TR-CS-08-03

Predicting Performance of Intel Cluster OpenMP with Code Analysis Method

Jie Cai, Alistair P. Rendell, Peter E. Strazdins, H'sien Jin Wong

November 2008

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical-DOT-Reports-AT-cs-DOT-anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-08-02 Paul Thomas. *Implementation of PIS*. June 2008.

TR-CS-08-01 Stephen M. Blackburn, Sergey I. Salishev, Mikhail Danilov, Oleg A. Mokhovikov, Anton A. Nashatyrev, Peter A. Novodvorsky, Vadim I. Bogdanov, Xiao Feng Li, and Dennis Ushakov. *The Moxie JVM Experience*. April 2008.

TR-CS-07-05 Peter Strazdins. *Research-Based Education in Computer Science at the ANU: Challenges and Opportunities*. August 2007.

TR-CS-07-04 Stephen M. Blackburn and Kathryn S. McKinley. *Immix Garbage Collection: Fast Collection, Space Efficiency, and Mutator Locality*. August 2007.

TR-CS-07-03 Peter Christen. *Towards Parameter-free Blocking for Scalable Record Linkage*. August 2007.

TR-CS-07-02 Sophie Pinchinat. *Quantified mu-calculus with decision modalities for concurrent game structures*. January 2007.

Predicting Performance of Intel Cluster OpenMP with Code Analysis Method

Jie Cai, Alistair P. Rendell, Peter E. Strazdins, H'sien Jin Wong
College of Engineering and Computer Science
The Australian National University
{Jie.Cai, Alistair.Rendell, Peter.Strazdins, Jin.Wong}@anu.edu.au

Abstract

Intel Cluster OpenMP (CLOMP) extends OpenMP programs onto clusters by using page-based software Distributed Shared Memory (sDSM) system. With the commercial release of CLOMP, the interest in cluster-enabled OpenMP systems is likely to increase. Page faults detection and servicing are the major overheads of such systems. This paper presents a code analysis method to estimate the number of page faults caused when running an OpenMP Program with CLOMP. Then, utilizing SDP model to predict performance of CLOMP. On a 4-node Intel cluster connected via both InfiniBand (IB) and Giga-Ethernet (Giga-Eth) interconnects, some NAS Parallel Benchmarks (NPB) are used to evaluate and predict CLOMP. The estimates show less than ∼10% prediction error on most benchmarks, which indicates that the code analysis method-SDP model is an effective approach to predict performance of CLOMP.

1. Introduction

Intel Cluster OpenMP (CLOMP) is a cluster-enabled OpenMP system which provides support for the shared memory OpenMP programming model on distributed memory systems [2]. This offers not only increased portability for existing OpenMP programs, but also an alternative programming paradigm that is generally considered to be easier compared to the message passing model normally used on clusters. With the commercial release of the CLOMP compiler in 2006, interest in cluster-enabled OpenMP implementations is only likely to increase. This paper will present a code analysis method to help with understanding the performance of CLOMP.

Typically, cluster-enabled OpenMP implementations are implemented over page-based software Distributed Shared Memory (sDSM) systems [2, 9, 6, 7, 4]. This is viable since the relaxed memory consistency model of OpenMP limits the need to move globally shared memory pages between nodes to well-defined consistency points (i.e. OpenMP barrier and flush operations). Accesses to these shared memory pages are detected using page protection, with protection states of "Invalid", "Read-Valid", and "Write-Valid". Memory consistency is maintained by page fault servicing. The overhead of this can be characterized by the number and cost of different types of page faults. In this way, we can rationalize the performance of a cluster OpenMP program by utilizing the SIGSEGV Driven Performance (SDP) model [1].

In this paper, a code analysis method for CLOMP will be presented to estimate the number of different types of page faults that will occur when running an OpenMP program on CLOMP. Based on the SDP model, the proposed code analysis model could be used to predict the performance of CLOMP. Experiments are run on a 4-node cluster connected via both InfiniBand (IB) and Giga-Ethernet (Giga-Eth) interconnections, and some NAS Parallel Benchmarks (NPB) programs are used to evaluate the performance of CLOMP and verify the code analysis method.

The rest of the paper will be organized in five sections. In section 2, the background knowledge of CLOMP will be introduced, followed by description of the proposed code analysis method in section 3. Some NAS Parallel Benchmarks are analyzed to estimate the number of page faults in section 4. In section 5, the performance evaluation results are demonstrated and discussed. Conclusions are drawn in section 6.

