Integrating Dynamic Memory Placement with Adaptive Load-Balancing for Parallel Codes on NUMA Multiprocessors

Paul Slavin* and Len Freeman

Centre for Novel Computing, School of Computer Science
The University of Manchester, Manchester, M13 9PL
{slavinp,lfreeman}@cs.man.ac.uk

Abstract. This Paper describes and evaluates a system of dynamic memory migraton for codes executing in a Non-Uniform Memory Access environment. This system of migration applies information about the load-imbalance within a workload in order to determine the affinity between threads of the application and regions of memory. This information then serves as the basis of migration decisions, with the object of minimising the NUMA distance between code and the memory it accesses. Results are presented which demonstrate the effectiveness of this technique in reducing the runtime of a set of representative HPC kernels.

1 Introduction

The Non-Uniform Memory Access (NUMA) environment presents the programmer of parallel HPC applications with a convenient and intuitive representation of memory in the form of a unified global address space comprised of several physically discrete memories which are transparently integrated by the actions of hardware. This layer of abstraction between the physical layout and conceptual presentation of memory in a NUMA system manifests itself in the form of a variable latency that differs according to the region of the address space accessed, as different regions are mapped by the operating system's page tables to different physical banks of memory. An access by a processor to a local bank completes in less time than if the same access were satisfied from a more remote memory to which requests and responses are mediated by an interconnect.

The placement of memory by a parallel program is therefore an important determinant of that program's performance [1] and it is desirable to have as great a proportion as possible of the memory that a process will access during its execution located in the same node of the NUMA machine as the processor executing this process [2]. The data in memory, on which the threads of a parallel program will operate, is termed the *workload* of the program. For applications with a regular and predictable workload, the optimal placement of data within memory

* This work was supported by the award of a Doctoral Training Studentship by the EPSRC.

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 274–284, 2008.

may be specified in advance, by way of source code directives or by the actions of a compiler that is aware of the physical memory layout of the NUMA system in question. A large class of applications have irregular, sparse, or otherwise unpredictable workloads whose optimal placement cannot be determined *a priori*. For these cases, *page migration* techniques have been developed which dynamically relocate memory between the nodes of a NUMA system with the objective of minimising the distance between threads and the data they operate upon.

2 Page Migration

Where the distribution in memory of a workload cannot be determined before runtime, page migration methods must examine the runtime behaviour of the parallel program in order to establish the affinities that exist between its threads and areas of the program's address space that are occupied by the partitioned workload that code will operate upon [3]. Then, having identified the affinity of processors for certain regions of the partitioned workload, these regions are relocated in physical memory to be as close as possible to the processors which access them. The *partition* of the workload in this sense refers to the division of the total of the program's data into a sequence of slices, each of which is allocated to a processor in the system.

As such, the task of a page migration scheme may be decomposed into two principal activities. The first is the gathering of information relating to the affinity of each of a parallel program's threads for certain regions of the workload. The second is the physical relocation of these regions of the workload to memory in the NUMA system that is in proximity to the appropriate threads. This paper deals with the first of these tasks and describes how an extended Feedback-Guided Dynamic Loop-Scheduling (FGDLS) [4] algorithm may be used to create a representation of a parallel program's workload which reveals the affinities of processors for regions of the workload.

Existing page migration schemes determine which of the processors participating in a parallel program exhibit an affinity for which regions of the address space by examining a sample of the accesses made by each thread to each page of virtual memory [5,6,7,8,9]. In such schemes a counter is maintained which records the number of accesses to that page originating from each node of the NUMA machine. When a sufficient number of accesses from a particular remote node are recorded, migration is triggered and the page in question is associated with a physical page frame local to (or at least closer to) the accessing node. We denote this technique as the *sampling/threshold* page migration technique. Although the technique of sampling and thresholds is widely used in migration schemes, there are aspects of real-world HPC applications which can cause the simple perspective on a program's memory accesses which this method offers to diverge from the real pattern of memory accesses that the program will exhibit. Foremost amongst these is the implicit and fundamental assumption of such schemes that observations about the historical behaviour of a program will continue to be relevant to the program's future behaviour. In many cases, the

underlying algorithms embodied by a program necessitate that the operations of the program are divided into several distinct "phases", within each of which the behaviour of the program is consistent yet between which behaviour differs considerably. It can be readily envisaged that migration performed in response to a program's behaviour in one phase may not be relevant to a subsequent phase.

