Preliminary Evaluation of Dynamic Load Balancing Using Loop Re-partitioning on Omni/SCASH

Yoshiaki Sakae
Tokyo Institute of Technology, Japan
sakae@is.titech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology/JST, Japan
matsu@is.titech.ac.jp

Mitsuhisa Sato
Tsukuba University, Japan
msato@is.tsukuba.ac.jp

Hiroshi Harada
Hewlett-Packard Japan,Ltd.
Hiroshi.Harada@hp.com

Abstract

Increasingly large-scale clusters of PC/WS continue to become majority platform in HPC field. Such a commodity cluster environment, there may be incremental upgrade due to several reasons, such as rapid progress in processor technologies, or user needs and it may cause the performance heterogeneity between nodes from which the application programmer will suffer as load imbalances. To overcome these problems, some dynamic load balancing mechanisms are needed. In this paper, we report our ongoing work on dynamic load balancing extension to Omni/SCASH which is an implementation of OpenMP on Software Distributed Shared Memory, SCASH. Using our dynamic load balancing mechanisms, we expect that programmers can have load imbalances adjusted automatically by the runtime system without explicit definition of data and task placements in a commodity cluster environment with possibly heterogeneous performance nodes.

1. Introduction

Recently, clusters of SMPs have become the majority in HPC machines[11]. In particular, commodity clusters are being used at many research organizations and universities for the cost performance and ease of management, etc. However, rapid progress of processor and network technology typically gives rise to *performance heterogeneity* due to incremental addition of nodes, incremental reinforcement of processors/memory, etc. Also, multi-user environment might result in such performance heterogeneity even if the nodes were homogeneous.

It would be difficult for a programmer to perform load balancing explicitly for every environment/application, and as such, automatic adaptation by the underlying runtime is indispensable. In this regard, we are developing technologies for commodity clusters that perform automatic rearrangement of data and dynamic load balancing, and employ OpenMP as the programming interface. More specifically, we have been developing Omni/SCASH[9], an implementation of OpenMP on SCASH, a Software Distributed Shared Memory, for commodity clusters. In this paper, we report on our implementation of performance monitoring feedback-based loop re-partitioning to cope with load imbalances in heterogeneous settings, and its measured performance. We also report on our ongoing work on dynamic data rearrangement based on SCASH page reference counting in the OpenMP *parallel* section. With these mechanisms, we expect that programmers can have load imbalances adjusted automatically by the runtime system without explicit definition of data and task placements.

2. Background

2.1. Omni OpenMP Compiler

The Omni OpenMP compiler is a translator, which takes OpenMP programs as input to generate a multi-threaded C program with runtime library calls. C-front and F-front are front-ends that parse C and Fortran codes into intermediate codes, called Xobject code. Exc Java tools is a Java class library that provides classes and methods to analyze and modify the program easily with a high level representation. The representation of Xobject code is a kind of AST (Abstract Syntax Tree) with data type information, each node of which is a Java object that represents a syntactical element of the source code, and that can be easily transformed. The translation from an OpenMP program to the target multi-threaded code is written by Java in the Exc Java tools. The generated program is compiled by the native back-end compiler and linked with the runtime library.

2.2. SCASH

SCASH[4] is a page-based software distributed shared memory system using the PM[10] low-latency and high bandwidth communication library for Myrinet[7] and memory management functions, such as memory protection, supported by the operating system kernel. SCASH is based on the Eager Release Consistency (ERC) memory model with multiple writer protocol and implemented as a user level runtime library. In the ERC memory model, the consistency of a shared memory area is maintained on each synchronization called the *memory barrier synchronization* point.

To realize memory consistency, invalidate and update page consistency protocols have been implemented. In the invalidate protocol, an invalidation message for a dirty page is sent to remote hosts where the page copy is kept at the synchronization point. In the update protocol, new data on a page is sent to remote hosts where the page copy is kept at the synchronization point.

In SCASH, the *home* node of a page is the node that keeps the latest data of the page and the page directory which represents the set of nodes sharing the page. The *base* is the node that knows the latest home node when the home migrates. All nodes know the base nodes of all pages.