2. CLOMP

The first commercial cluster-enabled OpenMP implementation is CLOMP, which is derived from the TreadMarks sDSM system [2], and uses a lazy release memory consistency model (LRC) [5]. It arguments OpenMP to include a new `sharable` directive that is used to identify variables that can be referenced by more than one OpenMP thread. These variables are placed in globally addressable memory, that is created, maintained, and synchronized across all OpenMP threads through the sDSM system. The TreadMarks sDSM system is *page-based*, partitioning the

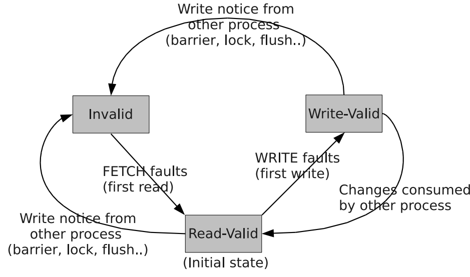

Figure 1: Page State Transfer Diagram for CLOMP (derived from [2], and experimental observation)

globally addressable memory into pages. Changes made to a page by other nodes are patched into the local view when required. The memory consistency model of CLOMP and performance models developed in [1] will be detailed in the rest of this section.

2.1. Memory Consistency Model

The LRC used in CLOMP is summarized in Figure 1 by showing the transitions between page states. Write notices are passed between threads when an OpenMP barrier, lock or flush directive is encountered. When a write notice is received for a shared page, it will be set to "Invalid". A subsequent read or write request to that page will then give rise to a *fetch* fault. This in turn requires diffs to be collected from the relevant other threads before the page can be used. Threads from which "diffs" have been requested (consumed) must change the protection for that page from "write-valid" to "read-valid" if necessary, indicating the start of a new "diff" reference point. Transitions from "read-valid" to "write-valid" occur the first time a write is made to a page, at which point a *write* fault will be issued, necessitating the creation of a copy or "twin".

2.2. SDP Performance Models

Two SIGSEGV Driven Performance (SDP) models were developed for cluster-enabled OpenMP systems in [1]. In the model, numbers and costs of different types of page faults have been used as the overhead to rationalize the performance of an OpenMP application running with cluster-enabled OpenMP. The estimated elapsed time for running an OpenMP program with such systems could be represented as the computation time plus the overhead. There are two types of SDP models. One utilizes the number of page faults caused along the critical path of running an OpenMP application, and another utilizes the aggregated number of page faults for the whole timed section. The critical path of running an OpenMP application is determined by the thread spending the longest time on handling page faults along each parallel region. Therefore, the critical path SDP model is more accurate but requires detailed knowledge of number of page faults caused on each processes within each parallel region. On the contrary, the aggregate SDP model only requires the aggregated number of page faults for the whole timed section.

A few assumptions are required to introduce the SDP models.

- The page faults caused on different processes are fully overlappable.
- The cost of servicing a fetch fault is constant regardless the number of processes involved.
- The computation time of p threads can be approximated by $\frac{T(1)}{p}$, where T(1) is the execution time on one thread. This in turn implies:
  - The OpenMP program is load-balanced.
  - The sequential portion within timed section is negligible.

In this case, the critical path SDP model for the total execution time on p threads can be expressed as:

$$T(p)^{crit} = \frac{T(1)}{p} + \sum_{r} Max_{i=0}^{p-1}\left(N_w^{r,i}C_w + N_f^{r,i}C_f\right) \quad (1)$$

The aggregate SDP model also assumes that an even number of page faults occur on each thread in each parallel region. under this model, the total execution time is given by:

$$T(p)^{agg} = \frac{T(1)}{p} + \frac{N_wC_w + N_fC_f}{p} \quad (2)$$

$N_w^{r,i}$ and $N_f^{r,i}$ stand for the number of write and fetch faults, respectively, occurring within parallel region r of thread i. N_w and N_f stand for the aggregate number of write and fetch faults respectively. C_w and C_f stand for the time spent on servicing a write and a fetch fault, respectively.

The page faults numbers can be obtained either by using segvprof.pl, a profile tool provided by Intel that reports the aggregated number of different type of page faults, or by analyzing OpenMP code directly. Additionally, the cost to service different type of faults can be measured by running a OpenMP testing program provided in [1][1].

[1]The source code of the program is available at http://ccnuma.anu.edu.au/dsm/segv_cost.

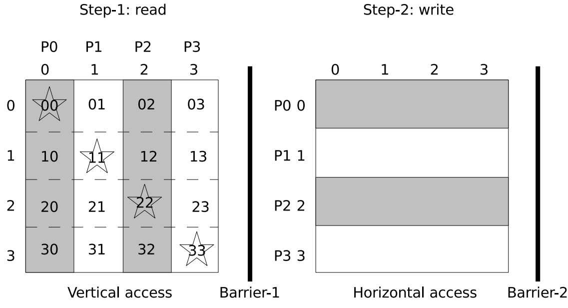

Figure 2: Memory Access Patterns on a shared matrix.

