ASOSI: Asymmetric Operating System Infrastructure

Yair Wiseman
Computer Science Department
The Open University of Israel
Raanana, 43107
ISRAEL
wiseman@openu.ac.il

Keywords: Operating Systems, Process Scheduling, Parallel and Distributed Systems, Memory Management.

ABSTRACT

At the present time, when the usage of super-computers has been brought into play and turned out to be more widespread, the research of Asymmetric Operating Systems has been gathering speed, because super-computers more often than not run an Asymmetric Operating System. In this paper some new algorithms and implementations designed in our group and based on our ASOSI framework are presented.

1. INTRODUCTION

Asymmetric Operating Systems [1] are designated for systems containing a cluster of computers. In such systems, there is one "master" computer, whereas the others are "slaves". The "master" sets the processes' scheduling of the "slaves". In such a manner most of the scheduling activity is done by the "master". However, the "slaves" also have an important responsibility. They inform the "master" about the nature of the running jobs. Based on these reports, the "master" can decide more efficiently about the scheduling resolution.

Distributed and parallel computing machines [2] put forward enhanced processing ability for their consumer by enabling the employment of several processors for each of an application' jobs. The global performance of such parallel/distributed systems is typically gauged by their job throughput, response time, and job wait time [3].

It is obvious that parallel systems strive to make the most of their resources, to augment throughput and to reduce response times. The various jobs are different in their computing resource consumption; therefore, the order in which the jobs are scheduled by the operating system and the manner they consume the system resources, have an effect on the overall functioning of the computing system.

The scheduler is a function of the operating system that admits jobs to the processors. The operating system also assigns other required computing resources for the jobs by other functions. The algorithms employed by the operating system with the intention of making a decision how to assign jobs to the system's processors influence the performance of the system.

When this issue is discussed with regard to super-computers, the challenge becomes even more intense. The price of a super-computer is usually very high, so a purchaser of such a system strives to obtain the best possible performance. Each additional job which the system is capable to execute is of advantage to the system purchaser.

Finding the optimal scheduling is impossible, because the operating system cannot know in advance what the needs of the processes will be. Gang scheduling [4] is a scheduling algorithm that strives for a better performance of parallel and distributed systems. The Gang Scheduling algorithm facilitates simultaneous scheduling of multiples jobs on the system's processors. Afterward, the group of the executed jobs will be switched after predefined time slices.

These activities, needless to say, put in additional computational and resource overheads on the operating system, that in charge of the general system's resources [5]. With the aim of making the Gang Scheduling algorithm applicable, systems must pay attention to these concerns. Consequently, several enhancements have been developed over the years. The enhancements come into being either by a deeper examination of the jobs' characteristics or by accompanying the fundamental algorithm with supplementary scheduling functions.

2. PAIRED GANG SCHEDULING

As has been mentioned one of the most important components in the operating system is the scheduler. The role of the scheduler is determining which process the CPU serves at any time. Asymmetric Operating Systems are designated for systems which contain a cluster of computers. There are several ways to implement the operating system scheduler. In this section we will introduce our scheduler.

When a computer cluster is used to run several parallel jobs concurrently, there are well-known performance benefits to be obtained if the process scheduling is coordinated so that all the processes of each parallel job run at the same time. Currently one of the most popular schemes for coordinated scheduling is Gang Scheduling. Gang scheduling enables processes in the same job to run at the same time. This usually yields better performance for communicating processes; however, there are many problems associated with conventional gang scheduling, such as wastefulness in resource employment and job performance, which have slowed down its widespread embracing.

Usually, using gang scheduling provides better performance for compute-bound communicating processes [6]; however, I/O-bound processes bring about the CPUs to be unoccupied at a significant percent of the time, whereas there are other processes that yearn for being executed. At one fell swoop, the influence on the disk behavior is the reverse: I/O-bound processes retain the disks full of activity, whereas compute-bound processes make the disk idle. As a matter of fact, it is not easy to keep upright the balanced use of the CPUs and the disks in applications that have large computation and I/O needs [7].

The focal point of Gang Scheduling is assigning as many processors to an application as are required at the same time. If this assignment is succeeded, it will allow the application to prevent processes from being blocked while they are waiting for the communications with other processes to come to an end, since it is assured that the looked-for process is running and making progress, so it is reasonable to wait for this process and since there is nothing else to be executed on the processor because all the job's processes are assigned.