The experimental results presented in Sect. 5 indicate that this class of page migration technique is unable to achieve substantial reductions in runtime for representative HPC applications on a commercially available parallel machine. Analysis of the behaviour of this traditional migration scheme with the SpeedShop profiling environment [10] indicates that, of the two components of the page migration procedure identified above, the physical relocation of memory between the nodes of a system is not time-consuming and does not impose a substantial overhead on code which makes use of page migration. As such, this implies that it is in the domain of the other component task, that of gathering accurate information about a program's affinities for regions of memory, that the sampling/threshold system of migration experiences problems.

This analysis agrees with that described in [3] where problems in tuning the threshold number of remote accesses required to trigger migration are encountered and no ready technique for determining the optimal sensitivity can be determined. Where the appropriate level of sensitivity for an application cannot be established, the sampling/threshold system of page migration will inevitably manifest 'false positives' due to overly-sensitive migration and similar 'missed-opportunities' where the potential to reduce latency by a migration is overlooked due to insufficient sensitivity to remote accesses. In view of this circumstance, we assert that the principal source of poor performance with existing page migration schemes is the problem of accurately determining *what* is to be migrated to *where* on the system, and to do so in a timely manner. The overhead of physically migrating data is minimal when compared to the consequences of making poor decisions as to the memory affinities of the program. This recognition has motivated our attempt to develop a source of information on which page migration decisions may be based that better describes the affinities between threads in a parallel program and the regions of the partitioned workload they operate upon.

3 Feedback-Guided Dynamic Loop Scheduling

Feedback-guided dynamic loop scheduling may be applied to a sequence of nested loops where a parallelised inner loop is contained within a sequential outer loop whose iterations typically represent change with respect to time, as depicted in the following code fragment:

```
DO SEQUENTIAL K = 1, NSTEPS
   DO PARALLEL I = 1, N
      CALL LOOP_BODY(I)
   END DO
END DO
```

In the case of a sparse workload, where equal extents of the workload do not contain equal amounts of work, the task of a FGDLS algorithm is to interrogate the program's historic execution profile to dynamically determine the amount of work that is represented by each partition of the workload.

After the first iteration of a nested loop is performed with an inital equipartition of the workload, the FGDLS algorithm compares the deviation of the execution time of each slice from the mean execution time denoted by $\frac{T}{P}$ and adjusts the boundaries of each slice according to whether it is over or under endowed with work. By this technique of gradual refinement, the FGDLS algorithm converges upon an equal allocation of work to each of the processors.

3.1 Integrating FGDLS with Page Migration

In a standard FGDLS algorithm of the type described in [4], each slice of work is represented by a data structure of the type indicated below. The *start* and *end* values represent the lower and upper boundaries respectively of a subdivision of the program's workload. After each outer loop iteration therefore, an execution time is associated with a region of the workload extending from the *start* to the *end* element, and so recording the execution time that is attributable to this delineated region of the data on which the program will operate.

```
struct slice{
    int start;
    int end;
    int cpuid;
    struct slice *next;
};
```

This load balancing operation results in an updated slice structure in which load imbalance will be reduced relative to the previous iteration. The values within this updated slice structure will be applied to each processor at the beginning of the next outer loop iteration. However, because of the representation of each processor's allocation as a contiguous region of the workload in memory, the *start* and *end* values in conjunction with the *cpuid* value in each slice define the region of the address space that the *cpuid*'th processor will access on the subsequent outer loop iteration.

