Towards Dynamic Load Balancing Using Page Migration and 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
Compaq Computer, Japan
Hiroshi.Harada@jp.compaq.com

Abstract

Increasingly large-scale clusters of SMPs continue to become majority platform in HPC field. Such a cluster environment, there may be load imbalances due to several reasons and mis-placement of data which bring performance bottlenecks. To overcome these problems, some dynamic load balancing mechanisms are needed. In this paper, we report our ongoing work on dynamic load balancing extention 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[13]. There, it is necessary to program considering the two level memory hierarchy, namely, shared memory within a node and distributed memory among nodes. Past work has compared several programming methods, such as (1) programming with message passing only, (2) programming with shared memory intra-node and with message passing inter-node, (3) programming solely with shared memory when the target machine has underlying shared memory, such as NUMA or Software Distributed Shared Memory (SDSM). Each such approach has shown drawbacks: with (1) while one attains good performance, programmer's burden will be high, with (2) while dynamic load balancing may become easier with shared memory in a node, a programmer has to cope with multiple paradigms and resulting improvement in performance with respect to (1) is typically negligible if at all[7, 3]. Although coding becomes easier with (3) it is harder to achieve high performance due to locality of data being difficult to express with OpenMP.

Moreover, 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 each environment/application, and automatic adaptation by the underlying runtime is indispensable. In this regard, we are investigating programming environment for commodity clusters that perform automatic rearrangemnt of data and dynamic load balancing, and employs OpenMP as the programming interface. More specifically we have been developing Omni/SCASH[11], an implementation of OpenMP on SCASH, a Software Distributed Shared Memory, for commodity clusters. In this paper, we report on our implementation of dynamic data rearrangement based on SCASH page reference counting in the OpenMP parallel section, and performance monitoring feedback-based loop re-partitioning to cope with load imbalances in heterogeneous settings. 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. Figure 1 shows the structure of our compiler. 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.

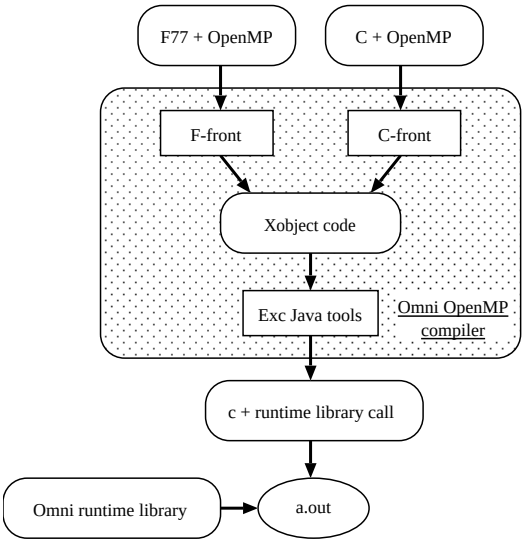

Figure 1. Omni OpenMP compiler

2.2 SCASH

SCASH[5] is a page-based software distributed shared memory system using the PM[12] low-latency and high bandwidth communication library for Myrinet[9] and memory management functions, such as memory protection, supported by the operating system kernel. SCASH is based on the Release Consistency (RC) memory model with multiple writer protocol and implemented as a user level runtime library. In the RC 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 "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 to 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 a viewpoint of the programmer, our implementation for the SDSM is almost similar in behavior to the hardware-supported SMP implementations, except for differences in various aspects such as granurality of coherence which may have performance implications.

2.4 Performance Degradation on Performance Heterogeneous Environment

Before progressing to the discussion on extensions for dynamic load balancing, we'll show performance degradation on heterogeneous settings.

Fig. 2 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.

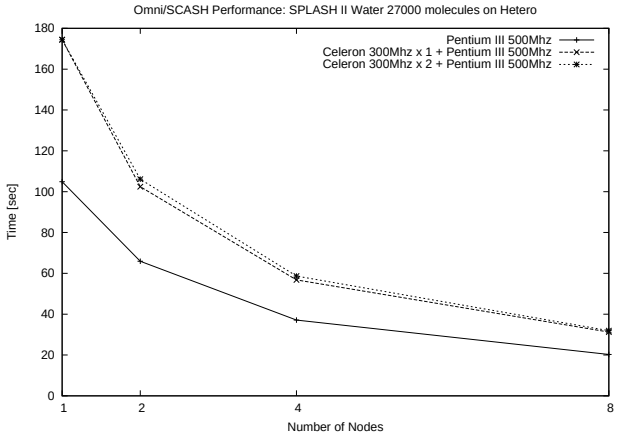

Figure 2. Execution Time of SPLASH II Water on Performance Heterogeneous Cluster

3 Dynamic Load Balancing

As mentioned in section 1, there are several conditions which gives rise to load imbalance: (1) the target application has load imbalance inherently, (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. In these cases, it is important to blance loads among the nodes to achieve sufficient performance. As static load balancing techniques would be insufficient, 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 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[6], 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 Extention for Loop Re-partitioning

We use loop re-partitioning as a runtime load balancing function for data parallel applications. Although *dynamic* and *guided* options of OpenMP can achieve load balancing to some extent, both involve some lock and memory flush operations when access to the range of loop indices allocated to each processors as a chunk of work must be managed centrally. 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])
```

When *profiled* scheduling is specified, a chunk of *chunk_size* iterations are assigned to each thread. When a thread finishes its assigned chunk of iterations, each thread calculates the next chunk size according to the result of runtime self-profiling. When no *chunk_size* is specified, it defaults to 1.

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 participated in the parallel region. In addition, Omni inserts time measurement code arround the loops for measuring precise execution time of iterations, then re-partition 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 informations such as index of subloop and upper/lower

```
#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

bounds of subloop, and these values are initialized by `_ompc_profiled_sched_init()`. `_ompc_profiled_sched_next()` calculates the 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:

1. When a profiled (loop) scheduling is performed:
   (a) Calculate 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 repartitioning according to 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 store 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 may not be performed anymore. Each thread merely calculates the next chunk of subloop based on `chunk_vector` calculated previously, and exit.
2. When a profiled schedule is't performed, each thread calculates the next chunk of subloop based on the `chunk_vector` calculated previously, end exit.

Of the operations above, (a) alone involves communication with other threads. When peformance 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 gain for profiled scheduling compared to dynamic or guided scheduling that must access to shared data to calculate next iteration space every time.

3.2 Page Migration Based on Page Reference Count

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, 8] proposes directives to specify explicit data placement; [6] presents a scheme to align the threads with data with affinity directives; [10] 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 hardware 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 targetting 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 couting 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.)

4 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[10]. 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[4]. 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[14] LU benchmark shows that their execution performance is advantageous over static home node allocation mechanisms, including optimal static placement on up to 8 nodes.

5 Conclusion and Future Work

We reported our ongoing work on a dynamic load balancing extention 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 dymamic load imbalance of application programs, namely load imbalance between nodes caused by multi-user envirionment, 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.

We are in the final stages of our prototype development, and expect to perform evaluation in the near future. In the evaluation, firstly, we will compare the effect of page migration to round-robin placement which is the default page placement of SCASH, and also with manual optimal data placement. Then, we will analyze the effect of loop re-partitioning with respect to the inherent load imbalance of the application. Since loop re-partitioning significantly affects the locality of data, it is important to investigate the balance between frequency of loop re-paritioning and page migration.

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] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Proceedings of Supercomputing '00*, Nov. 2000. Dallas, USA.

[4] 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.

[5] 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.

[6] 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.

[7] D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proceedings of Supercomputing '00*, Nov. 2000. Dallas, USA.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.