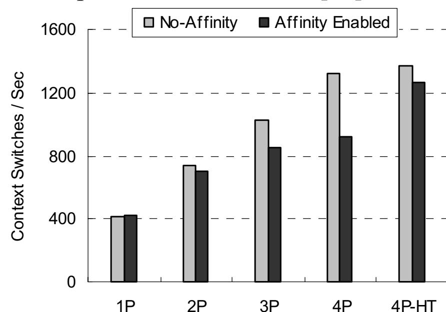

Figure 11. Context switches per seconds

On the other hand, as displayed in Fig 11, we find thread context switching increases significantly with the number of active threads. With binding the active application threads to the ideal physical processor, the thread context switches decrease steadily by preventing thread migration among different processors. Moreover, the Hyper-Threading enabled system can also benefit from thread affinity, not only reduce the number of thread context switches, but also avoid overlapping two active working threads on two logical processors residing on same physical processor.

To summarize, thread affinity can not only improve the data locality by reducing thread migration, but also reduce resources contention on Hyper-Threading enabled multiprocessor system by binding threads to different physical processors to avoid the overlap of threads on the same physical processor.

5. Conclusion

This study uses a video mining case to examine the performance of Java concurrent programming on Intel 4-way shared memory multiprocessor system , study the impact of multithreading on performance and memory system behavior, and investigate the performance of thread affinity mechanism. Our research on these aspects draws some similar conclusions as previous ones, and reveals some new observations: (1) it is easy to develop multithreaded application with Java concurrent programming; (2) The multithreaded Java application scales well; (3) thread synchronization and scheduling overhead is the dominant factor to prevent Java from achieving perfect scalability; (4) thread affinity mechanism is a good candidate for improving data locality by reducing thread migration and resources contention on memory hierarchy.

For future work, enhancement to thread scheduling in JVM and operating system are expected to mitigate the thread synchronization and scheduling overhead.

6. References

[1] H.J. Boehm, Threads cannot be implemented as a library. In ACM SIGPLAN conference on programming language design and implementation (PLDI), 2005

[2] C.W. Ngo, H.J. Zhang, T.C. Pong, Recent advances in content-based video analysis. International Journal of Image and Graphics, vol.1(3), pp. 445-468, 2001.

[3] S.W. Smoliar, H. Zhang, Content-based video indexing and retrieval, IEEE Multimedia, vol.1(2), pp.62-2,1994.

[4] B. Rychlik, J.P. Shen, Characterization of value locality in Java programs. In Proceedings of the 3th workshop on workload characterization, Austin, TX, 2000

[5] L. Yue, J.K. Lizy, Workload characterization of multithreded Java servers. In Proceedings of 2001 IEEE international symposium on performance

analysis of systems and software, Tucson, Arizona, 2001

[6] W. Huang, J. Lin, Z. Zhang, J.M. Chang, Performance characterization of Java applications on SMT processors. In Proceedings of 2005 EEE international symposium on performance analysis of systems and software, Austin, Texas, 2005

[7] L.A. Barroso, K. Gharachorloo, E. Bugnion, Memory system characterization of commercial workloads. In Proceedings of the 25th annual international symposium on computer architecture (ISCA), 1998

[8] Sun Corp,. Concurrent programming with with J2SE5.0. http://java.sun.com/developer/technicalArticles/J2SE/concurrency/

[9] Sun Corp, Performance enhancement. http://java.sun.com/j2se/1.5.0/docs/guide/performance/

[10] Sun Corporation, Java Media Framework API (JMF). http://Java.sun.com/products/Java-media/jmf/

[11] A. Bilas, J. Fritts, J.P. Singh, Real-time parallel MPEG-2 decoding in software. In Proceedings of the 11th international symposium on parallel processing, 1997

[12] E. Su, X.M. Tian, M. Girkar, et, Compiler support of the workqueuing execution model for Intel SMP architectures. In the fourth european workshop on OpenMP (EWOMP), 2002

[13] M.J. Holliman, E Li, Y.K. Chen, MPEG decoding workload characterization, in CAECW 2003

[14] J.H. Yuan, W.J. Zheng, L. Chen, et, Tsinghua University a TRECVID 2004: shot boundary detection and high-level feature extraction, 2004.

[15] Intel Corp, Intel® Integrated Performance Primitives, http://www.intel.com/software/products/ipp

[16] Intel Corp, VTune performance analyzer. http://www.intel.com/software/products/vtune

[17] ej-technologies GmbH. Java Profiler – JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html

[18] Y. Shi, Reevaluating Amdahl's law and Gustafson's law. http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html

[19] R.C. Kunz. PhD dissertation, performance bottlenecks on large-scale shared-memory multiprocessor. 2004

[20] IBM Corp, Technology article. Synchronization is not the enemy. http://www-128.ibm.com/developerworks/Java/library/j-threads1.html

[21] IBM Corp, Technology article. Java theory and practice: fixing the Java memory model. http://www-128.ibm.com/developerworks/library/j-jtp02244.html