It is this crucial observation which permits the information contained within the workload's partition to be applied to the placement of memory for the program in question. Because the values in each slice of the partition will dictate which processor operates upon which region of the workload, these same values also indicate the affinity of that processor for certain pages in memory. This is precisely the same information that a page migration scheme requires in order to migrate memory towards the processors which make most frequent use of it. As such, by migrating the region defined between the *start* and *end* boundaries of each slice to the same NUMA node as the *cpuid*'th processor which will access that region, it is possible to enhance the locality of memory placement at each

outer loop iteration in the same proportion as the FGDLS algorithm reduces load imbalance.

The following section describes how such a scheme combining load balancing and memory migration may be implemented on a representative shared-memory NUMA machine.

4 Implementation

The technique of page migration described in the sections above was implemented on an SGI Origin 3400 machine with 16 MIPS R12000 processors and 4GB of main memory. This system is comprised of four nodes, each containing four processors, and 1GB of local memory. Each processor has an 8MB level 2 cache consisting of 128 byte cache-lines with a two-way set associative hashing policy. The machine's operating system is IRIX 6.5.

4.1 Exposing Physical Topology to Userlevel Code

The NUMA API of such operating systems expose functions to the programmer which allow the naming and compartmentalisation of regions of the machine, each consisting of one or more nodes comprising processors and memory. This provides a mechanism for informing userlevel code of the physical layout of the machine and provides the conceptual basis that permits regions of the address space of a particular program to be selectively assigned to the physical resources of the NUMA machine of which it runs.

In the terminology of IRIX, these named and demarcated regions of the machine are termed *Memory Locality Domains (MLDs)*. These MLDs are an abstract representation of a machine's resources which are created dynamically at runtime and need not coincide with the physical topology of the NUMA machine. However, by configuring MLDs which coincide with the physical resources of the machine, userlevel applications may be given a method of addressing specific nodes of the machine rather than being restricted to the unified address space abstraction presented by the system's hardware.

As such, our first step is to use the *mld_create()* syscall to configure a MLD for each node in the system. Each MLD so created is provided with a *Resource Affinity List* which specifies its proximity to physical nodes of the system, as identified by the filenames of their corresponding entries in the /hw virtual filesystem. This results in a set of MLDs, each of which represents a physical portion of the machine's hardware.

4.2 Unifying Load Balancing and Page Migration

The computational kernel of the code in question consists of a sequence of nested loops. As described in Sect. 3, the FGDLS algorithm updates its 'map' of the program's workload after each outer loop iteration and modifies the quantity of

work allocated to each processor in response to any load imbalance that is identified. The schedule resulting from this reallocation is then applied to the next outer loop iteration. As such, a period exists where the FGDLS algorithm has calculated in advance the allocation of work to processors that will be applied to the $K+1$ outer loop iteration. As described in Sect. 3, this allocation determines the regions of memory which each processor will access, but at this point the workload in memory remains in the state determined by the K allocation. By migrating to the appropriate processor those regions of the address space which the forthcoming $K+1$ schedule will allocate to that processor, each memory access made during the $K+1$ iteration will be satisfied from local memory. The nested loop is modified to the following state by the inclusion of load balancing and subsequent migration:

```
DO SEQUENTIAL K = 1, NSTEPS
   DO PARALLEL I = 1, N
      CALL LOOP_BODY(I)
   END DO
   CALL FGDLS()
   CALL MIGRATE(FGDLS_PARTITION)
END DO
```

Section 5 demonstrates that for some applications, the cost of migrating this contiguous region of memory as a unit is substantially lower than the cost implied by having some proportion of the $K+1$ iteration's memory accesses satisfied from remote memories.

