Part I

Introduction and Background

This thesis presents my PhD research about evaluating, modeling and enhancing the performance of cluster OpenMP systems. This part consists of two chapters. In the first Chapter, we present the thesis motivation and objectives, followed by a summary of contributions. Chapter 2 describes background knowledge of:

- OpenMP programming model,
- Software Distributed Shared Memory (sDSM) systems,
- Intel Cluster OpenMP (CLOMP) implementation,
- Alternative approaches to sDSMs.

and reviews related work on:

- Existing performance models,
- Page prefetch techniques for cluster OpenMP systems,
- Run-length encoding methods.
Introduction

Contents

1.1 Motivation ................................................. 4
  1.1.1 Research Objectives ................................. 5

1.2 Contributions ............................................. 6
  1.2.1 Performance Evaluation of CLOMP ................. 6
  1.2.2 Region-based Performance Models ................. 7
  1.2.3 Region-based Prefetch Techniques ................. 8

1.3 Thesis Structure ............................... 9
Cluster OpenMP systems [38, 98, 83, 84, 50, 42] provide programming environments which enable the use of OpenMP programming model [75, 76] on distributed memory architectures such as clusters. These systems utilize software Distributed Shared Memory (sDSM) systems [58, 10, 32, 52, 87, 59, 89, 34, 40] to construct and maintain a virtual global shared memory space across all computer nodes. In late 2006, Intel released the first commercial Cluster OpenMP system, Intel Cluster OpenMP (CLOMP) [38], with support of Intel C/C++ and FORTRAN compiler. Consequently, the research interest on such systems increased dramatically.

In this thesis, a set of region-based techniques are investigated and designed to evaluate, model, and enhance the performance of page-based cluster OpenMP implementations such as CLOMP.

1.1 Motivation

Parallel applications usually exhibit irregular and/or dynamic behaviour. Development of these applications for distributed memory architectures using the de facto standard programming model, Message Passing Interface (MPI) [74, 67, 68], has been considered a hard task. This is because the MPI model requires that programmers explicitly specify all inter-process communications. This not only leads to good scalability and performance, but also can be very tedious and is potentially error prone.

On the other hand, the share memory programming model, such as OpenMP that utilizes a fork-join like parallelism approach, limits the need of data communication between threads to well defined consistency points (i.e. OpenMP barrier and flush operations). This has been shown to simplify the development of parallel applications [35]. This feature frees programmers from explicit data communication for synchronization operations which greatly improves programmability on distributed memory architectures. Hence, researchers have made many attempts to utilize the programmability advantage of OpenMP over clusters. One of these attempts is known as cluster OpenMP systems, which typically consist of a software Distributed Shared Memory (sDSM) system, a communication layer and a support compiler. Inheriting the good programmability of OpenMP programming model, cluster OpenMP systems are used as alternative programming models on distributed memory architectures.

Cluster OpenMP systems are usually built on page-based sDSM systems, such as IVY [58], Munin [10], TreadMarks [51, 5], SCASH [34, 73], SCLIB [97], Danui [98], ParADE [50], JIAJIA [41, 40] and Delphi [89, 88], which partition the virtual shared memory space into equally sized blocks, called pages. Based on the
1.1 Motivation

memory models used, page-based sDSMs can be further categorized as home-based and homeless system as follows.

- **Home-based** sDSM: each shared page is assigned a home where the master copy of the page is maintained. The status of the page at its home defines the most up-to-date view of that page. These systems include SCASH, SCLIB, Danui, ParADE, and JIAJIA.

- **Homeless** sDSM do not assign homes to shared pages. Instead, changes made to pages by other process are patched into the local view when required. Systems which take this approach include IVY, Munin, and TreadMarks.

Consequently, cluster OpenMP implementations are also divided into these two categories, home-based, such as Omni-SCASH [84, 83, 82], and homeless, such as CLOMP [38].

The most notable cluster OpenMP implementation is the Intel Cluster OpenMP (CLOMP), which is the first commercial such system released in 2006 with the Intel C/C++ and FORTRAN compilers [38]. The sDSM layer of CLOMP is derived from TreadMarks [51, 5, 6, 7, 24], which manages and maintains a virtual global shared memory space across different computing nodes. The communication layer of CLOMP utilizes socket APIs for Ethernet communication and the user Direct Access Programming Library (uDAPL) [22] for high performance interconnects with remote direct memory access (RDMA) support, such as InfiniBand [46]. CLOMP is the first cluster OpenMP system deployed for high performance interconnects.

Cluster OpenMP systems showed various performance for different applications. Reasonable performance was observed for some scientific application, such as selected workloads of Gaussian03 quantum chemistry code [100], and unsatisfactory performance was observed for other applications or benchmarks [92, 17, 96]. Therefore, it is important to understand the performance of cluster OpenMP system by identifying its major overhead, and effectively reduce it.

1.1.1 Research Objectives

Understanding the programmability benefits as well as the performance limitations of current cluster OpenMP systems, this thesis aims to provide a quantitative performance analysis for a current sDSM system, and then effectively reduce its major overhead and consequently enhance its performance.
Chapter 1: Introduction

1.2 Contributions

In order to achieve the research objectives, we investigated and designed a set of region-based techniques to evaluate, model and enhance performance of page-based cluster OpenMP systems, such as CLOMP.

Firstly, the performance of CLOMP is quantitatively evaluated using different programs on various hardware platforms. Secondly, based on the well defined OpenMP consistency points, we decompose OpenMP programs into parallel and sequential regions. Thirdly, two region-based performance models for the core operation in CLOMP are presented. Fourthly, a stride-augmented run-length encoding (sRLE) method is developed to effectively reconstruct and compress historical page misses records, and facilitate an efficient analysis of their temporal and spatial behaviour. Lastly, based on the proposed sRLE method, three region-based prefetch (ReP) techniques are designed and implemented to improve the performance of CLOMP. As a result, the ReP techniques significantly improve both the prefetch efficiency and the coverage based on some well-known page prefetch techniques. Moreover, the major overhead of CLOMP is significantly reduced by the ReP techniques.

Since CLOMP is the first commercialized cluster OpenMP implementation, and its supporting compiler, generates unique region IDs for OpenMP programs, it is selected as the target research platform. We use CLOMP as the target system to address these issues throughout this whole thesis. To be noted, these contributions are not limited to CLOMP, and they can be applied to other page-based cluster OpenMP systems as well. The details of these contributions are summarized in the following three sections.

1.2.1 Performance Evaluation of CLOMP

The CLOMP system is quantitatively evaluated using the NAS Parallel Benchmarks OpenMP suite (NPB-OMP) on different hardware platforms, and the page faults servicing cost, also known as the memory consistency cost, is identified as the major system overhead by breaking down elapsed time. Furthermore, since all existing OpenMP benchmarks only measure individual OpenMP operations, we co-designed a memory consistency cost benchmark, called MCBENCH [99, 96], to measure the memory consistency cost of OpenMP implementations. It is used in this thesis to characterise the system overhead of CLOMP.

In the evaluation, NPB-OMP and MCBENCH were run on clusters with different CPU architecture and interconnects, such as Gigabit Ethernet, DDR
1.2 Contributions

InfiniBand and QDR InfiniBand. In addition, MCBENCH is also used to compare the performance of different shared memory systems including both hardware and software implementations. In this thesis, MCBENCH is also used to measure and compare the memory consistency cost of the original CLOMP and the RePs enhanced CLOMP.

Partial details of the MCBENCH and the performance evaluation results have been published in the following articles:


1.2.2 Region-based Performance Models

To further understand the performance of cluster OpenMP systems, we have identified parallel and sequential regions of OpenMP programs, and developed two region-based SIGSEGV driven performance (SDP) models based on the page fault statistics of each OpenMP program region.

The first model uses critical path analysis [39, 86] and requires a detailed knowledge of the number and type of page faults occurring for each thread in each parallel region. The second takes a more holistic approach requiring just the aggregate number of page faults occurring for all threads.

These models propose that the performance of an OpenMP application running on a cluster can be estimated from the number and type of page faults encountered, and knowledge of the approximate cost of these events as obtained from running a simple OpenMP program on just two nodes of the cluster.

Partial details of the region-based SIGSEGV driven performance models have been published in the following article:
1.2.3 Region-based Prefetch Techniques

Based on patterns of memory accesses and page misses observed in different OpenMP programs, three region-based prefetch (ReP) techniques are developed to reduce the dominant overhead of cluster OpenMP systems. These techniques improves on existing page prefetch techniques in sDSM systems.

We have firstly developed the temporal ReP (TReP) and hybrid ReP (HReP) techniques to consider temporal and spatial memory access patterns, with the following contributions.

• Based on regions, more comprehensive observations were made for paging behaviours of running different OpenMP programs on cluster OpenMP systems, which lead to an accurate assumptions for prefetch techniques.

• Two region-based prefetch (ReP) techniques were designed based on historical execution records of different regions to address the above assumptions.
  – TReP addresses the temporal paging behaviour between consecutive executions of a region.
  – HReP addresses both the temporal paging behaviour between consecutive executions of a region and spatial paging behaviour within a region execution.

TReP and HReP greatly reduce number of page misses for the applications exhibiting temporal paging behaviour between consecutive executions of a region. However, for the applications exhibiting dynamic paging behaviour between consecutive executions of a region, neither technique performs well.

For the further improvement, we develop a novel stride-augmented run-length encoding (sRLE) method to reconstruct page misses records which facilitates much more accurate and efficient analysis for dynamic memory access patterns. Based on sRLE, the dynamic ReP (DReP) technique is developed which successfully addresses dynamic paging behaviour between executions of a region. The follows list the associated contributions.
1.3 Thesis Structure

- The dynamic paging behaviour, including both temporal and spatial locality between consecutive region executions, of OpenMP programs with CLOMP are summarised via detailed analysis of two LINPACK OpenMP benchmarks.

- A novel stride-augmented run-length encoding method is developed to effectively compress and restructure page miss records for executions of regions. It successfully facilitates an efficient analysis of dynamic paging behaviour.

- DReP addresses both the temporal and spatial paging behaviour between region executions.

- All three ReP techniques are implemented in the CLOMP runtime. The implementation issues and challenges are described in detail.

As data are aggregated by the ReP techniques, network latencies are amortized and the benefit brought by high performance interconnects is leveraged. Details of the how regions are identified and partial details of the prefetch techniques have been published in following articles:


1.3 Thesis Structure

Chapter 2 presents background material on the OpenMP programming model, cluster OpenMP systems, and related work on performance models and existing prefetch techniques for sDSM systems. Chapter 3 quantitatively evaluates the performance of Intel Cluster OpenMP. Chapter 4 presents how to identitied regions of OpenMP program, and two region-based performance models, which rationalise the performance of CLOMP to the number and type of page misses with their associated costs. Chapter 5 details the design of three region-based page prefetch techniques, and the offline simulations that compare the proposed ReP techniques with other well-known page prefetch techniques. Chapter 6 presents the implementation issues of ReP techniques, and evaluation of ReP enhanced CLOMP systems. Lastly, conclusions and future work are presented in Chapter 7.
Chapter 2

Background

Contents

2.1 OpenMP ........................................ 12
   2.1.1 OpenMP Directives ...................... 12
   2.1.2 Synchronization Operations ............ 16
2.2 Cluster OpenMP Systems ..................... 17
   2.2.1 Relaxed Memory Consistency ............ 18
   2.2.2 Software Distributed Shared Memory Systems  19
   2.2.3 Intel Cluster OpenMP .................... 23
   2.2.4 Alternative Approaches to sDSMs ....... 26
2.3 Related Work .................................. 29
   2.3.1 Performance Models ..................... 29
   2.3.2 Prefetch Techniques for sDSM Systems  31
   2.3.3 Run-Length Encoding Methods .......... 35
2.4 Summary ...................................... 37
Chapter 2: Background

This thesis is about evaluating, modeling and enhancing the performance of cluster OpenMP systems. This chapter provides background information on cluster OpenMP systems, especially Intel Cluster OpenMP (CLOMP). It also reviews related work to model and improve performance of these systems.

In general, cluster OpenMP systems extend OpenMP programs to clusters utilizing software Distributed Shared Memory (sDSM) [58, 10, 32, 52, 87, 59, 89, 34, 40] systems to create global virtual shared memory spaces across all processes (nodes) and maintain consistency by deploying the different memory consistency models. The virtual shared memory is protected against access if it is not up-to-date, and accesses to invalid shared memory are translated into inter-process communications (IPC) according the sDSM memory consistency protocols. Among different memory consistency protocols, a lazy release consistency model (LRC), derived from relaxed memory consistency models [23, 28], enables sDSMs to reduce the volume of IPC required by memory consistency work [52], and it is used by CLOMP to maintain its virtual global shared memory space [38].

In Section 2.1, the OpenMP programming model is briefly described. Some background knowledge of memory consistency models, sDSM systems, CLOMP and other approaches are described in Section 2.2. Related work on the performance models and some existing prefetch techniques for the sDSM layer and run-length encoding methods is reviewed in Section 2.3.

2.1 OpenMP

OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocess programming in C, C++ and Fortran on many architectures. It consists of a set of compiler directives, library routines, and environment variables that influence runtime behaviour. OpenMP has been widely recognized as the standard shared memory programming model and is supported by various vendors and open source communities [77], including GNU [29], IBM, Intel, Oracle/SUN and many more. A fork-join parallel mechanism is used by OpenMP, as shown in Figure 2.1.

2.1.1 OpenMP Directives

With the support of compilers, OpenMP uses directives and associated clauses to specify the program’s parallelism and data sharing. These directives can be grouped into several constructs, including the parallel construct, work-sharing construct, combined parallel and work-sharing construct, synchronization construct,
2.1 OpenMP

Figure 2.1: OpenMP fork-join multi-threading parallelism mechanism [93]

and threadprivate construct [75].

2.1.1.1 Parallel Construct

The purpose of a parallel directive is to specify a block of code that will be executed by multiple threads and forms a parallel section. Figure 2.2 shows the parallel directive with associated clauses in the C/C++ languages.

```c
#pragma omp parallel [clause ...] newline
  if (scalar_expression)
   private (list)
   shared (list)
   default (shared | none)
   firstprivate (list)
   reduction (operator: list)
   copyin (list)
   num_threads (integer-expression)
{
   ... block of code ...
}
```

Figure 2.2: OpenMP parallel directives and associated clauses in C and C++.

When a thread reaches a parallel directive, it forks a team of threads and becomes the master of the team with a thread ID 0. The block of code from the start to the end of the parallel region is duplicated, and all threads execute that
code. There is an implicit barrier at the end of a parallel section, and only the master thread continues execution past this point.

### 2.1.1.2 Work-sharing Construct

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. The work-sharing construct consists of multiple directives, including DO/for, section, WORKSHARE, and single directives. We will focus on introducing the most used DO/for directive. The DO/for directive specifies that iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor. The DO and for directives are used in Fortran and C/C++ languages respectively. Figure 2.3 shows the for directive with its associating clauses.

```c
#pragma omp for [clause ...] newline
    schedule (type [,chunk])
    ordered
    private (list)
    firstprivate (list)
    lastprivate (list)
    shared (list)
    reduction (operator: list)
    collapse (n)
    nowait
for_loop
```

**Figure 2.3:** OpenMP for directives and associated clauses in C and C++.

The DO/for work-sharing directive assumes that the program correctness must not depend upon which thread execute a particular iteration. In other words, there are no data dependencies between loop iterations. When the nowait clause is not used, there is an implicit barrier at the end of the DO/for loop.

### 2.1.1.3 Combined Parallel Work-sharing Construct

As the most widely used OpenMP construct is the work-sharing construct, OpenMP provides three combined directives merely for convenience. They are PARALLEL DO/parallel for, PARALLEL SECTIONS and PARALLEL WORKSHARE directives. We will focus on parallel for directive. As it combines parallel and for directives, it also
2.1 OpenMP

inherits their associated clauses. Figure 2.4 shows an example of using parallel for directive.

```c
#pragma omp parallel for 
    shared(a,b,c,chunk) 
    private(i) 
    schedule(static,chunk)
for (i=0; i < n; i++)
c[i] = a[i] + b[i];
...```

Figure 2.4: An example OpenMP program in C using parallel for directives.

When the parallel for directive is used, the whole parallel region is the for loop immediately after it. As for the parallel directive, there is an implicit barrier at the end of this parallel region. In addition, the parallel for directive also assumes that there are no data dependencies between different loop iterations.

2.1.1.4 Synchronization Construct

OpenMP also provides number of synchronization directives to maintain data consistency and program correctness between threads. They are master, critical, atomic, barrier and flush directives. In this section, barrier and flush directives are described in detail. Figure 2.5 shows these two directives.

```
(a) | (b)
#pragma omp barrier | #pragma omp flush (list)
```

Figure 2.5: OpenMP synchronization directives in C and C++ languages: (a) barrier, and (b) flush.

The barrier is the only global synchronization operation provided by OpenMP. When a barrier is reached, a thread is block-waiting until all others have reached the same barrier. It guarantees that all threads complete the parallel section above the barrier before any enter the section below it.

The flush directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point. Cache coherency mechanisms make certain that if one CPU executes a read or write instruction from/to memory, then all other CPUs in the system will get the same value from that memory address when they access it. All caches will show a coherent value. However, in the OpenMP standard,
there must be a way to instruct the compiler to actually insert the read/write machine instruction and not postpone it. Keeping a variable in a register in a loop is very common when producing efficient machine language code for a loop.

### 2.1.1.5 Threadprivate Construct

The `threadprivate` directive, as shown in Figure 2.6, is used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.

