Reduction of False Sharing by using Process Affinity in Page-based Distributed Shared Memory Multiprocessor Systems

K P Hung, N H C Yung and Y S Cheung

Department of Electrical and Electronic Engineering,
The University of Hong Kong.
Computer System Research Laboratory, 805 CYC Building,
Pokfulam Road, HONG KONG.

Email:{kphung,nyung,cheung}@hkueee.hku.hk

Abstract

In page-based distributed shared memory systems, a large page size makes efficient use of interconnection network, but increases the chance of false sharing, while a small page size reduces the level of false sharing, but results in an inefficient use of the network. This paper proposes a technique that uses process affinity to achieve data pages clustering so as to optimize the temporal data locality on DSM systems, and therefore reduces the chance of false sharing and improves the data locality. To quantify the degree of process affinity for a piece of data, a measure called process affinity index is used that indicates the closeness between this piece of data and the process. Simulation results show that process affinity technique improves the execution performance as page size increases due to the effective reduction of false sharing. In the best case, an order of magnitude performance improvement is achieved.

Keywords: Affinity scheduling, process affinity, data locality, spatial locality, temporal locality, false sharing, distributed shared memory

1. Introduction

The memory structure of today's parallel computers tends to be hierarchical consisting of several layers of different storage media, from processor caches, local memory to remote memory. In a non-uniform memory access (NUMA) system, such memory structure is created to lessen the contention of interconnection network between processing elements and shared memory. In specific, a distributed shared memory (DSM) system [1] (also called shared virtual memory system) introduces a single-shared address space on top of the distributed memory architecture to ease its programming effort and consequentially creates a memory hierarchy.

No matter what the rationale is for a memory hierarchy, system with memory hierarchy is characterized by their non-uniform memory access time. Such could have advantages as mentioned above, but for the same reason, remote memory accesses are likely to be time consuming and become the bottleneck of the system's performance. The techniques that can be used to alleviate this situation and improve the performance are such as prefetching, coherent caches, relaxed memory consistency models and multiple-contexts [2]. Of all these techniques, data locality optimization perhaps receives the least attention so far, notwithstanding its potential positive effect on system performance. Data locality can be classified into two types: spatial locality and temporal locality. Spatial locality refers to the use of data by accesses of nearby memory locations from a process, while temporal locality refers to the reuse of data by accesses of the same memory location from a process in different times.

Although data transfer rate of interconnection network between distributed memory modules is becoming faster and more efficient, the transfer latency is still considered high compared with local memory access latency. Therefore page-based DSM system relies on the spatial locality of data on the same page in order to hide the transfer latency. However, the page size of such a system cannot be too large as a well known side effect called false sharing may increase as a result. False sharing happens when a page is large and contains data that may be accessed by processes residing at other processing nodes. If this is the case, then the page will be transferred back and forth through the interconnection network, causing significant degradation in the system performance. The side effect of false sharing on large page size DSM system has been investigated in other approaches such as the array padding and data layout [3], the spatial data locality optimization, and the page aligned loop scheduling [4] techniques. These approaches have their strengths and limitations. For example, the page aligned loop scheduling technique can only be applied to loop scheduling, while reduction of false sharing is just a side effect of spatial data locality optimization; and array padding wastes a lot of memory storage. Similarly, work has been carried out on providing a scheme to quantify reuse and improve locality loops [5]. This approach analyses the multiple references to the same array element with affine subscript expressions.

0-7803-3529-5/96/$5.00

Beside spatial locality, research has also been done on exploring the temporal locality on DSM systems. For example, affinity scheduling is a well known technique to explore the temporal data locality in both parallel loops [6] and operating system tasks [7][8]. Affinity scheduling is a scheduling method to optimize temporal locality and tries to allocate execution entity (loop iteration or operating system task) on the processor that contains the necessary data in its local memory or cache. These use of affinity scheduling are sometimes called processor affinity because of the affinity exists between execution entity and processor cache. This affinity is due to the differences in data access performance between processor cache and main memory. KSR1 is an example of DSM systems that uses affinity scheduling technique.