4.3 Migrating Address Ranges between Nodes

Having identified the processors to which regions of the workload will be assigned in the next outer loop iteration, it remains to migrate the address ranges corresponding to these regions to the appropriate processors. The system call which effects this is *migr_range_migrate()* with the signature:

```
int migr_range_migrate(void* base_addr, size_t length,
                       pmo_handle_t pmo_handle)
```

The *base_addr* and *length* parameters specify a region of the virtual address space and the *pmo_handle* object represents a placed MLD. This MLD is the destination to which the designated range of virtual memory will be migrated. To determine whether such a migration is necessary, the *va2pa()* system call is passed a virtual memory address and returns the node of the system on which the physical page frame containing this address is located. In conjunction with the virtual addresses which correspond to the *start* and *end* boundaries that define each slice of the workload, *va2pa()* may be called to determine the current node of residence of the data described by this slice and then *migr_range_migrate()* called to reallocate this region to the node housing the processor that will operate upon this data in the subsequent loop iteration.

The procedure to implement scheduling guided page migration may therefore be summarised as follows:

1. Establish a set of Memory Locality Domains which correspond to the nodes of the NUMA system.
2. Implement an FGDLS algorithm which represents the program's workload as a set of contiguous slices, each associated with one processor.
3. Apply this FGDLS algorithm after each outer loop iteration to create a schedule for the next iteration.
4. According to this schedule, migrate each processor's workload to memory local to that processor.
5. Begin execution of the next outer loop iteration.

5 Experimental Evaluation

5.1 Method

The steps described above were applied to a set of source codes representing typical linear algebraic operations that constitute the computational kernel of real-world HPC applications. The first group of these is a parallel sequence of multiplications of sparse matrices of varying sizes by a dense vector. The second is the Conjugate Gradient benchmark from version 2.3 of the NAS Parallel Benchmark Suite. Trials were performed on a NUMA system as described in Sect. 4.

For each of these trials, the original source code is extended to incorporate the scheduling-guided page migration scheme described above. These extended codes are then compared with the original, unmodified version in terms of the absolute 'wallclock' runtime, and of the proportion of all memory accesses by each program that are made to remote memory. Furthermore, a series of trial are conducted with the inbuilt system of page migration in IRIX 6.5 enabled for each program. This page migration system bases its migration decisions on a sampling/threshold scheme such as that described in Sect. 2. Each trial is executed in the NUMA-aware SpeedShop [10] profiling environment. This utility has been configured to interrogate the Origin's hardware counters in the event of an L2 cache miss and to determine whether this miss is satisfied from local or remote memory.

5.2 Matrix-Vector Multiplication

This experiment consists of a sparse square matrix repeatedly multiplied by a dense vector. The Posix Threads library has been used to parallelise this operation such that each participating processor is assigned a contiguous set of the rows of the matrix.

The matrix is populated with work according to the Gaussian distribution which commonly occurs in statistical applications. The probability of a matrix

element being populated with a non-zero value is inversely proportional to the distance of its row coordinate from the central row of the matrix.

At the beginning of the multiplication, the matrix is divided into an equipartition where each division contains $\frac{N}{P}$ elements of the matrix, where P is the number of participating processors. The component calculations of the multiplication are then performed by a nested loop structure, with the allocation of work between each of the participating processors being modified by the FGDLS routine after completion of each outer loop iteration. Migration is then performed following each update to the schedule.

5.3 Conjugate Gradient

The CG application is an iterative solver which is conceptually similar to matrix-vector multiplication in that the CG algorithm is itself dominated by the multiplication of a dense vector by a positive definite sparse matrix.

In this code, the *rowstr* array is used to partition the main matrix and so performs the function in CG that is performed by the *slice* structure in the matrix-vector multiplication. One notable characteristic of the matrix used in this code is that the distribution of data is less regular and more fragmented than in the matrix-vector example. This implies that migration must be performed on a larger number of smaller, less contiguous regions of data. As described in the next section, the repercussions of this are evident in the observed results.

5.4 Results and Analysis

Tables 1 and 2 describe the effects of the inclusion of scheduling-guided migration within each benchmark's source code in terms of the proportion of memory accesses which are to remote memory and of the wallclock runtime for each program. Columns of each table describe these metrics for the case of the unmodified source code ("WithOut migration"), the scheduling-guided migration described in this paper ("SG migration") and the inbuilt operating-system version of sampling/threshold migration ("OS migration"). Matrix-vector multiplication is performed with square matrices of size 8000 and 12000 and sizes 'B' and 'C' of the CG application are evaluated.