3. Code Analysis Method

The memory consistency work is driven by detecting and servicing page faults in sDSM systems. However, the page faults only happen when a process tries to access protected pages. According to Figure 1, the page protection will be set to "Invalid" when one process receives write notices from other processes at a synchronization point, or be set "Read-Valid" when modification of a "Write-Valid" page has been consumed by other processes. Generally, changing the memory access pattern will result in page faults. Therefore, by analyzing the memory access pattern of a piece of OpenMP code, we will be able to estimate number of page faults that will occur for running that code.

Figure 2 shows a typical example of memory access pattern changes for a parallel matrix transpose program. We will use this example to demonstrate the code analysis method, and how to use the method to estimate number of page faults. Figure 3 shows the OpenMP code for matrix transpose program.

We assume there are p parallel processes running on the matrix transpose program. As shown in above code, the four processes read from then write to the shared matrix with different patterns at 2 steps, between which there is an implicit synchronization barrier. Each process will have a local copy of the shared matrix. To work out how many page faults may happen on this example, we need to give some assumptions:

- The matrix contains $pn_p = N^2 * \texttt{sizeof(int)}/P$ pages ($P = 4KB$ is system page size for CLOMP), which uses i and j as row and column indexes respectively.
- The data distribution boundary matches with the system page boundaries.

At the implicit barrier at start of parallel region for the first iteration, as the shared matrix (`S`) has been initialized by process 0 previously, write notices of `S` will be passed

```
#define N (1<<12) /* total 16384 pages */
static int P[N/nprocs][N];
#pragma omp threadprivate(P)
static int S[N][N];
#pragma intel omp sharable(S)
.
.
  initial(S);
#pragma omp parallel default(none) shared(S)
  {
    int i, j, k, iter;
    int tmp;
    for (iter = 0; iter < 10; iter++) {
      /* step-1: transpose a vertical stripe to
         local sub-matrix */
      k = 0;
      #pragma omp for
      for( j = 0; j < N; j++ ) {
        for( i = 0; i < N; i++ ) {
          P[k][i] = S[i][j];
        }
        k++;
      } /* implicit barrier-1 */

      /* step-2: copy transposed local sub-matrix
         back to shared matrix S*/
      k = 0;
      #pragma omp for
      for( i = 0; i < N; i++ ) {
        for( j = 0; j < N; j++ ) {
          S[i][j] = P[k][j];
        }
        k++;
      }/* implicit barrier-2 */
    }
  } /* end of parallel region */
```

Figure 3: Matrix transpose OpenMP program.

to all other processes, and the corresponding page state will be set to "Invalid". Each process reads from n_p pages (vertical stripe) at step-1 on their own copy. The pages states will be changed to "Read-Valid", and n_p fetch faults will be caused at all processes except process 0 (P0). Post barrier-1, each process except P0 writes to n_p pages, which results in n_p write page faults and state of all these pages will be set to "Write-Valid". During barrier-2, the write notices will pass across all processes, and the pages modified by other process(es) will be set to "Invalid" on their own copy. For example, P0 passes write notices of the horizontal stripe 0 to all other processes, and then all other processes will set to "Invalid" the pages within horizontal stripe 0.

At the following iterations, each process read from n_p pages (vertical stripe) at step-1 that results in 3 blocks of "Invalid" pages been read accessed. For instance, P0 will read the "Invalid" block 10, 20, and 30 on its own copy and cause $\frac{n_p(p-1)}{p}$ fetch page faults, where p is the number of processes involved. At same time, the state of these pages will be set to "Read-Valid". When the fetch page faults happen, the modifications on the page will be fetched from all other processes have modified it immediately. For P0, the modifications on blocks 01, 02, 03 at the previous iteration will be consumed at this step, and the page state will be set to "Read-Valid" as well. However, the blocks marked

with stars have been kept "Write-Valid" and no page faults will be caused on these pages, as these blocks are always accessed by only 1 process. At step-2, each process will write to n_p pages horizontally that results $\frac{n_p(p-1)}{p}$ write page faults and change the state of corresponding pages to "Write-Valid". During barrier-2, the write notices will be sent to all other processes, and then each process will set "Invalid" to the pages modified by other processes.

Therefore, we will have the following equations to estimate the aggregate number of write faults and fetch faults happened on this example program.

$$N_f = N_w = (iter - 1)n_p(p - 1) + n_p(p - 1) \quad (3)$$

As process 0 does not cause any page faults at first iteration, the critical path will be determined by any other processes. Therefore, the number of write and fetch page faults along critical path are both $(iter - 1)\frac{n_p(p-1)}{p} + n_p$.