2.3. Translation of OpenMP programs to SCASH

In the OpenMP programming model, global variables are shared by default. On the other hand, variables declared in the global scope are private for the processor in SCASH, and shared address space must be allocated explicitly by the shared memory allocation primitive at runtime. To compile an OpenMP program into the "shemem memory model" of SCASH, the compiler transforms code to allocate global variables in shared address space at runtime.

More specifically the compiler transforms an OpenMP program by the following steps:

1. All declarations of global variables are converted into pointers which contain the address of the data in shared address space.
2. The compiler rewrites all references to global variables to indirect references through the corresponding pointers.
3. The compiler generates global data initialization function for each compilation unit. This function allocates the objects in shared address space and stores these addresses into the corresponding indirect pointers.

The OpenMP primitives are transformed into a set of runtime functions which use SCASH primitives to synchronize and communicate between processors.

To translate a sequential program annotated with parallel directives into a fork-join parallel program, the compiler encapsulates each parallel region into a separate function. The master node calls the runtime function to invoke the slave threads which execute this function in parallel. All threads in each node are created at the beginning of execution, and wait for the fork operation on slave nodes. No nested parallelism is supported.

In SCASH, the consistency of all shared memory area is maintained at a barrier operation. This matches the OpenMP memory model. The lock and synchronization operations in OpenMP use the explicit consistency management primitives of SCASH on a specific object.

From the viewpoint of the programmer, our implementation for the SDSM is almost similar in behavior to the hardware-supported SMP implementations, except for several aspects such as granularity of coherence which may have performance implications.

2.4. Performance Degradation on Performance Heterogeneous Environment

Before we present extensions for dynamic load balancing, we demonstrate a typical performance degradation for performance heterogeneous settings.

Fig. 1 exemplifies the situation where the nodes are performance heterogenous. The cluster consists of 8 500MHz Intel Pentium III nodes, and 2 300MHz Intel Celeron nodes, and the earlier version of Omni/SCASH without any load balancing features are installed. The benchmark run is the SPLASH II Water benchmark. As one can observe from the Figure, the slower nodes dominate the critical path of the loop, and as a result, the entire cluster performs as if it were 8+2 300MHz nodes instead of performing as if it had the weighted average clockspeed of the two types of processors, which would have been ideal.

3. Dynamic Load Balancing

As mentioned in section 1, there are several conditions which gives rise to load imbalance: (1) the target application has inherently load imbalance, (2) there are differences in loads among the nodes due to multi-user environment, and (3) when an application is run on performance heterogeneous cluster. Since for cases (2) and (3) load imbalances cannot be determined a priori, static load balancing techniques would be insufficient, but rather dynamic load balancing techniques based on runtime performance measurement would be essential.

It is widely known that the locality of data affects performances in NUMA/SDSM environment, and as a result several data placement techniques have been proposed, such as (1) placing an extra initialization loop for some data