The first notable observation from these results is the failure of the operating-system's migration scheme to substantially affect either the proportion of accesses to remote memory or the wallclock runtime. This coincides with the observations made in [1] regarding the efficacy of page migration on this platform for real-world problems, rather than in the simulated environment described in [11].

In contrast, trials featuring a schedule-guided migration scheme show a marked reduction in both the proportion of remote memory accesses and in the overall execution time. While this is the case for both matrix-vector and CG programs, the extent of the reduction in remote memory accesses, and the corollary reduction in runtime, that is achieved by schedule-guided migration is observed to be notably more pronounced for the matrix-vector code than for the

Table 1. Effect of Schedule-Guided Migration on Percentage of Memory Accesses made to Remote Memory

Benchmark	WO Migr %	SG Migr %	OS Migr %
MV 8K	49.98	0.13	49.26
MV 12K	48.54	0.18	48.08
CG.B	38.96	3.24	36.94
CG.C	57.05	31.46	56.94

Table 2. Effect of Schedule-Guided Migration on Runtime

Benchmark	WO Migr (s)	SG Migr (s)	OS Migr (s)
MV 8K	272.66	211.2	271.83
MV 12K	840.66	673.96	837.57
CG.B	1523.73	1060.71	1491.46
CG.C	3789.73	2811.08	3711.66

CG code. Although both applications have a 'sparse' workload, the differences in the contiguity vs. fragmentation of each workload makes itself evident in this respect. The overhead of migrating one large region of the address space is significantly less than that required to migrate many smaller regions, as is the case for the CG application. This overhead is derived both from the cost of calling the *va2pa()* syscall to determine the current physical residency of a virtual memory page, and from the cost of the *migr_range_migrate()* function.

The influence of this characteristic of each workload becomes more evident as problem-size increases. While the matrix-vector trials with schedule-guided migration achieve a consistently high reduction in the proportion of remote memory accesses, essentially eliminating remote accesses for both sizes of matrix, the success of this same technique for the largest size of CG application is seen to decline relative to the results recorded for CG.B. This suggests that there exists a certain 'threshold of fragmentation' in a workload, beyond which the expense of migrating many small regions becomes sufficient to offset the latency that is saved by having these regions local the processor which accesses them. The repercussions of this phenomenon and its relation to the system's virtual memory page size are discussed in the following section.

6 Issues and Extensions

The results presented in the previous section demonstrate that for a workload which is partitioned so that each processor's allocation of work consists of a contiguous region of the address space, memory migration guided by a FGDLS load balancing algorithm is capable of producing a substantial reduction in remote memory accesses, but that this performance declines where the workload

is distributed less favourably. This section considers some of the limits on the general applicability of the concept of schedule-guided migration and some of the techniques which may be used to extend it to deal with these challenges.

6.1 Page Level False-Sharing

Of central importance in this respect is the effect that virtual memory page size exerts on the granularity with which data may be migrated between nodes. Where a virtual memory page is large relative to the size of the data elements it contains, many of these elements are likely to coexist on a single page. It is probable however that the boundaries of the workload partition assigned by a FGDLS scheme will not coincide with page boundaries but instead fall within a page. As the smallest unit that may be migrated between two nodes of a NUMA system is a single page, this implies that some portion of the data elements occupying a page are migrated *away* from the appropriate processor as they have the misfortune to share a page with elements which form part of the workload of another processor.

Although this problem affects a small proportion of the data assigned to each processor, its effect becomes progressively more apparent where the workload of a processor is comprised of many small regions of memory for which many separate migration operations are required. For each of these migrations, some portion of the data on each page containing a boundary of the workload's partition will be incorrectly migrated along with that page as a whole. This phenomenon is conceptually similar to the recognised issue of false-sharing that is applicable to coherent caches in a NUMA environment. This observation raises the prospect of whether similar data transformations may be applied to a workload in memory in order to restructure data into a form more amenable to page migration.