This paper proposes a technique that uses process affinity to achieve data pages clustering so as to optimize the temporal data locality on DSM systems. With that, the chance of reusing data on the same data page by the same process at different times is higher. Hence the probability in sharing part of the data page with another process is minimized, which means the false sharing effect is reduced. As a result, a larger page size can be used to improve the spatial data locality without serious degradation of execution performance. In our approach, the affinity between data and process is used, and it is due to the variations in performance of remote and local data accesses. This is different from some previous works that processes are scheduled to the data with the expectation that temporal data locality can be found. Therefore the effectiveness of previous affinity scheduling techniques is more determined by the nature of the problem being tackled. In this method, process affinity is quantified by some measures of data and process (DAP) relationship called process affinity indexes.

With this quantified measure, comparison of affinities can be objective and the ambiguity that arises in heuristic or empirical methods can be reduced. As a result, data item can be nominated to a process for which it has the highest process affinity index. In the next step, the data items are prioritized with descending relative process affinity index with respect to a particular process. With this priority order, this list of data items will be clustered into pages and assigned to that particular process. This inside-out approach is more flexible in optimizing the temporal data locality and is more detached from the inherent nature of the problem. To some extent, process affinity data clustering can also be considered as a type of data layout method. However, process affinity data clustering uses a quantified approach to determine the data clusters.

This paper is organized as follows: Section 2 gives a detail derivation of the approach using process affinity; and the related data clustering and page affinity indexes are described in Section 3. Section 4 presents some of the preliminary simulation results and their interpretations. This paper is concluded in Section 5.

2. Process Affinity Algorithm

2.1 Overview

A process is generally viewed as being responsible for manipulating data and producing result(s). Therefore data and processes are inherently related. In our attempt of using process affinity to cluster data into pages on a DSM system, the measure of data's process affinity is used. This so called process affinity index is derived from the data and process relationship, which essentially describes the number of read and write accesses per process. Using the process affinity index, the affinity of a data with respect to a certain process can be quantified and therefore comparisons can be made objectively. In addition, data clustering index, page affinity index and process cohesion index can also be derived which provide supplementary information for the run time DSM server to better manage the data pages.

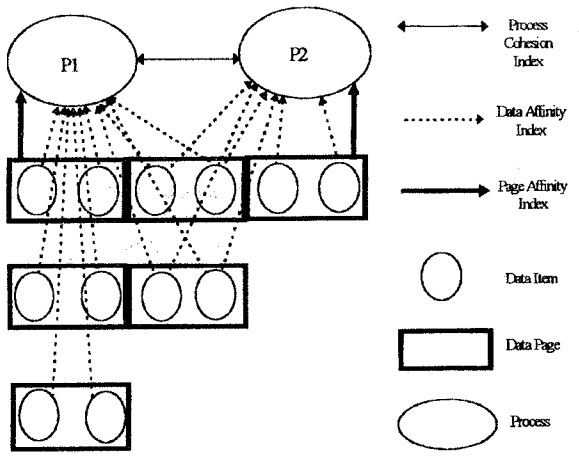

Figure 1. Affinity Indexes and Data-Page-Process Relationship

As depicted in Figure 1, data and process (DAP) relationship information is primarily the read and write frequencies of each process with respect to each data item. Based on this DAP information, process affinity index can be calculated which comprises of an absolute process affinity (APA) index and a relative process affinity (RPA) index. The RPA index is used as a measure of the closeness of a data item to a specific process compared with all other processes. This data item is then nominated to a process for which it has the highest RPA index. When all the data are exhausted, the nominated data list is sorted in a descending order according to the RPA index. Data in this prioritized list are then grouped together according to their RPA indexes to form pages and these pages are assigned to the process according to their priority in the list.