```c
#pragma omp threadprivate (list)
```

**Figure 2.6:** OpenMP threadprivate directive in C and C++ languages.

The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.

### 2.1.2 Synchronization Operations

OpenMP provides memory that is shared by all threads, which allows each thread to access the shared memory directly. However, there is no guarantee for the order of memory accesses between threads. In other words, writes to memory are allowed to overlap other computation, and reads from memory are allowed to be satisfied from a local copy of memory, referred to in the OpenMP 2.5 specification as the thread’s temporary view [75]. Each thread can also create thread-private variables that may not be accessed by any other thread. Therefore, synchronization operations are provided by OpenMP to maintain memory consistency.

As described in the previous section, OpenMP provides two explicit synchronization operations (the flush and barrier directives) which represent for non-global and global synchronization respectively. In addition, there are number of OpenMP directives containing implicit synchronization operations. These directives are summarised in Table 2.1.

When the `nowait` clause is not present, DO/for, sections, single directives contain an implicit global synchronization upon exit, and `critical` directive contains an implicit flush operation upon both entry and exit. All combined parallel work-sharing directives include a global synchronization upon exit.

Due to the explicit barrier directive including implicit flush operation in OpenMP
### 2.2 Cluster OpenMP Systems

#### Table 2.1: OpenMP synchronization operations.

<table>
<thead>
<tr>
<th>OpenMP Directives</th>
<th>Global Synchronization</th>
<th>Non-global Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>parallel</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DO/for</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td>sections</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td>single</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td>parallel DO/for</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>parallel sections</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>critical</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>flush</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>barrier</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

* without nowait clause

standard, all the directives listed in Table 2.1 include implicit non-global synchronizations.

#### 2.2 Cluster OpenMP Systems

A typical cluster OpenMP system consists of three major components, an OpenMP compiler, a sDSM runtime and a communication library. The supporting OpenMP compiler compiles OpenMP directives to the sDSM runtime. The sDSM runtime uses the communication library for inter-process communication and data exchanges.

Since the relaxed memory consistency model of OpenMP limits the need to move globally shared memory between nodes to well defined consistency points (i.e. OpenMP barrier and flush operations), it is possible to implement a software distributed shared memory (sDSM) version of OpenMP efficiently.

In this section, the relaxed memory consistency model is introduced with a focus on one of its derivations, the lazy release memory consistency model. It is followed by an overview of different sDSM systems. Finally, Intel Cluster OpenMP, as the target implementation, is described in detail.
2.2.1 Relaxed Memory Consistency

Memory consistency models [23, 28, 3, 11, 45] describe the behaviour that can be expected from a memory system. They are important for reasoning about the correctness of a program. In detail, a memory consistency model describes when the effect of a store operation has to be visible to load operations, and what the effect of a store operation implies about other load and store operations.

Usually, the correctness of a program is easier to maintain with a stricter memory consistency model, and the performance is potentially greater with a weaker consistency model [23, 28, 99]. For example, compared to program order\(^1\) [3] and sequential consistency\(^2\) [55] models, relaxed memory consistency models provide weak guarantees to the programmer but, in exchange, offer the potential of greater performance [23, 28]. This is achieved by adding synchronization semantics to the shared memory programming paradigm, such as OpenMP barrier and flush directives, that explicitly notifies the shared memory system when to make modifications of shared memory visible to other threads. The remainder of this section introduces one relaxed consistency model: release consistency (RC).

2.2.1.1 Release Consistency Model

Release consistency (RC) is a kind of relaxed memory consistency. It allows the effects of shared memory access to be delayed until certain competing (special) memory accesses occur. The special memory accesses are divided into synchronization and non-synchronization accesses, the synchronization accesses are in turn divided into acquire and releases. RC is defined by Gharachorloo et al. in [28] as follows:

A system is release consistent if it satisfied the following three conditions:

1. before an non-competing (ordinary) LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed,

2. and before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed,

---

\(^1\) The program order, a memory consistency model used by sequential programs, can be satisfied only when memory operations appear to happen in the order in which they occur in the actual program.

\(^2\) The sequential consistency, a memory consistency model used by parallel programs, is achieved when it appears that operations of all processes have been executed in some interleaved fashion that agrees with the constraints defined by program order.
2.2 Cluster OpenMP Systems

3. and special accesses are sequentially consistent with respect to one another.

To further explain RC, a RC system has the following memory access features. First, with respect to the local dependencies within the same process, the ordinary read/write operations following a release access are allowed to execute before the completion of the release access. The release synchronization access indicates the completion of memory accesses before it, and it has no effect to the order of any accesses after it. Second, since an acquire synchronization access does not give permissions to any other process to read/write the previous pending memory locations, it does not need to wait for the completion of the ordinary accesses before it. Thirdly, a non-synchronization special access does not affect the ordinary accesses. In detail, a non-synchronization special access does not wait for the completion of ordinary access before it, and does not delay any ordinary access after it. Fourthly, the special accesses are required to be sequentially consistent.

Lazy release consistency (LRC), developed by Keleher et al. in [52], further relaxes release consistency to achieve better performance. The main difference between RC and LRC is the order of special accesses.

RC maintains the sequential consistency between special accesses, while LRC delays the completion of non-synchronization accesses until another process performs an acquire access. In other words, all previous release accesses, and non-synchronizing accesses before these release accesses, must be complete to allow the acquire access to proceed. With this further relaxation, accesses only need to be completed with respect to the processes that require them to be complete, which results in fewer number of data transfers compared to that of RC.

2.2.2 Software Distributed Shared Memory Systems

As mentioned previously, cluster OpenMP systems utilize software distributed shared memory (sDSM) systems to create and maintain a virtual globally shared memory space across different processes (nodes). There are two typical methods to categorize sDSM systems.

By determining how the virtual shared memory is constructed, sDSM systems can be categorized as page-based and object-based as follows:

- **Page-based sDSM**: Shared memory is constructed as a shared address space that is partitioned into equally sized pages. Coherency is managed in units of
Chapter 2: Background

pages. These systems typically make use of the underlying operating system virtual memory mechanism. They include IVY [58], Munin [10], TreadMarks [51, 5], SCASH [34, 73], Danui [98, 97], ParADE [50], JIAJIA [41, 40], Delphi [89, 88].

- **Object-based** sDSM: On the other hand, for object-based sDSM systems, shared memory is an abstract storage space for storing shared objects of variable sizes. Coherency is managed with application specific granularity that is defined by objects. These systems are typically associated with object-oriented languages. They include SilkRoad [80], PJava [63], Adsmith [59, 60], DiSOM [32].

By the memory management model, sDSM systems can be categorized as home-based and homeless as follows:

- **Home-based** sDSM: Each page (or object) in the shared address space is assigned a home where the master copy of the page/object is maintained. The status of the page/object at its home node is what defines the most "up-to-date" view of that page/object. They include SCASH, Danui, ParADE, JIAJIA, Adsmith, SilkRoad.

- **Homeless** sDSM: In contrast, homeless sDSM systems do not assign page (or object) homes. Instead, changes made to a page/object by other processes (nodes) are patched into the local view when required. They include IVY, Munin, TreadMarks, DiSOM.

Our selected software platform, Intel Cluster OpenMP (CLOMP), uses page-based sDSM. This section will therefore provide an overview of some well-known page-based sDSM systems.

2.2.2.1 IVY

IVY was the first sDSM system described in the literature [58]. It was built into an operating system. The global virtual address space was formed by the physical memories of every processor, and this space is managed in units of pages, where the page size can be any multiple of that used by the underlying Memory Management Unit (MMU). Thus IVY is a page-based SDSM system.

IVY implemented sequential consistency. In other words, IVY maintained a total order on all memory accesses, and this total order is compatible with the program order of memory accesses in each individual process. In more detail, the
2.2 Cluster OpenMP Systems

State transition of virtual shared pages was maintained between read-only and read-writable. Read-only is the page state that invalidates shared pages. Each read-only page may have multiple copies residing in the physical memories of multiple processes. Each read-writable page may only have one copy residing at the processes that most recently modified it.

When a page is invalid, access to it caused a page fault. IVY brought an up-to-date copy of that page from its remote location into local memory and restarted the process. IVY also distinguished between read and write faults. For read faults, the page is replicated with read-only access for all replicas, and for write faults, an invalidation message is sent to all processes with copies of the page. Upon receiving this message, each process invalidated its copy of the page and sent an acknowledgment back to the writer, resulting the writer’s copy becomes the sole copy of the page. Therefore, it is important to keep track of the writer’s copy. Three algorithms, Centralized Manager, Fixed Distributed, and Dynamic Distributed, were described in [58] to track the single copy of the read-writable page.

Along with the advantages of simplicity and intuitive appeal of sequential consistency, IVY suffers from the consequent large amount of communication to keep this consistency, and also the “false sharing” problem, which describes the situation where more than one writer modified different regions of the same shared page.

2.2.2.2 Munin

Munin [10, 18] is a page-based sDSM system which allows a separate consistency protocol for each each shared variable, and the protocol for a variable can be changed over the course of execution of the program. This is achieved by source code annotation. One release consistency memory model was supported by Munin as well, which delays the propagation of modifications until a release access occurred.

When a release access is encountered, Munin tracks modifications to shared memory using a “twinning-diffing” mechanism. In twinning, a twin copy of a shared page is made before modifications to the page. During the diffing procedure, the twin page is compared to the modified page. The diffs between two copies are calculated, and then propagated by sending them to other processes.

Utilizing multiple memory consistency models, especially RC, greatly reduced the number of communications. Furthermore, the RC model enables the use of a twinning-diffing mechanism, which in turn solves the false sharing problem because changes made by different processes to different parts of the same page are tracked.
by differencing twin pages and may be merged at a later stage.

The main disadvantage of Munin is that the programmer must annotate each shared variable in the source code to get optimum performance. This can be very cumbersome and error prone for a large-scale applications.

### 2.2.2.3 TreadMarks

The research on Munin led to the most well known page-based sDSM system, TreadMarks [52, 51, 5, 6], in which lazy release consistency (LRC) was developed and utilized. The communication network traffic is further reduced by using LRC. Additionally, use of LRC achieves as good application performance as the adaptive multiple consistency protocols used by Munin. Therefore, it greatly simplified the task of writing applications for sDSM systems.

TreadMarks used an invalidation protocol together with the LRC model. The invalid virtual shared pages are protected against access using mprotect. Under the invalidation protocol, only write notices indicating the set of modified pages are sent to the acquiring process on the acquire access rather than the actual diffs. The pages listed in the write notices are invalidated upon arrivals of the write notices. The diffs for an invalidated page are only requested in the future if the page is actually accessed by the process.

Each process stores the diffs in order to send them to the remote requesters. Therefore, the stored diffs of the same shared memory region may overlap. To solve this, timestamps are used to order diffs temporally. In more detail, LRC divides the execution of each process into intervals with an unique interval index. These intervals are identified by a release or acquire access. In a synchronization, write notices indicating which page has been modified in a particular interval are sent to other processes.

Storing diffs in each process also raises the possibility that there may be too many diffs to store in available memory. TreadMarks uses a garbage collection mechanism to periodically remove the memory occupied by storing diffs. It is initiated at the end of barrier if the amount of memory used by TreadMarks exceeds a pre-defined threshold. During the garbage collection phase, each process will validate all shared pages modified by itself and update a copy-set for every page. Then it invalidates every other page and discards all diffs, write notices and intervals.
2.2 Cluster OpenMP Systems

2.2.3 Intel Cluster OpenMP

Intel Cluster OpenMP (CLOMP) was the first commercially released cluster OpenMP system. It is supported by Intel C/C++ and Fortran compilers and the Intel native OpenMP runtime.

The sDSM layer of CLOMP (TMK) was derived from TreadMarks [38] and consequently inherits many features from TreadMarks. It uses the same invalidation protocol, with the lazy release consistency (LRC) memory model, and garbage collection mechanism as TreadMarks. Moreover, CLOMP also has some extra features which TreadMarks does not have.

The TMK layer interacts with the communication layer of CLOMP (CAL) for IPC and data exchanges. The CAL layer can work individually on Linux operating systems and supports both Ethernet and InfiniBand interconnects [38].

In this section, we describe some features of CLOMP in detail, including its parallelism model, page state transition machine, and OpenMP directive to sDSM library mapping and its support profiling tools.

2.2.3.1 Parallelism Model

CLOMP deploys an process-thread parallelism model. In this model, when an OpenMP program is launched, the home process (where the program is executed) will create all processes on the involved computation nodes in a round-robin fashion. Figure 2.7 shows the relationship between nodes, processes and threads in CLOMP.

As shown in Figure 2.7, the number of processes in each node may not be the same, whereas the number of OpenMP threads in each process will be the same. Each process contains two types of threads:

![Figure 2.7: Processes and threads in CLOMP](image-url)
• *DSM support thread*: this is used to handle asynchronous I/O request, such as barrier, lock, flush, page/diffs requests. It is also used to manage shared memory access of its own copy detect heart beats and provide the timeout mechanism.

• *OpenMP thread*: this is the actual working thread which is in charge of the distributed working data computation.

In this model, virtual memory consistency is only maintained between different processes using the lazy release consistency model. The threads within a process are implemented using pthreads to exploit the smaller communication overhead provided by threading.

### 2.2.3.2 Page State Transition

CLOMP uses a lazy release memory consistency model (LRC) from Thread-Marks [52]. It augments 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 shared pages that are created, maintained, and synchronized across all OpenMP threads within the sDSM system.

Shared page access is detected by giving different protection types to the page, e.g. “read-valid” for read only access, “write-valid” for read write access, and “invalid” for full protection. Whenever accessing a protected (read-valid and invalid) page, a SIGSEGV signal is triggered. Upon a SIGSEGV signal, a CLOMP inserted signal handler is invoked to resolve the memory consistency and change the page protection. After the memory consistency is satisfied, the process is re-executed. Figure 2.8 shows the page state machine of CLOMP.

Transitions between the possible page states are given in Figure 2.8. Before shared pages are initialized, they stay in the “empty” state, and access to is protected. On the first access to these pages, fetch or/and write page faults will be raised, and full page requests are sent to the corresponding page manager and these pages are transferred to the requesters. Then, the page state will be set to “read-valid” or “write-valid” with respect to the type of access operation.

Write notices are passed between threads when OpenMP barrier, lock and flush directives are encountered. When a write notice is received for a shared page it will be set to “invalid” so that a subsequent read or write request to that page will give rise to fetch faults that require 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
2.2 Cluster OpenMP Systems

Figure 2.8: State machine of CLOMP (derived from [47], [38], and experimental observation).

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 twin copy.

During garbage collection, when a process does not have a most up-to-date copy of an “invalid” page, this page will be discarded and the page state will be set to “empty” and the protection remains.

2.2.3.3 OpenMP and sDSM Mapping

The key issue of mapping OpenMP to an sDSM system is to map both the explicit and implicit synchronization directives in OpenMP into corresponding sDSM operations. In CLOMP, the barrier and flush OpenMP directives are mapped.

OpenMP barriers are implemented in CLOMP using a two-level structure. A barrier between threads is done within each node, then across nodes. The nodes exchange lists of pages modified since the last synchronization, causing each node to protect the pages modified by other nodes. Write notices from all threads are sent to all other threads at barriers.

The flush operation is implemented similarly using a two-level structure, within a node and across nodes. During the flush, write notices are sent from the node which modified the variable to those nodes which have also performed a flush operation. CLOMP makes no attempt to implement the flush for specific
variables. In other words, flush directives in CLOMP will flush all visible shared variables, which sends the write notice from the node which made the changes to the nodes requesting the updated page.

2.2.3.4 SEGVprof Page Faults Profiling Tools

With the assistance of the Intel compiler and OpenMP runtime that generate unique region IDs, CLOMP provided the SEGVprof tool that reports a profile of the segmentation faults caused by a user's code on a per region basis. This allows a user to find out which part of their code takes the most time due to keeping memory consistency[47].

Major segmentation faults caused by memory consistency protocol are the FETCH fault and WRITE fault. The number of SIGSEGVs reported by SEGVProf is an aggregate.

Different costs of segmentation faults:

- FETCH faults: diffs or page transferring and applying.
- WRITE faults: Twin copy creation.

2.2.4 Alternative Approaches to sDSMs

This section reviews other alternative techniques to sDSM systems, including direct translation techniques that translate from OpenMP to MPI and Global Array at compile time [9, 43], partitioned global address space language (PGAS) [25, 101, 91, 21, 19], and single-system-image hardware virtualization [2].

2.2.4.1 Direct Translation Techniques

The direct translation techniques directly translate OpenMP programs into MPI or Global Array [71] programs to extend OpenMP applications on to clusters. They are briefly introduced in this section.

A compiler technique to translate OpenMP applications into MPI programs for execution on distributed memory system was presented in [9]. They allocated shared data in all nodes without management data (e.g. diffs between original copies and shadow copies). As the management data was not kept, the communication of the changed data was quite expensive in terms of the large messages. The evaluation of this technique was undertaken on two different platforms, 16 PIII 800
2.2 Cluster OpenMP Systems

MHz Linux nodes connected by 100MB Ethernet and 16 IBM SP2 WinterHawkII nodes connect by high performance switch. The results show variable performance based on different workloads.

Another source-to-source translation strategy was presented in [43] to implement OpenMP on clusters by translating OpenMP programs to Global Arrays (GA) programs. This technique uses GA to handle the shared data and communication across different nodes in a cluster. In addition to GA, MPI library calls (MPI_Send and MPI_Recv) were used in the translation to guarantee the execution order of processes, which increased the complexity of the translated code. With the larger number of processes, the overhead in communication will be non-trivial.

A linear speedup was obtained when running the Jacobi translated OpenMP program on a NERSC IBM cluster composed of 380 nodes, each of which consists of $16 \times 375$MHz POWER 3+ CPUs, connected to an IBM “colony” high speed switch via “GX Bus Colony”. A speedup of 28 on 40 processors is achieved obtained by a computational fluid dynamics OpenMP code that solves the Lattice Boltzmann equation on an Itanium2 cluster, which contains 24 nodes with dual 900MHz cores, with Scali interconnect.

2.2.4.2 Partitioned Global Address Space Languages

Partitioned Global Address Space (PGAS) languages partition a global shared memory address space and distribute it to all participating processes. On the other hand, sDSM systems do not explicitly specify whether the shared address space is partitioned. For example, each process of TreadMarks maintains a local view of the whole global shared address space. That is, the key difference is that PGAS languages make shared memory partitioning explicit to the programmer. Examples of PGAS languages are Unified Parallel C (UPC) [25], Co-Array FORTRAN [72], Titanium [101], Fortress [91], Chapel [21], and X10 [19].

PGAS languages are usually implemented in two different approaches which expose locality to users in different ways.

In the first approach, any part of the global address space is accessible to any process, regardless of where it is mapped (e.g. UPC). However, high overhead is associated with accesses to remote partitions. The parallel programs which access local partitions are preferable in this approach. Programmers are encouraged to minimize the number of accesses to remote partitions.

In the second approach, only local partitions are accessible. In order to access data residing in remote partitions, this approach allows processes to be started at remote locations (e.g. X10). Inter-location communication and data exchanges
are limited to instructions for creating new processes and transferring of the computing results. Compared to the first approach, since this approach only allows access to local partitions, it enforces locality.

2.2.4.3 Single System Image Hardware Virtualization

ScaleMP developed the Versatile Symmetric Multi-Processors (vSMP), a software-based computing-architecture, to virtualize a single-system-image on number of physical x86 computers [2].

Unlike other hardware virtualization hypervisors, such as Xen and VMware ESX, that allow multiple virtual hosts running on the same physical computer [8, 1], ScaleMP creates a single operating system on multiple physical computers connected via interconnects, which aggregates the compute, memory and I/O capabilities of each system and presents a unified virtual system to both the operating system and the applications running above the OS. The cache coherency between different compute cores is maintained by retrieving data between physical computers through interconnects.

In fact, ScaleMP provides an alternative approach to run shared memory applications on distributed memory architecture. For different applications, different systems can be created by ScaleMP. For example, a homogeneous system with large memory size and number of compute cores can be used for compute intensive applications, and an imbalanced heterogeneous system can be used for memory or I/O intensive applications.

In terms of handling inter-process communication and data exchange, ScaleMP is similar to sDSM systems, because both techniques remove the explicit control of data exchange between compute nodes from the programmers. Moreover, in terms of implementation, ScaleMP presents an single operating system to programmers, while sDSM systems are runtimes.

There is very limited academic literature available on ScaleMP. The most relevant is that Schmidl et al. evaluated ScaleMP vSMP using both kernel benchmarks and real-world applications in [85]. The kernel benchmarks including page access benchmark, memory bandwidth benchmark, sparse matrix multiplication benchmark, allocation time benchmark, and EPCC syncbench [14], explored the different core operations of a computer system. The evaluation results revealed a number of performance characteristics of ScaleMP vSMP as follows:

- Reading and writing remote memory page is about 20 times slower than local memory access through InfiniBand interconnects.
2.3 Related Work

- The OpenMP synchronization constructs are around two orders of magnitude slower on the ScaleMP vSMP compared to physical SMP machines.

The two real-world applications, FIRE and SHEMAT-suite, benefitted from the high core count and the aggregated memory bandwidth of the ScaleMP machine. In summary, the ScaleMP vSMP revealed obvious cc-NUMA behaviour.

2.3 Related Work

There is limited related work on performance models and they are not in the context of OpenMP. On the contrary, there were different approaches developed to improve the performance of sDSM systems, such as improving memory consistency protocols, and deploying page prefetch techniques.

Over the past couple of decades, the memory consistency protocols of sDSM systems have been extensively discovered, which also reflects the developing and improving road-map of sDSM system, from IVY to Munin and then TreadMarks. This is already discussed in previous sections.

In this section, we will focus on some previous research work on performance modeling and prefetching techniques for cluster OpenMP or its sDSM layer.

2.3.1 Performance Models

The purpose of performance evaluation is to understand the behaviour of a system, and the purpose of performance modeling is to provide a quantitative view of the system and confirm this understanding. There is very limited research work on performance models for cluster OpenMP systems. In this section, we review some hardware and software DSM system performance models.

2.3.1.1 Hardware DSM Performance Model

Waheed et al. proposed a performance model to characterize the the parallelization overhead of a compiler directives-based parallel program in [95]. Then this performance model was used to evaluate the performance of NPB benchmarks on the SGI Origin2000 hardware DSM system with a cache coherent non-uniform memory access (ccNUMA) architecture.

Waheed et al. decomposed a sequential program into \( N \) blocks, and use \( T_i \) to denote the time spent on \( i \)-th block. Therefore, the sequential execution time
can be denoted as $T_s = \sum_{i=1}^{N} T_i$. Furthermore, it was assumed that each block may be parallelized, and the parallel execution time $T_p$ can be derived as sum of sequential and parallel computation with addition parallelization overhead, as $T_p = \sum_{i=1}^{N} (T_{si} + T_{pi}) + T_o$.

As per Amdahl's law, the theoretical parallel execution time is calculated based on the calculated parallelism coverage ($PC$), the ratio of parallelizable computation load to the total sequential computation work load. Then, the parallelization overhead can be given as:

$$T_o = T_p - \frac{T_s}{\bar{p}} (PC + p(1 - PC))$$  \hspace{1cm} (2.1)

where $T_p$ measures parallel execution time and $p$ stands for number of processors. The derived parallelization overhead ($T_o$) contains following factors:

- aggregate synchronization time between threads during execution of a parallelized program;
- number of parallel loops;
- aggregate load imbalance between threads during execution of a parallelized program;
- non-local memory accesses by each thread; and
- resource contention between a thread and other users on the system.

The aim of this performance model is to measure the overhead of DSM system, rather than quantitatively evaluate the overall execution time.

### 2.3.1.2 Software DSM Performance Models

Parsons et al. presented a performance model to predict the performance changes of page-based sDSM on different hardware platforms based on fine-grained sDSM operations in [78]. This model divided the execution time of an application into busy time and overhead. The busy time refers to the time spent by the application on actual computation, and the overhead refers to the time spent on sDSM operations. TreadMarks was used as an example to validate this model. The sDSM operations for TreadMarks were decomposed into following components:

- **Fault handling** refers to the time consumed by handling read or write faults of the application to maintain memory consistency across processors. It in turn consists two major components:
2.3 Related Work

- request sending,
- waiting for replies.

- **Empty time** refers to the time consumed in handling first time misses for pages accessed by each processor, which corresponds to page state transition from “empty” to “read-valid” or “read-write”.

- **Garbage collection time** refers the time consumed by garbage collection, which is initiated at the end of a barrier if the amount of memory consumed by TreadMarks data structures exceed a pre-defined threshold.

- **Lock acquisition and release time** refers to the time consumed in acquiring and releasing locks, which are typically used to serialized access to shared data.

- **Sigio time** is consumed by handling asynchronous I/O requests from remote processors resulting from sDSM operations, such as barriers, faults, or locks.

- **Barrier time** refers to the time consumed by barrier operations, which are the global synchronization points. The barrier operation is further divided into the following sub-components:
  
  - the time for last processor to send a message to the master,
  - the time to merge write notices,
  - the time for the master to send a message to all the slaves.

The above-listed components of TreadMarks were measured, which revealed that sDSM overhead can be very significant, around 60% of overall execution time. As the hardware platform was limited the cost of barrier, sigio and locking operations were high. However, this model was not presented in a numerical and quantitative form in [78].

Another performance model was developed in [80] for object-based sDSM. In this model, computation time, scheduling overhead and global synchronization overhead have been considered. This model was evaluated on NQueens problem which showed around 10% difference from the actual measurements. Since this model was designed for an object-based sDSM system, the “twinning-diffing” overhead of page-based sDSM systems was not considered at all.

2.3.2 Prefetch Techniques for sDSM Systems

In this section, the prefetch taxonomy and some highly relevant page prefetch techniques for sDSM systems will be reviewed, including Dynamic Aggregation technique [7], B+ [12], Adaptive++ techniques [13], and third-order DFCM [88].
Chapter 2: Background

An effective prefetch technique can significantly reduce number of page faults and associated page fault handling costs, and also is able to aggregate pages and diffs to take advantage of interconnects with reduced latency and higher bandwidth.

2.3.2.1 Taxonomy of Data Prefetching Mechanism

In [15], a taxonomy of data prefetching mechanism has been introduced. In order to perform a proper prefetch, five issues that need to be addressed are:

- what data to prefetch?
- when to prefetch?
- what is the prefetching source?
- what is the prefetching destination?
- who initiates a prefetch?

The five issues are fundamental for any prefetching strategy, and will definitely concern the most important component for page/diff prefetch for CLOMP as well.

2.3.2.2 Dynamic Aggregation Technique

The Dynamic Aggregation technique [7] records all page access faults as a fault sequence list at process basis, that is divided into multiple groups with a pre-defined group size.

After a global synchronization point, when the first access fault occurs to any page in a group, prefetches are issued for all other pages in that group. In order to record the page fault history, only the faulting page is set to valid after the requested data arrives. After the group is fetched, it is freed. All prefetches need to be guaranteed to arrive before the next synchronization point. At the next synchronization point, groups are re-calculated based on the access faults experienced by the processor prior to the synchronization point with the following rules.

Page groups are computed at each synchronization point based on the access faults experienced by the processor prior to the synchronization. A fault occurs on every first access to an invalid page. All access faults form a fault sequence that is divided into groups in such a way that a group is completely filled before the next is created. The maximum size of page groups is defined by the user.
2.3 Related Work

Besides prefetching, Dynamic Aggregation also improves performance by combining multiple diff requests to the same processors, thus reducing the number of messages involved in fetching diffs to validate the pages of a group.

2.3.2.3 SCASH Prefetch Technique

The SCASH sDSM system utilized a simple historical based prefetch technique in [73]. Similar to Dynamic Aggregation, this technique prefetched all page misses that happened in the previous iteration. A communication thread is used to perform the prefetches. The pages to be prefetched are passed to the communication thread using an explicit prefetch instruction. The communication thread checks, locks and starts prefetch communication if the page does not exist in its local view of global memory. Upon receive of the page data, pages are unlocked and protection is changed to read only. The computation thread resumes after detecting the arrival of prefetched pages.

2.3.2.4 B+ Technique

B+ [12] is an invalidation-driven prefetch technique, which uses page invalidation to guide prefetching. B+ is a very straightforward prefetching technique because it assumes that a page that has been recently accessed by a processor and is later invalidated by another processor will likely be referenced again in the near future. Thus, the B+ technique prefetches diffs for each of the pages invalidated at synchronization points. Prefetches are issued right after synchronization points, including barrier and lock acquire operations. Upon arrival of prefetched pages, the page is marked as valid at the local node.

In summary, B+ has the following features and limitations:

- it issues prefetches after both lock acquire and barrier points;
- it is driven by page invalidation;
- it is not suitable for applications with irregular memory access and page fault patterns;

2.3.2.5 Adaptive++ Technique

In [13], the Adaptive++ technique is developed, which relies on two recorded page fault lists and two modes of operation to predict which pages to prefetch. These two page fault lists are maintained for the previous two regions. Then, during
the current barrier, the similarity between the two lists is calculated, and the page fault list for the previous region ($p_{list}$) will be chosen if the similarity is greater than 50%. Otherwise, the list for the “before previous” region ($bp_{list}$) will be chosen.

The first mode is named the repeated-phase mode. In the current barrier, $p$ predefined pages from the chosen list (starting from the first page) will be prefetched. After the barrier, at each page fault, if the page is in the chosen list, the $q$ predefined pages following the faulting page from the chosen list will be prefetched$^3$.

The second mode is named the repeated-stride mode. The most frequent page fault stride of the chosen list is used to determine the pages to prefetch in the next phase. Post the current barrier, at each page fault, if the faulting page is in the chosen list, the next $q$ pages with a multiple of the most frequent stride from the faulting page are prefetched.