4. Code Analysis of Some NAS Parallel Benchmarks

The NAS Parallel Benchmarks were derived from Computational Fluid Dynamics (CFD) codes by NASA [3]. The whole suite of NPB OpenMP implementations consists of eight benchmarks. There are six different problem sizes for these benchmarks, which are class S, W, A, B, C and D. Class S refers to the smallest problem, and class C and D refers to largest problem size.

Some relatively easy NPB OpenMP benchmarks will be analyzed with the proposed code analysis method in the rest of this section, such as EP, IS, FT and CG. For the following analysis, N_f and N_w stand for number of estimated aggregated fetch and write faults respectively, and p is the number of processes.

4.1. EP

EP is an **E**mbarrassingly **P**arallel OpenMP program that generates pairs of Gaussian random deviates. The timed section of EP code only contains one parallel region with a parallel do loop and a reduction clause. There is no memory consistency work involved except the reduction of two scalar variables. Therefore, no page fault will occur for this benchmark.

4.2. IS

IS is an **I**nteger **S**ort OpenMP program. The main shared memory has been allocated as a two-dimensional array of size $p \times$ MAX_KEY. Each process is assigned a sub-array of MAX_KEY elements, and the number of pages contained in this sub-array is $n = \frac{\texttt{MAX_KEY*sizeof(int)}}{P}$. The timed section is 10 iterations of the major subroutine, a ranking function that contains a single parallel region with an explicit OpenMP barrier. The memory access pattern before the barrier and after the barrier are different. The access pattern before the barrier is shown below.

```
Pattern-1:
#pragma omp parallel
myid=omp_get_thread_num();
for( i=0; i<MAX_KEY; i++ )
    key_buff1[myid][i] = 0;
```

After the whole array been zeroed out, process myid creates a histogram of the input array in key_buff1[myid] (the pages corresponding to the input array are already "Read-Valid", so this causes no page faults). The access pattern after the barrier is as follows.

```
Pattern-2:
for( myid=1; myid<num_procs; myid++ ) {
    #pragma omp for nowait
    for( i=0; i<MAX_KEY; i++ )
        key_buff1[0][i] += key_buff1[myid][i];
}
```

Relating these to Figure 2, the first pattern corresponds to the step-2, where each process writes to one horizontal stripe of key_buff1[myid]. Access pattern-2 corresponds to step-1; for example, P0 sums the elements within blocks 10, 20, 30 to corresponding elements in block 00.

Within timed section before explicit barrier:

For process 0, the part of key_buff1[0] accessed by other process(es) before the timed section will be in an "Invalid" state. For other process(es), the part of key_buff1[myid] accessed by other process(es) before the timed section will be in an "Read-Valid" state. Therefore, when all processes write 0 to the sub-array owned by themselves with pattern-1, $n(p - 1)/p$ write and fetch faults will be caused on key_buff1[0] (process 0), and $n(p - 1)(p - 1)/p$ write faults will be caused on all other sub-arrays (other processes). Correspondingly, the state of memory page will be "Write-Valid" for the process that wrote to it, and "Invalid" for other processes after barrier.

Within timed section post explicit barrier:

The access pattern is changed to pattern-2, resulting $n(p - 1)/p$ write and fetch faults on key_buff[0] (all processes excepting process 0), and $n(p - 1)(p - 1)/p$ fetch faults on the $p - 1$ other sub-arrays (process 0 contributes $n(p - 1)/p$ fetch faults). Additionally, the state of key_buff1[0] will be set to "Invalid", and the state of other sub-arrays will be set to "Read-Valid" before the next iteration.

Hence, the aggregated number of page faults can be estimated as follows.

$$N_f = N_w = r \times n\frac{(p - 1)(p + 1)}{p} \quad (4)$$

In Equation (4), $r = 10$ stands for number of total repetitions. The parameter MAX_KEY and calculated number of pages n are listed in Table 1.

To be noted, process 0 will cause $rn(p-1)/p$ write faults and $2rn(p-1)/p$ fetch faults, whereas one of other process will cause rn write faults and $rn(p-1)/p$ fetch faults. Thus process 0 causes twice the fetch faults as other process, and the critical path of IS is determined by process 0.