Essentially, if the scheduler assigns two or more processes to each processor, it may cause circumstances where one process has to wait for another process to be rescheduled, because it is not currently being executed. Therefore, Gang Scheduling does its best to get the most out of processors for the current job, at a potential sacrifice of global system performance demotion.

Another option for scheduling rather than Gang Scheduling can be using the local scheduling autonomously on each processor of the cluster. The local scheduler can be any algorithm such as Round Robin, or a priority-based algorithm such as the LINUX scheduler or UNIX scheduler. If this alternative is chosen, a process that has to wait for another process should be blocked, because the awaited process is most likely not being executed and the processor has more important tasks to execute rather than keeping itself executes loops of busy wait.

Practically, local scheduling prioritizes global system performance, at the potential price tag of harming the performance of jobs that execute many communication operations; however, the added context switches induced by the fine grain communication may cause an extra overhead and an improper use of the processors [5].

If the scheduler is able to identify the behavior of every gang, this information can be utilized to facilitate a balance between the CPU-bound processes and the I/O-bound processes and keeping both the CPU and the disks busy.

ASOSI implements the scheme of matching pairs of gangs [8,9], a compute-bound gang and an I/O-bound gang. The motivation for such a matching is that such gangs will almost not interfere with each other's resource consumption, as they make use of different devices; hence, these gangs will be of the opinion that they are unaccompaniedly executed in the system. If the I/O execution time is not negligible in the processor time, an overlap of the I/O execution along with the processor activity may produce a better functioning [10].

Paired gang scheduling endeavors to find the middle ground between the scheme of gang scheduling and the scheme of local scheduling, with the purpose of make the most of the system resources without causing meddling with the processes of different jobs. It meets both of the schemes halfway. On one hand the processes will not wait a long time because a process requiring the processor during most of its execution will be matched with a process that in most of its execution requires an I/O device, so they will not meddle with each other's needs. On

the other hand, the processor and the I/O devices will be kept busy if there are jobs that are necessitated to be executed.

3. PARALLEL JPEG DECOMPRESSION

Another implementation that was developed on the ASOSI framework is the parallel JPEG decompression system.

JPEG[11] is a lossy image compression method. JPEG compresses the images in several steps. In a first step, the picture is split into a sequence of blocks of size 8X8 pixels. Each block is then compressed by the following sequence of transformations:

- Applying a *Discrete Cosine Transform* (DCT) [12] to the set of 64 values of the pixels in the block.
- Applying *Quantization* to the DCT coefficients; thereby producing a set of 64 smaller integers. This step causes a loss of information but makes the data more compressible.
- Applying an *entropy encoder* to the quantized DCT coefficients. Baseline JPEG uses Huffman [13] coding in this step, but the JPEG standard specifies also the arithmetic coding [14] as a possible alternative.

The decompression process simply reverses the procedure and its order. It first applies Huffman decoding, then dequantizes the coefficients and finally uses an inverse DCT to obtain the original set of values. Because of the quantization step, the reconstructed image includes only approximated values.

In [15] we describe an innovative idea on how to split the decompression task of JPEG images. The results of the paper compare between sequential decompression and parallel decompression. We start by splitting the image into several portions and assigning different processors, each working on a different portion of the image. The synchronization problems mentioned in [16] appear here as well and even more harshly. Not only the beginning of the block to be decoded by a specific processor unnecessarily corresponds with the beginning of a Huffman codeword, but even if it begins, synchronization is not guaranteed, because the block boundary could be located within the codeword representing the length of the DC coefficient or the block boundary could be located within the stored DC value at the beginning or the block boundary could be located within a codeword representing a pair used for the AC coefficients or the block boundary could be located at the beginning or within a stored AC value. Just if the block starts with a codeword for the length of the stored DC value, the block will be correctly decoded.

The processor would attempt to recognize a Huffman codeword representing the length of a DC value and would thus probably erroneously interpret the first bits; however, as can be seen in [16], the first few decoded elements are usually wrong, but typically a synchronization point is found almost immediately, after which the decoded items are correct.

This idea can be implemented in a cluster of computers. In [15] some tests on an SMP machine are presented. SMP machines obviously have a fast connection and a connection between all the processors is provided; however, actually, our algorithm does not require a connection between all the computers and some weaker topologies can be applied too.