The decision of which mode to use for the next phase is made during the barrier. If the repeated-phase mode is used for the last region, the efficiency of previously issued prefetches is calculated, using page fault information collected during the execution of the previous regions. If the repeated-phase mode is not used for the last region, the efficiency of what would be issued by this mode is calculated. If the efficiency of the repeated-phase mode is greater than the frequency of the most common stride for the chosen list, the repeated-phase mode is chosen; otherwise the repeated-stride mode is chosen. If neither is greater than 50%, prefetching is avoided until the next barrier.

Figure 2.9 illustrate the chosen list and the derived strides list, as well as

$^3$In [13], $p$ and $q$ are set to 24 and 4 respectively.
2.3 Related Work

the two prefetch modes of Adaptive++. The repeated-stride mode can issue pages which are not listed in the chosen list, while repeated-phase mode can only issue pages experienced before. This difference results in more pages potentially being prefetched with repeated-stride mode, and it has some ability to exploit the spatial data locality.

2.3.2.6 Third Order Differential Finite Context Method

The third-order differential finite context method (TODFCM) is used to predict the most likely page to be accessed next for the Delphi sDSM system [88]. A predictor that continuously monitors all misses to globally shared memory is implemented for TODFCM. For any three consecutive page misses, the predictor records the page number of the next miss in a hash table. During a prediction, a table lookup determines which page miss followed the last time the predictor encountered the same three most recent misses. The predicted page needs to be prefetched before the next actual page miss.

The predictor contains two levels of records. The first level retains the page numbers of the three most recent misses. However, only the most recent page number is stored as an absolute value. The remaining values are the strides between consecutive page numbers. The second level is a hash table which stores the target stride to calculate the next possible page. The entry of second level can be calculated by a given hash operation on the strides stored in first level. The records will be updated when the prediction is not correct. The target stride is replaced by the new stride and the corresponding first level records are updated with the new stride as well.

2.3.3 Run-Length Encoding Methods

Run-Length Encoding (RLE) [30] is a lossless data compression method, in which runs of data are stored as single data value and the number of runs (run length) rather than the original format. The following shows an example of RLE.

Input original format: WWWWWWWBBBBBBBBBBBBBB

Output RLE format: 12W1B12W3B

In the above example, the original data is interpreted as 12 W’s, 1 B, 12 W’s and 3 B’s by RLE, which compresses the original 28 characters in only 10.
RLE method is widely used in image processing [54, 53, 70, 69, 44], and some other application areas, such as network security [27] and pattern recognition [37].

In [70], Messom et al. utilized the following RLE format to preserve the position, size and color of objects in an image.

\[(\text{Color, StartIndex, EndIndex, ObjectID})\]

As shown in above format, they augmented the traditional RLE method with i) the start and the end index of a contiguous region of same colour pixels in a line, and ii) an object identifier that is unique for each element on an image. The RLE algorithm achieved linear time complexity in both height and width of the image. It shows good performance in the case of robotics applications where there are a small number of small objects.

In [44], Xu et al. used a RLE matrix and its texture descriptor to perform volumetric texture analysis in three dimensions computed tomography (CT) images. For a given 3D image, the RLE matrix $P$ is defined as follows: each element $P(i, j)$ represents the number of runs with pixels of gray level intensity equal to $i$ and length of run equal to $j$ along the $d(x, y, z)$ direction. The size of the matrix $P$ is $n \times k$, where $n$ is the maximum gray level $n$ in the CT image and $k$ is equal to the possible maximum run length in corresponding image. Based on the RLE matrix, eleven descriptors were extracted to reflect specific characteristic in the image, such as the distribution of short runs in an image etc. Their preliminary results showed that the run-length features calculated from the volumetric run-length matrix are capable of capturing the texture primitives properties for different structures in 3D image data.

In [37], Hinds et al. reduced the computational cost of Hough transforms, RLE was used to compress data within a document image through the computation of its horizontal and vertical black run-lengths. Histograms of these run-lengths are used to determine whether the document is in portrait or landscape orientation. A grey scale “burst image” is created from the black run lengths that are perpendicular to the text lines by placing the length of the run in the run’s bottom-most pixel. This data reduction procedure decreased the processing time of the Hough transform and reduced the effects of non-textual data on the determination of skew and interline spacing.

In [27], Qun et al. presented a RLE based dynamic trust model for P2P networks, which compressed peers’ behaviour history along the time dimension to compute trust. This strategy helped the model to increase the information amount while decreasing the data amount.
2.4 Summary

This chapter reviews the relevant background knowledge to this thesis, which includes:

- OpenMP shared memory programming model and its synchronization operations;
- the memory consistency models which is utilized in software distributed shared memory systems;
- the evolution and development road-map of software distributed shared memory systems;
- some detailed features of the first commercialized Intel Cluster OpenMP systems (CLOMP);
- alternative approaches to sDSM systems;
- the existing performance models for both software and hardware DSM systems;
- some relevant prefetch techniques for sDSM systems;
- the run-length encoding method with some of its applications.

The cluster OpenMP systems implement the OpenMP version of sDSM systems mainly through mapping OpenMP synchronization directives into sDSM operations. The relaxed memory consistency model deployed by OpenMP programming model makes its sDSM based implementation viable.

In the review of the road-map of sDSM systems and their memory consistency models, the lazy release consistency (LRC) with invalidation protocol had the best performance. The first commercialized cluster OpenMP system, CLOMP, is derived from TreadMarks to inherit its LRC and the invalidation protocol.

There are no existing performance models for the sDSM systems were designed for the OpenMP context. Most performance models were designed for specific purposes, such as evaluation of system overhead and performance prediction. Moreover, some models does not have numerical expression, and some models did not include all major sDSM operations.

The two major approach to improve performance of sDSM system were covered. One is to improve memory consistency and the other is to utilize effective prefetch techniques. The former is reviewed by introducing the road-map of development
of sDSM systems, and four well-known prefetch techniques for sDSM systems were described for the latter. These different prefetch techniques will be further evaluated and compared in Chapter 5.
Performance Issues of Intel Cluster OpenMP

To understand the performance of a Intel Cluster OpenMP (CLOMP) system, a three-steps approach is utilised. In the first step, the widely used NAS Parallel Benchmark OpenMP suite is used to measure the performance of CLOMP. The measured elapsed time is broken down to investigate the major overhead. In the second step, we have developed a micro-benchmark to characterise this overhead. In the last step, two performance models are developed to further analyse the different parts of the major overhead in a quantitative approach.
Chapter 3

Performance of Original Intel Cluster OpenMP System

Contents

3.1 Hardware and Software Setup ................................. 42
3.2 Performance of CLOMP .................................. 43
  3.2.1 NPB OpenMP Benchmarks Sequential Performance .... 44
  3.2.2 Comparison of CLOMP and Intel Native OpenMP on a Single Node ........................................ 44
  3.2.3 CLOMP with Single Thread per Compute Node ........ 48
  3.2.4 CLOMP with Multiple Threads per Compute Node .... 48
  3.2.5 Elapsed Time Breakdown for NPB-OMP Benchmarks ... 53
3.3 Memory Consistency Cost of CLOMP ....................... 55
  3.3.1 Memory Consistency Cost Micro-Benchmark – MCBENCH 56
  3.3.2 MCBENCH Evaluation of CLOMP ....................... 57
3.4 Summary ..................................................... 60
Chapter 3: Performance of Original Intel Cluster OpenMP System

Cluster OpenMP systems extend OpenMP shared memory programs on to clusters, which inherit the good programmability of OpenMP programming model. Since the first commercialized cluster OpenMP implementation was released by Intel at 2006, the interest on the performance of such system raised rapidly.

As many other parallel systems, an program execution time on cluster OpenMP implementations can be decomposed into the sequential part, the parallelizable part, and the system overhead [4, 33]. In this chapter, the first commercialized cluster OpenMP implementation, Intel Cluster OpenMP (CLOMP) selected as the target system, is comprehensively evaluated with different NAS Parallel OpenMP benchmarks (NPB-OMP) on different hardware platforms.

To obtain a deeper understanding of the performance and discover the major overhead, the measured elapsed time on CLOMP is broken down. This overhead is further characterised by a proposed micro-benchmark (MCBENCH).

3.1 Hardware and Software Setup

A widely used OpenMP benchmark suite, NAS Parallel OpenMP benchmarks (NPB-OMP) [26, 49] is used to measure the performance of CLOMP. The measure elapsed time for NPB-OMP benchmarks are broken down to identify the major system overhead of CLOMP. Then, a memory consistency cost micro-benchmark (MCBENCH) [96] is developed to characterise such overhead.

Both NPB-OMP and MCBENCH has been ported to CLOMP by specifying the global shared variables using the sharable directive. In addition, malloc function calls in C benchmarks have been replaced with the kmp_sharable_malloc function provided by CLOMP. Moreover, in order to ensure that the runtime system overhead corresponds to the reported benchmarking time, parallel regions which are not included in the timed section of the benchmark are serialized. This is a particularly important for the IS and FT benchmarks.

Two different supercomputers, hosted locally and in the Australian National Computational Infrastructure National Facility (NCI NF), are used to run the experiments.

As shown in Table 3.1, the two platforms exhibit different CPUs, size of memory, interconnects, operating systems and file systems. Intel CPUs with different micro-architectures are deployed in NCI NF supercomputers, known as XE and VAYU. Both clusters are connected/managed with Gigabit Ethernet (GigE). Additionally, XE and VAYU are also connected via high performance interconnect, InfiniBand (IB), with different data rate. Double Data Rate (DDR) IB is deployed
3.2 Performance of CLOMP

| Table 3.1: Evaluation experimental hardware platforms. |
|---------------------------------|--------|--------|
| Platform | XE | VAYU |
| CPU Model | Intel Xeon E5472 | Intel Xeon X5570 |
| Clock Speed | 3.0GHz | 2.93GHz |
| # of Cores | 2x Quad Core (8) | 2x Quad Core (8) |
| L2 Cache | 6MB (shared) | 8MB (shared) |
| Memory | 16GB | 16GB |
| Operating System | CentOS 5.6 | CentOS 5.6 |
| File System | Lustre | Lustre |
| Interconnect | GigE, DDR IB | GigE, QDR IB |

on XE, and Quad Data Rate (QDR) IB is used on VAYU.

NCI NF provides a shared computing service to large number of users from different research discipline. The maximum number of physical nodes that can be used for my PhD project is limited to 8, which consist of 64 cores in total.

3.2 Performance of CLOMP

Performance of CLOMP is evaluated using benchmarks from the NPB-OMP suite on XE and VAYU supercomputers hosted by NCI NF.

As we mentioned in Section 2.2.3, CLOMP uses both processes and threads. A process \((p)\) can contain multiple threads \((t)\). Therefore, the experiments can be configured in two ways.

The first is by using a single thread for each process denoted as \(p \times 1\), where the equivalent number of OpenMP threads is \(p\). Similarly, the second configuration is denoted as \(p \times t\), where the equivalent number of OpenMP threads is \(p \times t\).

Based on the above feature of CLOMP, the rest of this section is managed in four aspects:

- the sequential elapsed time for NPB-OMP benchmarks,
- performance comparison between CLOMP and Intel native OpenMP on a multicore shared memory computer,
Chapter 3: Performance of Original Intel Cluster OpenMP System

Table 3.2: Sequential elapsed time (sec) of NPB with CLOMP.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Size</th>
<th>BT</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>XE</td>
<td>A</td>
<td>79.7</td>
<td>10.3</td>
<td>11.4</td>
<td>0.4</td>
<td>60.8</td>
<td>66.1</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>341.7</td>
<td>41.3</td>
<td>n/a</td>
<td>3.6</td>
<td>523.9</td>
<td>296.9</td>
<td>122.1</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1441.6</td>
<td>165.0</td>
<td>n/a</td>
<td>29.9</td>
<td>2188.4</td>
<td>1216.4</td>
<td>334.2</td>
</tr>
<tr>
<td>VAYU</td>
<td>A</td>
<td>58.2</td>
<td>8.3</td>
<td>8.37</td>
<td>0.4</td>
<td>46.3</td>
<td>35.12</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>251.2</td>
<td>33.1</td>
<td>n/a</td>
<td>2.2</td>
<td>219.2</td>
<td>150.1</td>
<td>76.7</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1055.4</td>
<td>128.0</td>
<td>n/a</td>
<td>16.4</td>
<td>910.7</td>
<td>882.3</td>
<td>227.4</td>
</tr>
</tbody>
</table>

- performance of CLOMP on multiple nodes with a single thread per node \((p \times 1)\),
- performance of CLOMP on multiple nodes with multiple threads per node \((p \times t)\).

The details of performance of CLOMP is demonstrated and discussed in the rest of this section.

3.2.1 NPB OpenMP Benchmarks Sequential Performance

Due to the NPB-OMP benchmark classes S and W being too small (sequential elapsed time is usually less than couple of seconds), the larger classes A, B and C are used to present different data sizes.

The sequential elapsed time of NPB benchmarks on three different platforms are shown in Table 3.2. This is measured by running NPB-OMP benchmarks over CLOMP with a single OpenMP thread. ¹

3.2.2 Comparison of CLOMP and Intel Native OpenMP on a Single Node

For the \(1 \times t\) scenario, CLOMP is compared with native Intel OpenMP directives on a single compute node. The comparison, as shown in Figure 3.1 and 3.2, is presented in terms of speedup based on the sequential elapsed time listed in Table 3.2.

¹FT class B and C can not be compiled on our system due to memory limits. Since MG can not be compiled, it is excluded from the experiments.
3.2 Performance of CLOMP

Figure 3.1: Comparison of performance between native Intel OpenMP and CLOMP on a XE compute node.
Figure 3.2: Comparison of performance between native Intel OpenMP and CLOMP on a VAYU compute node.
3.2 Performance of CLOMP

According to Figure 3.1 and 3.2, CLOMP and Intel native OpenMP show the same performance trend on XE and VAYU. Both CLOMP and native Intel OpenMP obtained reasonable speedup for most NPB-OMP benchmarks except IS, SP and CG. As we have described in Section 2.2.3, each CLOMP process has a dedicated daemon thread (sDSM thread) for event handling and heart beat detection. This additional work results in a small performance penalty.

In general, CLOMP shows similar scalability to native Intel OpenMP with some additional software overhead. This overhead increases with the number of threads. When the number threads is less than 4, the performance difference between CLOMP and the native Intel OpenMP is less than \( \sim 2\% \) on average. In contrast, a larger performance difference is observed when the number of threads approaches the total number of available cores on both XE and VAYU for most benchmarks, excluding EP. For XE, CLOMP shows \( \sim 8.7\% \), \( \sim 18\% \) and \( \sim 13.2\% \) less speedup on average for CG, FT class A, and LU benchmarks respectively.

A similar trend is observed on VAYU with CLOMP showing larger performance penalty when the number of threads is more than 4. Here, CLOMP shows \( \sim 20\% \), \( \sim 50\% \), \( \sim 17\% \), \( \sim 17.5\% \), \( \sim 55\% \) less speedup on average for BT, IS, FT, and LU benchmarks respectively. Since VAYU exhibits more NUMA effects than XE, such scalability issues are to be expected.

Both CLOMP and the native Intel OpenMP show almost linear speedup for EP. It is because that EP benchmark represents the embarrassing parallel implementation of Gaussian random number generation, which requires minimal software effort to keep data consistent. Neither CLOMP nor the native Intel OpenMP shows good scalability for SP on both XE and VAYU platforms. To be precise, only \( \sim 1.9 \) and \( \sim 1.8 \) speedup achieved by the native Intel OpenMP and CLOMP on VAYU respectively. While this number is even lower on XE, \( \sim 1.3 \) for both Intel OpenMP and CLOMP. Nevertheless, the LU benchmark class B shows superlinear speedup on XE for both CLOMP and the native Intel OpenMP due to L1 cache effects.

Moreover, NCI NF uses software for system monitoring and job scheduling purpose, including bobMonitor [31], Ganglia [65], and Portable Batch System (PBS) [79, 90]. These system processes will have some affect on the performance, which is known as the “OS jitter” problem [64, 81]. In the rest of the thesis, we avoid fully subscribing all cores with OpenMP threads. In other words, a condition, \( t < N_{\text{cores}} \), will be applied to the second setup of experiments \( (p \times t) \).
3.2.3  CLOMP with Single Thread per Compute Node

In this section, CLOMP is evaluated on multiple compute nodes with a single thread on each over both Gigabit Ethernet and InfiniBand. The results shown in Figure 3.3 and 3.4 are represented in speedup based on the sequential elapsed time listed in Table 3.2.

The first observation from Figures 3.3 and 3.4 is that there is no benchmark scaling well with CLOMP on either XE or VAYU, except EP. The largest speedup (∼2.7) is observed for CG class C over IB on XE 8 nodes. CLOMP only achieves speedup for BT, CG, IS and FT benchmarks over IB connections on XE. In contrast, CLOMP does not show obvious ×1.5 speedup for all benchmarks on VAYU except the speedup observed for CG and BT class C over IB with 8 processes.

Another observation is that CLOMP performs better over IB interconnects rather than Gigabit Ethernet. As described by Amdahl’s law, the lower the overhead introduced by the parallel system, the higher the speedup obtained. InfiniBand is much faster than Gigabit Ethernet. The theoretical bandwidth of 4x DDR InfiniBand and 4x QDR InfiniBand is 16 Gigabit/s and 32 Gigabit/s respectively. This number for Gigabit Ethernet is only 0.8 Gigabit/s.²

Moreover, CLOMP shows different scalability for different benchmarks. For example, BT on IB interconnects scales with increasing number of processes, while there is no obvious performance improvement for IS with increasing number of processes. This is due to the homeless lazy release memory consistency model deployed by CLOMP [5, 38]; the reasons are discussed in detail in Section 3.3.

For the EP benchmark, similar to the 1 × t configuration, we again observed linear speedup on both XE and VAYU. Therefore, in the rest of this thesis, EP and other similar embarrassingly parallelized benchmarks are excluded from the performance experiments.

3.2.4  CLOMP with Multiple Threads per Compute Node

The second configuration of the experiment is p×t. As we discussed in Section 3.2.2, we will not fully subscribe compute cores (t = 8) to avoid “OS jitter” effect. Hence, in this section each CLOMP process will not fork more than 4 OpenMP threads.

Since the performance behaviour of the NPB benchmarks does not have much different behaviour for this configuration, BT, LU and SP NPB-OMP benchmarks are presented in this section. Figure 3.5 illustrates the performance of CLOMP

²Both InfiniBand and Gigabit Ethernet network interface cards are connected PCI(e) bus, which utilizes 8bytes/10bytes encoding mechanism. It results in lower theoretical bandwidth.
3.2 Performance of CLOMP

Figure 3.3: Performance of CLOMP on XE with a single thread per compute node.

on different $p \times t$ configurations of Gigabit Ethernet and InfiniBand interconnects
Figure 3.4: Performance of CLOMP on VAYU with a single thread per compute node.

on XE. Figure 3.6 shows the corresponding performance data on VAYU. In these
3.2 Performance of CLOMP

figures, $P \times 1$, $P \times 2$ and $P \times 4$ stand for the case each process (node) has 1, 2 and 4 OpenMP threads respectively.

Similar the performance behaviour observed in Section 3.2.3, the larger problem sizes usually show better speedup. Besides the effect of problem sizes, the other two different performance behaviours can be observed from these figures. The first is the scalability across different compute nodes with the same number of threads per node, represented by each plotted line. The second is the scalability with the same number of compute nodes and an increasing number of threads per node, which does not directly plotted, however it can be find out on the figure as well.

On XE over both GigE and IB networks, CLOMP shows worse scalability when more threads are utilized in each node, e.g. $P \times 4$ scales less than $P \times 2$, and $P \times 2$ scales less than $P \times 1$. This is not surprising because the first data point of each plot of $P \times t$ is the speedup achieved on a SMP which involves zero inter-process communication. CLOMP usually achieves only very slight speedup when more OpenMP threads are deployed as that of the single thread per process case for most benchmarks.

There are two factors that have major influence. The first is that CLOMP only maintains the consistency of virtual shared memory between processes. Multiple threads within the same process communicate by physical shared memory, whose overhead is much smaller than that of network data transfer. The second is that when different threads within the same process send requests to update different invalid pages from the same destination (process), the data transfer is serviced by the sDSM daemon thread of the destination process in serial and the requested data is arrived at the request process in serial. This highly inhibits the level of parallel processing, and, unfortunately, many applications exhibit this kind of data distribution and memory behaviour. As a result, multiple threads per process used in CLOMP shows roughly the same speedup as that of the single thread per process cases for some benchmarks. These two factors balance out each other, which results a slightly different performance behaviour observed for different benchmarks. For example, more threads performs slightly better for BT benchmarks, and roughly equal speedup for LU and SP benchmarks.

Although CLOMP achieves better speedup with InfiniBand compared to GigE, the benefit of using more threads and single thread per compute node is not very obvious. For some benchmarks, CLOMP even shows the smaller speedup when more threads are deployed, such for the SP benchmark.

On VAYU, a similar behaviour is observed. For more benchmarks, CLOMP with single OpenMP thread per process performs a little bit better than the case
that multiple OpenMP threads used per process. This is due to VAYU deploying much faster CPU than that of XE, which results in communication overhead of CLOMP becoming the dominant portion of the whole elapsed time. Hence, poorer scalability is exhibited.

In summary, multiple threads deployed per CLOMP process does not improve its scalability for the NAS OpenMP benchmarks.
3.2 Performance of CLOMP

According to Section 2.2.3, CLOMP utilizes the processes-threads parallelism model. Threads within a process communicate with each other via shared memory. The memory consistency is only maintained between processes. We conjecture that the major system contention is contributed by memory consistency of CLOMP. To validate this point of view, we breakdown the elapsed time of the NPB-OMP benchmarks obtained on XE with single thread per node to show that of page fault handling, including FETCH and WRITE page faults (see Section 2.2.3 for more}

Figure 3.6: Performance of CLOMP on VAYU with multi-threads per compute node.

3.2.5 Elapsed Time Breakdown for NPB-OMP Benchmarks
Moreover, as discussed in Section 3.2.4, utilizing multiple threads per process does not clearly improve the scalability of CLOMP for the NPB benchmarks. Therefore, the page fault handling costs when multiple threads deployed per process is further broken down to find out the associated overhead.

3.2.5.1 Single Thread per Process

Table 3.3 shows the page fault handling costs of CLOMP for some NPB-OMP benchmarks on both GigE and DDR IB networks. In the table, the columns labeled as “Timing” represents the time in seconds reported by the NPB-OMP benchmarks. The columns labeled as “SEGV Cost” represents the time spent on the CLOMP instrumented segmentation fault handler over the whole life span of NPB-OMP benchmarks. This cost covers both FETCH and WRITE page faults. “Timing” is shown in seconds, and “SEGV Cost” is shown as a ratio to the corresponding “Timing”.

Corresponding to Table 3.3, the page fault handling costs dominates the elapsed time for all NPB-OMP benchmarks. In average of all benchmarks, CLOMP spends ~77% and ~55% of the execution time in maintaining memory consistency on GigE and DDR IB networks respectively. FT benchmark poses the highest average page fault handling costs on both GigE (83%) and DDR IB (64%). In contrast, on average, CG achieves the lowest page fault handling costs ratio on both GigE (63%) and DDR IB (39%) interconnects.

There are three common trends of the page fault handling ratio. The first is with increasing number of processes, the ratio of page fault handling increases slightly for most benchmarks and problem sizes. The second is, with increasing problem sizes, the ratio decreases slightly for all benchmarks. The third trend is that, due to the higher bandwidth and latency provided by DDR IB compared to GigE network, the page fault handling costs on DDR IB is around ~20% to ~25% less than that on GigE.

3.2.5.2 Multiple Threads per Process

In Table 3.4, “Timing” represents the elapsed time in seconds. “SEGV” represents the time spent on the page faults handling costs as a ratio to the elapsed time. Additionally, the column labeled as “SEGV Lock” represents the time spent on pthread mutex locking to guarantee the correctness of diffs/pages updates between threads within the same process, and it is represented as a ratio to the total page
3.3 Memory Consistency Cost of CLOMP

Fault handling costs. As the trends between the benchmark classes is similar to Table 3.3, only class A benchmarks are shown in Table 3.4. The page fault handling costs dominate the elapsed time when multiple threads are utilized within each process.

Moreover, in contrast to the single thread case, excluding CG when 2 threads are deployed, the portion of page fault handling costs are reduced by around 13% and 7% on average for GigE and DDR IB networks respectively. This portion is further reduced to \(~24\%\) on GigE when 4 threads are deployed. However, on the DDR IB network, this the page fault handling costs is not further reduced. Nevertheless, it increased around 8% for BT benchmark compared to the single thread case. For CG, the page faults handling cost on the DDR IB network is increased around \(~10\%\) on average, and this ratio maintains on GigE.

Additionally, there is no locking cost when there is only one thread used per process, while this cost is increasing with number of threads deployed in the same process, and number of processes. For the CG benchmark, locking cost dominates the total page fault handling costs.

To summarize, when multiple threads are deployed, the page fault handling costs reduced slightly on GigE network and not much affect is observed on DDR IB network because an extra locking overhead is introduced. This overhead increases with both the number of threads and processes.

The above observations for the time spent page fault handling reflect the poor scalability of CLOMP. The memory consistency costs significantly limit CLOMP’s performance. More detailed characterization of memory consistency cost in CLOMP are described in the next section.

3.3 Memory Consistency Cost of CLOMP

As show in the previous section, CLOMP performs well in generally when programs were not running cross different compute nodes except when a compute node is fully subscribed.

According to what is described in the background of CLOMP, the memory consistency of virtual shared memory space is maintained among processes. Moreover, the discussion of page fault handling costs of NPB-OMP benchmarks in Section 3.2.5 shows that the page fault handling costs dominates the elapsed time for all NPB-OMP benchmarks.

Therefore, the conclusion is made that the major overhead of CLOMP system is the memory consistency overhead. To detailed characterize the memory
consistency overhead, I worked with Dr. H'sien Jin Wong to publish a micro benchmark (MCBENCH) which measures the memory consistency costs of OpenMP implementations including cluster-enabled OpenMP systems [96, 99].

In the rest of this section, the details of this benchmark is described and the memory consistency costs of CLOMP are evaluated using it.

### 3.3.1 Memory Consistency Cost Micro-Benchmark – MCBENCH

In the OpenMP standard, individual threads are allowed to maintain a temporary view of memory that may not be globally consistent. Rather, global consistency is enforced either at synchronization points (OpenMP barrier operations) or via the use of the OMP flush directive.

The goal of the Memory Consistency Benchmark (MCBENCH) is to measure the overhead that can be attributed to maintaining memory consistency for an OMP program. To do this, memory consistency work is created by first having one OMP thread make a change to shared data and then flush that change to the globally visible shared memory; and then having one or more other OMP threads flush their temporary views so that the changes made to the shared data are visible to them.

As noted above it is important that the readers’ flushes occur after the writer’s flush, otherwise OMP does not require the change to have been propagated. Both these requirements are met by the OMP `barrier` directive since this contains both synchronization and implicit flushes [75]. Accordingly, the general structure used by MCBENCH is a series of change and read phases that are punctuated by OMP `barrier` directives (where implicit flushes and synchronization occurs) to give rise to memory consistency work.

Since the above includes other costs that are not related to the memory consistency overhead, it is necessary to determine a reference time. This is done by performing the exact same set of operations but using private instead of shared data. The difference between the two elapsed times is then the time associated with the memory consistency overhead.

To ensure that the same memory operations are performed on both the private and shared data, the MCBENCH kernel is implemented as a routine that accepts the address of an arbitrary array. Figure 3.7 shows that this array of \( a \) bytes is divided into chunks of fixed size \( c \) which are then assigned to threads in a round-robin fashion. In the Change phase, each thread changes the bytes in their respective chunks. This is followed by a barrier, and the Read phase where the round-robin distribution used in the Change phase is shifted such that, had the
3.3 Memory Consistency Cost of CLOMP

Figure 3.7: MCBENCH – An array of size \( a \)-bytes is divided into chunks of \( c \)-bytes. The benchmark consists of Change and Read phases that can be repeated for multiple iterations. Entering the Change phase of the first iteration, the chunks are distributed to the available threads (four in this case) in a round-robin fashion. In the Read phase after the barrier, each thread reads from the chunk that its neighbour had written to. This is followed by a barrier which ends the first iteration. For the subsequent iteration, the chunks to Change are the same as in the previous Read phase. That is, the shifting of the chunk distribution only takes place when moving from the Change to Read phases.

array been a shared one, each thread will now read the chunks previously changed by their neighbours. The size of the shared array is the total number of bytes that was modified during the Change phase, and this is also the total number of modified bytes that must be consistently observed in the subsequent Read phase. Thus, this number represents the memory consistency workload in bytes that the underlying memory system must handle.

Therefore, at each iteration, each process will have page faults in \( \frac{a}{c \cdot p} \) shared pages, where \( p \) is number of processes deployed.

3.3.2 MCBENCH Evaluation of CLOMP

The memory consistency cost of CLOMP is evaluated with MCBENCH on XE. Three shared array sizes \( (a) \) are used (64KB, 4MB and 8MB). Three different chunk sizes \( (c) \) are used as well (4B, 2KB and 4KB). The comparison of memory consistency cost among different chunk size \( (c) \) and network interconnects is illustrated for each array size \( (a) \) in Figure 3.8. Since the memory consistency
is only maintained among processes by CLOMP, the $p \times 1$ configuration is used.

According to the evaluation results of using different sized shared arrays, the chunk size of 4 bytes poses largest memory consistency overhead, followed by the chunk size of 2KB. When the chunk size of 4KB, CLOMP achieves the lowest memory consistency cost.

Corresponding to the background of CLOMP described in chapter 2, the system page size of CLOMP is 4KB, and memory consistency of the virtual shared memory is maintained for each shared page. When 4 bytes chunk is applies, all processes will read and write all shared pages, which results in an each process will need to retrieve data from all other processes for each shared page. A $O(p)$ scalability is observed for 4 bytes chunk size where $p$ stands for number of processes, which results in a linear increment of memory consistency cost in Figure 3.8 for all array sizes and network interconnects with increasing number of processes.

When the chunk size was increased to 2KB, each page contains two chunks and will be read and written by two processes, which in turn results in data transfer from either one or two process(es) for $\frac{a}{c_p}$ shared pages. Therefore a “semi-constant” scalability is expected. Figure 3.8 reflects my expectation for this case. The communication cost is reduced rapidly compared to 4 bytes chunk size. The memory consistency cost does not arise with increasing number of processes.

For 4KB chunk size, a system page of CLOMP contains only one chunk, which will be read and written by one process during one iteration. Therefore, a ad-hoc communication pattern applied for data transfers post barrier for each process with $\frac{a}{c_p}$ shared pages. The memory consistency cost remains constantly as the cost to transfer a system page between two processes, which is the lowest observed cost among three different chunk size cases.

As shown in Figure 3.8, a faster interconnect, 4x DDR InfiniBand, significantly reduces memory consistency cost of CLOMP. The memory consistency cost observed for InfiniBand is around 50% of that of Ethernet for 64KB shared array. This ratio dropped to $\sim$40% when the shared array size increase to 4MB and 8MB.

The memory consistency cost is also proportional to the shared array size ($a$). The larger sized array contributes more page/diff transfers. Let $A$ denote the number of shared pages ($\frac{a}{4096}$), and $C$ denote the number of writers for each shared page ($\frac{4096}{c}$), where 4096 is the system page size used in CLOMP. Hence, the memory consistency cost, noted as $T_{mc}$, has the following relation with the shared array size ($a$) and the chunk size ($c$), as shown in Equation 3.1.

$$T_{mc} \propto A$$
3.3 Memory Consistency Cost of CLOMP

Figure 3.8: MCBENCH evaluation results of CLOMP on XE with both Ethernet and InfiniBand interconnects: 64KB, 4MB and 8MB array sizes are used in these three figures respectively; comparison among difference chunk sizes 4B, 2KB and 4KB is illustrated in each figure for both Ethernet and InfiniBand.
Chapter 3: Performance of Original Intel Cluster OpenMP System

\[ T_{mc} \propto C \]  

(3.1)

3.4 Summary

In this chapter, Intel Cluster OpenMP (CLOMP) is comprehensively evaluated using the NPB-OMP benchmarks on both the XE and VAYU clusters hosted at NCI NF, which explores both different CPUs and interconnects. Three different configurations, based on the process-thread parallelism model of CLOMP, are used.

The first configuration \((1 \times t)\) deploys single process and multiple threads within this process on a multi-core compute node. In this case, CLOMP is compared with the native Intel OpenMP. When number of threads \((t)\) is not more than 4, CLOMP shows a compatible scalability to the native Intel OpenMP with less than 2\% software performance penalty on both XE and VAYU. Due to the “OS jitter” caused by many system monitoring daemons used on NCI NF machines, CLOMP performs much worse than the native OpenMP when 8 threads are deployed.

Secondly, CLOMP is evaluated with single thread per process (node) on multiple physical compute nodes \((p \times 1)\). CLOMP shows no scalability at all for all NPB-OMP benchmarks on GigE network. On IB network, some improvement in terms of speedup is observed for all benchmarks, and a small scalability is observed for some NPB-OMP benchmark with the largest problem size (class C). The same trend has observed on both XE and VAYU. To understand the performance of CLOMP in this case, the elapsed time of different benchmarks obtained on XE has been broken down to show the page fault handling costs. On average, 75\% and 55\% of the total elapsed time has been spent on page fault handling via GigE and DDR IB networks respectively.