Table 1: Parameters of different class size of IS, FT and CG benchmarks

Class	IS		FT				CG		
	MAX_KEY	n	d_1	d_2	d_3	n	n_a	r_{out}	n
A	2^{19}	512	256	256	128	32768	14000	15	27.34
C	2^{23}	8192	512	512	512	524288	150000	75	292.97

4.3. FT

FT performs a **F**ourier **T**ransform on a 2D plane for a 3D data structure. The main shared memory used in FT benchmark has been allocated to two 3D double precision complex arrays (u0, u1), and the indexes of the 3 dimensions are d_1, d_2, and d_3. The major arrays contain n shared pages, where $n = d_1 d_2 d_3 * \texttt{sizeof}(\texttt{structcomplex})/P$. The timed section of FT contains a non-loop section and a loop section following it. There are four different memory access patterns in the timed section, which are shown as Figure 4.

```
Pattern-1:                   | Pattern-2:
!$omp parallel do            | !$omp parallel do
do k = 1, d3                 | do k = 1, d3
   do j = 1, d2              |    do j = 1, d2
      access x(1, j, k)      |       do i = 1, d1
   enddo                     |          access x(i,j,k)
enddo                        |       enddo
                             |    enddo
                             | enddo
-------------------------------------------------------
Pattern-3:                   | Pattern-4:
!$omp parallel do            | !$omp parallel do
do j = 1, d2                 | do j=1,1024
   do k = 1, d3              |    q = mod(j, d1)+1
      do i = 1, d1           |    r = mod(3*j,d2)+1
         access x(i,j,k)     |    s = mod(5*j,d3)+1
      enddo                  |    access x(q,r,s)
   enddo                     | enddo
enddo                        |
```

Figure 4: Access pattern for NPB FT benchmark

The transitions between Pattern-2 and Pattern-3 correspond to Figure 2 with $d1 \times d2$ and $d3$ dimensions providing the effectively two-dimensional structure. It can be noted that, as $d1$ is sufficiently large and a power of two, no page boundaries span the row divisions, indicated on the step-2 of the figure.

Non-loop section:

u1 has been written to with pattern-1, and written again with pattern-2, then read with pattern-3. The total number of write faults caused is $\frac{n(p-1)}{d_1 p}$l; this is the same as number of fetch faults. However, as these number of page faults is negligible compared to n (see Table 1), these will be ignored.

u0 is accessed by calling subroutine fft within the non-loop section. As u0 was written to with pattern-2 prior to fft has been called, the page state of u0 is "Invalid". Then u0 is written to with pattern-3 in subroutine fft, resulting in $n(p-1)/p$ fetch and $n(p-1)/p$ write faults. The page state is still "Invalid" when the non-loop section completed.

Loop section: the number of iteration is 6.

u1 is accessed by calling subroutines evolve, fft and checksum. Within evolve, u1 is written to with pattern-2, resulting in $n(p-1)/p$ write faults and page state changed to "Invalid". While in the fft subroutine, u1 is firstly read and written with pattern-3 followed by being read and written with pattern-2, which causes $2n(p-1)/p$ write and $2n(p-1)/p$ fetch faults and the page faults has been changed to "Invalid". u1 is accessed in checksum with the pattern-4, which can cause a maximum of 1024 page faults. As this becomes negligible with increasing problem size (3.1% of class A and 0.2% of class C shared pages), this can be ignored.

u0 is accessed by calling subroutine evolve. Each process read from u0 with pattern-2. Therefore, $n(p-1)/p$ fetch faults happened at the first iteration.

To sum up, the estimates of aggregated number of fetch and write page faults could be calculated by following equation.

$$N_f = N_w = (2r + 2)\frac{n(p-1)}{p} \quad (5)$$

In Equation (5), r stands for number of iterations which is 6. The parameter d_1, d_2, d_3, and calculated number of page n are listed in Table 1 for different problem sizes.

As each process contribute same number of page faults, the critical path model for FT is equivalent to the aggregate model.

4.4. CG

The CG benchmark utilizes **C**onjugate **G**radient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. However, the sparse matrix array does not contribute any page faults for the timed section. The major shared arrays are the 1-dimensional double precision arrays x,z,p,q, and r with size of n_a. The number of pages contained in one of the arrays is $n = \frac{n_a \times \texttt{sizeof(double)}}{P}$. The timed section of CG is a do-loop with r_{out} iterations, which contains a inner loop with

r_{in} iterations. There are two different memory access patterns within the section, shown as below.

```
Pattern-1:            | Pattern-2:
!$omp parallel do| !$omp parallel do
do j = 1, na+1      | do j = 1, na+1
   access x(j)      |    do k=rowstr(j),rowstr(j+1)-1
enddo               |       access p(colidx(k))
                    |    enddo
                    | enddo
```

Within timed section and before inner loop:

The arrays are written (z, p, q), read (x) and read/write accessed (r) with pattern-1. As the arrays x, z, p, q have been accessed with pattern-1 before the timed section, no page faults will be caused, and the state of pages accessed by other processes is set to "Invalid". However, the z will cause n write faults except on the first iteration of the main loop.