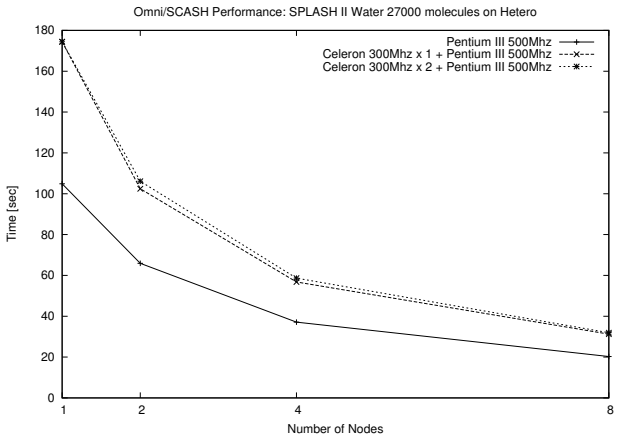

Figure 1. Execution Time of SPLASH II Water on Performance Heterogeneous Cluster

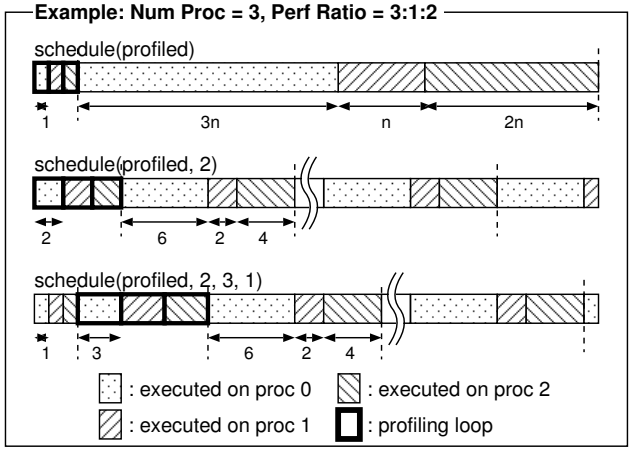

Figure 2. Example of profiled scheduling

which are accessed in the main loop as a preamble, to utilize the first-touch memory allocation mechanism supported by the system, (2) annotating affinity info between data and threads[5], and (3) application programmers describing data placement explicitly with directives. These are all static techniques, and consequently cannot deal with access pattern changes at runtime or dynamic load changes in nodes, especially those with most frequent access to certain data.

Instead, we propose a mechanism that combines dynamic loop re-partitioning based on runtime self-profiling of performance for dynamic load balancing, and dynamic page migration based on page reference counting for dynamic data placement. Because excessive loop re-partitioning may cause unnecessary page migrations, it is very important to investigate the balance between loop re-partitioning and page migration.

3.1. Directive Extension for Loop Re-partitioning

We use loop re-partitioning as a runtime load balancing function for data parallel applications. Although *dynamic* and *guided* scheduling options of OpenMP can achieve load balancing to some extent, both involve some lock and memory flush operations when access and update to the chunk queue managed centrally. Therefore they reveal relatively big scheduling overhead in a distributed memory environment like especially a cluster. Instead, we achieve load balance by adjusting the chunk size for each processor early on in the loop according to runtime performance profiled from the initial static OpenMP scheduling.

We have added the new scheduling policy **profiled** to the *schedule* clause as follows:

```
schedule(profiled[, chunk_size[,
                                eval_size[, eval_skip]]])
```

When *profiled* scheduling is specified, the number of times iteration specified by *eval_skip* is normally performed by each thread, and performance measurement is performed for the next number of times iteration specified by *eval_size*. Then, each thread will calculate the runtime peformance and the remaining iteration space will be partitioned cyclically to each thread according to a performance ratio with making *chunk_size* into a basis bottom chunk size.

In addition, you can omit eval_skip, eval_size and chunk_size in this order. When eval_skip is omitted or 0, the runtime peformance measurement is peformed from top iteration. When eval_size is omitted, the runtime peformance measurement is performed supposing eval_size = 1. When chunk_size is omitted or 0, the remaining iteration space will be partitioned in a block manner according to runtime performance ratio.

Fig. 2 is an example of profiled scheduling. In this case, the total number of processors is 3 and its performance ratio is 3:1:2 respectively.

When an application programmer specifies *profiled* scheduling for some parallel loop, Omni makes new subfunctions for each parallel region, and these are invoked on slave threads participating in the parallel region. In addition, Omni inserts time measurement code around the loops for measuring precise execution time of iterations, then repartitions the loop according to performance variations between threads for the target loop as in Fig. 3. For precise performance measurement, we utilize hardware real time counter supported by CPU via PAPI[2].

Each thread manages loop related information such as index of subloop and upper/lower bounds of subloop, and these values are initialized by `_ompc_profiled_sched_init()`. `_ompc_profiled_sched_next()` calculates the

```
#pragma parallel omp for schedule(profiled)
  for (i = 0; i < N; i++) {
      LOOP_BODY;
  }
```

```
static void __ompc_func(void **__ompc_args){
  int i;
  {
    int lb, ub, step;
    double start_time = 0.0, stop_time = 0.0;
    lb = 0, ub = N, step = 1;
    _ompc_profiled_sched_init(lb, ub, step, 1);
    while (_ompc_profiled_sched_next(&lb, &ub, start_time, stop_time)) {
      _ompc_profiled_get_time(&start_time);
      for (i = lb; i < ub; i += step) {
          LOOP_BODY;
      }
      _ompc_profiled_get_time(&stop_time);
    }
  }
}
```

Figure 3. Code translation when profiled is specified as scheduling policy

next iteration space of subloop for each thread. `_ompc_profiled_sched_next()` adjusts iteration space between threads according to performance ratio measured by `_ompc_profiled_get_time()` to achieve loop re-partitioning.