Lastly, CLOMP is also evaluated with multiple threads per process \((p \times t)\). A slightly better performance is achieved compared to that of the single thread \((p \times 1)\) case. However, the page fault handling costs still dominates the elapsed time. On average, \(~60\%\) and \(~50\%\) of the total elapsed time is spent on page fault handling for GigE and DDR IB networks correspondingly. An extra cost associated with locking to maintain the correctness of pages/diffs updates between threads within the same process is introduced. The locking cost increases with increasing \(p\) and \(t\), and it is more dominant on GigE network \((\sim 30\%\) when \(t = 4\)) compared to that on DDR IB network \((\sim 20\%)\).

Based on the detailed performance evaluation, it has been discovered that the major system overhead of CLOMP is the page fault servicing cost, also known as the memory consistency cost. Therefore, a micro-benchmark, MCBENCH, was
3.4 Summary

developed to characterise this overhead. The memory consistency cost of CLOMP is proportional to the number of shared pages, and the number of writers to the same shared page.

In summary, due to high memory consistency costs, CLOMP does not show good scalability for most NPB-OMP benchmarks except EP. Utilizing high performance interconnects improve the scalability slightly. With multiple threads deployed, performance of CLOMP does not improve significantly due to the extra locking cost to maintain the correctness of diffs/pages update between threads within the same process. Since there are many types of the memory consistency costs, we will quantitatively modeled them in the next chapter to obtain deeper understanding.
# Table 3.3: Performance of Original Intel Cluster OpenMP System

<table>
<thead>
<tr>
<th>Class</th>
<th>interconnects</th>
<th>nprocs</th>
<th>BT Timing SEGV Cost</th>
<th>IS Timing SEGV Cost</th>
<th>FT Timing SEGV Cost</th>
<th>LU Timing SEGV Cost</th>
<th>SP Timing SEGV Cost</th>
<th>CG Timing SEGV Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DDR IB</td>
<td>2</td>
<td>73.2 39.0</td>
<td>0.9 59.8</td>
<td>18.3 58.1</td>
<td>164.2 53.4</td>
<td>90.6 55.4</td>
<td>3.3 38.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>56.5 53.9</td>
<td>0.9 65.3</td>
<td>14.7 62.8</td>
<td>158.9 64.2</td>
<td>84.4 67.8</td>
<td>4.2 56.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>48.3 66.6</td>
<td>1.5 76.6</td>
<td>11.7 70.6</td>
<td>180.0 50.6</td>
<td>76.5 71.5</td>
<td>8.3 67.3</td>
</tr>
<tr>
<td>B</td>
<td>DRR IB</td>
<td>2</td>
<td>294.0 36.2</td>
<td>4.5 49.8</td>
<td>n/a n/a</td>
<td>640.0 49.8</td>
<td>352.1 49.6</td>
<td>77.4 15.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>211.0 49.4</td>
<td>4.6 67.1</td>
<td>n/a n/a</td>
<td>528.6 62.3</td>
<td>300.0 61.7</td>
<td>59.7 36.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>164.6 61.4</td>
<td>5.7 64.1</td>
<td>n/a n/a</td>
<td>481.1 59.8</td>
<td>266.9 69.2</td>
<td>103.3 46.1</td>
</tr>
<tr>
<td>C</td>
<td>DRR IB</td>
<td>2</td>
<td>1220.6 35.1</td>
<td>27.8 32.8</td>
<td>n/a n/a</td>
<td>2793.3 42.9</td>
<td>1389.1 48.0</td>
<td>201.5 12.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>841.2 47.3</td>
<td>24.7 58.9</td>
<td>n/a n/a</td>
<td>1897.7 59.0</td>
<td>1092.9 58.7</td>
<td>133.0 27.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>641.8 52.4</td>
<td>26.7 56.6</td>
<td>n/a n/a</td>
<td>1579.1 65.2</td>
<td>939.7 60.9</td>
<td>125.2 51.6</td>
</tr>
</tbody>
</table>

Note: The table provides the timing and SEGV cost for various benchmarks on a single node with a single thread.
### 3.4 Summary

Table 3.4: Page faults handling cost breakdown for CLOMP for class A NPB benchmarks with multiple threads per process on XE. “SEGV” represents the ratio of page faults handling cost to the corresponding elapsed time; “SEGV Lock” in turn represents a ratio of pthread mutex cost within “SEGV”.

<table>
<thead>
<tr>
<th>Network</th>
<th>p × t</th>
<th>BT</th>
<th>IS</th>
<th>FT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Timing</td>
<td>SEGV (%)</td>
<td>SEGV Lock (%)</td>
</tr>
<tr>
<td>GigE</td>
<td>2x2</td>
<td>107.4</td>
<td>61.4</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>4x2</td>
<td>110.2</td>
<td>66.1</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>8x2</td>
<td>98.0</td>
<td>66.5</td>
<td>4.9</td>
</tr>
<tr>
<td></td>
<td>2x4</td>
<td>87.1</td>
<td>53.6</td>
<td>5.2</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>95.5</td>
<td>57.5</td>
<td>8.6</td>
</tr>
<tr>
<td></td>
<td>8x4</td>
<td>89.5</td>
<td>53.5</td>
<td>15.0</td>
</tr>
<tr>
<td>DDR IB</td>
<td>2x2</td>
<td>48.6</td>
<td>40.5</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>4x2</td>
<td>46.1</td>
<td>56.3</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>8x2</td>
<td>42.1</td>
<td>63.6</td>
<td>6.6</td>
</tr>
<tr>
<td></td>
<td>2x4</td>
<td>40.8</td>
<td>48.2</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>45.8</td>
<td>58.4</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td>8x4</td>
<td>44.4</td>
<td>63.4</td>
<td>18.4</td>
</tr>
<tr>
<td>GigE</td>
<td>2x2</td>
<td>347.5</td>
<td>60.9</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>4x2</td>
<td>400.3</td>
<td>63.3</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>8x2</td>
<td>484.0</td>
<td>53.2</td>
<td>12.9</td>
</tr>
<tr>
<td></td>
<td>2x4</td>
<td>273.8</td>
<td>54.8</td>
<td>8.3</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>367.9</td>
<td>53.0</td>
<td>11.6</td>
</tr>
<tr>
<td></td>
<td>8x4</td>
<td>548.9</td>
<td>37.3</td>
<td>31.9</td>
</tr>
<tr>
<td>DDR IB</td>
<td>2x2</td>
<td>135.7</td>
<td>48.2</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>4x2</td>
<td>161.6</td>
<td>49.3</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>8x2</td>
<td>216.4</td>
<td>33.3</td>
<td>9.7</td>
</tr>
<tr>
<td></td>
<td>2x4</td>
<td>135.7</td>
<td>47.4</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>183.7</td>
<td>37.1</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>8x4</td>
<td>285.6</td>
<td>22.8</td>
<td>19.9</td>
</tr>
</tbody>
</table>
Chapter 4

Region-Based Performance Models

Contents

4.1 Regions of OpenMP Programs ........................................... 66
4.2 SIGSEGV Driven Performance (SDP) Models ....................... 67
   4.2.1 Critical Path Model ........................................... 68
   4.2.2 Aggregated Model ........................................... 70
   4.2.3 Coefficient Measurement ....................................... 71
4.3 SDP Model Verification ................................................. 72
   4.3.1 Critical Path Model Estimates .................................. 73
   4.3.2 Aggregate Model Estimates ..................................... 74
4.4 Summary ................................................................. 75
Chapter 4: Region-Based Performance Models

The performance of CLOMP was comprehensively evaluated in Chapter 3, and the major overhead of CLOMP has been identified as the page fault handling cost, which is also known as the memory consistency cost. Its characteristics are described by the measurements of the memory consistency micro-benchmark (MCBENCH). In order to better understand the performance of CLOMP, a quantitative approach ought to be utilized [36].

In this chapter, the OpenMP parallel and sequential regions is firstly reviewed. Secondly, based on the idea of parallel and sequential regions, we have developed two performance models for the cluster-enabled OpenMP systems which are both driven by number of SIGSEGV signals handled by sDSM layer of Cluster-enabled OpenMP systems. These two models provide a quantitative view of the performance for cluster-enabled OpenMP systems. These models are developed for both a home-less and home-based sDSM system as stated in [17]. In this chapter, we will focus on that for home-less sDSM system, e.g. CLOMP. The details of the two performance models will be described and verified in this chapter.

4.1 Regions of OpenMP Programs

A major challenge for shared memory programming is determining what data will be manipulated by which process/thread and how this is coordinated between the processes/threads. Synchronization operations are used to exchange and keep the consistency of the data. Often a parallel program is decomposed into several regions separated by global synchronization points in a fork/join programming model. Fig. 4.1 illustrates this for an OpenMP parallel program.

In Fig. 4.1, parallel regions can be distinguished by an explicit barrier within a fork-join section, such as parallel region #1 and #2. Moreover, a parallel region can also be distinguished by fork-join operations, such as parallel region #3. The remaining regions which have only one thread are sequential regions. Some cluster-enabled OpenMP system runtimes can generate an unique ID for each region (parallel or sequential), e.g. CLOMP.

The regions in Fig. 4.1 can often be enclosed in a loop, which will cause each region to be executed multiple times. Executions of the same region will have a memory access pattern determined by the loop index.
In this section two sDSM performance models are described. The first uses critical path analysis [39, 86] and requires detailed knowledge of the number and type of page faults occurring for each process in each parallel region. The second takes a more holistic approach requiring just the aggregate number of page faults occurring for all processes. Both the homeless and home-based sDSM systems will be considered, although the models will be developed initially within the context of the homeless sDSM system. We refer to both these models as SIGSEGV Driven Performance (SDP) models, reflecting the fact that the page faults give rise to SIGSEGV signals on POSIX compliant systems.

For both models, $N^w$ and $N^f$ will be used to denote the total number of WRITE and FETCH page faults respectively. The corresponding costs of these will be represented as $C^w$ and $C^f$. For home-based sDSM systems, a further distinction is made for write faults depending on whether the fault happens for a local ($N^{wl}$ and
Chapter 4: Region-Based Performance Models

$C^{wi}$ or remote ($N^{wr}$ and $C^{wr}$) page. It should be noted that the values for $N$ are platform-independent, and if needs these values may be gathered on a single node, provided the required number of sDSM processes are specified.

The total execution time of an OpenMP program can be broken down into the serial time ($T^s$), and the parallel time ($T^p$). The parallel time ($T^p$) in turn broken down to the computation ($T^c$) and overhead ($T^o$), as shown in Equation (4.1).

$$Tot = T^s + T^p = T^s + (T^c + T^o)$$ (4.1)

For the critical path SDP model the time spent in the parallel regions is further broken down into computation time ($T^c$, i.e. time spent doing useful work), overhead time ($T^o$, i.e. time spent servicing the various page faults), and idle time ($T^i$, i.e. time spent waiting for other processes). This is illustrated in Figure 4.2. The model assumes that page faults occurring on different processes can be fully overlapped, with a total per process cost that can be estimated by knowing the cost of a page fault of a given type as measured when using just two nodes.

Both models assume that overheads, such as those associated with the creation of a parallel region, are small, at least in comparison to the memory consistency costs. Also, and although not central to the SDP models, we will make the assumptions that the OpenMP application is perfectly load-balanced (i.e. $T^i = 0$ and $T^c$ is constant across processes) and that the total serial time is zero.

### 4.2.1 Critical Path Model

If the total serial time is zero, the total elapsed time is simply $\sum_r T^\text{par}_r$, where $T^\text{par}_r$ is the runtime for the $r^{th}$ parallel region. Any given $T^\text{par}_r$ is determined by the process in parallel region $r$ that has zero idle time. In the critical path model this is also the process with the maximum value for $T^c + T^o$. Denoting this time as $T^\text{crit}_r$, the value of $T^\text{crit}_r$ for parallel region $r$ is given by:

$$T^\text{crit}_r = \max_{i=0}^{p-1} (T^c_{r,i} + T^o_{r,i})$$ (4.2)

where $i$ is used to indicate process id. (In Figure 4.2, $T^\text{crit}_r$ occurs on $P1$ and $P3$ for parallel regions 1 and 2 respectively.)

The critical path model assumes that the values for $T^o$ are determined solely by the number and type of page faults. For the homeless sDSM system these are either write or fetch page faults. Thus for parallel region $r$ and process $i$ we have:
4.2 SIGSEGV Driven Performance (SDP) Models

Figure 4.2: Schematic illustration of timing breakdown for parallel region using the SDP model

\[ T_{r,i}^o = N_{r,i}^w C^w + N_{r,i}^f C^f \]  
(4.3)

If we also assume that the code is perfectly parallelized, \( T^o \) in Equation (4.5) can be replaced by \( T(1)_{r,\text{par}}^o / p \), where \( T(1)_{r,\text{par}}^o \) is the elapsed time for parallel region \( r \) when the application is run using just one process. Hence

\[ T_{r,\text{crit}}^o = \frac{T(1)_{r,\text{par}}^o}{p} + \max_{i=0}^{p-1} (N_{r,i}^w C^w + N_{r,i}^f C^f) \]  
(4.4)

and the total execution time on \( p \) processors becomes

\[ Tot(p)_{\text{crit}}^o = \frac{Tot(1)}{p} + \sum_r \max_{i=0}^{p-1} (N_{r,i}^w C^w + N_{r,i}^f C^f) \]  
(4.5)
In the above we have also used the fact that the total serial time is zero so \( \sum_r T^{\text{par}}_r = T^{\text{tot}}_1 \).

### 4.2.2 Aggregated Model

Evaluating the time for the critical path SDP model requires a detailed knowledge of the number and types of page faults occurring in every parallel region. In our second, simplified SDP model, this is avoided. Instead only the aggregate SIGSEGVs count is required, and the total execution time is approximated as:

\[
T^{\text{agg}}_p = \frac{T^{\text{tot}}_1}{p} + (N^w C^w + N^f C^f)(f + \frac{1 - f}{p})
\]

where \( f \) is a factor with a value between zero and one. The rationale behind the aggregate model is the idea that for any given application some fraction, \( f \), of the total page faults will be serialised, while the remaining \( (1 - f) \) fraction will be overlapped. The fraction will be application and environment dependent, and may vary as a function of the number of processes used. In this sense the model is empirical.

In comparison to the critical path approach we may expect:

\[
T^{\text{agg}}_p|_{f=0} \leq T^{\text{crit}}_p \leq T^{\text{agg}}_p|_{f=1}
\]

Fully overlapped page faults \( (f = 0) \) will not occur if several processes request diffs from the same location at the same time. In which case we would expect the critical path approach to underestimate the total execution time and a large value for \( f \) to be necessary in the aggregate model.

For both models the cost of fetching a page, \( C^f \), is based on a two-node measurement. For the homeless sDSM system, however, it is possible that diffs need to be fetched from many nodes. Recent work to design a memory consistency benchmark for OpenMP[96] suggests that if this occurs the cost will increase linearly with the number of nodes involved. This will cause the critical path model to underestimate the execution time as the number of processes increases. For the aggregate model it will require the value of \( f \) to increase as the number of processes increases in order to fit the observed data.
4.2 SIGSEGV Driven Performance (SDP) Models

LEGEND: execution time assignment ←

1. \( D_w \leftarrow \text{WRITE}(R) \)
2. \( D_r \leftarrow \text{READ}(R) \)
3. \( \text{READ}(S) \)
4. \( \text{barrier} \)
5. \textbf{if} process-0:
6. \( C_{\text{w}} \leftarrow (\text{WRITE}(S) - D_w)/\text{npages} \)
7. \( \text{barrier} \)
8. \textbf{if} process-1:
9. \( C_f \leftarrow (\text{READ}(S) - D_r)/\text{npages} \)
10. \( C_{w\text{r}} \leftarrow (\text{WRITE}(S) - D_w)/\text{npages} \)

\textbf{Figure 4.3:} The algorithm used to determine the SDP coefficients. The code shown is
in a parallel region. \( R \) is a private array while \( S \) is a shared one. Variables \( D_w \) and \( D_r \)
represent reference times for accessing private array \( R \).

4.2.3 Coefficient Measurement

The model detailed in section 4.2 consists of several coefficients that need to be measured. This is done using an OpenMP C program\(^1\) that involves various write and read operations on shared and private arrays. The first operation, \text{WRITE}(A)\) writes to the elements of \( A \) and returns the time it takes to complete that operation. Similarly, \text{READ}(A)\ will read the elements in \( A \) and return the time required.

\text{Figure 4.3} show how the coefficients can be measured. The algorithm uses private (\( R \)) and shared (\( S \)) arrays of equal length. For the home-based sDSM, runtime options are used to ensure that the pages in the shared array \( S \) will have home locations on the same node as process-0. The time it takes to perform the \text{WRITE} and \text{READ} operations on the private array \( R \) are recorded as \( D_w \) and \( D_r \) respectively (lines 1 and 2). This provides a reference time for performing the memory operations without shared memory consistency concerns.

The measurements for \( C_{w}, C_f, C_{w\text{r}}, \) and \( C_{\text{w}} \) can now be done. Two processes are used in this code. In line 3 and 4, a \text{READ}(S)\ is done by both processes followed by a barrier operation. For CLOMP, this step ensures that the pages of \( S \) begin in a read-only state. Next process-0 performs a write operation on \( S \). For the homeless sDSM system the value recorded is \( C_{w} \), while for the home-based sDSM system the time recorded is \( C_{\text{w}} \). At the end of the write phase a barrier operation is performed.

Lines 9 and 10 are then executed by process-1 to give values for coefficients \( C_f \) and \( C_{w\text{r}} \). Note also that, as with the previous write measurement, the value of \( C_{w\text{r}} \)

\(^1\)The source code of the program is available at http://ccnuma.anu.edu.au/dsm/segv_cost.
on the home-based sDSM system is the same as $C^w$ on the homeless sDSM system.

For CLOMP we have used the SEGVprof profiling tool that is provided as part of the distribution. This tool creates profile files (.gmon files) for all CLOMP processes, reporting the segmentation faults occurring for each process in each parallel region. SEGVprof provides a script (segvprof.pl) that reports aggregated results, and this was extended to produce per-process results.

### 4.3 SDP Model Verification

To test the applicability of the SDP models, we ran the OpenMP version of the NAS Parallel Benchmark (NPB)[26, 49] (NPB-OMP) on an 8-node AMD cluster. Each cluster node has Athlon dual core 2.2 GHz CPU with 4 GB of memory. The nodes where connected using Gigabit Ethernet. The OSU benchmark [61] gave a ping-pong latency and bandwidth for the cluster of 62.5us and 103.4MB/s respectively.

The code outlined in Section 4.2.3 was used to measure the timing coefficients for the SDP models. This gave values of $C^w=21.6us$ and $C^f=320.1us$ for the homeless sDSM, and $C^{wr}=40.5us$, $C^{wl}=20.9us$ and $C^f=295.3us$ for the home-based sDSMs. Results for the homeless and home-based systems will now be considered in detail.

The EP, SP, BT, FT, LU, IS and CG class A and C benchmarks from the NPB-OMP suite were used. Both elapsed time and the associated fault counts have been recorded at a per process level for the parallel regions within the timed section of each NPB benchmark. The number of read and write page faults along the critical paths are shown in Table 4.1. The observed sequential and parallel execution times together with those predicted from the critical path and aggregate SDP models are given in Table 4.2.

The most immediate observation from Table 4.1 is the lack of page faults for EP. This suggests this benchmark will scale equally well on CLOMP as for a non-sDSM OpenMP implementation. Differences between the number of write faults and fetch faults gives an estimate of the number of read only pages. In most cases this difference is relatively small, the exception being for the IS benchmark where fetches typically are more than twice of the number of writes. In a few cases fetch faults are less than the number of writes, this suggests that there are local changes to globally shared pages that are not subsequently requested by other processes. For good scalability the page fault count along the critical path should decrease as the number of processes increases. This is true to some level for all benchmarks, except IS.
4.3 SDP Model Verification

Table 4.1: Critical path page faults counts for the NPB-OMP benchmarks run using CLOMP

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>2 processes</th>
<th>4 processes</th>
<th>8 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write</td>
<td>Fetch</td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP</td>
<td>8.51E+5</td>
<td>8.38E+5</td>
<td>7.36E+5</td>
</tr>
<tr>
<td>BT</td>
<td>4.91E+5</td>
<td>4.93E+5</td>
<td>4.07E+5</td>
</tr>
<tr>
<td>FT</td>
<td>1.18E+5</td>
<td>1.18E+5</td>
<td>8.83E+4</td>
</tr>
<tr>
<td>LU</td>
<td>1.12E+6</td>
<td>1.17E+6</td>
<td>8.59E+5</td>
</tr>
<tr>
<td>IS</td>
<td>2.56E+3</td>
<td>5.10E+3</td>
<td>3.84E+3</td>
</tr>
<tr>
<td>CG</td>
<td>8.54E+3</td>
<td>8.95E+3</td>
<td>6.21E+3</td>
</tr>
<tr>
<td>Class C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP</td>
<td>1.17E+7</td>
<td>1.16E+7</td>
<td>9.57E+6</td>
</tr>
<tr>
<td>BT</td>
<td>7.31E+6</td>
<td>7.31E+6</td>
<td>5.56E+6</td>
</tr>
<tr>
<td>LU</td>
<td>1.58E+7</td>
<td>1.68E+7</td>
<td>9.68E+6</td>
</tr>
<tr>
<td>IS</td>
<td>4.10E+4</td>
<td>8.19E+4</td>
<td>6.14E+4</td>
</tr>
<tr>
<td>CG</td>
<td>3.02E+5</td>
<td>3.03E+5</td>
<td>1.60E+5</td>
</tr>
</tbody>
</table>

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 4.1 are found to vary by around four orders of magnitude between the different benchmarks in a given class. On the other hand the sequential execution times given in Table 4.2 vary by about two orders of magnitude. This suggests that the different benchmarks will show very different behaviour when run using CLOMP. This is indeed the case as evident from the observed times given in Table 4.2, where speedups for eight processors range from around eight for EP to a slowdown of two or more for CG class A or LU class C. For this reason the NPB OMP suite represents an interesting and challenging problem set for cluster-enabled OpenMP implementations.

4.3.1 Critical Path Model Estimates

The estimated elapsed times obtained using the critical path model are shown in Table 4.2. With the exception of EP, and the class C IS and CG benchmarks the observed parallel performance is poor, suggesting that overhead dominates
performance. While this implies that applications of this type and size should not be run in parallel on this cluster using CLOMP, the objective of the work presented here is different; namely to consider how well the proposed models predict the above scalability, whatever that is.

For class A the critical path model predicts speedup reasonably well, except for LU and CG. For these cases the predicted performances are better than the observed ones. The fact that the errors in the predicted values get larger with increasing process count can be attributed to the onset of contention; something that is ignored in the critical path model. The relatively poor agreement for two processes indicates, however, that for these two benchmarks there are other issues beyond contention. For LU the problem appears to be significant load imbalance that is not accounted for by the current version of the critical path model. (When LU is run using two processes with a non-sDSM OpenMP implementation on one of the dual core nodes of the AMD cluster a speedup of just 1.3 is observed.) For CG the problem is that this benchmark contains significant serial regions (∼1.4sec out of 5.8sec) and this is again not included in the current version of the critical path model.

The results for the class C benchmarks show similar trends to those observed for class A. For CG the sequential time is, however, a much smaller fraction of the overall execution time, so the predicted values are now in slightly better agreement with the observed results. The results for IS are interesting, in that the critical path model noticeably underestimates the observed parallel performance. This appears to be due to cache effects that give smaller than predicted computational times on multiple processors (i.e. less than $T_c/p$). (Using performance counters the level 2 cache hit ratio for IS is ∼9% when using one process, but ∼26% on the master process when using eight processes.)

In summary, the relative errors for the class C NPB are smaller than for class A. For both classes the error in the critical path model increases with number of processes used, but this is expected given that the model assumes no contention. The largest deviations occur for LU and CG, with the other benchmarks giving errors that are typically 10% or less.

### 4.3.2 Aggregate Model Estimates

In contrast to the critical path model the aggregate model includes an empirical parameter $f$ that can be adjusted to account for contention. Results for various values of $f$ are shown in Table 4.2. As EP contains no page faults, the predicted results are independent of $f$. For all other benchmarks, except IS, the aggregate model with two processes and $f = 0$ agrees very well with the critical path model.
4.4 Summary

This indicates that the numbers of both fetch and write faults occurring on each process are roughly equal. For IS this is not true, rather there are roughly double the number of fetch faults on one process as the other, and a similar but opposite imbalance for the write faults.

With increasing process count the difference between $S_{agg}(0)$ and $S_{crit}$ tends to increase, indicating greater imbalance between the numbers of each type of page faults on each process. In all cases Equation (4.7) holds.

While differences between $S_{agg}(0)$ and $S_{crit}$ reflect imbalance in the numbers of page faults of a given type occurring on the different processes, contention will cause both $S_{agg}(0)$ and $S_{crit}$ to over estimate $S_{obs}$. Contention is expected to increase with increasing process count, implying that the value of $f$ required by the aggregate model in order to fit the observed data is likely to increase with process count. This trend is apparent for SP, BT and FT, where we find $S_{agg}(f=0)$ to be very accurate for two processes, but a non zero value of $f$ to be more applicable on eight processes (even accounting for the difference between $S_{crit}$ and $S_{agg}(0)$). For LU, IS and CG a similar analysis is much harder given that the values for $S_{obs}$ are contaminated by the load imbalance, serial regions, and cache issues discussed above. Overall, however, the results for all the NPB benchmarks given in Table 4.2 clearly show that the basic cost of servicing the various page faults primarily determines performance (or lack thereof), and contention issues are secondary. On average, $f = 0$ gives most accurate estimation results for the NPB-OMP benchmarks.

4.4 Summary

In this chapter, we developed two region-based SIGSEGV driven performance (SDP) model to rationalize numbers and types of page faults to the performance of page-based cluster OpenMP systems based on following assumptions.

- Page faults occurring on different processes can be fully overlapped, with a total per process cost that can be estimated by knowing the cost of a page fault of given type as measured when using just two nodes.
- The overheads, such as those associated with the creation of a parallel region, are small and negligible in comparison to the memory consistency cost.
- OpenMP application is perfectly load-balanced ($T^c = \frac{Tot(1)}{p}$).

The SDP models enhanced our understanding in the performance of page-based cluster OpenMP implementations. According to measured cost of different
page fault types, a FETCH fault is much more expensive compare to a WRITE fault by more than an order of magnitude.

Moreover, we summarise the overall accuracy of the critical path and aggregate model \((f = 0)\) for the CLOMP system using a variety of different process counts in Table 4.3. These results show that for CLOMP the model based on critical path analysis is slightly more accurate than the simpler model based on aggregate page fault counts. For computations where the page fault overhead from different processes is highly overlapped, the models are generally accurate to within 10\%, and when this is not the case, the models are optimistic.

The accuracy of the current models are known to be limited for applications that contain significant serial regions and/or load imbalance. We resolve these limitations for the critical SDP model by incorporating sophisticated profiling data that generated by modified CLOMP runtime. Please refer to Chapter 6 for more details.
### 4.4 Summary

#### Table 4.2: Comparison between observed and estimated speedup for running NPB class A and C on the AMD cluster with CLOMP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Seq. Time (sec)</th>
<th># of thds</th>
<th>$S_{Obs}$</th>
<th>$S_{Crit}$</th>
<th>$S_{agg}(f)$ for various $f$</th>
<th>0</th>
<th>0.25</th>
<th>0.5</th>
<th>0.75</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class A</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>26.47</td>
<td>2</td>
<td>2.05</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>8.25</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
</tr>
<tr>
<td>SP</td>
<td>137.4</td>
<td>2</td>
<td>0.41</td>
<td>0.39</td>
<td>0.39</td>
<td>0.32</td>
<td>0.28</td>
<td>0.24</td>
<td>0.21</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.45</td>
<td>0.46</td>
<td>0.47</td>
<td>0.28</td>
<td>0.20</td>
<td>0.16</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.50</td>
<td>0.59</td>
<td>0.61</td>
<td>0.23</td>
<td>0.15</td>
<td>0.11</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>BT</td>
<td>145.4</td>
<td>2</td>
<td>0.64</td>
<td>0.60</td>
<td>0.60</td>
<td>0.51</td>
<td>0.45</td>
<td>0.40</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.77</td>
<td>0.77</td>
<td>0.80</td>
<td>0.50</td>
<td>0.36</td>
<td>0.28</td>
<td>0.23</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.95</td>
<td>1.06</td>
<td>1.11</td>
<td>0.44</td>
<td>0.28</td>
<td>0.20</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>FT</td>
<td>12.0</td>
<td>2</td>
<td>0.27</td>
<td>0.26</td>
<td>0.26</td>
<td>0.21</td>
<td>0.18</td>
<td>0.16</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.37</td>
<td>0.36</td>
<td>0.36</td>
<td>0.21</td>
<td>0.15</td>
<td>0.12</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.59</td>
<td>0.61</td>
<td>0.62</td>
<td>0.24</td>
<td>0.15</td>
<td>0.11</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>LU</td>
<td>197.2</td>
<td>2</td>
<td>0.29</td>
<td>0.40</td>
<td>0.40</td>
<td>0.33</td>
<td>0.28</td>
<td>0.25</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.31</td>
<td>0.52</td>
<td>0.53</td>
<td>0.32</td>
<td>0.23</td>
<td>0.18</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.31</td>
<td>0.65</td>
<td>0.70</td>
<td>0.27</td>
<td>0.17</td>
<td>0.12</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>IS</td>
<td>3.4</td>
<td>2</td>
<td>1.03</td>
<td>1.00</td>
<td>1.13</td>
<td>1.02</td>
<td>0.93</td>
<td>0.85</td>
<td>0.79</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.10</td>
<td>1.00</td>
<td>1.36</td>
<td>0.91</td>
<td>0.68</td>
<td>0.55</td>
<td>0.46</td>
<td>0.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1.06</td>
<td>1.00</td>
<td>1.58</td>
<td>0.66</td>
<td>0.41</td>
<td>0.30</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>CG</td>
<td>5.8</td>
<td>2</td>
<td>0.78</td>
<td>0.97</td>
<td>0.98</td>
<td>0.87</td>
<td>0.78</td>
<td>0.71</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.64</td>
<td>1.00</td>
<td>1.04</td>
<td>0.67</td>
<td>0.49</td>
<td>0.39</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.39</td>
<td>1.05</td>
<td>1.07</td>
<td>0.43</td>
<td>0.27</td>
<td>0.19</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td><strong>Class C</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>428.5</td>
<td>2</td>
<td>2.04</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4.08</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>8.19</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
</tr>
<tr>
<td>SP</td>
<td>2346.9</td>
<td>2</td>
<td>0.48</td>
<td>0.46</td>
<td>0.46</td>
<td>0.38</td>
<td>0.33</td>
<td>0.29</td>
<td>0.26</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.59</td>
<td>0.60</td>
<td>0.62</td>
<td>0.38</td>
<td>0.27</td>
<td>0.21</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.71</td>
<td>0.76</td>
<td>0.87</td>
<td>0.34</td>
<td>0.21</td>
<td>0.15</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>BT</td>
<td>2767.3</td>
<td>2</td>
<td>0.75</td>
<td>0.71</td>
<td>0.71</td>
<td>0.61</td>
<td>0.54</td>
<td>0.48</td>
<td>0.43</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
<td>0.66</td>
<td>0.49</td>
<td>0.39</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1.35</td>
<td>1.42</td>
<td>1.59</td>
<td>0.66</td>
<td>0.42</td>
<td>0.30</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>LU</td>
<td>3284.6</td>
<td>2</td>
<td>0.33</td>
<td>0.44</td>
<td>0.44</td>
<td>0.37</td>
<td>0.32</td>
<td>0.28</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.43</td>
<td>0.67</td>
<td>0.68</td>
<td>0.42</td>
<td>0.30</td>
<td>0.24</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.60</td>
<td>1.13</td>
<td>1.14</td>
<td>0.46</td>
<td>0.29</td>
<td>0.21</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>IS</td>
<td>113.9</td>
<td>2</td>
<td>1.50</td>
<td>1.35</td>
<td>1.46</td>
<td>1.37</td>
<td>1.29</td>
<td>1.21</td>
<td>1.15</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.87</td>
<td>1.64</td>
<td>2.08</td>
<td>1.53</td>
<td>1.21</td>
<td>1.00</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>2.15</td>
<td>1.85</td>
<td>2.72</td>
<td>1.26</td>
<td>0.82</td>
<td>0.61</td>
<td>0.48</td>
<td>0.48</td>
</tr>
<tr>
<td>CG</td>
<td>1385.7</td>
<td>2</td>
<td>1.68</td>
<td>1.74</td>
<td>1.74</td>
<td>1.68</td>
<td>1.63</td>
<td>1.58</td>
<td>1.54</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.71</td>
<td>2.80</td>
<td>2.80</td>
<td>2.29</td>
<td>1.93</td>
<td>1.67</td>
<td>1.48</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>3.24</td>
<td>4.03</td>
<td>4.04</td>
<td>2.16</td>
<td>1.48</td>
<td>1.12</td>
<td>0.90</td>
<td>0.90</td>
</tr>
</tbody>
</table>
Table 4.3: Average relative errors for the predicted NPB speedups evaluated using the critical path and aggregate \((f = 0)\) SDP models and data from Tables 4.2.

<table>
<thead>
<tr>
<th>nprocs</th>
<th>Crit</th>
<th>agg((f=0))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CLOMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class A</td>
<td>0.12</td>
<td>0.20</td>
</tr>
<tr>
<td>Class C</td>
<td>0.10</td>
<td>0.13</td>
</tr>
<tr>
<td>Without</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class A</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>LU &amp; CG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class C</td>
<td>0.05</td>
<td>0.04</td>
</tr>
</tbody>
</table>
Part III

Optimizations: Design, Implementation and Evaluation

The key to enhance performance of a parallel system is to reduce its overhead as much as possible.

In Chapter 3 and 4, the performance of Intel Cluster OpenMP (CLOMP) system is evaluated and modeled. The memory consistency cost has been identified as the major system overhead, especially the FETCH page fault servicing (page/diff transfer) cost. According to the performance evaluation results, a faster network helps reducing this overhead. However, the performance are still not satisfied.

To address this problem, we design and implement three region-based page prefetch (ReP) techniques for CLOMP to further reduce this overhead. Each ReP technique is described in detail and compared with some well known existing page prefetch techniques in Chapter 5. The implementation are discussed and evaluated in Chapter 6.
Chapter 5

Region-Based Prefetch Techniques

Contents

5.1 Limitations of Current Prefetch Techniques for sDSM Systems ........................................ 82
  5.1.1 Parallel Application Examples ................................ 82
  5.1.2 Limitations ........................................ 85
  5.1.3 Prefetch Technique Design Assumptions ................. 88
5.2 Evaluation Metrics of Prefetch Techniques ....................... 89
5.3 Temporal ReP (TReP) Technique ............................... 90
5.4 Hybrid ReP (HReP) Technique ............................... 90
5.5 ReP Technique for Dynamic Memory Accessing Applications (DReP) ................................. 93
  5.5.1 Stride-augmented Run-length Encoded Page Fault Records 93
  5.5.2 Page Miss Prediction ............................... 95
5.6 Offline Simulation ........................................ 97
  5.6.1 Simulation Setup ................................ 97
  5.6.2 Simulation Results and Discussions ................. 98
5.7 Summary ........................................ 106
Since the dominant overhead of the Intel Cluster OpenMP (CLOMP) system has been identified as memory consistency cost, especially the FETCH page fault servicing cost, the best approach to optimize CLOMP is to reduce its FETCH page fault servicing cost. An effective page prefetch technique can significantly reduce this overhead.

The limitations of the existing prefetch techniques reviewed in Chapter 2 are discussed in this chapter by analyzing two different OpenMP implementations of LINPACK. It is then followed by details of design on three region-based prefetch (ReP) techniques. Temporal ReP (TReP) considers temporal paging behaviour only; Hybrid ReP (HReP) considers both temporal and spatial paging behaviour; Dynamic ReP (DReP) considers dynamic paging behaviour. At the end of this chapter, the three ReP technique will be verified preliminarily with off-line simulations.

5.1 Limitations of Current Prefetch Techniques for sDSM Systems

There are some obvious limitations to some of the existing sDSM prefetch techniques. To discover and analyze them, we use two OpenMP implementations of LINPACK as examples. The example LINPACK implementations are described first, then the limitations of different sDSM prefetch techniques are discussed.

5.1.1 Parallel Application Examples

To understand better the execution paging behaviour of (parallel) regions, we analyze two different implementations of the LINPACK OpenMP benchmarks as parallel application examples. ¹

The LINPACK benchmark is an implementation of the LU decomposition with partial pivoting to solve a dense $N \times N$ system of linear equations:

$$Ax = b$$  \hspace{1cm} (5.1)

with $A \in \mathbb{R}^{n\times n}$, and $x, b \in \mathbb{R}^n$. The solution is obtained by Gaussian elimination with partial pivoting. We have implemented a LINPACK OpenMP benchmark using a blocked LU decomposition algorithm, resulting in a floating point workload of $2/3N^3$.

¹This OpenMP LINPACK benchmarks are available at http://cs.anu.edu.au/~Jie.Cai.
5.1 Limitations of Current Prefetch Techniques for sDSM Systems