Within timed section and within inner loop:

The array p is read accessed with pattern-2. As pattern-2 is difficult to analyze, a simulation was run that found out that $n(p-1)$ fetch faults are caused and the state of all pages is set to "Read-Valid". Then, p is written to with access pattern-1, which results n write faults on all processes and after the following barrier the state of the pages accessed by other processes is set to "Invalid". As some other arrays (q, r, z) are accessed with pattern-1, no page faults are caused on these shared arrays.

Within timed section and after inner loop:

The array z is read accessed with pattern-2, which causes $n(p-1)$ fetch faults and the state of all pages has been set to "Read-Valid". The arrays r, x are accessed with pattern-1. Therefore, no page faults are caused on r and x.

Hence, the aggregated number of page faults for timed section of CG could be estimated by following equation.

$$\begin{aligned} N_f &= r_{out}(1 + r_{in})n(p-1) \\ N_w &= (r_{out}(1 + r_{in}) - 1)n \end{aligned} \tag{6}$$

In Equation (6), r_{in} is 25 for all classes. The parameter r_{out}, n_a and calculated number of page n of different problem sizes are listed at Table 1.

As each process contribute same number of page faults, the critical path model for CG is equivalent to the aggregate model.

5. Experiments and Evaluation

A cluster, containing 4 Intel nodes each with Core2 Quad-core Q6600 2.4 GHz CPU and 4 GB memory, is employed to run NPB-OMP 3.2 benchmarks. The cluster is connected by both 4x DDR InfiniBand (IB) and Gigabit Ethernet (Eth).

The Intel C/Fortran compiler 10.0 was used in the experiments. The benchmarks were compiled with -O2 and -cluster-openmp flags. NPB-OMP 3.2 benchmarks EP, IS, FT and CG are used in the experiments. NPB OpenMP C program is ported to CLOMP by adding "sharable" directive for shared variables, and Fortran code is ported by compiling with -clomp-sharable-commons -clomp-sharable-localsave -clomp-sharable-modvars flags addition to adding "sharable" directives. segvprof.pl is used to profile the numbers and types of page faults.

The prediction and actual performance data of NPB-OMP class A and C is demonstrated and analyzed against number of page faults estimated by code analysis method in this section.

5.1. Number of Page Faults Estimation

Based on Equations (4), (5) and (6), we can estimate the aggregate number of page faults (aggr) that occurred in the timed section of the IS, FT and CG benchmarks. The number of page faults along critical path (crit) for FT and CG are estimated by averaging the aggregate numbers. The critical path number of page faults for IS, as we discussed in the end of section 4.2, is determined by process 0. The actual number of page faults are obtained by using the segvprof.pl tool. The comparison between the estimates and the actual number of page faults is shown in Table 2. The estimates is presented as percentage error to actual number of page faults (0 means exactly same).

The most immediate observation from Table 2 is the lack of page faults for EP in both the actual and estimated case. This suggests this benchmark will scale equally well on CLOMP as for a non-sDSM OpenMP implementation.

Only for the aggregate number of page faults of CG class A, a big difference between actual and estimated results are observed which becomes larger with increase number of processes. This is due to two major factors. The first factor is that the code analysis assumes that the boundary of pages distribution among all processes will match the system page boundaries. The second factor is that the shared memory contains only a few pages (∼27.34 for CG). As the shared memory is spread over a non-integral number of pages for CG, this scenario is even worse. Moreover, the uncertain components of code analysis estimate of FT and CG (please refer to section 4) contribute to the difference between estimated and actual number of page faults as well.

With the increasing problem sizes (class C), the estimates agree with the actual numbers of page faults better. There is no difference for IS class C. Due to statically allocated memory space is too large to fit in the Linux .bss file, FT class C can not be compiled at our machine. The difference for CG decreases from ∼40% to ∼5% with problem size increasing from A to C. However, the estimates for rest benchmarks (including MT) show a small percentage error ($< 7\%$)

Table 2: Comparison of actual and estimated aggregate number of page faults counts for the EP, IS, FT and CG benchmarks and matrix transpose example program (MT)