Some pages are closer to a process and therefore have the higher privilege not to be replaced first in the page replacement scheme. The page affinity index is used to measure this metric and it is simply the arithmetic mean of all the individual data items. The last index is called

process cohesion index. It is used to determine the degree of similarity between two processes in data access pattern. If they share a large portion of the same data, then they will have a higher process cohesion index, and therefore more plausible to be scheduled together.

2.2 Data and Process (DAP) Relationship

The relationship between data and process is basically determined by memory access. A memory access can be one of the two types, namely read access and write access. Each of these accesses may change the data distribution of the underlying machine in a certain way, and therefore they are the primary information used to deduce the process affinity index. Broadly, DAP relationship information may be defined as representing the per process number of read and write accesses. To extract such information from a parallel program, it is perhaps best carried out either at the compiler level or the programming language level. These two possible directions are discussed below.

The first direction is at the compiler level. For example, FORTRAN 90D [9] defines a data parallel programming language that offers no explicit expression of data distribution to processes. The compiler is supposed to be intelligent enough to resolve the data distribution problem automatically or to totally ignore the data distribution and suffer from significant effect of remote data accesses. Since it is not easy to construct efficient FORTRAN 90D compiler, High Performance FORTRAN [10] was defined to supplement mainly the expression of data distribution strategy in the source program. Similarly, DAP information can be extracted either by explicitly indication in the source program by program developer or by using an automatic data access calculation process with the aid of heuristic methods. Essentially, the read and write access information acquired for the simulation presented in this paper is obtained through the latter.

Secondly, at a programming language level, object oriented programming language can also be considered as an alternative for representing the DAP relationship information. For example in C++, it allows program developer to categorize the data as public, private, or protected type such that the scope of data sharing is well defined. Furthermore, with program entities expressed as objects, it should aid to determine the data read and write frequencies by a process systematically.

After the extraction, the data and process relationship information can be constructed into a DAP matrix . Let symbols P_i represents the process i for $i=1,2,\ldots,n$ and D_j to represents the data item j for $j=1,2,\ldots,m$, where n is the total number of parallel execution processes and m is the total number of data items used by the processes. Moreover, a tuple is defined as (R_{ij},W_{ij}) to represent the DAP relationship information of process P_i, with respect to data item D_j, where R_{ij} is the read access frequency and W_{ij} is the write access frequency. A typical DAP matrix is illustrated below:

	D_1	D_2	D_3	D_4
P_1	(5, 7)	(1, 2)	(4, 6)	(0, 2)
P_2	(1, 0)	(4, 5)	(5, 4)	(3, 0)
P_3	(5, 7)	(3, 4)	(3, 5)	(2, 3)

Typical DAP matrix

From the matrix, the DAP relationship can be obtained, for example, P_2 has 5 read accesses and 4 write accesses to D_3.

2.3 Process Affinity Index

With the DAP relationship information being made available, process affinity index can be derived. To measure the degree of data affinity to a process, it is important to understand how data accesses can influence the existing data distribution. This influence in turn depends on at least the following factors: data read frequency, data write frequency, and the number of processes related to the same data item.

Two process affinity indexes are defined as a measure of how close the relationship between a piece of data is to a process. They are the absolute process affinity index (APA Index) and the relative process affinity index (RPA Index), and the proposed method is sketched as below.

First, the absolute process affinity (APA) index of D_j with respect to P_i is defined as:

$$APA(P_i,D_j) = Cr_{ij}R_{ij} + Cw_{ij}W_{ij} \qquad (1)$$

where Cr_{ij} and Cw_{ij} are the weighing factors for read and write access of P_i on D_j respectively, and the relative process affinity (RPA) index of data j, D_j, with respect to process i, P_i as:

$$RPA(P_i,D_j) = \frac{(Cr_{ij}R_{ij} + Cw_{ij}W_{ij})}{\sum_{k=1}^{n} (Cr_{ij}R_{kj} + Cw_{ij}W_{kj})} \qquad (2)$$

where n is the total number of processes.