`_ompc_profiled_sched_next()` calculates loop re-partitioning as follows (see Fig. 4):

1. When a profiled (loop) scheduling is performed:
   (a) Calculate the execution speed of each thread for previous chunk of loops, and broadcast the info to all other threads.
   (b) Estimate the optimal time taken to perform the remaining loops when performing the loop re-partitioning according to the performance ratio, and also when the loop repartitioning is not performed.
   (c) When the performance improvement is above a certain threshold:
      i. each thread calculates the number of chunks for all threads and stores these in the `chunk_vector`.
      ii. each thread adjusts its own loop index, loop upper/lower bounds, etc., and exit.
   (d) Otherwise a flag is set to indicate that loop re-partitioning should not be performed further. Each thread merely calculates the next chunk of subloop based on `chunk_vector` calculated previously, and exit.
2. When a profiled schedule isn't performed, each thread merely calculates the next chunk of subloop based on the `chunk_vector` calculated previously, end exits.

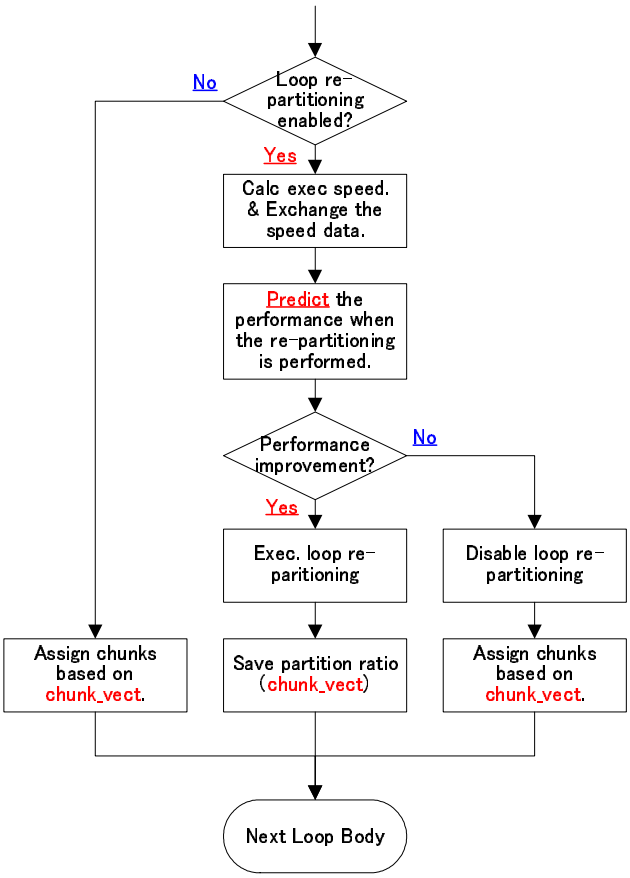

Figure 4. Overview of loop re-partitioning algorithm

Of the operations above, (a) alone involves communication with other threads. When performance difference between threads is not caused by the inherent load imbalance of the application itself, but is caused by the performance difference between nodes or loads, the communication occurs only for the first chunk of iterations, and remain static afterwards. As such, we expect performance advantage for profiled scheduling, compared to dynamic or guided scheduling that must access to shared data to calculate next iteration space every time.

4. Evaluation

We have used OpenMP versions of EP and CG from NAS Parallel Benchmarks (NPB-2.3) to evaluate profiled scheduling. The conversion to OpenMP version from original Fortran version was done by RWCP.

EP The EP kernel benchmark is a kernel that could be used in a Monte Carlo type procedure. $2n$ pseudorandom

Table 1. Evaluation Environment: Performance Heterogeneous Cluster

	Fast nodes	Slow node
CPU	Pentium III 500MHz	Celeron 300MHz
Cache	512KB	128KB
Chipset	Intel 440BX	Same as "Fast"
Memory	SDRAM 512MB	Same as "Fast"
NIC	Myrinet M2M-PCI32C	Same as "Fast"

floating point numbers are generated and some computations which depend only on pairs of neighboring elements are performed. The data dependency is minimal, hence the problem is called embarrassingly parallel. Therefore we use EP to evaluate the pure efficiency of profiled scheduling.