```c
/* A is a column-major n x n matrix,
 nb is the blocking factor */
for (j = 0; j < n; j += nb) {
  /* region 1 -- sequential */
  /* factor the current block column,
   apply row swap to left */
  read/write(A[j:n, 0:j+nb]);

  /* access the current block column */
  read(A[j+1:n, j:j+nb]);

  /* apply tow swap to right */
  read/write(A[j+nb:n, j:j+nb]);

  /* region 2 -- parallel */
  #pragma omp parallel default(shared) private(i, jk, jh) {
    tid=omp_get_thread_num();
    nthr=omp_get_num_threads();
    chk = (n-j-nb)/nthr
    /* sub matrix update by the production of the two panels */
    read(A[j+nb:n, j:j+nb]);
    read(A[j:j+nb, j+nb+tid*chk:j+nb+(tid+1)*chk]);
    write(A[j+nb:n, j+nb+tid*chk:j+nb+(tid+1)*chk]);
  } /*end of region 2*/
} /*end of most outter loop*/
```

Figure 5.1: Pseudo code to demonstrate the memory access patterns of the naive LINPACK OpenMP benchmark implementation for an $n \times n$ column-major matrix A with blocking factor $nb$.

5.1.1.1 Naive Implementation

Figure 5.1 shows pseudo code for the parallelized section of the naive LINPACK OpenMP implementation.

In this scheme, the parallel region is re-executed several times. The number of pages accessed in each iteration is decreased, and not all pages accessed in the current execution of the region will be accessed again in the next execution. However, the access pattern changes in a regular fashion. Figure 5.2 illustrates the paging behaviour of the LINPACK benchmark for four threads over two iterations. In order to simplify the scenario, we set $n = 8nb$ in Figure 5.2.

All write operations to the matrix exploit both temporal and spatial data locality. All read operations explore only spatial data locality. However, as the memory of the matrix A is allocated contiguously, both write and read operations
performed by one thread will explore multiple spatial localities (different stride patterns). The stride along rows is 1 page, and the stride along column is \( \frac{n}{\text{SystemPageSize}} \) pages. However, when \( \frac{n}{nb} \) is large, the dominant stride of the
5.1 Limitations of Current Prefetch Techniques for sDSM Systems

According to Figure 5.2, a large portion of the whole memory area of matrix is missed in every region of each iteration. This creates significant memory consistency costs in CLOMP. Moreover, in the second region of each iteration, all slave threads ($T_1$ to $T_3$) will need to request pages/diffs from thread-0. Due to each CLOMP process deploying single sDSM daemon thread to handle communication requests, the pages/diffs requesting behaviour will have to be served by the sDSM thread of $T_0$. As a result, the computation done by slave threads is serialized. The above mentioned two reasons result in low parallelism efficiency of the naive LINPACK implementation.

Additionally, the page fault area of each process contract in every second iteration. This pattern exhibits good temporal locality and some spatial locality between executions of a region.

5.1.1.2 Optimized Implementation

The details of the memory access pattern of the optimized implementation is shown in Figure 5.3.

Based on the parallelism method of the optimized LINPACK program, the consecutive page fault areas are illustrated in Figure 5.4: (a) demonstrates the memory access areas for different iterations, and (b) illustrates the page fault areas for different iterations.

In this LINPACK OpenMP implementation, regions are repeatedly executed in each iteration, which is referred to as region-executions. As we can see from Figure 5.4 (b), between region-executions, for each thread, there is no overlap in the page fault area. Additionally, the number of page faults keeps reducing with increasing number of region-executions.

5.1.2 Limitations

The Dynamic Aggregation technique assumes that the page faults that occurred in the current parallel region will occur again in the consecutive parallel region. The B+ technique has a similar assumption. This is obviously not suitable for most parallel applications.

The Adaptive++ technique assumes that a region will experience the same page access strides (same fault pages) as either the previous region or the region before that. Thus, Adaptive++ improves Dynamic Aggregation and the B+ techniques
/* A is a column-major n x n matrix, 
   nb is the blocking factor */

for (j = 0; j < n; j += nb) {

    /* region 1 -- sequential 
       factor the current block column, 
       apply row swap to left */
    read/write(A[j:n, 0:j+nb]);

    /* region 2 -- parallel */
    #pragma omp parallel default(shared) private(i, jk, jh) {
        tid=omp_get_thread_num();
        nthr=omp_get_num_threads();
        chk = (n-j-nb)/nthr

        /* access the current block column */
        read(A[j+1:n, j:j+nb]);

        /* apply row swap to right */
        read/write(A[j:n, j+nb+tid*chk:j+nb+(tid+1)*chk]);

        /* update trailing sub-matrix, 
           require access of block column*/
        read/write(A[j:n, j+nb+tid*chk:j+nb+(tid+1)*chk]);

    } /*end of region 2*/

} /*end of most outter loop*/

Figure 5.3: Pseudo code to demonstrate the memory access patterns of the optimized LINPACK OpenMP benchmark implementation for an n x n column-major matrix A with blocking factor nb.

by having page fault records for the two most previous regions and choosing one as target region for prediction, but it is still not adequate for complex parallel applications with multiple parallel regions.

TODFCM is a more generic prefetch technique; however, it does not consider the characteristics of the sDSM memory consistency models. Furthermore, as it only predicts one page fault at a time, only one page can be prefetched in advance, which limits the overlap of computation and communication. Some attempts have been done by Speight et al. to reduce the number of network communications via prefetching multiple pages at one time instead of only one page. However, these attempts resulted in a significant reduction in the prefetch efficiency (~45% reduction for 8 pages and ~25% reduction for 4 pages) [88].

Additionally, previous sDSM prefetch techniques all assumed that either a future page fault has happened in a previously executed parallel region or a future
5.1 Limitations of Current Prefetch Techniques for sDSM Systems

![Diagram of memory access areas for different iterations](image)

![Diagram of page fault areas for different iterations](image)

**Figure 5.4:** Optimized OpenMP LINPACK program: (a) memory access areas for different iterations illustrated on a $n \times n$ matrix panel. (b) page fault areas for different iterations illustrated on the $n \times n$ matrix panel.

Page fault can be predicted based on the ID of a previously missed page.
Chapter 5: Region-Based Prefetch Techniques

The optimized implementation of the LINPACK OpenMP benchmark exhibits a paging behaviour which both page fault area and number of page faults are totally different between region-executions. This is again not covered by any of the assumptions made by the existing techniques.

5.1.3 Prefetch Technique Design Assumptions

The naive and optimized LINPACK implementations for cluster-enabled OpenMP systems provide an interesting pattern of page accesses to test prefetch techniques on.

Based on observations of the above LINPACK program and the NPB-OMP suite, we found that these applications exhibit the three major types of behaviour:

1. A region executed previously is likely to be executed again in the near future.

2. The page accesses within a region-execution will either show good temporal or spatial locality. In other words, either the page faults or the strides between the consecutive page faults in an execution of a region are likely to be repeated in the future execution of the same region.

3. Executions of the same region usually exhibit the most similar paging behaviour if they are temporally proximate; i.e. in the two previous executions of that region.

4. Between executions of the same region, paging behaviour is likely to be either temporal or spatial. In other words, the temporal paging behaviour exhibits the same page misses between executions, while the spatial paging behaviour exhibits the different page misses with or without overlaps between executions. However, for the spatial paging behaviour between executions, the paging patterns are the same.

In order to address the limitations of those existing sDSM prefetch techniques and fulfill the observations for paging and region execution behaviour of parallel applications, we designed three Region-Based Prefetch (ReP) techniques. The first ReP technique only considers the temporal paging behaviour between consecutive region-executions, the second ReP technique addresses both temporal paging behaviour between consecutive region-executions and spatial paging behaviour within a region-execution, and the last ReP technique addresses both temporal and spatial paging behaviour between region-executions.
5.2 Evaluation Metrics of Prefetch Techniques

Some widely agreed metrics to evaluate and validate different prefetch techniques are introduced in this section.

A useful prefetch: is a page prefetch issued before the actual access. When a prefetch is not useful for the next execution of a region, it may still be useful for a later execution if the page is not invalidated during the period.

Efficiency: this is the ratio of prefetches which were useful. Prefetch techniques need to improve this metric to avoid unnecessary prefetches and to eliminate the associated overhead. It can be calculated as follows.

\[
E = \frac{N_u}{N_p}
\]  

(5.2)

In Equation (5.2), \(E\) stands for efficiency, while \(N_u\) and \(N_p\) denote the number of useful prefetches, and the number of prefetched pages.

Coverage: this is the ratio of useful prefetches to the total number of page misses. Prefetch techniques need to improve this metric to achieve a better overall hit rate. It can be calculated as follows.

\[
C = \frac{N_u}{N_f}
\]  

(5.3)

In Equation (5.3), \(C\) stands for coverage, while \(N_u\) and \(N_f\) denote the number of useful prefetches and the number of total page misses.

Before describing ReP techniques, we briefly introduce some metrics which will be used for ReP techniques.

Similarity: the similarity of two page fault lists (\(l_1\) and \(l_2\)) is calculated as follows.

\[
S_{l_1 l_2} = \frac{N_{\text{same}}}{N_{l_1}}
\]  

(5.4)

In Equation (5.4), \(S_{l_1 l_2}\) denotes the ratio of \(N_{\text{same}}\) against \(N_{l_1}\). \(N_{\text{same}}\) stands for number of fault pages belonging to both list, and \(N_{l_1}\) stand for the number of page faults in \(l_1\). Similarly, \(S_{l_2 l_1}^{l_1}\) can be calculated. A similarity of greater than 50% means both \(S_{l_1 l_2}\) and \(S_{l_2 l_1}^{l_1}\) need to be greater than 50%.

Frequency: it represents how often a stride appears in a page fault list. The most common stride frequency of a given page fault list can be calculated as follows.
In Equation (5.5), $F_c$ stands for frequency of the most common stride ($c$). $N_c$ denotes the number that the most common stride ($c$) appears in a given page fault list, and $N_s = N_l - 1$ denotes the number of strides in the list.

### 5.3 Temporal ReP (TReP) Technique

Only the temporal paging behaviour is considered in the Temporal ReP (TReP) technique.

All experienced page faults are recorded on a per region basis for the TReP predictor. Each record entry contains the region ID, the number of page faults in the region, and the fault page IDs, as shown in Figure 5.5.

Immediately after a barrier (at the beginning of a region execution), the TReP predictor will look-up the previous executions of the current region ID in the records. If there are at least two previous executions, the TReP will issue prefetches for this region. Otherwise, the TReP will not do any page prefetch operations in this region.

To issue prefetches, TReP treats the two recent executions of the current region as two page fault lists. If the two lists are “highly similar”, the application is deemed to show good temporal locality, and the whole list of pages for the most recent previous execution of the current region is prefetched. Otherwise, TReP will not prefetch any pages for the current region. The term “highly similar” means that the similarity of these two lists is above a pre-defined threshold (this will be defined and examined in Section 5.6).

### 5.4 Hybrid ReP (HReP) Technique

Based on TReP, we designed a more complex prefetch technique to target both temporal and spatial data locality and all observations listed in Section 5.1.3.
5.4 Hybrid ReP (HReP) Technique

This prefetch mechanism firstly verifies whether the current region has been executed previously. If the current region was executed previously (even only once), the predictor will predict possible pages that will be accessed in the region by utilizing a hybrid prefetch technique combining TReP and Adaptive++. Therefore, we name it the Hybrid ReP (HReP) technique. The details of the HReP prefetch mechanism are as follows.

In the HReP predictor (Figure 5.6), there are three prefetch modes in total, whole-phase prefetch mode, repeated-phase prefetch mode [13], and repeated-stride prefetch mode [13]. The whole-phase prefetch mode utilizes the TReP technique, while the repeated-phase mode and repeated-stride mode utilize the prefetch mechanisms from the Adaptive++ technique (refer to Section 2.3.2.5). Three paging behaviours were considered: full temporal locality (whole-phase), partial temporal locality (repeated-phase) and spatial locality (repeated-stride).

When $p_list$ and $bp_list$ (refer to Section 2.3.2.5 and Figure 5.6) are “highly similar”, whole-mode is used. Otherwise, either the repeated-phase or the repeated-stride mode is used. Similar to the Adaptive++ technique, picking the repeated-phase or the repeated-stride mode depends on the paging behaviour of the chosen list. When the efficiency of the prefetches that the repeated-phase mode would have issued for the last execution of this region is greater than or equal to the most common stride frequency (temporal locality is more dominant than spatial locality), the repeated-phase mode is picked. Otherwise, the repeated-stride mode is picked. If the chosen list does not show either good temporal or spatial locality, the HReP will not issue any prefetches for this region. Good temporal or spatial locality is determined by whether the efficiency or the frequency is higher than a pre-defined value; this value will be determined empirically in Section 5.6.

Although the HReP predictor is developed based on both TReP and Adaptive++ techniques, there are a few differences which improve the prediction compared to the Adaptive++.

Firstly, $p_list$ and $bp_list$ are determined from the first and second most recent executions of the region with same region ID as the current region (in Adaptive++, they are determined from the executions of the first and second most recent regions). Since $p_list$ and $bp_list$ are determined from the two most recent executions of the region with the same ID, it is much easier to predict the page faulting pattern for the same region.

Secondly, the determination of the chosen list is different. If only one previous region is found having the same ID as the current region, $p_list$ will be chosen.

Finally, if $p_list$ is “highly similar” to $bp_list$, HReP will issue prefetches immediately for all the fault pages of $p_list$ at once, then it exits the HReP predictor.
Chapter 5: Region-Based Prefetch Techniques

Figure 5.6: A flowchart of the HReP predictor.
5.5 ReP Technique for Dynamic Memory Accessing Applications (DReP)

This is the whole-phase prefetch mode.

5.5 ReP Technique for Dynamic Memory Accessing Applications (DReP)

To address the paging behaviour which exhibits the dynamically changed page miss area and the length of the area as we illustrated in Section 5.1.1.2, it is difficult to analyze the pattern of these changes using a linked list like page miss ID record. Therefore, we propose a novel stride-augmented run-length encoding (sRLE) method to compress and reconstruct missed page IDs on a per region-execution basis, and then the encoded page miss records are stored together with the region ID and a flag to indicate whether it is a parallel or sequential region. At the beginning of a region execution, DReP uses the encoded record to determine the pattern of the page fault area and length changes, and predicts the possible page misses for the current region execution.

The details of utilizing the stride-augmented run-length encoding method to reconstruct page fault records and prefetch predictions and issues are described in the rest of this section.

5.5.1 Stride-augmented Run-length Encoded Page Fault Records

Page faults usually do not occur in an orderly fashion, and the same page may be missed in the same region execution due to flush and lock operations. In order to observe paging behaviour at the region execution level, the missed page fault IDs are firstly sorted, and then the minimum strides between consecutive pages are calculated. Based on the minimum strides, the sorted page fault IDs are broken into many arrays to have consecutive pages with same stride placed in one array, as shown in Figure 5.7 (a).

After the sorted page fault records have been broken into small arrays, we utilize the run-length encoding method augmented with the common stride of these arrays to encode them into a first level format as shown in Figure 5.7 (b), which contains a start page of the array, the common stride between consecutive pages in the array, and run-length of the array (number of elements of the array).

Some of the first level encoded entries have the same common stride and run length. Again, we further encode these first level encoded entries, augmented with the stride between their start pages, into a more compressed format, named as the second level encode entry. As shown in Figure 5.7 (c), each second level encoded
entry contains a start first-level encoded entry, the common stride between the start pages of those first level entries, and number of first level entries encoded.
5.5 ReP Technique for Dynamic Memory Accessing Applications (DReP)

<table>
<thead>
<tr>
<th>Region ID</th>
<th>Parallel?</th>
<th>Num of Fault Pages</th>
<th>Fault Page ID</th>
</tr>
</thead>
</table>

Before reconstruction

![Diagram]

After Run-Length Encoding Reconstruction

<table>
<thead>
<tr>
<th>2^2 level encode entry #1</th>
<th>2^2 level encode entry #2</th>
<th>2^2 level encode entry #3</th>
<th>2^2 level encode entry #4</th>
<th>2^2 level encode entry #5</th>
</tr>
</thead>
</table>

Figure 5.8: Page fault record of region execution reconstructed via run-length encoding method.

Each second-level encoded entrie represent a 2D rectangle view of page fault area. It largely increases the accuracy of analysis for dynamic paging patterns.

Once all encoding processes are completed, a page fault record for a region-execution can be reconstructed as shown in Figure 5.8, which contains the ID of the region, a flag to indicate whether it is a parallel region, number of total page faults, and a list of second level encoded entries. The algorithms, used in the three steps of sRLE to compress and restructure a page fault record of an execution of a region, are presented in Appendix A.

5.5.2 Page Miss Prediction

The DReP technique predicts page misses for the current region at the beginning of each region executions. DReP will first look up the records list to find out whether the current region has been previously executed at least twice.

If the region has not been previously executed at least twice, there is no prefetch issued. Otherwise, DReP compares the two sRLE encoded page fault records of the two previous region executions and issues prefetches. DReP loops through each the second level encoded entry contained in the record of the most recent previous execution (p_list) and compares it to that of the one before most recent execution record (bp_list). This process compares all the entry pairs formed by the two records. An entry pair is a pair of the second level encoded entries.
contain one entry from each list. For a given entry in \( p_{list} \), it can form a set of entry pairs. The comparison can result in three different cases as discussed below (see Appendix A for detailed algorithms).

For the first case, when an entry is common to both lists, this entry is selected to calculate an encoded entry for prefetching. DReP moves to the next set of entry pairs.

For the second case, if there is not a common entry to both lists, but there is an entry pair that have the same strides and run lengths, this pair is selected to predict a second level encoded entry for prefetching. The predicted entry has the same strides and run lengths as that of any entry in the entry pair, while its start page is calculated as follows:

\[
P = P_p + (P_p - P_{bp})
\]

where \( P \) stands for the start page of the predicted entry, \( P_p \) stands for the start page of the entry recorded in \( p_{list} \), and \( P_{bp} \) stands for the start page of the entry recorded in \( bp_{list} \).

For the last case, if neither of the first and second cases are satisfied, but there is an entry pair that the contained two entries have the same strides, and their run lengths are “highly similar” (the similarity is larger than a threshold), this pair is also can also be used to predict a second level encoded entry for prefetching. The predicted entry has the same strides as that of any entry in the pair, while its start page can be calculated using Equation (5.6) as the second case, and its run lengths can be calculated as follows:

\[
L1 = L1_p + (L1_p - L1_{bp}) \\
L2 = L2_p + (L2_p - L2_{bp})
\]

where \( L1 \) and \( L2 \) stand for run length of the first and second level encoding of the predicted entry respectively, \( L1_p \) and \( L2_p \) stand for the run length of the first and second level encoding of the entry in \( p_{list} \) respectively, and \( L1_{bp} \) and \( L2_{bp} \) stand for the run length of the first and second level encoding of the entry in \( bp_{list} \) respectively. Note that with this mechanism, one second level encoded entry in \( p_{list} \) can be similar to multiple \( bp_{list} \) entries, which may potentially introduce unnecessary prefetches.

If none of the above cases is satisfied, DReP would not issue any prefetches for this entry pair set.

After all entry pair sets are compared, a list of second level entries are predicted. These predicted entries are decoded into a list of page IDs, and these pages are prefetched.
5.6 Offline Simulation

5.6.1 Simulation Setup

According to a comparison of B+, Dynamic Aggregation and Adaptive++, Adaptive++ performs better than the other two techniques in terms of efficiency [13]. Therefore, we will simulate and compare five different sDSM techniques: TReP, HReP, DReP, Adaptive++ and TODFCM. In [88], evaluation of TODFCM showed that prefetch efficiency is decreased by \( \sim 45\% \) and \( \sim 25\% \) when 8 pages and 4 pages are prefetched at each fault. Therefore, in the simulation for TODFCM, only one page is prefetched at each fault. In the repeated-phase and repeated-stride modes of both Adaptive++ and HReP, 4 pages will be prefetched at each page fault.

5.6.1.1 Offline Page Fault Records

The five different sDSM prefetch techniques have been implemented and evaluated in an offline simulator. The naive LINPACK OpenMP benchmark (\( n = 2048 \) with \( nb = 64 \) and \( nb = 16 \)), the optimized OpenMP benchmark (\( n = 2048 \) with \( nb = 64 \)) analyzed in Section 5.1.1, and some of class A OpenMP NAS Parallel Benchmarks (OMP-NPB) were chosen to generate page fault records on a per thread and per region basis using CLOMP. These records were then used as inputs for the offline simulations.

5.6.1.2 Pre-defined Thresholds for ReP Techniques

The thresholds used in ReP techniques to be defined in this section are:

- the degrees of similarity to
  - identify “highly similar” lists for both TReP and HReP,
  - identify “similar” lists for HReP,
  - identify “highly similar” run lengths for DReP,

- the degrees of efficiency and frequency to identify whether a region’s execution shows good temporal or spatial data locality for HReP.

Table 5.1 shows the effect of different values for the similarity of “highly similar” lists for the naive LINPACK OpenMP implementation (\( n = 2048 \) and \( nb = 64 \)). The results show that DReP, TReP, and HReP have an increasing efficiency \( (E) \) with increasing threshold for “highly similar”. However, the number
of prefetches issued ($N_p$) dropped with increasing threshold. A similar effect is also observed for NPB-OMP benchmarks. Therefore, in order to achieve a balance between the number of prefetches and its efficiency, we chose 80% for TReP and HReP, and 90% for DReP as the threshold of “highly similar”.

Similar tests were done to determine the threshold of “similar” lists and optimal efficiency and frequency for HReP. The naive LINPACK OpenMP program and NPB-OMP benchmarks were used. We found that the the optimal similarity threshold for two “similar” lists is 50%. The optimal efficiency and frequency thresholds, indicating good temporal and spatial data locality, were found to be both 50%. These results are similar to what been tested and chosen by Bianchini et al. in [13] and Lee et al. in [56].

5.6.2 Simulation Results and Discussions

In this section, the offline simulation results will be discussed in four aspects: reduction of network communications, prefetch efficiency, prefetch coverage, and effective page miss reduction. A breakdown analysis for the HReP technique is also performed.

In general, the fewer the useless prefetches are issued (high efficiency) and the more the page faults are reduced (high coverage), the more effective a prefetch technique is. These metrics correspond to the well-known metrics of precision and recall in text retrieval (see Section 5.2 for more details). The effective page miss

---

<table>
<thead>
<tr>
<th>Prefetch Techs</th>
<th>highly similar (%)</th>
<th>2 threads $E$ (%)</th>
<th>2 threads $N_p$ (x1000)</th>
<th>4 threads $E$ (%)</th>
<th>4 threads $N_p$ (x1000)</th>
<th>8 threads $E$ (%)</th>
<th>8 threads $N_p$ (x1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DReP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>94.2</td>
<td>81</td>
<td>93.9</td>
<td>131</td>
<td>93.2</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>96.0</td>
<td>78</td>
<td>95.7</td>
<td>123</td>
<td>95.1</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>97.6</td>
<td>76</td>
<td>96.9</td>
<td>112</td>
<td>95.9</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>TReP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>87.5</td>
<td>83</td>
<td>84.3</td>
<td>90</td>
<td>87.9</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>88.0</td>
<td>80</td>
<td>88.2</td>
<td>86</td>
<td>91.2</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>90.6</td>
<td>51</td>
<td>90.9</td>
<td>59</td>
<td>91.5</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>HReP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>83.2</td>
<td>97</td>
<td>81.2</td>
<td>142</td>
<td>80.8</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>85.0</td>
<td>97</td>
<td>81.8</td>
<td>139</td>
<td>81.8</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>85.2</td>
<td>91</td>
<td>82.3</td>
<td>130</td>
<td>82.1</td>
<td>146</td>
<td></td>
</tr>
</tbody>
</table>
5.6 Offline Simulation

reduction is a combination of these metrics corresponding to the impact on prefetch and page fault overhead on execution time.

5.6.2.1 Reduction of Network Communications

As the TReP and DReP techniques only prefetch once per region execution, it maximally reduces the number of network transfers required to serve page misses. HReP is not quite as good, because the repeated-phase and repeated-stride modes are used for runtime prefetch as well. HReP can achieve the same number of reduction in network accesses as other ReP techniques only when whole-phase mode is used; and it would reduce network communications by a factor of $q$ when either repeated-phase or repeated-stride mode is used. By contrast, Adaptive++ has the same number of network communications as the worse case of HReP. TODFCM cannot reduce any network communications, as it prefetches a single page at a time.

5.6.2.2 Efficiency