The weighing factors, Cr_{ij} and Cw_{ij}, are the adjustments to reflect the significance of read access and write access of D_j by P_i. In determining these factors, the write invalidate coherence protocol is assumed as it is a common implementation on a lot of DSM systems [11]. In essence, a write operation to a local page modifies the data of its page and invalidate all other shared pages in the system. If a read operation is requested later on by another process, a new page of the data would have to be reloaded from the owner of the page. By assuming that the read and write operations on local memory are of comparable speed, if the

read and write requests are distributed uniformly, then the need for an extra reload operation will depend on the relative numbers of read and write access frequencies.

Let us consider a data item D_j and a process P_i. A reload of D_j will occur on a read request of P_i if D_j is not local to P_i. There are two possibilities that this may occur:

- It is the first time that P_i issues a read request for D_j.
- The read request from P_i is preceded by a write request on D_j by P_k and k is not equal to i.

The first possibility depends on the initial distribution of data and this can occur at most once in a process execution. For the second possibility, its frequency is determined by the comparison of the number of read accesses on D_j by P_i and the summation of write frequencies in all P_k where k=1,..,n and k≠i.

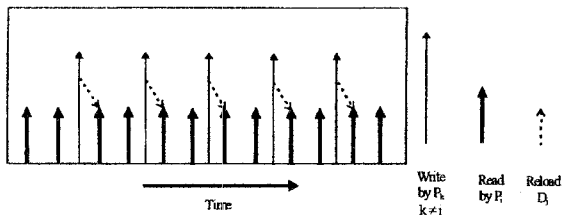

Figure 2. Process i read frequency is larger than summation of write frequencies in all processes k other than i

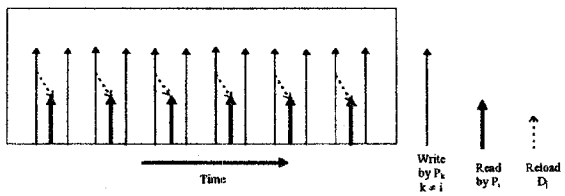

Figure 3. Summation of write frequencies in all processes k other than i is larger than process i read frequency

A reload operation of D_j by P_i should be executed after a write operation on D_j by P_k $(k \neq i)$ and before a read operation of D_j by P_i can be performed. With the assumption that the read and write operations are separated uniformly, Figure 2 shows the situation when R_{ij} is greater than $S_{ij} = \sum W_{kj} (k \neq i)$. After each write operation on D_j by P_k $(k \neq i)$, it must be followed by a read operation of D_j by P_i followed. Hence, the total number of extra reload operations is determined by S_{ij}. On the other hand, Figure 3 depicts the situation when R_{ij} is smaller than S_{ij}. Using the similar argument as before, the total number of extra reload operations in this case is determined by R_{ij}. In short, the number of extra reload operations due to all read requests on D_j by process P_i can be expressed as:

$$RL_{ij} = \min(R_{ij}, S_{ij}) \text{ where } S_{ij} = \sum_{all k \neq i} W_{kj} \quad (3)$$

where min(a, b) is an operation that returns the minimum value among a and b. This reload operation is a costly operation because of the fact that D_j have to be transferred from a remote processing node and therefore incurred a remote memory access latency. In a DSM system, this is the latency of replicating a remote page.

Assume the local memory access (LMA) latency be t_l and the ratio of remote memory access (RMA) latency to LMA be c. Then, the total number of read requests, R_{ij}, on D_j by process P_i introduces a reload cost:

$$\text{Reload cost} = c.RL_{ij}.t_l \quad (4)$$

A local memory access is required in a write operation, while a local memory access with addition to a reload cost are required for a read operation. As a result, we have the weighing factors as:

$$Cw_{ij} = t_l \quad (5a)$$

$$Cr_{ij} = (1 + (c.RL_{ij})/R_{ij})t_l \quad (5b)$$

or normalized to

$$Cw_{ij} = 1 \quad (6a)$$

$$Cr_{ij} = 1 + (c.RL_{ij})/R_{ij} \quad (6b)$$

From this argument, the RPA index given by equation (2) and the ADA index given by equation (1) of P_i against D_j can be calculated and they are the primarily measures of data D_j's process affinity to process P_i.