CG The CG kernel benchmark uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. There are substantial frequent accesses to the shared arrays in kernel loop of CG, so that the locality of data has significant influence on performance. This means that the possibility of performance degradation could overwhelm the gain for profiled scheduling.

4.1. Evaluation Environment

In Tab. 1, we show our evaluation environment. Our performance heterogeneous cluster has differences only in CPU performance such that some nodes are equipped with Pentium III 500MHz and the others are equipped with Celeron 300MHz.

We use RedHat 7.2 as an OS and SCore cluster system and gcc-2.96 as a compiler with the -O option.

In the following evaluation results, we only show the best case when chunk_size, eval_size and eval_skip are all omitted for profiled scheduling.

4.2. Results for Homogeneous Settings

Firstly, in order to examine the overhead of profiled scheduling itself, we conducted benchmarks on homogeneous settings using EP whose performance is not influenced by data placement.

In Fig. 5, while dynamic scheduling exhibits the overhead due to remote access to update the loop index at every chunk, we can see that profiled scheduling roughly exhibits no overhead equaling static scheduling performance.

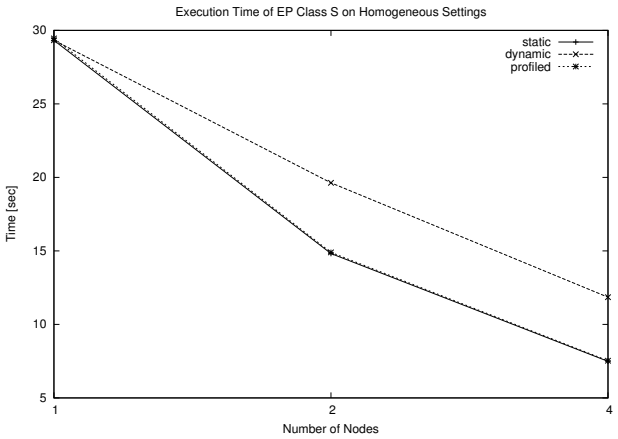

Figure 5. Execution Time of EP Class S on Homogeneous Settings (Pentium III nodes only)

4.3. Results for Heterogeneous Settings

Fig. 6 shows the execution time of EP class S with static, dynamic and profiled scheduling without specifying chunk size. The dashed line with square shows the EP performance with static scheduling when all nodes are "Fast", i.e. the upper bounds of performance. The solid line with plus sign shows the EP performance with static scheduling on heterogeneous settings and this is the lower bounds of performance.

Because the target loop's iteration space of EP is rather small, so that dynamic scheduling is feasible and exhibits the better performance than static scheduling.

When performing profiled scheduling, the target parallel loop body of EP is rather large, and we confirmed that the chunk_vector exactly corresponds to performance ratio of the two node classes. Profiled scheduling also shows the best performance for heterogeneous settings.

Fig. 7 shows the execution time of CG class A with static and profiled scheduling without specifying the chunk size.

The CG kernel benchmark involves frequent accesses to shared array, and as such it is important to allocate the task and its data on the same node, especially on a cluster environment where remote access has relatively high overhead. Moreover, because loop re-partitioning may occur at every "parallel for" region, such as a data initializing phase and data consumption phase, it is probable that affinity between the task and its data could be broken. Consequently, in the case of CG, profiled scheduling shows worse performance comparing to static scheduling. We investigate the influence which profiled scheduling has on data locality in the next subsection.

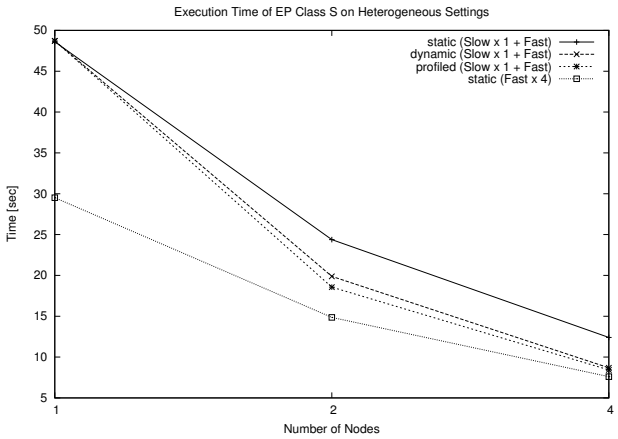

Figure 6. Execution Time of EP Class S on Heterogeneous Settings (one Celeron node + Pentium III nodes)