Bench-marks			2 processes		4 processes	
			Write	Fetch	Write	Fetch
MT	aggr	Actual	8.60E+4	8.60E+4	1.26E+5	1.32E+5
		Estimate	4.8%	4.8%	2.4%	6.8%
	crit	Actual	4.51E+4	4.92E+4	3.17E+4	3.48E+4
		Estimate	0	8.3%	0	8.9%
Class A						
EP	aggr	Actual	0	0	0	0
		Estimate	0	0	0	0
	crit	Actual	0	0	0	0
		Estimate	0	0	0	0
IS	aggr	Actual	7.68E+3	7.67E+3	1.92E+4	1.92E+4
		Estimate	0	0	0	0
	crit	Actual	2.56E+3	5.10E+3	3.84E+3	7.66E+3
		Estimate	0	0	0	0
FT	aggr	Actual	2.34E+5	2.34E+5	3.52E+5	3.56E+5
		Estimate	2.1%	2.1%	2.3%	3.4%
	crit	Actual	1.18E+5	1.18E+5	8.83E+4	8.96E+4
		Estimate	2.5%	2.5%	2.6%	4.0%
CG	aggr	Actual	1.71E+4	1.75E+4	2.44E+4	4.96E+4
		Estimate	38.0%	38.9%	56.6%	35.5%
	crit	Actual	8.54E+3	8.95E+3	6.21E+3	1.32E+4
		Estimate	37.9%	40.2%	57.3%	39.4%
Class C						
EP	aggr	Actual	0	0	0	0
		Estimate	0	0	0	0
	crit	Actual	0	0	0	0
		Estimate	0	0	0	0
IS	aggr	Actual	1.23E+5	1.23E+5	3.07E+5	3.07E+5
		Estimate	0	0	0	0
	crit	Actual	4.10E+4	8.19E+4	6.14E+4	1.23E+5
		Estimate	0	0	0	0
CG	aggr	Actual	6.03E+5	6.05E+5	6.39E+5	1.80E+6
		Estimate	5.3%	5.3%	10.6%	5.0%
	crit	Actual	3.02E+5	3.03E+5	1.60E+5	4.53E+5
		Estimate	5.3%	5.6%	10.6%	5.5%

For the number of page faults along critical path, we can observe the similar pattern with aggregate numbers. With the increasing problem size, the estimates agree better with observations. In summary, the code analysis method estimates show a small percentage error for most benchmarks (5% in average).

Whether a particular benchmark performs well using CLOMP will depend on how the total number of page faults compares with the overall execution time. Ignoring EP, the page fault counts given in Table 2 are found to vary by around three orders of magnitude between the different benchmarks in a given class. On the other hand the sequential execution times given in Table 3 vary by about two orders of magnitude. This suggests that the different benchmarks will show very different behavior when run using CLOMP. This is indeed the case as evident from the observed times given in Table 4, where speedups for four processors range from around four for EP to a slowdown of two or more for CG class A. For this reason the NPB OMP suite represents an interesting and challenging problem set for cluster-enabled OpenMP implementations and is a good candidates to validate our prediction approach.

Table 3: Sequential elapsed time (sec) for NPB-OMP 3.2 benchmarks and matrix transpose example program (MT)

Benchmarks	CG	EP	FT	IS	MT
A	2.73	17.81	5.94	0.38	4.23
C	347.19	288.53		45.14	

5.2. Performance Prediction and Evaluation

The sequential elapsed time for these benchmarks is obtained by running NPB-OMP 3.2 with 1 CLOMP threads, which is shown in Table 3. To be noted, the data listed in this section are minimum of three runs, and the maximum deviation among all benchmarks is $\sim$12%.

As demonstrated in the previous section, the number of page faults could be approximately estimated via code analysis. We will further utilize the estimated number of page faults in Table 2 with the SDP models to predict the performance of NPB benchmarks. The measured C_w on the 4-node cluster is $\sim 10\mu$s, and C_f is $\sim 67\mu$s for IB connection and $\sim 171\mu$s for Giga-Eth connection. The observed speedups (obs), the aggregate model predicted speedup (aggr) and the critical path model predicted speedup (crit) for running NPB-OMP benchmarks on the 4-node cluster are listed in Table 4.

Firstly, we concentrate on the observed speedups. The most immediate observation is that all class A benchmarks, except EP, performs very poor. This is due to a few reasons. The first reason is that the sequential execution time for all class A benchmarks (except EP) is very small (less than 6 second except EP) compared to the overhead brought by servicing page faults is even smaller. The second reason is that the overhead brought by memory consistency work (servicing page faults) dominate the overall performance. As the data distribution boundary may not be as same as the system page boundary for class A, more than one process will access the same page at same time. Then, post synchronization point, all processes accessing the page will request data from all other processes modified the page, resulting in $O(p)$ communication for every single process, where n is number of processes accessed the page [8].

The situation is getting better for larger problem size, class C, which is because that the sequential execution time sharply increased and the chance for the scenario that multiple processes accessing one page has been significantly reduced. For example, CG and IS show 1.63 and 1.32 speedups, respectively, with 4 processes. On IB connections, all benchmarks show speedup on 4 processes and the performance is much better than on Giga-Eth connection, such as 2.11 for CG, 2.2 for IS. The relatively smaller cost for servicing fetch page fault contributes to the difference of performance between IB connection and Giga-Eth connection.