3. Data Clustering and Page Affinity

3.1 Data Clustering Method

Up to this point, process affinity index of all data items can be determined. The next step is to make use of these calculated process affinity index values to cluster data items into data pages. To begin with, each data item is nominated to its closest process, which has the highest process affinity index with respect to that process. The nomination decision is given by the following:

If $RPA(P_i, D_j) = \max_{all\ P_k \in S} \{ RPA(P_k, D_j) \}$

where S = set of processes

then D_j -> P_i, data item D_j is nominated to process P_i

where max {...} is an operation that returns the maximum value from a set of values.

In addition, the nominated data list per process is sorted in a descending order according to the RPA index. Such data items are then clustered into pages according to the page size of the process. The reason for this arrangement is to prevent the data items with very small RPA index mixed with data items with large RPA index. This is because data items with small RPA index will introduce large amount of page transfer in the system, and data items with large RPA index are localized to their corresponding processes. Mixing them together may cause a lot of unnecessary data transfers of data items with large RPA index, thus adversely affect the system performance.

After clustering the sorted data list into pages, data pages are allocated to the corresponding process in order. In the case of extra pages that cannot be allocated to a process due to memory space limitation, the remaining pages are then allocated sequentially according to the space availability in other processes.

3.2 Page Affinity Index

Page affinity index (PA), PA_i, is used to measure a page's affinity to process P_i. This page affinity index is the arithmetic mean of individual data item's RPA index. Pages with higher PA index is scheduled closer to the corresponding process at system startup, and remained with its process as long as possible by employing a suitable page replacement scheme.

$$PA_{ij} = \left[\sum_{\substack{\text{all } D_k \\ \text{in} \\ \text{page } j}} RDA(P_i, D_k) \right] / \text{pagesize} \quad (7)$$

3.3 Process Cohesion Index

The last index that can be derived from the RPA index is the process cohesion Index (PC Index). It is used to measure the degree of data sharing between two processes and is defined as follows:

$$PC[P_i, P_j] = \sum_{\text{all } k} RDA(T_i, D_k) \times RDA(T_j, D_k) \quad (8)$$

With process cohesion index, process scheduling algorithm can be designed to optimize the data locality when assigning different processes into the same processing node. By doing so, the execution performance may be improved.

4. Preliminary Simulation Results and their Interpretation

The purpose of the simulation is to demonstrate the usefulness of the process affinity algorithm, and the preliminary area of focus is the effect of process affinity to false sharing in a page-based DSM system.

Matrix multiplication, Gaussian elimination [12] and fast Fourier transform(FFT) [13] are selected as test vehicles in this simulation study. The experiments tested and compared are:

- matrix multiplication with matrix size of 64x64 and 128x128,
- linear equation solver using Gaussian elimination with 64 and 128 equations,
- and fast Fourier transform of 1024 points,

based on varying the size of a page and the resultant execution time to complete solving the problems, between the case of sequential arrangement of data into pages and evenly distributing the pages into different processing nodes, and the case of using the process affinity indexes and other related affinity indexes to determine the allocation of data to process, clustering of data into a page and the distribution of pages to processes as described in section 3.