Table 5.2 shows simulation results using a different number of threads for the naive LINPACK implementation (nLPK), the optimized LINPACK implementation (oLPK) and the NPB-OMP benchmarks. The total number of page faults ($N_f$) and the number of prefetches issued ($N_p$) are presented in thousands ($\times 1000$), and the number of useful prefetched pages is presented as a ratio to $N_f$, which is also known as the coverage. The efficiency of each prefetch technique is calculated based on Equation (5.2).

As can be seen from Table 5.2, almost all prefetch techniques shows good prefetch efficiency for the naive LINPACK benchmark ($\sim 80\%$) except Adaptive++. The efficiency of Adaptive++ decreases from $\sim 81\%$ at $nb = 16$ to $\sim 76\%$ at $nb = 64$. A similar trend is observed for TReP and HReP as well. Recalling the memory access and page fault pattern of the naive LINPACK analysis in Section 5.1.1, the memory region accessed at each iteration of the parallel region changes more with increasing block size, thus reducing temporal locality. On the contrary, TODFCM and DReP show a roughly stable prefetch efficiency as the block size is changed. This is because TODFCM only issues prefetch when the same consecutive faults appears three times, which avoids useless prefetches in this case. As DReP issues prefetches based on the page fault changing patterns, the different block sizes has less effect to DReP technique.
Moreover, Adaptive++, TODFCM, and DReP prefetch techniques roughly maintain their efficiency with increasing number of threads. On the other hand, HReP shows a slight decrement in prefetch efficiency with increasing number of threads. The efficiency of TReP increases when \( nb = 64 \) and decreases when \( nb = 16 \). As with a finer grain data distribution (more working threads), TReP and the whole-phase mode of HReP will result in a greater portion of useless prefetches, which directly contributes to the decrement of the prefetch efficiency for both TReP and HReP. However, with more threads employed, the data is partitioned into finer chunks, and the memory access pattern changes less each iteration, and results in the increase of efficiency. These two factors balance out the effect of each other for TReP and the whole-phase mode of HReP. Therefore, for TReP, a increasing efficiency is observed when \( nb = 64 \) and decreasing efficiency is observed when \( nb = 16 \). Additionally, for HReP, with a finer grain data distribution, exploiting spatial locality becomes more complex. The portion of consecutive faults with a stride \( \frac{n}{\text{SystemPageSize}} \) becomes more significant, resulting in an increase in the number of useless prefetches.

The observed prefetch efficiency for the optimized LINPACK benchmark is very different with that of the naive LINPACK implementation. Adaptive++ and HReP pose very poor efficiency, less than 30%, for different number of threads. It is because that the repeated-stride mode of both Adaptive++ and HReP issues the most prefetches based on the common stride. However, because the page faults are totally different between region executions for the optimized LINPACK implementations, most of these issued prefetches become useless. TODFCM and TReP show good prefetch efficiency due to their every strict prefetch issue conditions. Since the DReP assumes that application will have dynamic changing page miss area, it also shows a decent efficiency.

For the NPB-OMP benchmarks, the prefetch efficiency of Adaptive++ shows large variance. Adaptive++ shows good efficiency, 98.1% and 94.7%, for FT and IS respectively. However, less than 45% efficiency is achieved for all other benchmarks. The major reason for this is that its assumptions, namely a consecutive or alternating region repeat pattern, do not hold for the NPB-OMP benchmarks.

In contrast with Adaptive++, the TODFCM prefetch technique shows good prefetch efficiency, better than \(~90\%\), for all NPB-OMP benchmarks except CG. For CG, a \(~70\%\) efficiency is observed, due to the paging behaviour of CG being irregular due to sparse matrix access [48]. This breaks TODFCM’s assumption of a regular stride.

As TReP will only prefetch when the most recent two executions of the current
region are “highly similar”, it has achieved very good efficiency for all NPB-OMP benchmarks. TReP shows greater than 99% efficiency for IS, SP, BT and LU, and greater than 96% and 92% efficiency for FT and CG respectively.

HReP performs very well on FT, IS, and BT, with efficiency greater than 96.8%. For SP, the achieved efficiency is 94.1%, 91.9%, and 96.5% for 2 threads, 4 threads and 8 threads respectively. HReP shows reasonable efficiency on CG, from 82.8% to 92.1% for differing number of threads. HReP shows an 88.8% efficiency for LU on 2 threads, and a dramatic decrease to 56.1% efficiency on 8 threads. The reason is that the LU benchmark utilizes a lot of flush and lock synchronizations, which will result in the same page faulting multiple times in a region, breaking the repeated-stride mode assumptions.

Similar to other ReP techniques, DReP performs well for NPB-OMP benchmarks, except LU. Again, due to the excessive use of flush operations in LU, the page fault pattern breaks the assumption of DReP. This effect becomes more apparent with increasing number of threads, which results in a corresponding lower efficiency.

Comparing the different prefetch techniques, we find that Adaptive++ shows the worst efficiency for all benchmarks. TReP shows the best efficiency on the optimized LINPACK, IS, CG, SP, BT, and LU, while TODFCM shows the best efficiency on FT and the naive LINPACK implementation. DReP shows the best efficiency for IS. Nevertheless, ReP techniques and TODFCM are quite comparable with each other.

### 5.6.2.3 Prefetch Coverage

The prefetch efficiency represents how efficient a prefetch technique is. In other words, the higher efficiency represents less useless prefetches. However, it is not sufficient to compare prefetch techniques by only using efficiency. A good prefetch technique should: i) issue minimal useless prefetches, and ii) maximally reduce the number of page misses. Therefore, another metric, prefetch coverage, is introduced.

The prefetch coverage corresponds to the ‘hit rate’ of the pages that would have otherwise faulted in the absence of prefetch. A major observation from Table 5.2 is that the coverage ($N_u/N_f$) of the ReP techniques are much higher than both Adaptive++ and TODFCM.

In more detail, as mentioned in the previous section, TODFCM shows best prefetch efficiency for the naive LINPACK benchmark and FT benchmarks, and TReP shows best prefetch efficiency in the optimized LINPACK and other benchmarks. However, the coverage of TODFCM for naive LINPACK benchmark
and FT is significantly less than that of ReP techniques. Moreover, the coverage of TReP for the optimized LINPACK benchmark is negligible (1.1%), while DReP is the only technique shows good coverage (∼55%).

On average, TReP shows 39% and 30% better coverage compared to Adaptive++ and TODFCM respectively, while HReP has 50% and 41% better coverage compared to Adaptive++ and TODFCM respectively. DReP has 52% and 43% better coverage compared to Adaptive++ and TODFCM respectively.

HReP and DReP have the best coverage overall. This is largely due to HReP’s use of the repeated-stride mode, enabling it to take advantage of spatial locality when page faults, which are not predictable by temporal locality, occur. DReP takes the advantage brought by page fault record reconstruction, which break page faults into different records with common stride for each region execution, and prefetches are issued based on matching individual sub-record rather the whole region execution record.

5.6.2.4 Effective Miss Rate Reduction

The main objective of an sDSM prefetch technique is to effectively reduce the number of page misses. This is equivalent to the number of effective prefetches, which can be defined using Equation (5.8).

\[ N_e = N_u - (N_p - N_u) \]  

(5.8)

\( N_e \) stands for the number of effective prefetches, \( N_p - N_u \) stands for the number of useless prefetches. This definition reflects the (worst-case) scenario where the cost of prefetching a page is equivalent to the cost of servicing a page fault.

Subsequently, we can calculate the effective miss rate reduction via Equation (5.9), in which \( R_{mr} \) stands for the effective miss rate reduction, \( N_f \) stand for the number of total fault pages.

\[ R_{mr} = \frac{N_e}{N_f} \]  

(5.9)

The effective miss rate reduction based on total page faults (see Table 5.2) is shown in Figure 5.9.

This shows that Adaptive++ effectively reduces 24% of the page misses for the naive LINPACK benchmark \( nb = 64 \) at 2 threads and maintains it with less than 4% difference for 4 and 8 threads. It effectively reduced ∼32% of the page misses for the naive LINPACK benchmark \( nb = 16 \). ∼50% of the page misses are
effectively reduced for IS benchmark, and \(~4\%\) of the page misses are reduced for FT. However, for the other benchmarks (the optimized LINPACK, CG, SP, BT and LU), Adaptive++ shows a negative value for the effectively reduced page miss rate, which means that the program execution may slow down from using Adaptive++.

TODFCM effectively reduces \(~18\%\) and \(~28\%\) of the page misses for the naive LINPACK benchmark with \(nb = 64\) and \(nb = 16\) respectively. Less than 5%
effective page miss reduction is achieved for the optimized LINPACK benchmark. Between 12% and 40% page misses are effectively reduced for the NPB-OMP benchmarks. Although TODFCM shows the best efficiency for FT and LINPACK, and comparable efficiency for IS, SP, BT, and LU benchmarks in Table 5.2, the number of page misses effectively reduced is very much less than ReP techniques for almost all cases, except for HReP on LU with 8 threads.

TReP effectively reduces more page misses than Adaptive++ and TODFCM for all benchmarks except the optimized LINPACK benchmark, where TReP shows a negative $R_{mr}$. 58.4% to 28.0% and 89.0% to 76.0% of page misses are effectively reduced for the naive LINPACK benchmark for $nb = 64$ and $nb = 16$ respectively. ~40% to ~95% page misses are effectively reduced for NPB-OMP benchmarks.

HReP effectively reduces the page miss rate the most for most benchmarks, except for the all LINPACK benchmarks and LU with 8 threads. 65.2% to 44.5% of the page misses are effectively removed for the naive LINPACK benchmark with $nb = 64$, and 90.0% to 79.0% of the page misses are effectively removed for $nb = 16$. This observation on effectively reduced page miss rate reflects on the HReP efficiency observed for the naive LINPACK benchmarks in the above section. However, almost negligible $R_{mr}$ is achieved for the optimized LINPACK benchmark. Moreover, ~56% to ~97% of the page misses are effectively reduced for NPB-OMP benchmarks, except for LU with 8 threads. To analyze the reason, we will break down prefetches issued by HReP to find out what is the contribution from different prefetch modes in next section.

DReP poses the best $R_{mr}$ for all LINPACK benchmarks, especially for the optimized implementation. DReP is the only technique that achieves reasonable effective reduction rate. For all other benchmarks, DReP is comparable with other ReP techniques and performs better than Adaptive++ and TODFCM.

On average, TReP effectively reduces page misses 45% and 28% more than Adaptive++ and TODFCM respectively. HReP technique effectively reduces 52% and 35% more page misses than Adaptive++ and TODFCM respectively. DReP effectively reduces 54% and 37% more page misses than Adaptive++ and TODFCM respectively.

5.6.2.5 Prefetch Mode Usability Analysis for HReP

Table 5.3 shows the number of prefetches issued by different modes and using different chosen lists in the HReP technique.

In Table 5.3, Tot Pref stands for total prefetches issued for the benchmark. W-phase, R-phase, and R-stride stand for the number of prefetches issued by the
5.6 Offline Simulation

whole-phase prefetch mode, the repeated-phase mode, and the repeated-stride mode respectively (refer to Section 5.4). \( N_p \) and \( N_{bp} \) stands for the number of prefetches issued by using \( p \_list \) and \( bp \_list \) as the chosen list respectively.

Except the total prefetches, all other data is presented as a ratio of the total prefetches. For most benchmarks, the \( W \)-phase is the dominant part of the total prefetches, except the optimized LINPACK with in all cases, the naive LINPACK benchmark with \( nb = 64 \) and LU on 8 threads. \( R \)-phase only contributes less than 12% of the total prefetches for most benchmarks, except the optimized LINPACK on 8 threads, CG, LU, the naive LINPACK benchmark (\( nb = 64 \)) with 4 and 8 thread. Moreover, \( R \)-stride contributes more than 30% for FT and SP with 2 threads, and LU and LINPACK with 8 threads. \( R \)-stride dominates the prefetches for the optimized LINPACK benchmark (> 74%) for all cases. This is also explains the reason of HReP achieves poor prefetch efficiency for the optimized LINPACK benchmark in Table 5.2.

The contribution to total prefetches by \( N_p \) and \( N_{bp} \) equals that of \( R \)-phase and \( R \)-stride. As shown by Table 5.3, the ability to select the \( bp \_list \) is necessary for the FT and SP benchmarks. This is because an alternating behaviour of page misses occurs for several regions of thread 0, making \( bp \_list \) a better predictor than \( p \_list \).

LU uses a number of flush and lock synchronization operations, which will not start or end a region but does invalidate pages. This will cause the same page to fault multiple times within a region. When data is more finely partitioned (the 8 threads case), this effect becomes more distinct and more frequent. Both temporal locality and spatial locality become worse with an increasing number of threads. Particularly, with 8 threads, the temporal locality becomes worse; however, the most common stride still shows more than a 50% frequency, and so contributes 36.1% of the total prefetches. As a result of this, more useless prefetches are issued.

5.6.2.6 Flush Filtering for ReP Techniques

It has been verified by simulating HReP after removing the page misses caused by flush and lock operations from the collected page fault records of LU. This is done by inserting a filter into ReP predictor. The filter searches two most recent execution records of the current region, and find whether there are pages missed multiple times in those records. If either records have page(s) missed multiple times, these page misses are filtered out from the record before prediction is made. Consequently, the prefetch predictions are made based on the filtered records.

\(^2\)This is because that CLOMP flushes all shared memory instead of a single variable.
Chapter 5: Region-Based Prefetch Techniques

To take HReP as an example to prove this concept, simulation results for LU benchmark of HReP with filter (F-HReP) is compared with native HReP in Table 5.4.

As we removed some page misses, the number of total faults is reduced, as well as the number of prefetches issued. As shown in Table 5.4, F-HReP shows much better efficiency (∼ 99%) compared to HReP for all cases. On the contrary, the effectively reduced page miss rate ($R_{mr}$) of F-HReP is smaller HReP for 2 and 4 thread cases, and F-HReP shows better $R_{mr}$ for 8 threads case. The filter can be also applied to TReP and DReP as well.

It should be emphasized here that by using region-based techniques, pages subject to invalidations due to flushes and locks can be predicted in a similar fashion, so such improvements are realizable in a practical sDSM system.

5.7 Summary

In this chapter, we have firstly reviewed the limitations of some well-known existing page prefetch techniques and the page accessing behaviour of distinct parallel regions of a variety of benchmarks. Then, based on the above observations we have designed three region-based page prefetch (ReP) techniques, TReP, HReP and DReP, for cluster-enabled OpenMP systems. The ReP prefetch techniques are further validated via an offline simulation.

TReP utilizes the temporal locality of pages accesses from the current parallel region to prefetch all pages deemed likely to fault in its current execution. The fact that this can be done in bulk promises reduced per-page prefetch overhead, due to the reduction of network communications and the effective utilization of the bandwidth provided by high performance interconnects.

HReP combines this with the Adaptive++ techniques to prefetch a limited number of pages each time a page fault occurs. The repeated-phase mode exploits temporal locality in the faulting page (but unlike in Adaptive++, the same region’s previous history is used, rather that of previous two regions, which are not necessarily the same as that of the faulting page). The repeated-stride mode uses spatial locality within the current region (and so is the same as in Adaptive++).

DReP deploys our proposed stride-augmented run-length encoding method to encode page fault records twice, and then based on the encoded page fault records for each region executions to issue prefetches. Unlike other ReP techniques, DReP addresses both static and dynamic memory accessing patterns.

We ran offline simulations using page fault records collected by CLOMP with
5.7 Summary

two different LINPACK OpenMP benchmarks and some NPB-OMP benchmarks to evaluate our proposed prefetch techniques, comparing these with Adaptive++ and TODFCM. On average, TReP, HReP and DReP effectively reduced page misses by 54%, 62% and 64% for all benchmarks respectively. This represents an improvement of 45% and 28% (TReP), and 52% and 35% (HReP), and 54% and 37% (DReP) on Adaptive++ and TODFCM respectively. In terms of efficiency, TReP showed the best efficiency overall, followed closely by TODFCM. The main difference in effective page miss reduction was however due to coverage, with DReP achieving 2% and 14% better coverage than HReP and TReP, largely due to its capability of handling dynamic page miss patterns. Moreover, HReP achieves 12% better coverage than TReP, which is because of its exploitation of spatial locality. TReP in turn achieved 35% and 46% better coverage than TODFCM and Adaptive++ respectively.

Pages that are invalidated from locks and flushes, such as in the LU benchmark, cause page misses that cannot be avoided using prefetch techniques. These contributed to the miss rates for all methods, and, in some cases, decreased the efficiency as well. However, they can be predicted using region-based techniques and removed from consideration of prefetching.
Chapter 5: Region-Based Prefetch Techniques

Table 5.2: Simulation prefetch efficiency (E) and coverage (C) for Adaptive++, TODFCM (1 page), TReP, HReP and DReP techniques.
5.7 Summary

Table 5.3: Breakdown of prefetches issued by different prefetch modes and chosen list deployed in HReP.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Tot Pref</th>
<th>W-phase</th>
<th>R-phase</th>
<th>R-stride</th>
<th>( N_{p,t} )</th>
<th>( N_{p,p} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>nLPK ((nb = 64))</td>
<td>96562</td>
<td>82.4%</td>
<td>8.4%</td>
<td>9.2%</td>
<td>17.4%</td>
<td>0.1%</td>
</tr>
<tr>
<td>nLPK ((nb = 16))</td>
<td>406954</td>
<td>96.3%</td>
<td>1.4%</td>
<td>2.3%</td>
<td>3.7%</td>
<td>0.0%</td>
</tr>
<tr>
<td>oLPK ((nb = 64))</td>
<td>893</td>
<td>16.2%</td>
<td>0.0%</td>
<td>83.8%</td>
<td>4.0%</td>
<td>79.4%</td>
</tr>
<tr>
<td>FT</td>
<td>147653</td>
<td>68.2%</td>
<td>1.1%</td>
<td>31.7%</td>
<td>9.2%</td>
<td>22.6%</td>
</tr>
<tr>
<td>IS</td>
<td>6612</td>
<td>86.9%</td>
<td>0.1%</td>
<td>13.0%</td>
<td>13.1%</td>
<td>0.0%</td>
</tr>
<tr>
<td>CG</td>
<td>36625</td>
<td>87.5%</td>
<td>10.9%</td>
<td>1.6%</td>
<td>9.4%</td>
<td>3.0%</td>
</tr>
<tr>
<td>SP</td>
<td>1607569</td>
<td>53.3%</td>
<td>0.9%</td>
<td>45.9%</td>
<td>0.3%</td>
<td>46.3%</td>
</tr>
<tr>
<td>BT</td>
<td>970365</td>
<td>87.0%</td>
<td>0.5%</td>
<td>12.6%</td>
<td>0.2%</td>
<td>12.8%</td>
</tr>
<tr>
<td>LU</td>
<td>3084373</td>
<td>87.3%</td>
<td>11.9%</td>
<td>0.8%</td>
<td>12.5%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Tot Pref</th>
<th>W-phase</th>
<th>R-phase</th>
<th>R-stride</th>
<th>( N_{p,t} )</th>
<th>( N_{p,p} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>nLPK ((nb = 64))</td>
<td>138966</td>
<td>62.2%</td>
<td>18.1%</td>
<td>19.7%</td>
<td>36.8%</td>
<td>10.0%</td>
</tr>
<tr>
<td>nLPK ((nb = 16))</td>
<td>618882</td>
<td>95.8%</td>
<td>1.5%</td>
<td>2.8%</td>
<td>4.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>oLPK ((nb = 64))</td>
<td>2732</td>
<td>88.4%</td>
<td>0.2%</td>
<td>11.5%</td>
<td>11.6%</td>
<td>0.0%</td>
</tr>
<tr>
<td>FT</td>
<td>230226</td>
<td>80.0%</td>
<td>0.1%</td>
<td>19.9%</td>
<td>9.0%</td>
<td>11.0%</td>
</tr>
<tr>
<td>IS</td>
<td>16880</td>
<td>88.4%</td>
<td>0.2%</td>
<td>11.5%</td>
<td>11.6%</td>
<td>0.0%</td>
</tr>
<tr>
<td>CG</td>
<td>105403</td>
<td>71.5%</td>
<td>27.3%</td>
<td>1.2%</td>
<td>25.9%</td>
<td>2.6%</td>
</tr>
<tr>
<td>SP</td>
<td>3137665</td>
<td>73.4%</td>
<td>0.8%</td>
<td>25.8%</td>
<td>0.6%</td>
<td>25.9%</td>
</tr>
<tr>
<td>BT</td>
<td>1710157</td>
<td>93.8%</td>
<td>0.3%</td>
<td>5.9%</td>
<td>0.5%</td>
<td>5.7%</td>
</tr>
<tr>
<td>LU</td>
<td>4343248</td>
<td>53.7%</td>
<td>46.0%</td>
<td>0.3%</td>
<td>35.5%</td>
<td>10.9%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Tot Pref</th>
<th>W-phase</th>
<th>R-phase</th>
<th>R-stride</th>
<th>( N_{p,t} )</th>
<th>( N_{p,p} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>nLPK ((nb = 64))</td>
<td>147558</td>
<td>48.4%</td>
<td>18.3%</td>
<td>33.3%</td>
<td>45.9%</td>
<td>5.4%</td>
</tr>
<tr>
<td>nLPK ((nb = 16))</td>
<td>742836</td>
<td>91.4%</td>
<td>5.0%</td>
<td>3.6%</td>
<td>8.1%</td>
<td>0.4%</td>
</tr>
<tr>
<td>oLPK ((nb = 64))</td>
<td>6960</td>
<td>8.8%</td>
<td>16.9%</td>
<td>74.4%</td>
<td>18.9%</td>
<td>71.7%</td>
</tr>
<tr>
<td>FT</td>
<td>269677</td>
<td>84.3%</td>
<td>0.1%</td>
<td>15.6%</td>
<td>9.5%</td>
<td>6.2%</td>
</tr>
<tr>
<td>IS</td>
<td>36033</td>
<td>89.0%</td>
<td>0.4%</td>
<td>10.6%</td>
<td>11.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>CG</td>
<td>269278</td>
<td>53.5%</td>
<td>45.4%</td>
<td>1.1%</td>
<td>44.0%</td>
<td>2.4%</td>
</tr>
<tr>
<td>SP</td>
<td>5095622</td>
<td>87.0%</td>
<td>6.4%</td>
<td>6.7%</td>
<td>1.1%</td>
<td>11.9%</td>
</tr>
<tr>
<td>BT</td>
<td>2715739</td>
<td>96.7%</td>
<td>0.3%</td>
<td>2.9%</td>
<td>0.6%</td>
<td>2.7%</td>
</tr>
<tr>
<td>LU</td>
<td>7934938</td>
<td>39.0%</td>
<td>24.9%</td>
<td>36.1%</td>
<td>15.9%</td>
<td>45.0%</td>
</tr>
</tbody>
</table>

Table 5.4: Comparison of F-HReP and HReP with the LU benchmark.

<table>
<thead>
<tr>
<th>Techniques</th>
<th># of threads</th>
<th>Total Faults (x1000)</th>
<th>Prefetches (x1000)</th>
<th>Efficiency</th>
<th>( R_{inv} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-HReP</td>
<td>2</td>
<td>2865.4</td>
<td>1478.5</td>
<td>98.6%</td>
<td>43.7%</td>
</tr>
<tr>
<td>4</td>
<td>4347.3</td>
<td>2349.3</td>
<td>99.4%</td>
<td>40.7%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5803.9</td>
<td>3123.4</td>
<td>99.1%</td>
<td>33.1%</td>
<td></td>
</tr>
<tr>
<td>HReP</td>
<td>2</td>
<td>3287.7</td>
<td>3084.4</td>
<td>88.8%</td>
<td>72.8%</td>
</tr>
<tr>
<td>4</td>
<td>5700.8</td>
<td>4343.2</td>
<td>86.2%</td>
<td>55.2%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9248.8</td>
<td>7934.9</td>
<td>56.1%</td>
<td>10.4%</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6

Implementation and Evaluation

Contents

6.1 ReP Prefetch Techniques Implementation Issues . . . . . . . 112
  6.1.1 Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . 113
  6.1.2 New Region Notification . . . . . . . . . . . . . . . . . . . . . 114
  6.1.3 Record Encoding and Flush Filter enabled Decoding . . . . 116
  6.1.4 Prefetch Page Prediction . . . . . . . . . . . . . . . . . . . . . 116
  6.1.5 Prefetch Request and Event Handling . . . . . . . . . . . . . 117
  6.1.6 Page State Transition . . . . . . . . . . . . . . . . . . . . . . . 118
  6.1.7 Garbage Collection Mechanism . . . . . . . . . . . . . . . . . 119

6.2 Theoretical Performance of the ReP Enhanced CLOMP . . . 120

6.3 Performance Evaluation of the ReP Enhanced CLOMP . . . 123
  6.3.1 MCBENCH . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
  6.3.2 NPB OpenMP Benchmarks . . . . . . . . . . . . . . . . . . . . 130
  6.3.3 LINPACK Benchmarks . . . . . . . . . . . . . . . . . . . . . . 138
  6.3.4 ReP Techniques with Multiple Threads per Process . . . . . 142

6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
In this chapter we present the implementation issues for the three region-based prefetch (ReP) techniques designed in Chapter 5. Additionally, the ReP techniques, as implemented in the CLOMP runtime, are evaluated again using the NPB-OMP benchmarks, two LINPACK benchmarks, and MCBENCH.

The theoretical performance of the prefetch technique enhanced CLOMP is discussed based on the SDP critical path performance model. The measured overhead of the ReP enhanced CLOMP are compared with both the original CLOMP and the calculated theoretical for page prefetch techniques. In addition, this overhead is broken down to investigate introduced cost of deploying the ReP techniques.

6.1 ReP Prefetch Techniques Implementation Issues

The CLOMP runtime consists of three components. The first is Intel OpenMP compilers and supporting runtime. The second is the sDSM layer, called TMK, which creates and maintains the global virtual shared memory system. The third is the communication layer, called CAL, which is invoked by TMK to perform inter-process communications. The structure of CLOMP is shown in Figure 6.1.

![Figure 6.1: Intel Cluster OpenMP runtime structure.](image)

The TReP, DReP and HReP techniques are implemented into the sDSM layer (TMK) of the CLOMP runtime. At the beginning of each execution of regions, named as a region-execution, the recorded page faults for the last region-execution will be encoded and stored in a linked list. Each node of the linked list represents a region-execution. The ReP techniques will predict and issue prefetches immediately after the encoding process. One of the ReP prediction functions is invoked to predict page misses for the upcoming region-execution. The predicted pages are then requested for prefetch, and at the same time, the sDSM daemon thread may be invoked to service the incoming prefetch requests. After pages/diffs have been
6.1 ReP Prefetch Techniques Implementation Issues

prefetched, all accesses to the prefetched pages will be recorded as page misses for future predictions, except when no data transfer is required for these misses. The implementation of RePs raises issues in relation to:

- data structures,
- new region notification,
- record encoding and flush filter enabled decoding,
- prefetch page prediction,
- prefetch request and event handling,
- page state transition,
- garbage collection mechanism.

The details of each aspect will be discussed in the rest of this section.

6.1.1 Data Structures

In order to reduce memory usage due to storing all experienced page faults, the *stride-augmented run-length encoding* (sRLE) method is used as a common strategy to encode and segment the original page fault record on a per region-execution basis.

At the beginning of each region-execution, the ReP techniques sort the page fault record of the previous region-execution, and then use the sRLE method to encode the sorted page fault record twice, as shown in Figure 5.7.

The encoded page fault records contain a list of second level sRLE encoded entries and each second level encoded entry in turn contains a first level encode entry. The detailed data structure is shown in Figure 6.2.

As we have already illustrated in Figure 5.7 (b), the struct `l1_encode_cols` refers to the first level encoded page record data structure, contains the starting page ID, the common stride and the run length. The struct `l2_encode_cols` is the data structure for the second level encoded entry consisting of a starting first level entry, the common stride between the consecutive first level entries, and the run length of the first level entries. Each region-execution may contain a number of second level encoded entries. Therefore, the encoded page fault struct `l2_encode_region` contains a linked list of second level encoded entries.

Consequently, an additional decoding process will be required for TReP and HReP to find out the page fault list similarity and issuing prefetches.
Chapter 6: Implementation and Evaluation