7 Related Work

The concept of dynamically relocating memory pages between the nodes of a NUMA machine in response to a program's runtime behaviour is first described in [12]. Other authors have developed this theme whilst continuing to treat page migration as a function of the operating system, for example in [11,9]. The alternative technique of implementing page migration in userspace is advocated in [6] and a related userspace technique employing a software virtual-memory layer is described in [13]. More recent reflections on migration as a solution to the latency problem in distributed shared-memory machines have returned to the theme of operating-system modification, as in [7], and even hardware assistance to migration, as in [14]. A page-migration scheme which employs Solaris' inbuilt migration system calls on a Sun Fire system to improve locality for a particular PDE solving routine is described in [2].

The issue of work scheduling for the NUMA environment has been widely considered, with the FGDLS technique in particular described in [4]. Although the techniques of page migration and load balancing have heretofore been

considered in isolation, [1] recognises their separate contributions to the performance of parallel code on a NUMA machine.

References

1. Jiang, D., Singh, J.P.: Scaling application performance on a cache-coherent multiprocessor. In: ISCA 1999: Proceedings of the 26th annual international symposium on Computer architecture, pp. 305–316. IEEE Computer Society, Los Alamitos (1999)
2. Nordén, M., Löf, H., Rantakokko, J., Holmgren, S.: Geographical locality and dynamic data migration for OpenMP implementations of adaptive PDE solvers. In: Müller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315. Springer, Heidelberg (2008)
3. Scheurich, C., Dubois, M.: Dynamic page migration in multiprocessors with distributed global memory. IEEE Trans. Comput. 38(8), 1154–1163 (1989)
4. Bull, J.M.: Feedback guided dynamic loop scheduling: Algorithms and experiments. In: Pritchard, D., Reeve, J.S. (eds.) Euro-Par 1998. LNCS, vol. 1470, pp. 377–382. Springer, Heidelberg (1998)
5. Bartal, Y., Charikar, M., Indyk, P.: On page migration and other relaxed task systems. Theoretical Computer Science 268(1), 43–66 (2001)
6. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguado, E.: A case for user-level dynamic page migration. In: ICS 2000: Proceedings of the 14th international conference on Supercomputing, pp. 119–130. ACM Press, New York (2000)
7. Corbalan, J., Martorell, X., Labarta, J.: Evaluation of the memory page migration influence in the system performance: the case of the SGI Origin 2000. In: ICS 2003: Proceedings of the 17th annual International Conference on Supercomputing, pp. 121–129. ACM Press, New York (2003)
8. LaRowe Jr., R.P., Wilkes, J.T., Ellis, C.S.: Exploiting operating system support for dynamic page placement on a NUMA shared memory multiprocessor. In: Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Williamsburg, VA, April 1991, vol. 26(7), pp. 122–132 (1991)
9. Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M.: Scheduling and page migration for multiprocessor compute servers. In: ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, pp. 12–24. ACM Press, New York (1994)
10. SGI Incorporated: Speedshop user's guide. Technical Report 007-3311-003, SGI, Mountain View, CA (2003)
11. Verghese, B., Devine, S., Gupta, A., Rosenblum, M.: Operating system support for improving data locality on ccNUMA compute servers. In: ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, pp. 279–289. ACM Press, New York (1996)
12. Black, D., Sleator, D.: Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University (1989)
13. Petersen, K., Li, K.: An evaluation of multiprocessor cache coherence based on virtual memory support. In: Proceedings of the 8th International Symposium on Parallel Processing, pp. 158–164. IEEE Computer Society, Los Alamitos (1994)
14. Tikir, M.M., Hollingsworth, J.K.: Using hardware counters to automatically improve memory performance. In: SC 2004: Proceedings of the ACM/IEEE SC2004 Conference (SC 2004), p. 46. IEEE Computer Society, Los Alamitos (2004)