The simulation was written in the C++ programming language and executed as 15 processes running on 15 SGI Indy workstations. The assumptions made in the simulation are that first, remote memory access (RMA) latency that includes replicating a remote page and performing a local memory (LMA) access operation is assumed to be 100 execution time units, while local memory access latency is assumed to be 2 execution time units. As the RMA and LMA latencies are still substantially different in many distributed memory architectures today, these execution time assumptions are deemed to be reasonable. Second, an arithmetic operation such as the multiplication or addition of two floating point numbers is assumed to be 1 execution time unit. Third, the remote memory access latency should theoretically vary as page size changes, however it is assumed that the RMA latency is varying within a small range especially for small page size.

From the results shown in Figures 4 and 5, a number of points can be observed. First, the performance difference between the execution time units with process affinity and without process affinity is minimal for page size equal to or smaller than the matrix row size. From the calculated relative process affinity (RPA) index, it was discovered that the number of data items with the highest RPA index that clustered together is closed to the row size. This quite simply highlights the fact that false sharing does not occur at these page sizes. In other words, process affinity has no impact at all when the above condition is satisfied. The minor discrepancies between the two cases are more due to the initial data distribution.

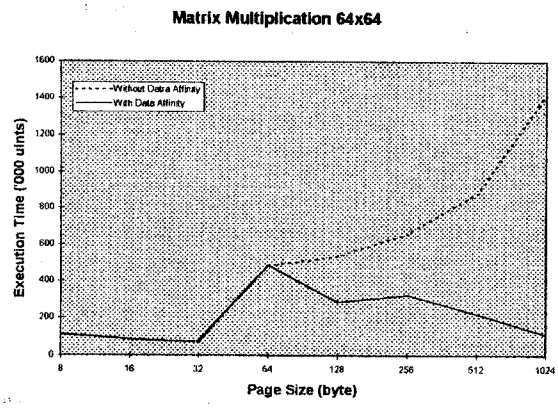

Figure 4. Matrix multiplication (64x64) execution time unit versus page size

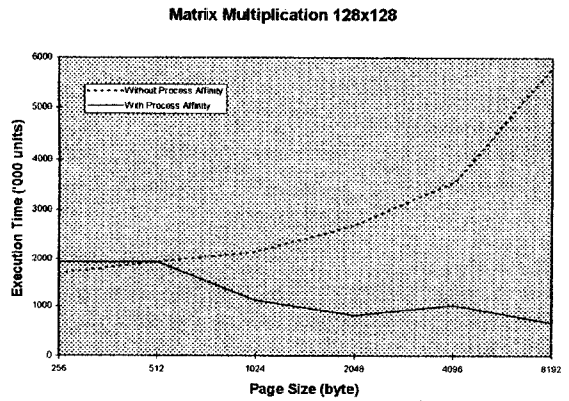

Figure 5. Matrix multiplication (128x128) execution time unit versus page size

Second, the performance difference between the two cases widens when the page size increases beyond the matrix row size. In these cases, false sharing becomes a dominating factor and as a result, the case of without process affinity requires a lot more execution time to handle the remote memory accesses that are the consequence of a process now has to share data with some other processes. On the other hand, with process affinity, this false sharing phenomenon is drastically reduced to a level that gives a performance better than the case where page size is equal to or smaller than the matrix row size (i.e. 512 bytes for 128 floating point numbers) for the matrix size equals to 128. Using this as our discussion basis, it is noted that for page size of 1024 bytes, the reduction in execution time units is by almost 50% when the process affinity algorithm is adopted. In the best case where the page size equals to 8192 bytes, the number of execution time unit for the process affinity case is about one-tenth of the matrix multiplication without process affinity. Similar results hold for the 64 by 64 matrix multiplication.

For the test case of linear equation solver using Gaussian elimination, results are depicted in Figures 6 and 7. We can observed that the performance difference between the execution time units with process affinity and without process affinity is minimal for page size less than or equal to 1024 bytes for the number of equations equals to 128, and is increased as the page size increases beyond this point. The argument for this observation is similar to that of matrix multiplication case. When the page size is small, false sharing phenomenon is insignificant and the execution time to solve the Gaussian elimination with using data affinity and without using data affinity are almost the same. However, false sharing is becoming significant when page size increases. The execution time of solving the problem without using data affinity is increased correspondingly, but that execution time is kept almost constant in the case data affinity employed.