The results can be also executed on ASOSI and any other cluster with faster communication cards. We can change the Operating System, so the topology of the cluster will be actually different. Also, another issue that has been coped with is the large amount of data that is read to the memory for each JPEG chunk calculation. Therefore, we have used Super-Pages. Super-Pages are a development of the famous paging notion. Super-Page is larger page that is pointed to by the TLB [17]. Multimedia applications like the parallel JPEG decompression application frequently have large chunks of memory that are clustered in few areas. Such an application can benefit Super-Paging very much [18].

We suggested the AMSQM algorithm [19] for handling the Super-Pages. This algorithm employs a reservation-based technique, in which segments are reserved for a super-page at the page fault time and a promotion is made whenever the number of the base pages within a super-page arrives at a predefined promotion threshold. The algorithm strives to have the possibility for a partially populated super-page to be promoted, so the pronouncement of a super-page candidate's reservation preemption or a swapping out a super-page candidate's base-pages is made as a result of the super-page "recency" in the page lists and not as a result of the number of resident

base-pages that the super-page is currently consisting of. This feature makes AMSQM accomplishing a higher TLB coverage and in addition a better page fault ratio.

4. SCALABLE PARALLEL COLLISION DETECTION

The ASOSI was also a framework for a parallel collision detection application. A number of implementations for parallel collision detection have been developed over the years [20,21]. The implementations typically completely depend on the parallel infrastructure. Minimizing the dependency will significantly increase the scalability of the implementation [22]. Also, the dependency can harm the portability of the simulation. Thus, we implemented a scalable and portable parallel algorithm for collision detection simulation that will be suitable to any infrastructure, even with a small support for parallelism [23].

The focal point of the proposed implementation is keeping the scalability approach while not leaving behind the locality principle and the load balancing of the cluster.

A common algorithm for Bounding Volumes hierarchy can be employed for testing out of an intersection of two models or a collision. Let us call the minimum "work unit" for one course of action e.g. collision detection of a complex geometry model or one course of action of two complex geometry models.

As a matter of fact, the proposed implementation employed a low-grade split into bigger units unlike the author of [24] has suggested; however, the execution time of one "work unit" that we propose is still not big, even if the geometry model is multifarious. Experiments show that if the implementation splits the geometry models into overly tiny units, too much overhead can be generated.

Let us call "processing unit" for one process that gets some portions of the collision detection course of action and sends back the outcome to the master process. Any process in ASOSI can migrate from one processor to an alternative processor in the same SMP or migrate from one node to an alternative node in the same cluster.

The algorithm employs the Vector Space technique [25] to discover similarity of scenarios ("work units") and processors ("processing units") like the technique of queries in document sets in the Information Retrieval research area.

For any specified complex geometry models, the implementation can become aware of an intersection in a short execution time. The proposed implementation reduces the preliminary overhead of a parallel collision check between complicated geometries on a computer cluster like ASOSI. The overhead is reduced by minimizing the dependency of data transfer augmentation and by reducing the number of processing units in the cluster. Consequently, the suggested implementation scales up in a good way in respect to the cluster size and the geometries size, whereas standard implementations do not succeed to scale up suitably. Decreasing the number of clients' memory allocation is another advantage of the proposed implementation. This reduction lets the implementation have the possibility of being put into operation on many various parallel infrastructures and ASOSI is just a case in point.

Our work about compressing the transferred information in the communication channel [26] can be also integrated into this project in order to facilitate a reduced transfer time.

5. DISTRIBUTED SHARE MEMORY

A more general application that was developed on the ASOSI framework is a common support for distributed shared memory. Many researches about Distributed shared memory [27] has been published and many changes for the better have been made over the years [28]. E.g. in [29] the author suggests a way for several Distributed Shared Memory applications to run their Distributed Shared Memory functions in 10%-30% of PVM run time.

Unexpectedly, using Distributed Shared Memories in clusters is not as widespread as it we would have anticipated. Besides the explanations listed in [30], the requirement from programmers to study a new Distributed Share Memory scheme for each Distributed Shared Memory system and the intricacy of revision of SMP applications like the application presented in [31], into Distributed Shared Memory programs have caused this neglection.

An analogous concern has come about in electronic mail systems. Many standards had been wandered around before the MIME format [32] was accepted as standard. After the acceptation of the MIME standard, email usage was significantly greater than before.