Table 4: Observed and predicted speedups for NPB-OMP 3.2 benchmarks and matrix transpose (MT) example program

benchmarks	nprocs	Class A			Class C		
		obs	aggr	crit	obs	aggr	crit
Giga-Eth Connection							
MT	2	0.35	0.43	0.39			
	4	0.40	0.60	0.58			
CG	2	0.31	1.21	0.92	1.30	1.72	1.52
	4	0.23	1.10	0.91	1.63	2.35	2.09
EP	2	2.10	2.00	2.00	2.09	2.00	2.00
	4	4.17	4.00	4.00	4.18	4.00	4.00
FT	2	0.21	0.25	0.24			
	4	0.27	0.34	0.34			
IS	2	0.32	0.88	0.35	1.17	1.71	1.22
	4	0.24	0.61	0.26	1.32	2.28	1.37
InfiniBand Connection							
MT	2	0.65	0.78	0.72			
	4	0.93	1.18	1.14			
CG	2	0.54	1.57	1.33	1.40	1.87	1.76
	4	0.47	1.92	1.67	2.11	3.11	2.92
EP	2	1.99	2.00	2.00	2.06	2.00	2.00
	4	3.97	4.00	4.00	4.14	4.00	4.00
FT	2	0.45	0.50	0.49			
	4	0.63	0.70	0.69			
IS	2	0.60	1.26	0.68	1.52	1.85	1.58
	4	0.54	1.16	0.58	2.20	3.01	2.23

Secondly, we concentrate on the SDP model prediction results and its comparison with observations. As discussed above, the memory consistency cost (page servicing) may dominate the performance when problem size is not large, the number of page faults would be a useful determinant to predict the performance. In Table 4, for class A, aggregate SDP model prediction results is far from the observation, except EP and FT. This is because multiple processes accessing the same page is occurring frequently for CG and the number of page faults does not evenly distributed cross processes for and IS (P0 causes twice the number of fetch faults than other processes), both of which break the assumptions for the aggregated SDP model. However, the critical path model predictions are accurate except for CG, which is due to the serial sequential part of CG program within the timed section being non-negligible [1]. More importantly, as the page number is non-integral for CG and the data distribution boundary does not match the system page boundaries, multiple processes will access different parts of the same page. For class C, the aggregate SDP model still performs worse than critical path model. However, both models show an improvement for CG, as the two factors mentioned above become less significant. Overall, the SDP model shows relative error less than 10% for most benchmarks on both class A and C. Whereas the aggregated model shows a worse prediction on IS, which suggests a trade-off between effort put into code analysis and the prediction accuracy.

The prediction error for MT is due to uneven distribution of page faults, as we discussed before, other thread cause more page faults than P0.

6. Conclusion

Intel Cluster OpenMP shows reasonable performance on NPB-OMP 3.2 class C benchmarks on InfiniBand connection. However, the performance of smaller problem size (class A) is not satisfactory on both Giga Ethernet and InfiniBand cluster. With the assistance of code analysis method and SDP model, we understand that the memory consistency work may dominate the overall performance of an OpenMP program when the computation time is relatively small. Additionally, the proposed code analysis method has been verified together with SDP models to be able to predict performance of OpenMP program on CLOMP. Moreover, the code analysis method would also be used as a guide for designing cluster OpenMP programs.

7. Acknowledgment

This research is funded by ARC Linkage Grant LP0669726 with support from Intel Corporation and APAC National Facility at the ANU.

References

[1] J. Cai, A. P. Rendell, P. E. Strazdins, and H. J. Wong. Performance model for cluster-enabled OpenMP implementations. In *13th IEEE Asia-Pacific Computer Systems Architecture Conference*, page accepted, 2008.

[2] J. P. Hoeflinger. Extending openmp to clusters. *White Paper, Intel Corporation*, 2006.

[3] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. *Technical Report: NAS-99-011*, 1999.

[4] Y.-S. Kee, J.-S. Kim, and S. Ha. Parade: An openmp programming environment for smp cluster systems. In *Proceedings of the ACM/IEEE SC2003 conference on High Performance Networking and Computing*, page 6, 2003.

[5] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[6] M. Sato, H. Harada, and Y. Ishikawa. OpenMP compiler for a software distributed shared memory system SCASH. In *WOMPAT2000*, July 2000.

[7] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99*, pages 32–39, Sept. 1999.

[8] H. J. Wong, J. Cai, A. P. Rendell, and P. E. Strazdins. Micro-benchmarks for cluster OpenMP implementations: Memory consistency costs. In *IWOMP 2008, LNCS 5004*, pages 60–70, 2008.

[9] H. J. Wong and A. P. Rendell. The design of mpi based distributed shared memory systems to support openmp on clusters. In *Cluster07, IEEE Catalog Number 07EX1855C*, pages 231–240, 2007.