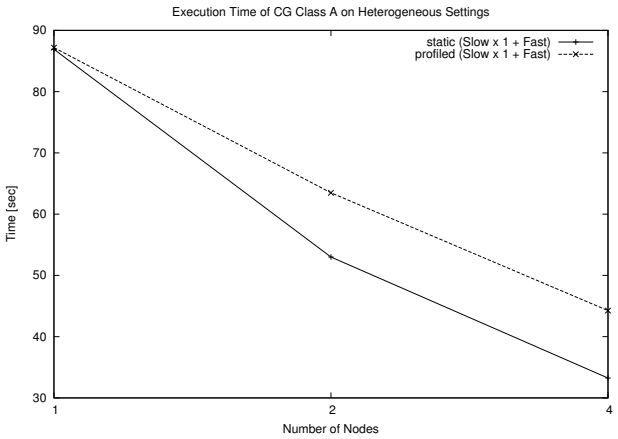

Figure 7. Execution Time of CG Class A on Heterogeneous Settings (one Celeron node + Pentium III nodes)

Table 2. Memory Behavior of the CG Class A on Celeron$\times$1 + Pentium III$\times$3 settings for static/profiled scheduling

	static	profiled
L2 miss ratio	29.6%	31.1%
Page Fault at SCASH	16456	27201
Barrier	5088	8006

4.4. Influences of Profiled Scheduling on Data Locality

Tab. 2 shows the numbers of barrier and SCASH level page fault on one node and L2 cache miss ratio. We compute L2 cache miss ratio with PAPI as follows:

$$\text{L2 cache miss ratio} = \frac{\text{PAPI\_L2\_TCM}}{\text{PAPI\_L1\_DCM}} \times 100$$

PAPI_L1_DCM means Level 1 data cache misses and PAPI_L2_TCM means Level 2 total cache misses.

Although static and profiled scheduling L2 cache miss ratios are similar, the number of SCASH level page fault of profiled scheduling is much greater than that of static scheduling. This is because affinity of a memory and a task may break whenever loop re-partitioning takes place with profiled scheduling. As a result, more remote memory accesses will occur and performance will degrade.

The number of barriers increases for the following reason: recall that our profiled scheduling uses barrier to broadcast the loop re-partitioning info to all the nodes. Now, because of remote references *without* page migration in the current setting, the predicted ratio of CPU performance may not be accurately reflected in the actual loops in the latter iterations, because of remote references occurring or not occurring. Such discrepancy will result in repeated loop re-partitioning, resulting not only in bad performance but also excessive barriers.

The results indicated that, it seems almost essential to combine page migration with dynamic loop re-partitioning in order to reestablish data locality. Also, we are considering loop re-partitioning to occur more gradually instead of re-partitioning immediately and exactly according to the instrumented performance; this will have a dampening effect which could prevent repeated repartitioning to occur, much in the way some numerical optimization algorithm work with dampening effects.

5. Ongoing Work: Page Migration Based on Page Reference Count

As mentioned in section 3, the locality of data affects performance especially in SDSM environment. However, there is no explicit method for a programmer to specify data placement with the current OpenMP standard, originally intended for shared memory environment. There have been proposals to improve OpenMP data placement locality on NUMA or SDSM environment: [1, 6] proposes directives to specify explicit data placement; [5] presents a scheme to align the threads with data with affinity directives; [8] migrates pages to a node on which the thread that most frequently access the data on the page reside using hardware page reference counters. For our work however, because we aim to employ loop re-partitioning, such static techniques would be difficult to apply directly.

Moreover, counting every page reference without hard-

ware support would result in considerable overhead in a SDSM environment. Instead, we count the number of page faults at the SDSM level, that is, the number of times when non-local memory has been accessed, and migrate the page to the node with the most number of (dynamic) remote references to the given page. Because we lose precision over direct counting of page references with hardware support, there is a possibility of increase in actual references to remote pages due to page migration. However, because we are targeting SPMD applications, we can assume that memory access pattern in kernel parallel loop will not typically change over each iteration. As such, we may safely assume that our approximated reference counting will have sufficient precision for our purpose.