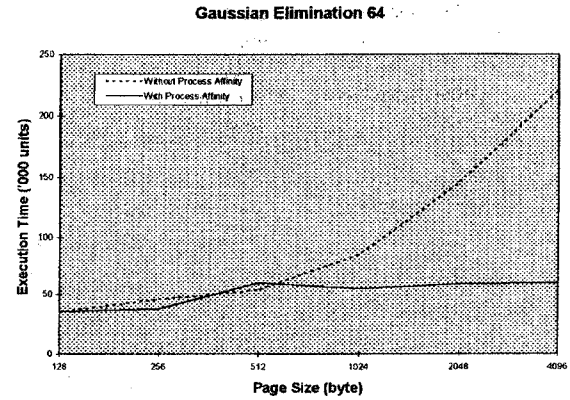

Figure 6. Gaussian elimination (64 equations) execution time unit versus page size

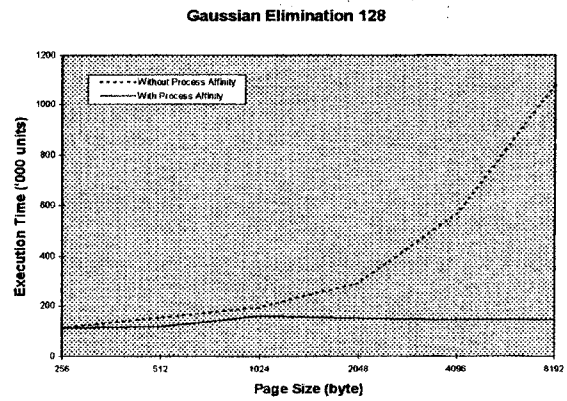

Figure 7. Gaussian elimination (128 equations) execution time unit versus page size

As shown in Figure 7, the reduction in execution time units is by more than 50% when process affinity is employed with page size of 2048 bytes. In the best case where page size is 8192 bytes, the number of execution time unit for the case using process affinity is about one-seventh of the case without using process affinity. Similar observations can be found in the 64 linear equations solver using Gaussian elimination.

The last test case is to solve the fast Fourier transform with 1024 points and the results are depicted in Figure 8. The execution time performance difference for solving this problem with using data affinity and without using data affinity is small when page size is below 32 bytes, and this difference widens as the page size is increased. The turning point of page size 32 bytes is small compared to that of the cases of matrix multiplication and Gaussian elimination. From the calculated relative data affinity (RDA) index, it is found that its data items show less affinity to a specific process. This means the RDA indexes of a piece of data item with respect to different processes are close to each other. As a result, data sharing between different processes are common and false sharing effect can be significant in a relatively small page size.

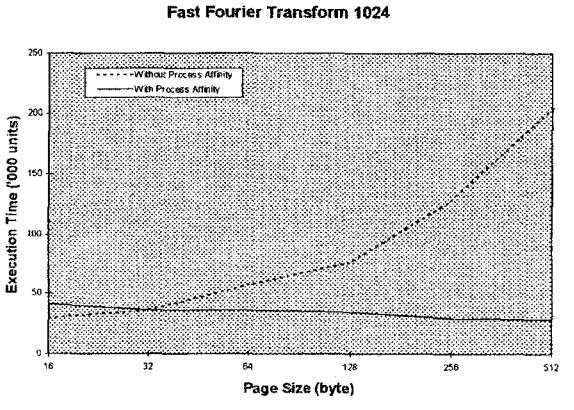

Figure 8. Fast Fourier transform (1024 points) execution time unit versus page size

As depicted in Figure 8, the execution time reduction in the FFT test case is by more than 50% when process affinity is employed with page size of 128 bytes. In the best case where page size is 512 bytes, the number of execution time unit for the case using process affinity is about one-seventh of the case without using process affinity.