```c
// first level encoding struct for a region
typedef struct l1_encode_cols {
    unsigned int start_page;
    int stride;
    int run_len;
} l1_en_cols;

// second level encoding structure for a region
typedef struct l2_encode_cols {
    l1_en_cols l1_en_col;
    int stride;
    int run_len;
    struct l2_encode_cols *next;
} l2_en_cols;

/* stride-augmented run-length encoded page fault record of a region-execution */
typedef struct l2_encode_region {
    int region_id;
    int is_parallel;
    l2_en_cols *head;
    l2_en_cols *curr;
    int num_cols;
    int total_faults;
    struct l2_encode_region *next;
    struct l2_encode_region *prev;
} l2_en_reg_t;

// list of page fault record of all region-executions
typedef struct rec_regs {
    l2_en_reg_t *first_region;
    l2_en_reg_t *curr_region;
    pthread_mutex_t lock;
} rec_regions_t;
```

**Figure 6.2:** Data structure for stride-augmented run-length encoded page fault records.

Figure 6.3 shows the linked list used to keep the statistical prefetch information including the number prefetches issued for this region and the number of useful prefetches.

### 6.1.2 New Region Notification

An API is designed to allow applications to directly notify the sDSM layer (TMK) about the start of a region and whether it is a parallel region or not by accepting a single integer parameter. It has two functionalities. First, it sets a flag in TMK layer to indicate the start of a new region. Second, it retrieves the region ID for later use. This interactive function needs to be invoked by the user program immediately after the start of each sequential and parallel region. Figure 6.4 shows
6.1 ReP Prefetch Techniques Implementation Issues

typedef struct predict_page_record {
    int is_parallel;
    int region_id;
    int num_pref_pages;
    int useful;
    # if REGION_PREF_STATS
    int num_faults; /* record for the number of faults 
        (including prefetched pages) */
    # endif
    int prefetched; // any prefetches issued?
    int completed; // prefetch for current region completed?
    struct predict_page_record *next;
} page_predict_record_t; //one entry per region

typedef struct predict_region {
    page_predict_record_t *first_region;
    page_predict_record_t *curr_region;
    pthread_mutex_t lock;
} pred_region_t;

Figure 6.3: ReP prefetch record data structure.

an example of the interactive user interface in the NPB-OMP BT benchmark.

BT/x_solve.f:

......

!$omp parallel default(shared) shared(isize)
!$omp private(i,j,k,m,n)
    call KMP_USER_NOTIFY_NEW_REGION(1)
!$omp do
    do k = 1, grid_points(3)-2

......

Figure 6.4: User interactive interface of new region notification.

There are two major reasons that motivated this new design. The first is that since Intel compilers are closed source, there is no sufficient information about how OpenMP directives are compiled onto the CLOMP runtime libraries. Additionally, no intermediate code is produced with the current Intel C/FORTRAN compilers, which results in no information about the starting point of a region. Secondly, the OpenMP global synchronization operation (barrier) is implemented using a number of reduce operations in TMK, which means that there is not a unique TMK global synchronization function where we can explicitly distinguish a start/end point of a region.¹

¹It would be simple in principle to modify the compiler to achieve this functionality, if the source is available.
6.1.3 Record Encoding and Flush Filter enabled Decoding

Once a new region notification is received, the recorded page fault IDs for the previous region-executions are sorted and then encoded using the proposed stride-augmented run-length encoding method. The encoded page fault record is stored in a linked list of encoded history records as illustrated in Figure 6.2.

The ReP techniques were designed based on regions distinguished by implicit and explicit barriers. Non-global synchronization operations occurred in a region can cause two problems. The first is that a page can be missed twice within a region. According to the simulation results presented in Section 5.6, the prefetch efficiency of RePs are affected consequently. The second is that prefetching a page that is requested during a non-global synchronization operation can cause incorrect numerical results.

To solve the first problem, the flushed page filter (mentioned in Section 5.6.2.6) is implemented into the decoding process of the $p_{list}$ and $bp_{list}$ entries for TReP and HReP, and the predicted entry for DReP. During these processes, all duplicated page IDs are removed from the record. The second problem can be solved by identifying individual pages that were flushed in the previous region-execution.

6.1.4 Prefetch Page Prediction

After the page faults of the previous region-execution are encoded and stored, the ReP techniques are immediately invoked to predict page misses and issue prefetches.

The TReP and DReP techniques predict and issue prefetches once at the beginning of each region-execution. On the other hand, HReP determines which mode will be used for the current region. If the whole-phase mode is selected, the behaviour of HReP is the same as TReP, i.e. it predicts and prefetches only once. Otherwise, HReP will predict and/or issue prefetches every time a page miss occurs in the current region-execution.

The ReP techniques predict and issue prefetches based on the flush filtered records. Moreover, to eliminate useless prefetches and maintain data dependencies, a page state check is made to guarantee that only pages in the invalid and empty states will be prefetched.
6.1 ReP Prefetch Techniques Implementation Issues

6.1.5 Prefetch Request and Event Handling

As the ReP techniques issue many page prefetches each time they are invoked, the number of pages/diffs can be large. Therefore, pages to be prefetched are broken into chunks and prefetched on a per chunk-basis. The prefetch requests are sent out by the OpenMP working threads and responded to by a sDSM daemon thread of a remote process (see Section 2.2.3 for types of threads used in CLOMP).

The original communication layer (CAL) of CLOMP reserves 32 bytes for a header message which can only contain 8 integer page IDs. When a prefetch is requested, the aggregated diffs of 8 pages is only in the range of 128 to 32864 bytes.\(^2\) To effectively leverage the available bandwidth provided by GigE and IB, the size of aggregated diffs needs to be 262144 bytes on average, which is equivalent to 128 pages. To achieve this goal, we modified the data structure of the header message to accommodate 128 pages.

Additionally, due to the homeless model used by CLOMP, each process might need to request prefetches from all other processes [96]. Therefore, to minimize network latency, prefetches are requested amongst the processes in a round robin fashion. The details of this implementation is shown in Fig. 6.5.

As shown in Figure 6.5, each process requests prefetches from the daemon thread of its right neighbour, and then move to next right process until all other processes have been requested.

Figure 6.5 shows that the requesting process will issue the prefetch request to the sDSM daemon of the first process to its right, followed by the second process to its right, and so on until all processes have received the request. The sDSM daemon thread of each process handles the incoming prefetch requests and respond with the requested pages/diffs. Also note that, as diffs are generated on a per page-basis, servicing a page miss may resolve multiple diffs. Therefore, in the rest of this chapter, we only account for the number of prefetched pages rather than prefetched diffs. Consequently, this number is used to calculate prefetch efficiency and coverage.

The prefetch initiator will update the page state of prefetched pages after received them from the responders. The details of the page state transition is described in the next section.

\(^2\)A diff consists 4 bytes length, 8 bytes address offset and at the actual diff in 4 bytes granularity.
6.1.6 Page State Transition

In order to keep track of all prefetched pages, two new page states are introduced, prefetched_diff and prefetched_page. The state prefetched_diff implies that the diff(s) for the page has been prefetched, while the state prefetched_page implies that the page has been prefetched. Unlike the old page state transition machine (Figure 2.8), the page state transitions from the invalid and the empty states to the prefetched_diff and the prefetched_page states when the diffs and the page has been prefetched respectively, as shown in Figure 6.6. The prefetched_page state can transition to the empty state after a garbage collection operation. All out-going transitions from the invalid and empty states can be directly applied to the prefetched_diff and prefetched_page states.

Pages in the prefetched_diff and the prefetched_page states are protected by mprotect as well. After a region begins executing, all page faults caused by accessing pages in the invalid, empty, prefetched_diff, and prefetched_page state
6.1 ReP Prefetch Techniques Implementation Issues

![Diagram of page state machine]

Figure 6.6: New page state machine after introduced Prefetched_diff and Prefetched_page states.

are recorded.

Two page sub-states pref_diff_locked and pref_page_locked are used to avoid redundant page prefetches when multiple OpenMP working threads are deployed.

6.1.7 Garbage Collection Mechanism

As CLOMP inherited the garbage collection mechanism from TreadMarks [5], we also adapt our prefetch implementation to this garbage collection mechanism. During the garbage collection, each page in the prefetched_diff state is examined by each process to determine whether the page is most recently written by the process. If this is the case, all other modifications will be transferred to this process and the page state will be changed to valid. Otherwise, the prefetched page will be discarded, and the page state will be changed to empty. For the page in the prefetched_page state, the prefetched page will be directly discarded and the page state will be reset to empty.
6.2 Theoretical Performance of the ReP Enhanced CLOMP

Corresponding to the breakdown the elapsed time for the NPB-OMP benchmarks represented in Section 3.2.5, the SDP critical path model can be reformulated as follows:

\[
T_{\text{tot}}(p)^{\text{crit}} = \frac{T_{\text{tot}}(1)}{p} + T_{\text{segv}}
\]

\[
T_{\text{segv}} = T_w + T_f
\]

\[
T_w = \sum_r \max_{i=0}^{p-1} N^w_r C^w
\]

\[
T_f = \sum_r \max_{i=0}^{p-1} N^f_r C^f
\]

(6.1)

where \(T_{\text{segv}}\) denotes the time spent on memory consistency (page fault servicing) which consists of two components, the time spent on WRITE page faults \((T_w)\) and that spent on FETCH page faults \((T_f)\). As defined in Section 4.2, \(N^w\) and \(N^f\) are used to denote the total number of WRITE and FETCH page faults respectively. The corresponding costs are represented as \(C^w\) and \(C^f\).

Since the local servicing cost of FETCH page fault is as the same as \(C^w\), and the communication cost \((C^f)\) consists of requesting and receiving diff/page updates. \(C^f\) can be extended as:

\[
C^f = C^w + C_{\text{req}} + C_{\text{rsp}}
\]

(6.2)

where \(C_{\text{req}}\) and \(C_{\text{rsp}}\) represent the round-trip communication cost of requesting and receiving diff/page updates.

An effective prefetch technique can improve performance by reducing the page fault servicing overhead, which is \(T_{\text{segv}}\) in Equation (6.1), along the critical path. The communication overhead, \(C_{\text{req}}\) and \(C_{\text{rsp}}\), overhead caused by FETCH page fault \((T_f)\), can be reduced through aggregating both diffs and/or pages requests and the actual data, and then transferring them in large chunk of data (128 pages) to achieve the higher interconnect bandwidth.

In theory, the best prefetch technique has the following feature:

- it prefetches all pages missed;
- it has 100% prefetch efficiency and coverage;
6.2 Theoretical Performance of the ReP Enhanced CLOMP

Table 6.1: Bandwidth and latency measured by the communication layer (CAL) of CLOMP.

<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>XE GigE</th>
<th>XE DDR IB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bandwidth (MB/s)</td>
<td>Latency (µs)</td>
</tr>
<tr>
<td>512</td>
<td>5.2</td>
<td>92.6</td>
</tr>
<tr>
<td>2048</td>
<td>18.6</td>
<td>105.2</td>
</tr>
<tr>
<td>4096</td>
<td>33.0</td>
<td>118.6</td>
</tr>
<tr>
<td>262144</td>
<td>89.8</td>
<td>2783.3</td>
</tr>
<tr>
<td>524288</td>
<td>93.0</td>
<td>5377.2</td>
</tr>
</tbody>
</table>

- it has no additional overhead for predicting and prefetch issuing;
- it can fully utilize network bandwidth.

Under these assumptions, the $C_{req}$ and $C_{rsp}$ in Equation (6.2) can be reformulated as follows to calculate the theoretical lowest cost.

\[
C_{req} = \frac{L_{req}}{\alpha}, \\
C_{rsp} = \frac{s_{rsp}B_p}{B_p}
\]  (6.3)

where $\alpha$ stands for the number of pages prefetched at each time, which is 128 as limited by the communication layer of CLOMP. $B_p$ is the peak bandwidth that can be achieved by the communication layer (CAL) of CLOMP. $L_{req}$ denote the CAL measured latency for the size of prefetch request message (520 bytes). $s_{rsp}$ denotes the average size of a diff (2054 bytes) and a page (4096 bytes). Some measured values of $L_{req}$ and $B_p$, which is used in this thesis, are shown in Table 6.1, and please refer to Appendix E for more performance data of the CAL layer.

After replacing $C_f$ in Equation (6.1) using Equation (6.2 and 6.3), the theoretical peak performance, in terms of elapsed time, of the ReP enhanced CLOMP can be derived based on the SDP critical path model as follows:

\[
Tot(p)^{crit}_{pref} = \frac{Tot(1)}{p} + T_{segv} \\
T_{segv} = T_w + T_f \\
T_w = \sum_r \max_{i=0}^{p-1} N_{r,i}C^w \\
T_f = \sum_r \max_{i=0}^{p-1} N_{r,i}(C^w + \frac{L_{req}}{\alpha} + \frac{s_{rsp}}{B_p})
\]  (6.4)

where $Tot(p)^{crit}_{pref}$ represents the theoretical elapsed time for $p$ processes that assumes there is no overlap between computation and prefetch communication.
In Equation (6.4), $\frac{N_f L_{req}}{\alpha}$ represents the theoretical lowest cost of requesting prefetches, and $\frac{N_f S_{rsp}}{B_p}$ represents the theoretical lowest cost of transferring diffs/pages transfer cost.

Due to the load balancing assumption made by the SDP model, it is not entirely accurate to say that the computation time is $\frac{\text{Tot}(1) p}{p}$. This will affect the accuracy of theoretical peak estimation of elapsed time ($\text{Tot}(p)_{\text{pref}}^{\text{crit}}$). Therefore, as shown in Equation (6.5), Equation (6.4) is further refined to address this problem.

$$\text{Tot}(p)_{\text{pref}}^{\text{crit}} = T_{\text{comp}} + T_{\text{segv}}$$

$$= T_{\text{comp}} + T_{\text{segv, local}} + T_{\text{comm}}$$

$$= T_{\text{local}} + T_{\text{comm}}$$

$$T_{\text{segv, local}} = T_w + \sum_{r=0}^{P-1} \max_p N_f L_{req}^{rw}$$

$$T_{\text{comm}} = \frac{N_f L_{req}}{\alpha} + \frac{s_{\text{total}}}{B_p}$$

$$s_{\text{total}} = \frac{N_f S_{rsp}}{B_p}$$

(6.5)

where $T_{\text{segv}}$ is decomposed into $T_{\text{segv, local}}$ and $T_{\text{comm}}$, and the computation time is denoted by $T_{\text{comp}}$ rather than $\frac{\text{Tot}(1) p}{p}$. $T_{\text{comp}}$ and $T_{\text{segv, local}}$ form the overall local operation cost ($T_{\text{local}}$).

Since the ReP techniques use mprotect to protect the prefetched pages, the local page fault cost still applies to them, and only the $T_{\text{comm}}$ can be reduced. With the above modification, $T_{\text{comp}}$ and $T_{\text{segv, local}}$ are measured at runtime to eliminate the inaccurate estimation given by $\frac{\text{Tot}(1) p}{p}$, and $T_{\text{comm}}$ is calculated with peak network performance and total number of FETCH page faults. Finally, the theoretical elapsed time can be estimated by combining these different components.

Hence, the theoretical speedup due to the application of by ReP techniques ($S(p)$) for $p$ processes is given by:

$$S(p) = \frac{\text{Tot}(1)}{\text{Tot}(p)_{\text{pref}}^{\text{crit}}}$$

(6.6)

in which, $\text{Tot}(1)$ is the elapsed time measured when single OpenMP is deployed.

The theoretical speedup given by Equation (6.6) will be used to evaluate the ReP enhanced CLOMP.
6.3 Performance Evaluation of the ReP Enhanced CLOMP

The ReP enhanced CLOMP is evaluated in this section. Firstly, the micro benchmark, MCBENCH, is used to measure the memory consistency cost of ReP enhanced CLOMP that is compared with the original CLOMP. Then, some benchmarks from NPB-OMP suite with different problem sizes and the two LINPACK OpenMP benchmarks analyzed in Section 5.1.1 are utilized to evaluate the ReP enhanced CLOMP.

The benchmarks are run on XE cluster at NCI NF, which is connected via both GigE and IB. Due to the limited allocated number of physical nodes (refer to Section 3.1 for details), 8 computing nodes are used in the performance evaluation. Additionally, as we described in Section 6.1.2, all benchmarks are modified using the user interactive interface to notify TMK layer the start of a new region.

6.3.1 MCBENCH

The major feature of the MCBENCH is that it measures the total cost difference of read/write a shared and a private array. Therefore, this difference effectively represents the effect of utilizing ReP techniques in CLOMP runtime. Similar to Section 3.3, three shared array sizes (64KB, 4MB and 8MB) and three typical chunk sizes (4B, 2KB and 4KB) are used. The measured memory consistency cost of ReP enhanced CLOMP is compared the original CLOMP. Again, due to the memory consistency is only maintain between processes, only single thread is deployed in each compute node (process).

Table 6.2: ReP techniques prefetch efficiency and coverage for MCBNECH with 4MB array.

<table>
<thead>
<tr>
<th>Chunk Sizes</th>
<th>nproc</th>
<th>Faults $(N_f)$ (×1000)</th>
<th>TReP $N_u/N_f$ (%)</th>
<th>E (%)</th>
<th>HReP $N_u/N_f$ (%)</th>
<th>E (%)</th>
<th>DRReP $N_u/N_f$ (%)</th>
<th>E (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4B</td>
<td>2</td>
<td>168</td>
<td>90.1</td>
<td>97.5</td>
<td>96.2</td>
<td>100.0</td>
<td>90.1</td>
<td>97.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>340</td>
<td>95.1</td>
<td>100.0</td>
<td>96.2</td>
<td>99.9</td>
<td>95.1</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>684</td>
<td>94.6</td>
<td>100.0</td>
<td>95.6</td>
<td>99.9</td>
<td>94.6</td>
<td>100.0</td>
</tr>
<tr>
<td>2KB</td>
<td>2</td>
<td>168</td>
<td>95.6</td>
<td>100.0</td>
<td>96.2</td>
<td>99.9</td>
<td>95.6</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>177</td>
<td>26.6</td>
<td>59.0</td>
<td>18.1</td>
<td>36.2</td>
<td>44.5</td>
<td>49.3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>195</td>
<td>5.8</td>
<td>14.0</td>
<td>10.1</td>
<td>20.5</td>
<td>39.4</td>
<td>48.0</td>
</tr>
<tr>
<td>4KB</td>
<td>2</td>
<td>86</td>
<td>0.0</td>
<td>n/a</td>
<td>92.8</td>
<td>99.5</td>
<td>92.3</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>95</td>
<td>0.0</td>
<td>n/a</td>
<td>9.1</td>
<td>76.7</td>
<td>83.8</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>113</td>
<td>0.0</td>
<td>n/a</td>
<td>9.2</td>
<td>46.1</td>
<td>70.4</td>
<td>99.0</td>
</tr>
</tbody>
</table>
The measure prefetch efficiency and coverage of ReP techniques are shown in Table 6.2 with the total page fault represented in thousands ($\times 1000$) and the coverage denoted as $N_u/N_f$. Since difference array sizes do not affect the prefetch performance, a 4MB array are used as an example in the table.

6.3.1.1 4 Bytes Chunk

In this section, the comparison of MCBENCH evaluation results are made for the 4B chunk size with different array sizes and over both GigE and DDR IB, as shown in Figure 6.7, in which (a), (b) and (c) representing 64kB, 4MB and 8MB array sizes respectively.

For the 64KB array, the memory consistency cost of the original CLOMP scales linearly with increasing number of threads. It reaches around 24$ms$ and 11$ms$ at 8 threads on GigE and DDR IB respectively. In contrast, all ReP techniques have significantly reduced this cost to around 5$ms$ and 3.5$ms$ at 8 threads on GigE and DDR IB respectively. In other words, any ReP technique can reduce the memory consistency cost of the original CLOMP to $\sim$ 21% and $\sim$ 32% for the GigE and DDR IB networks respectively.

Similar trends have been observed for 4MB and 8MB arrays with a more significant improvement which also reflects the ReP prefetch efficiency and coverage listed in Table 6.2. For 4MB array, ReP techniques reduced the memory consistency cost to $\sim$ 12% and $\sim$ 18% for the GigE and DDR IB networks respectively. Moreover, for 8MB array, these figures are further reduced to $\sim$ 11% and $\sim$ 17% on GigE and DDR IB networks respectively.

All ReP techniques show very good improvement in the memory consistency cost for the 4 bytes chunk case. As we discussed in Section 3.3, when 4 bytes chunk is used, all shared pages are read/written by all processes and in the next iteration (region-execution) will be missed and requested by each process from all other processes to maintain the memory consistency. All region-executions experience the same page faults. Therefore, TReP, DReP and HReP are all able to predict well and achieve good performance. To be noted, the whole-phase mode is the only mode used by HReP for 4 bytes chunk case.

Table 6.3 shows the maximum number of messages sent by each processes for the three array sizes, 64KB, 4MB and 8MB, in thousands ($\times 1000$). The ratio of the message sent by the RePs enhanced CLOMP to that of the original CLOMP is $\sim$ 54% for 64KB array. This ratio dropped to around 4% to 11% for 4MB and 8MB arrays because that the larger working array reduced the portion of page faults contributed by other data (“noise”). This observation also explains the reason of
6.3 Performance Evaluation of the ReP Enhanced CLOMP

![Graphs showing the performance evaluation of ReP Enhanced CLOMP vs Original CLOMP with different chunk sizes and array sizes over GigE and IB networks.](image)

**Figure 6.7:** RePs VS. Original CLOMP: MCBENCH with 4B chunk size over both the GigE and IB networks. (a) 64KB array size, (b) 4MB array size, (c) 8MB chunk size.

that ReP techniques reduce more memory consistency cost for the larger array.
Table 6.3: Message transfer counts (×1000) comparison between RePs enhanced CLOMP and the original CLOMP for MCBENCH with 4B chunk

<table>
<thead>
<tr>
<th>nprocs</th>
<th>64KB array</th>
<th>4MB array</th>
<th>8MB array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>TReP</td>
<td>HReP</td>
</tr>
<tr>
<td>2</td>
<td>5.1</td>
<td>2.7</td>
<td>2.7</td>
</tr>
<tr>
<td>4</td>
<td>15.2</td>
<td>8.1</td>
<td>8.0</td>
</tr>
<tr>
<td>8</td>
<td>40.0</td>
<td>22.6</td>
<td>22.5</td>
</tr>
</tbody>
</table>

6.3.1.2 2048 Bytes Chunk

When a 2048 bytes (2KB) chunk size is used, each page (4096 bytes) will be read/written by 2 processes, which results in a change in the memory access pattern of MCBENCH with increasing number of processes.

When 2 processes are deployed, all shared pages will be read/written by all processes in each iteration. Therefore, each process will have page faults at all shared pages, and must then requests all shared pages from another process. This is the same as the 4B chunk case. In Table 6.4, the number of messages sent by the ReP techniques are also the same as what is observed for 4B chunk cases. All ReP techniques are able to predict well for this case. Figure 6.8 shows that on both GigE and DDR IB, all ReP techniques reduced the memory consistency cost significantly. Similar to the 4B chunk case analyzed in previous section, ReP performs better for the larger arrays.

Figure 3.7 shows the case of 4 processes. Each process misses on the pages with either even or odd IDs in each iteration. Moreover, the page fault pattern is in the sequence of “···–even–odd–odd–even–···” over different iterations. For such a page miss pattern, the ReP techniques perform quite differently. The details of RePs prefetch efficiency and coverage for 4MB array are shown in Table 6.2.