In the current prototype, variables subject to migration and its size must be specified with directives. Because page reference information that affects page migration is largely caused by accesses in the (rather stable) main loop that is dominant with respect to the overall performance, we expect that good locality can be attained if the compiler can reduce the # of false sharings (which is achievable with appropriate loop partitions.)

6. Related Work

Nikolopoulos et al. proposed a data placement technique for dynamic page migration based on precise page reference counting in a parallel loop of OpenMP programs using hardware page reference counter supported by SGI Origin 2000[8]. Based on the accurate value of page reference counter during the proper code section, exact and timely page migration is attained. The results show that their method show better performance than dynamic page migration supported by OS with some programs of NPB. Our proposal will extend their results to commodity clustering environment where such hardware support does not exist.

Harada et al. implemented home reallocation mechanism base on the amount of page data changes to SCASH[3]. In their method, the system detects the node which has made the most changes of each page at every barrier synchronization point, then alters the "home" node of the page to be that node to reduce remote memory access overheads. The evaluation results with SPLASH2[12] LU benchmark shows that their execution performance is advantageous over static home node allocation mechanisms, including optimal static placement on up to 8 nodes.

7. Conclusion and Future Work

We reported our ongoing work on a dynamic load balancing extension to Omni/SCASH which is a implementation of OpenMP on Software Distributed Shared Memory, SCASH. We aim to provide a solution for solving inherent and dynamic load imbalance of application programs, namely load imbalance between nodes caused by multi-user environment, performance heterogeneity in nodes, etc. Static approaches for such situations would not be adequate and instead, we propose dynamic load balancing mechanism with loop re-partitioning based on runtime performance profiling for target parallel loops, and page migration based on efficient approximated page reference counting during the target loop sections. Using these techniques we expect that user programmers can achieve non-optimal load balance corrected by the runtime environment without explicit definition of data and task placement.

The results of our preliminary evaluation indicates that, when data locality is not a question, our re-partitioning scheme with profiled scheduling works well, causing almost no overhead compared to static techniques, and performing best when there is load imbalance due to performance heterogeneity. However, when locality is lost due to re-partitioning, performance instead degrades due to increased remote DSM references, and possibly more occurrences of barrier synchronizations. This suggests that profiled scheduling needs to be combined with dynamic page migration to restore locality; we have proposed a simple scheme where lightweight page reference count can be made without hardware support, at the sake of some accuracy. We are currently implementing our proposed page migration algorithm into Omni/SCASH, and hope to conduct extensive performance measurements in order to validate the effectiveness of our combined approach.

References

[1] J. Bircsak, P. Craig, R. Crowell, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA Machines: The Language. In *Proceedings of Workshop on OpenMP Applications and Tool (WOMPAT'2000)*, July 2000. San Diego, USA.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[3] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory system. In *Proceedings of IEEE 4th HPC ASIA 2000*, pages 158–163, may 2000.

[4] H. Harada, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, and Y. Ishikawa. SCASH: Software DSM using High Performance Network on Commodity Hardware and Software. In *Proceedings of Eighth Workshop on Scalable Shared-memory Multiprocessors*, pages 26–27. ACM, May 1999.

[5] A. Hasegawa, M. Sato, Y. Ishikawa, and H. Harada. Optimization and Performance Evaluation of NPB on Omni

OpenMP Compiler for SCASH, Software Distributed Memory System (in Japanese). In *IPSJ SIG Notes*, 2001-ARC-142, 2001-HPC-85, pages 181–186, Mar. 2001.

[6] J. Merlin. Distributed OpenMP: Extensions to OpenMP for SMP Clusters. In *Invited Talk. Second European Workshop on OpenMP (EWOMP'00)*, Oct. 2000. Edinburgh, Scotland.

[7] http://www.myri.com/.

[8] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP? In *Proc. of Supercomputing 2000*, Nov. 2000. Dallas, TX.

[9] M. Sato, H. Harada, and Y. Ishikawa. OpenMP compiler for Software Distributed Shared Memory System SCASH. In *Proceedings of Workshop on OpenMP Applications and Tool (WOMPAT'2000)*, July 2000. San Diego, USA.

[10] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. In P. Sloot and B. Hertzberger, editors, *High-Performance Computing and Networking '97*, volume 1225, pages 708–717. Lecture Notes in Computer Science, Apr. 1997.

[11] http://www.top500.org/.

[12] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.