In ASOSI we propose a technique of prevailing over the difficulty of nonstandard Distributed Shared Memory by making the same inter-process communication interfaces that the programmers are familiar with on SMP machines available [33].

Several techniques for easier remote objects like semaphores have been introduced e.g. [34]. However, those techniques keep distributed applications very similar to the original application. In ASOSI we suggest that each IPC will be autonomous, i.e. not bundled within other IPCs like scores of existing Distributed Shared Memory systems use to do. This autonomy is of the essence because IPCs are employed not only for a safe shared memory use, but also for other synchronization tasks, as they are employed in SMPs.

As was mention below, scores of implementations of Distributed Shared Memories have been built up over the years. All the implementations take into consideration, the performance concerns of Distributed Shared Memory more than other concerns and specially neglect the portability of applications and standard API. All the Distributed Shared Memory systems have APIs that suit only a particular system and require that the programmer will familiarize himself to these particular APIs.

Nowadays, a migration of an application from one system to another will require a significant amendment of the application. Furthermore, applications that have been written for SMPs need to be rewritten to facilitate a scale up of the application from an SMP to a cluster of computers. The ASOSI framework shows a conception of a distributed system, in which an API for any IPC system will be indistinguishable or at least very similar to the standard SMP's API [35]. This concept can habituate programmers to distributed systems straightforwardly and in addition this concept will get better the integration of current SMP applications to clusters.

6. CONCLUSION

ASOSI - an Asymmetric Operating System Infrastructure has been presented. ASOSI supports many applications in a wide spectrum of parallel and distributed disciplines. The paper shows the important aspects of ASOSI for the various applications. We believe that ASOSI can be very beneficial for many more applications; especially for applications with massive parallelism and extensive computing requirements.

7. ACKNOWLEDGEMENT

The author would like to thank the graduate student students in his research group – Moshe Itshak, Ilan Grinberg, Moti Geva, Oshi Keren-Zur and Pinchas Weisberg for their help.

The author would also like to thank SUN Microsystems for their donation.

8. REFERENCES

[1] S. Muir and J. Smith "AsyMOS - An Asymmetric Multiprocessor Operating System", Proceedings of IEEE Conference on Open Architectures and Network Programming, (OPENARCH '98), San Francisco, pp. 25–34, 1998.

[2] D. G. Feitelson and L. Rudolph, "Parallel job scheduling: issues and approaches". In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), pp. 1-18, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[3] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling ". In IPPS'97 Workshop Job Scheduling Strategies for Parallel Processing, Geneva, Switzerland, April 1997.

[4] J.K. Ousterhout, "Scheduling techniques for concurrent systems". In 3rd Intl. Conf. Distributed Comput. Syst., pp 22-30, Oct 1982.

[5] A. Hori, H. Tezuka, and Y. Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. Job Scheduling Strategies for Parallel Processing, LNCS 1459:217-230, 1998.

[6] Feitelson D. G. and Rudolph L., Gang scheduling performance benefits for fine-grain synchronization., Journal of Parallel and Distributed Computing Vol. 16(4), pp. 306-318, 1992.

[7] Rosti E., Serazzi G., Smirni E., and Squillante M. S., Models of Parallel Applications with Large Computation and I/O Requirements, IEEE Transactions on Software Engineering, Mar 2002.

[8] Wiseman Y. and Feitelson D. G., Matching Parallel Jobs in Asymmetric Operating Systems, JPDPS-2001, pp. 13-16, 2001.

[9] Wiseman Y. & Feitelson D. G, *Paired Gang Scheduling*, IEEE Transactions on Parallel and Distributed Systems, Vol. 14(6), pp. 581-592, 2003.

[10] Rosti E., Serazzi G., Smirni E. and Squillante M. S., The Impact of I/O on Program Behavior and Parallel Scheduling SIGMETRICS Conference of Measurement and Modeling of Comput. Systems, pp. 56-65, 1998.