5. Conclusions

In this paper, we introduced the concept of data to process affinity and proposed a set of equations that enables a number of indexes to be calculated, namely: data and process affinity index, page affinity index and process cohesion index. These indexes allow the affinity relationships between data and process to be deduced as well as how these data should be grouped together to form pages, and how these pages are grouped together and adhered to a process.

Preliminary simulation was conducted to demonstrate the usefulness of the process affinity concept, particularly in the area of reducing or eliminating false sharing in page-based DSM systems. By using a matrix multiplication of size 64^2 and 128^2, it can be concluded system performance can be improved substantially by using a page size larger than the matrix row size and the process affinity algorithm as depicted in the early sections. Such improvement can be as large as an order of magnitude in the case of page size equals to 2048, matrix size equals to 128 as shown in figure 5. Similar results are obtained for the case of using other problems such as the linear equation solver using Gaussian elimination and the fast Fourier transform.

Furthermore, there are two more issues that are worth considering. First, as some data are shared quite equally among many processes, placing such data into a page may introduce a huge amount of page transfer over the DSM system. Therefore it may be useful to adopt a mixed page-based and shared variable DSM implementation strategy [14]. In this case, the extensively shared data can be shared between processes in shared variable entities and communicated using message passing method so that network transfer loading may be reduced and system performance may be improved.

The second issue is parallel loop scheduling. The process affinity algorithm may be applied to scheduling parallel loop, and combined with ideas of multithreaded self scheduling [15] to formulate a more effective loop scheduling technique designed for DSM systems.

6. References

[1] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors, *PhD Thesis,* Yale University, Sept 1986.

[2] Kai Hwang, "Advanced Computer Architecutre - Parallelism, Scalability, Programmability", *McGraw-Hill International*, 1993, pp.475-544.

[3] J. Torrellas, M. S. Lam, J. L. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates," *Proceedings of International Conference on Parallel Processing,* pp. 266-270 1990.

[4] E. D. Granston, H. Wijshoff, "Managing Pages in Shared Virtual Memory Systems: Getting the Compiler into the Game," *Proceedings of International Conference on Supercomputing,* pp. 1993.

[5] M. E. Wolf, M. S. Lam, "A Data Locality Optimizing Algorithm," *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30-40 June 1991.

[6] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, pp. 379-400 5(4) April 1994.

[7] E. D. Lazowska, M. Squillante, "Using Processor-cache Affinity in Shared- Memory Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, pp. 131-143 4(2) Feb 1993.

[8] R. Vaswani, J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp.26-40 1991.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M. Wu, "Fortran D language specification," *Technical Report TR90-141*, Dept of Computer Science, Rice University, 1990.

[10] High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," *Technical Report CRPC-TR92225*, Center for Research on Parallel Computation, Rice University, 1993.

[11] B. Nitzberg, V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, pp. 52-61 Vol. 24 Aug 1991.

[12] M. Cosnard, D. Trystram, "Parallel Algorithms and Architectures", International Thomson Computer Press, 1995, pp.481-488.

[13] S. G. Akl, "The Design and Analysis of Parallel Algorithms", Prentice-Hall, Englewood Cliffs, NJ, 1989, pp.231-241.

[14] F. B. Bodin, T. Priol, D. Gannon, P. Mehrotra, "Directions in parallel programming: HPF, Shared Virtual Memory, and Object Parallelism in PC++," *Parallel Computer: Theory and Practice*, pp. 183-215, IEEE Computer Society Press 1996.

[15] K. P. Hung, N. H. C. Yung, Y. S. Cheung, "Multithreaded Self-Scheduling: Application of Multithreading on Loop Scheduling for Distributed Shared Memory Multiprocessor," *Proceedings of IEEE 1st International Conference on Algorithms and Architectures for Parallel Processing*, pp.680-689 Vol. 2 April 1995.