[11] Wallace G.K., The JPEG Still Picture Compression Standard, Communication of the ACM Vol. 34 pp. 30-44, 1991.
[12] Rao K.R. and Yip P., Discrete Cosine Transform Algorithms, Advatages, Applications, Academic Press Inc., London, 1990.
[13] Huffman D., A Method for the Construction of Minimum Redundancy Codes, it Proc. of the IRE, Vol. 40 pp. 1098-1101, 1952.
[14] Witten I.H., Neal R.M. and Cleary J.G., Arithmetic Coding for Data Compression, Comm. of the ACM, Vol. 30, pp. 520-540, 1987.
[15] Klein S. T. & Wiseman Y., Parallel Huffman Decoding with Applications to JPEG Files, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 46(5), pp. 487-497, 2003.
[16] Klein S. T. & Wiseman Y., Parallel Huffman Decoding, Proc. Data Compression Conference DCC-2000, Snowbird, Utah, USA, pp. 383-392, 2000.
[17] Y. A. Khalidi, M. Talluri, M. N. Nelson and D. Williams. Virtual memory support for multiple page sizes. In Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems, Napa, California, October 1993.
[18] Abouaissa H., Delpeyroux E., Wack M. and Deschizeaux P., "Modelling and integration of resource communication in multimedia applications with high constraints using hierarchical Petri nets", Proceedings of IEEE International Conference on Systems, Man, and Cybernetics (SMC-99), pp. 220-225, vol. 5, Tokyo, Japan, 1999.
[19] Itshak M. & Wiseman Y., "AMSQM: Adaptive Multiple SuperPage Queue Management", Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008), Las Vegas, Nevada, 2008.
[20] P. Jiménez, F. Thomas, and C. Torras, "3d Collision Detection: A Survey", Computers and Graphics, Vol. 25(2), pp. 269-285, 2001.
[21] S. Brown, S. Attaway, S. Plimpton, and B. Hendrickson, "Parallel Strategies for Crash and Impact Simulations" Computer Methods in Applied Mechanics and Engineering, Vol. 184, pp. 375-390, 2000.
[22] Yehezkael R. B., Wiseman Y., Mendelbaum H. G. & Gordin I.L., "Experiments in Separating Computational Algorithm from Program Distribution and Communication", LNCS of Springer Verlag Vol. 1947, pp. 268-278, 2001.
[23] Grinberg I. & Wiseman Y., Scalable Parallel Collision Detection Simulation, Proc. Signal and Image Processing (SIP-2007), Honolulu, Hawaii, pp. 380-385, 2007.
[24] M. Figueiredo and T. Fernando. "An Efficient Parallel Collision Detection Algorithm for Virtual Prototype Environments". Proc. ICPADS'04, Newport Beach, California, USA, pp. 249-256, July 2004.
[25] Salton, G., Wong, A., and Yang, C. S.. "A Vector Space Model for Automatic Indexing". Commun. ACM vol. 18(11), pp. 613-620, Nov. 1975.
[26] Y. Wiseman, K. Schwan & P. Widener, "Efficient End to End Data Exchange Using Configurable Compression", Proc. The 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, pp. 228-235, 2004.
[27] Li K., Hudak P., *Memory Coherence in Shared Virtual Memory Systems*, Proc. of the Fifth Annual ACM Symposium on Princiles of Distributed Computing, pp. 229-239, Calgary, Alberta, Canada, August 1986,
[28] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. *Message passing versus distributed shared memory on networks of workstations.* Proc. SuperComputing '95, December 1995.
[29] Beguelin A., Dongarra J. J., Geist A., Otto S., Walpole J., *PVM: Experiences, Current Status and Future Direction*, Proc. Supercomputing '93, pp. 765-766, November 1993.
[30] John B. Carter, Dilip Khandekar, and Linus Kamb. *Distributed shared memory: Where we are and where we should be headed.* Proc of the Fifth Workshop on Hot Topics in Operating Systems, pp. 119-122, May 1995.
[31] Klein S. T. & Wiseman Y., "Parallel Lempel Ziv Coding", Journal of Discrete Applied Mathematics, Vol. 146(2), pp. 180-191, 2005.
[32] Borenstein, N., *Implications of MIME for Internet Mail Gateways*, RFC 1344, Bellcore, June 1992.
[33] Geva M. & Wiseman Y., Distributed Shared Memory Integration, Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2007), Las Vegas, Nevada, pp. 146-151, 2007.
[34] Aldrich J., Dooley J., Mandelsohn S., and Rifkin A., Providing Easier Access to Remote Objects in Client Server Systems. In Thirty-first Hawaii International Conference on System Sciences, Hawaii, January 1998.
[35] Geva M. & Wiseman Y., A Common Framework for Inter-Process Communication in a Cluster, Operating Systems Review, Vol. 38(4), pp. 33-44, 2004.