Since TReP deploys the very strict predict conditions, it only issues prefetches for the every second iteration, in other words, when the page faults in the two previous executions have the same parity. These prefetched pages will be accessed after two iterations. However, if the prefetched pages are not accessed in the consecutive region-executions, they may be discarded by the garbage collection operations. This results in 59% efficiency and 26% coverage. HReP adapts the whole-phase, repeated-phase, and repeated-stride modes. Therefore, in every 6 iteration, it issues prefetches for three iteration and correctly predicts once, which results in 18.1% coverage and 36% efficiency. Since DReP calculates the movements of page fault area, it performs slightly better than TReP and HReP in both coverage and efficiency with ~50% efficiency and coverage.
6.3 Performance Evaluation of the ReP Enhanced CLOMP

![Graphs showing performance comparison between RePs and Original CLOMP for different array sizes and networks.](image)

**Figure 6.8**: RePs VS. Original CLOMP: MCBENCH with 2048 bytes chunk size over both the GigE and IB networks. (a) 64KB array size, (b) 4MB array size, (c) 8MB chunk size.

As the efficiency and coverage for RePs are not good on 4 processes, the number of messages sent are greater than that of the original CLOMP, especially for the 4MB and 8MB arrays. The ReP prefetch performance reflects the memory consistency cost as shown in Figure 6.8. RePs enhanced CLOMP performs similar to the original CLOMP with some variation between the GigE and IB.
Table 6.4: Message transfer counts (×1000) comparison between RePs enhanced CLOMP and the original CLOMP for MCBENCH with 2KB chunk

<table>
<thead>
<tr>
<th>nprocs</th>
<th>64KB array</th>
<th></th>
<th></th>
<th>4MB array</th>
<th></th>
<th></th>
<th></th>
<th>8MB array</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig TReP</td>
<td>HReP</td>
<td>DReP</td>
<td>Orig TReP</td>
<td>HReP</td>
<td>DReP</td>
<td>Orig TReP</td>
<td>HReP</td>
<td>DReP</td>
<td>Orig TReP</td>
<td>HReP</td>
</tr>
<tr>
<td>2</td>
<td>5.1 2.7</td>
<td>2.7</td>
<td>2.7</td>
<td>168.4 18.1</td>
<td>18.1</td>
<td>18.1</td>
<td>334.2 35.8</td>
<td>35.8</td>
<td>35.8</td>
<td>324.8 351.4</td>
<td>351.4</td>
</tr>
<tr>
<td>4</td>
<td>9.4 9.2</td>
<td>9.5</td>
<td>9.4</td>
<td>149.7 148.1</td>
<td>156.9</td>
<td>153.0</td>
<td>226.4 215.2</td>
<td>215.2</td>
<td>215.2</td>
<td>226.4 215.2</td>
<td>215.2</td>
</tr>
<tr>
<td>8</td>
<td>23.0 23.0</td>
<td>22.9</td>
<td>23.0</td>
<td>117.1 118.6</td>
<td>120.6</td>
<td>115.2</td>
<td>226.4 215.2</td>
<td>215.2</td>
<td>215.2</td>
<td>226.4 215.2</td>
<td>215.2</td>
</tr>
</tbody>
</table>

interconnects.

When the number of processes is increased to 8, the page fault pattern for each process is “··· mod(ID)==0 - mod(ID)==1 - mod(ID)==2 - mod(ID)==3 - mod(ID)==3 - mod(ID)==3 - mod(ID)==2 - mod(ID)==0 - mod(ID)==1 - mod(ID)==2 - mod(ID)==3 - mod(ID)==3 - mod(ID)==3 - mod(ID)==2 - mod(ID)==0”. Both the prefetch efficiency and coverage dropped dramatically for TReP and HReP techniques. DReP largely maintains its prefetch efficiency and coverage. This is reflected in Table 6.4 and Figure 6.8, for most cases, TReP and HReP have not reduced the memory consistency costs much, while DReP shows a slight improvement.

When 2048 bytes chunk size is applied, on $p$ processes, a $\text{mod}(p/2)$ page fault pattern will be observed for MCBENCH. In general, both TReP and HReP do not perform well for this kind of page fault pattern, while DReP improves the performance.

6.3.1.3 4096 Bytes Chunk

When a 4096 bytes (4KB) chunk is used, each shared page contains only 1 chunk. As a result, the modular type ($\text{mod}(p)$) page fault pattern happens. For example, for 2 processes, a “···-even-odd-even-odd···” page fault pattern is exposed, while for 4 processes, a “···-mod(ID)==0 - mod(ID)==1 - mod(ID)==2 - mod(ID)==3 - mod(ID)==3 - mod(ID)==2 - mod(ID)==0” page fault pattern occurs.

TReP, which relies on the “high similarity” between the two most recent previous execution records of the same region, is not able to make any predictions for the above modular type page fault patterns. This is shown by the 0 coverage and n/a efficiency in Table 6.2. Since TReP does not issue any prefetches, the number of messages sent associated with TReP enhanced CLOMP is almost the same as that of the original CLOMP, as shown in Table 6.5. It only introduces a small software overhead, while this overhead becomes more negligible with larger array sizes. On both the GigE and IB networks, its performance aligns with the original CLOMP in Figure 6.9.

The repeated-stride mode of HReP predicts well for 4KB chunk case on 2
### 6.3 Performance Evaluation of the ReP Enhanced CLOMP

Table 6.5: Message transfer counts ($\times 1000$) comparison between RePs enhanced CLOMP and the original CLOMP for MCBENCH with 4KB chunk

<table>
<thead>
<tr>
<th>nprocs</th>
<th>64KB array</th>
<th>4MB array</th>
<th>8MB array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>TReP</td>
<td>HReP</td>
</tr>
<tr>
<td>2</td>
<td>3.8</td>
<td>3.8</td>
<td>2.7</td>
</tr>
<tr>
<td>4</td>
<td>8.1</td>
<td>8.1</td>
<td>8.1</td>
</tr>
<tr>
<td>8</td>
<td>22.4</td>
<td>22.4</td>
<td>22.4</td>
</tr>
</tbody>
</table>

processes with 99.5% prefetch efficiency and 92.8% coverage, as shown in Table 6.2. Consequently, as shown in Table 6.5, the number of message sent by HReP is significantly less than the original CLOMP. A $\sim 50\%$ memory consistency cost reduction is achieved by HReP over both GigE and IB for 4MB and 8MB arrays in Figure 6.9. However, HReP increases memory consistency costs for a 64KB array on both GigE and IB networks. It is because, when repeated-stride or repeated-phase modes are used, the overhead introduced by HReP is relatively high. This overhead becomes more obvious when the array size is small. The total page fault servicing cost of MCBENCH with 64KB is $\sim 0.36$ and $\sim 0.13$ seconds on GigE and IB respectively, where HReP adds $\sim 0.18$ seconds software overhead. As such, for 4 and 8 processes, none of HReP prefetch modes can handle well, resulting in very low prefetch coverage and a decreasing efficiency, which again is reflected in Figure 6.9. HReP shows larger or equal memory consistency cost as that of the original CLOMP.

As DReP technique is able to predict for the program with dynamic page fault area, DReP performs very well when 4KB chunk size is deployed. On the average, DReP reduces $\sim 40\%$ and $\sim 33\%$ of the memory consistency cost of the original CLOMP on the GigE and DDR IB networks respectively.

Moreover, prefetch efficiency and coverage of DReP are very high, and it effectively reduces memory consistency cost for most cases except 8 threads over the IB network as shown in Figure 6.9. There are number of reasons. Firstly, according to Table 6.2 and 6.5, as the prefetch coverage is reduced along with an increasing number of processes, which results in a reduced number of message reductions and higher memory consistency cost. Secondly, as each process is contiguously accessing shared pages, the diffs will have the maximum size (4108 bytes), which is around 70% peak bandwidth of DDR IB network [16]. Therefore, the benefit of larger bandwidth brought by prefetch aggregation is not as high as what it was for GigE network. Moreover, for the 64KB array, the software overhead introduced by DReP dominates the memory consistency cost for such case which results in a little higher memory consistency cost.
Figure 6.9: RePs VS. Original CLOMP: MCBENCH with 4KB chunk size over both the GigE and IB networks. (a) 64KB array size, (b) 4MB array size, (c) 8MB chunk size.

6.3.2 NPB OpenMP Benchmarks

As it was shown in Section 5.6, ReP techniques performs well in general for NPB benchmarks. In this section, we use the NPB OpenMP BT benchmark as a model to evaluate the performance of RePs enhanced CLOMP and validate our previous
6.3 Performance Evaluation of the ReP Enhanced CLOMP

...simulation results. Then, the comparison of page fault handling costs \( T_{segv} \) between the enhanced and original CLOMP runtimes are presented for the other NPB-OMP benchmarks.

Three different problem sizes are used, class A, B and C representing small, medium and large data set respectively. The evaluation results are compared to the corresponding original CLOMP performance presented in Section 3.2.1. The performance of ReP enhanced CLOMP is also compared with the theoretical speedup which is evaluated by Equations (6.5) and (6.6). The required parameters \( T_{segv,local} \) and \( N_f^{total} \) are measured at runtime, and shown in Appendix B. Since the memory consistency only needs to be maintained among processes, only one OpenMP thread is deployed on each compute node (process).

The page fault handling costs of the other NPB-OMP benchmarks are summarised and compared to the theoretical lowest cost at the end of this section.

6.3.2.1 BT Benchmark Elapsed Time Breakdown

The measured elapsed time of the BT benchmark on ReP enhanced CLOMP is broken down in terms of the computation and page fault handling costs. It is compared to the the original CLOMP and the calculated theoretical lowest cost.

In Table 6.6, the comparison is made for each individual problem sizes and different network interconnects. The computational time and page fault handling time for the original CLOMP \( (T_{segv}^{orig}) \) are represented in seconds \( (sec) \). The theoretical page fault handling costs \( (T_{segv}) \) (calculated by Equation 6.5) are represented as a percentage of the reduced cost, e.g. \( \frac{T_{segv}^{orig} - T_{segv}}{T_{segv}^{orig}} \). The percentages of RePs in the table are calculated similarly using the measured value \( T_{segv} \).

According to Table 6.6, ReP techniques have significantly reduced the page fault handling overhead compared to the original CLOMP. On average, a \(~62\%\) and \(~40\%\) overhead reduction rate is achieved by ReP techniques for the GigE and DDR InfiniBand network respectively. This number is very close the theoretical overhead reduction rate with \(~9\%\) difference on GigE, and a \(20\%\) difference on the DDR IB network.

Corresponding to the bandwidth and latency measured for the communication layer (CAL) of CLOMP in Table 6.1, the peak bandwidth is \(~5.1\) and \(~4.1\) times the bandwidth achieved when message size is the average diff size (2054 bytes) on GigE and IB respectively. In turn, this results in the \(~10\%\) lower theoretical overhead reduction rate on DDR IB.

On the other hand, the page fault handling costs of ReP techniques contains
## Chapter 6: Implementation and Evaluation

### Table 6.6: Page fault handling costs comparison for BT benchmark among the original CLOMP, the theoretical and the ReP techniques enhanced CLOMP. The computation part of elapsed time is common to all compared items. The page fault handling costs of the original CLOMP is presented in second, and that of others are presented as a reduction ratio (e.g. \(\frac{ Orig - TReP \, Orig }{ Orig }\)).

<table>
<thead>
<tr>
<th>network</th>
<th>class</th>
<th>nprocs</th>
<th>Computation (sec)</th>
<th>Page Faults Handling Cost</th>
<th>Orig (sec)</th>
<th>Theoretical (%)</th>
<th>TReP (%)</th>
<th>HReP (%)</th>
<th>DReP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GigE</td>
<td>A</td>
<td>2</td>
<td>48.5</td>
<td></td>
<td>112.9</td>
<td>71.5</td>
<td>62.7</td>
<td>59.6</td>
<td>62.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>31.5</td>
<td></td>
<td>106.5</td>
<td>73.7</td>
<td>58.6</td>
<td>62.1</td>
<td>57.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>22.1</td>
<td></td>
<td>105.0</td>
<td>76.6</td>
<td>55.4</td>
<td>61.0</td>
<td>54.6</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
<td>199.2</td>
<td></td>
<td>435.5</td>
<td>68.7</td>
<td>64.7</td>
<td>62.1</td>
<td>64.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>125.2</td>
<td></td>
<td>372.1</td>
<td>69.1</td>
<td>59.9</td>
<td>63.2</td>
<td>59.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>77.3</td>
<td></td>
<td>339.5</td>
<td>72.6</td>
<td>57.9</td>
<td>69.3</td>
<td>63.4</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>883.7</td>
<td></td>
<td>1744.9</td>
<td>68.4</td>
<td>68.3</td>
<td>62.0</td>
<td>48.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>498.4</td>
<td></td>
<td>1424.1</td>
<td>69.6</td>
<td>63.9</td>
<td>64.9</td>
<td>63.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>315.9</td>
<td></td>
<td>1203.2</td>
<td>67.0</td>
<td>64.6</td>
<td>63.1</td>
<td>64.5</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>70.8</td>
<td>61.8</td>
<td>63.0</td>
<td>60.0</td>
<td></td>
</tr>
<tr>
<td>DDR IB</td>
<td>A</td>
<td>2</td>
<td>48.5</td>
<td></td>
<td>28.5</td>
<td>53.6</td>
<td>45.4</td>
<td>28.6</td>
<td>45.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>31.5</td>
<td></td>
<td>30.6</td>
<td>63.7</td>
<td>49.3</td>
<td>36.8</td>
<td>58.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>22.1</td>
<td></td>
<td>32.1</td>
<td>71.4</td>
<td>50.6</td>
<td>7.5</td>
<td>59.5</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2</td>
<td>199.2</td>
<td></td>
<td>106.4</td>
<td>51.1</td>
<td>43.8</td>
<td>27.3</td>
<td>42.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>125.2</td>
<td></td>
<td>104.2</td>
<td>60.6</td>
<td>47.9</td>
<td>36.1</td>
<td>44.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>77.3</td>
<td></td>
<td>101.0</td>
<td>67.3</td>
<td>43.0</td>
<td>37.4</td>
<td>43.0</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>883.7</td>
<td></td>
<td>428.6</td>
<td>51.9</td>
<td>47.6</td>
<td>36.1</td>
<td>18.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>498.4</td>
<td></td>
<td>398.2</td>
<td>61.2</td>
<td>39.5</td>
<td>38.8</td>
<td>46.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>315.9</td>
<td></td>
<td>336.2</td>
<td>61.8</td>
<td>24.0</td>
<td>44.7</td>
<td>28.7</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60.3</td>
<td>43.5</td>
<td>32.6</td>
<td>42.9</td>
<td></td>
</tr>
</tbody>
</table>

The extra overhead of page faults encoding/decoding, predicting and prefetch issuing. This overhead becomes more dominant when DDR IB interconnect is used. Consequently, the overhead reduction rate of ReP techniques on DDR IB is around 20% less than that of GigE.

Corresponding to Table 5.2, HReP achieves the best effective page miss rate reduction in the offline simulation. It reflects the best ReP page fault handling cost reduction rate achieved by HReP on GigE in Table 6.6. However, due to the above reason (HReP has highest extra overhead), HReP poses the lowest overhead reduction on DDR IB network.
6.3 Performance Evaluation of the ReP Enhanced CLOMP

6.3.2.2 BT Benchmark Scalability: Speedup

The scalability, in terms of speed, of the RePs enhanced CLOMP measured with the BT benchmark is compared with the original CLOMP, as well as the theoretical best for using prefetch technique.
As shown in Figure 6.10, the original CLOMP is unable to achieve speedup for all problem sizes on GigE network. All ReP techniques improve the speedup of CLOMP dramatically, and the enhanced CLOMP shows scalability for all problem sizes on GigE network. For class A, due to small overall elapsed time, the speedup difference between ReP techniques and theory is large. For classes B and C, due to the much larger amount of data to be maintained, the ReP techniques show more benefit which results in speedup as that predicted in theory.

On the DDR InfiniBand network, the original CLOMP achieves $\sim 1.5$ to $\sim 2.2$ speedup on 8 processes for different problem sizes. ReP techniques improves these numbers to $\sim 2.3$ and $\sim 3$ respectively. Similar to what we have observed for the overhead reduction rate in Table 6.6, the difference in speedup between RePs enhanced CLOMP and that in theory is larger than what is observed for GigE network. However, this difference decreases with increasing problem size.

Comparing to the native Intel OpenMP, the theoretical peak performance utilizing prefetch techniques for CLOMP is still $\sim 42\%$ lower (please refer to Chapter 3 for the native Intel OpenMP performance). This indicates that, with support of effective prefetch techniques, the system overhead of CLOMP still suffers from limited interconnect speed. However, optimizing other components of CLOMP such as the CAL communication layer can improve performance further (please refer Appendix E for its bandwidth and latency data in detail).

### $T_{segv}$ of Other NPB-OMP Benchmarks

The page fault handling costs ($T_{segv}$) is represented as a overhead reduction rate compared to that of the original CLOMP. In Table 6.7, the average of such ratio of ReP techniques are compared with that in theory for each benchmark on different networks.

For IS and FT, smaller page fault cost reduction rates are observed compared to other benchmarks especially on the DDR IB network. This is due to the total elapsed time for these two benchmarks being relative small, which results in prefetch prediction and issue becoming dominant. This situation is more apparent for the DDR IB network.

In contrast, the page fault handling costs has been significantly reduced by ReP techniques for LU and SP benchmarks. Similar to what is achieved for BT benchmark, ReP achieves only $\sim 10\%$ less overhead reduction rate compared to the theoretical largest overhead reduction rate via GigE network for both LU and SP. This number on the DDR IB network is $\sim 20\%$ for LU and $\sim 11\%$ for SP.

The measured prefetch efficiency and coverage for CG is consistent with what
6.3 Performance Evaluation of the ReP Enhanced CLOMP

Table 6.7: Page fault handling costs reduction ratio \( \frac{T_{\text{orig}} - T_{\text{segv}}}{T_{\text{orig}}} \) comparison for other NPB benchmarks

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Networks</th>
<th>Theoretical (%)</th>
<th>TReP (%)</th>
<th>HReP (%)</th>
<th>DReP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>GigE</td>
<td>65.9</td>
<td>37.0</td>
<td>45.2</td>
<td>38.1</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>60.7</td>
<td>21.9</td>
<td>22.3</td>
<td>18.9</td>
</tr>
<tr>
<td>FT</td>
<td>GigE</td>
<td>67.0</td>
<td>25.8</td>
<td>30.4</td>
<td>27.6</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>68.0</td>
<td>10.6</td>
<td>12.0</td>
<td>4.6</td>
</tr>
<tr>
<td>LU</td>
<td>GigE</td>
<td>67.4</td>
<td>55.4</td>
<td>58.3</td>
<td>56.8</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>58.7</td>
<td>38.5</td>
<td>37.2</td>
<td>38.1</td>
</tr>
<tr>
<td>SP</td>
<td>GigE</td>
<td>70.4</td>
<td>61.4</td>
<td>59.7</td>
<td>60.1</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>63.9</td>
<td>37.7</td>
<td>30.3</td>
<td>35.7</td>
</tr>
<tr>
<td>CG</td>
<td>GigE</td>
<td>66.3</td>
<td>-33.4</td>
<td>-47.7</td>
<td>-42.5</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>66.6</td>
<td>-110.0</td>
<td>-340.3</td>
<td>-247.0</td>
</tr>
</tbody>
</table>

is presented in the simulation results (Table 5.2), and the number of sent messages is dramatically reduced by deploying RePs (from 806 thousands to 209 thousands on average). However, none of the ReP techniques reduces the page fault handling costs of CG. For some unascertained reasons, CLOMP communication layer spent 1022\( \mu \)s per transfer when the ReP techniques are deployed, which is \( \sim 7 \) times more than that for the original CLOMP (161\( \mu \)s) on the XE cluster. Another profiling data obtained by using the time Linux command shows that the user and system CPU time of CG benchmark on the ReP enhanced CLOMP is roughly as the same as that of the original CLOMP. While for the estimated communication time \( (\text{real} - (\text{sys} + \text{user})) \), the ReP enhanced CLOMP is \( \sim 8 \) times more. However, ReP enhanced CLOMP improves the speedup of CG on a 4-node Intel cluster, as shown in Appendix C.

6.3.2.4 Overhead Analysis of the ReP Techniques

According to Table 6.7, the overhead introduced by RePs is more apparent for small benchmarks, such as IS and FT. We select IS as a candidate to analyse the introduced overhead of the ReP techniques on different networks and problem sizes.

The page fault handling costs of CLOMP is broken down into six components in Table 6.8 and Table 6.9 for IS benchmarks class A and C respectively. These components include communication time of the TMK layer (TMK Comm), the local software overhead of TMK (TMK local), the communication time of data prefetching (ReP Comm), and the local software overhead introduced by ReP (ReP local). “TMK Comm” and “ReP Comm” are further broken down to time spent on
Table 6.8: Detailed $T_{segv}$ breakdown analysis of the IS Class A Benchmark for the ReP techniques. Overall $T_{segv}$ stands for overall CLOMP overhead. “TMK Comm” stands for the communication time spent by TMK for data transfer. “TMK local” stands for the local software overhead of TMK layer. “ReP Comm” stands for the communication time spent on prefetching data. “ReP local” stands for the local software overhead introduced by using the ReP prefetch techniques. $T_{segv}$ is presented in seconds and its components are presented as a ratio to the overall $T_{segv}$.

<table>
<thead>
<tr>
<th>Overhead Breakdown</th>
<th>2 processes</th>
<th>4 processes</th>
<th>8 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig TReP HReP DReP</td>
<td>Orig TReP HReP DReP</td>
<td>Orig TReP HReP DReP</td>
</tr>
<tr>
<td>Overall $T_{segv}$ (sec)</td>
<td>1.5 1.3 1.1 1.1</td>
<td>1.9 1.1 0.9 1.1</td>
<td>2.3 1.2 1.2 1.2</td>
</tr>
<tr>
<td>TMK Comm (%)</td>
<td>diff page</td>
<td>diff page</td>
<td>diff page</td>
</tr>
<tr>
<td></td>
<td>88.2 80.4 74.4 75.9</td>
<td>81.7 66.7 62.4 66.4</td>
<td>79.1 43.3 33.5 64.7</td>
</tr>
<tr>
<td>TMK local (%)</td>
<td>5.2 4.3 4.7 5.8</td>
<td>2.3 3.2 3.9 3.3</td>
<td>2.0 1.6 1.6 2.6</td>
</tr>
<tr>
<td>ReP Comm (%)</td>
<td>diff page</td>
<td>diff page</td>
<td>diff page</td>
</tr>
<tr>
<td></td>
<td>n/a 5.9 8.9 6.0</td>
<td>n/a 7.0 5.7 2.5</td>
<td>n/a 15.8 19.6 16.5</td>
</tr>
<tr>
<td>ReP local (%)</td>
<td>n/a 1.0 1.5 1.2</td>
<td>n/a 1.0 1.2 0.7</td>
<td>n/a 1.6 2.3 0.6</td>
</tr>
</tbody>
</table>

diff and page transferring respectively. These communication costs are the periods of time purely spent in the CAL communication layer. “TMK local” includes time spent on diffing-twinning, garbage collection, and other costs. “ReP local” mainly consists of time spent on page fault record compression/reconstruction cost and prefetch prediction. The overall page fault handing costs ($T_{segv}$) is represented in
6.3 Performance Evaluation of the ReP Enhanced CLOMP

Table 6.9: Detailed $T_{segv}$ breakdown analysis of the IS Class C Benchmark for the ReP techniques. Overall $T_{segv}$ stands for overall CLOMP overhead. “TMK Comm” stands for the communication time spent by TMK for data transfer. “TMK local” stands for the local software overhead of TMK layer. “ReP Comm” stands for the communication time spent on prefetching data. “ReP local” stands for the local software overhead introduced by using the ReP prefetch techniques. $T_{segv}$ is presented in seconds and its components are presented as a ratio to the overall $T_{segv}$.

<table>
<thead>
<tr>
<th>Overhead Breakdown</th>
<th>GigE</th>
<th>DDR IB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>TReP</td>
</tr>
<tr>
<td>Overall $T_{segv}$ (sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMK Comm (%)</td>
<td>50.6</td>
<td>65.9</td>
</tr>
<tr>
<td>TMK local (%)</td>
<td>7.8</td>
<td>5.3</td>
</tr>
<tr>
<td>ReP Comm (%)</td>
<td>n/a</td>
<td>0.0</td>
</tr>
<tr>
<td>ReP local (%)</td>
<td>n/a</td>
<td>0.6</td>
</tr>
<tr>
<td>Overall $T_{segv}$ (sec)</td>
<td>25.2</td>
<td>19.8</td>
</tr>
<tr>
<td>4 processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMK Comm (%)</td>
<td>42.0</td>
<td>66.7</td>
</tr>
<tr>
<td>TMK local (%)</td>
<td>2.1</td>
<td>3.8</td>
</tr>
<tr>
<td>ReP Comm (%)</td>
<td>n/a</td>
<td>13.8</td>
</tr>
<tr>
<td>ReP local (%)</td>
<td>n/a</td>
<td>0.9</td>
</tr>
<tr>
<td>Overall $T_{segv}$ (sec)</td>
<td>28.3</td>
<td>13.6</td>
</tr>
<tr>
<td>8 processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMK Comm (%)</td>
<td>41.4</td>
<td>26.4</td>
</tr>
<tr>
<td>TMK local (%)</td>
<td>2.2</td>
<td>1.8</td>
</tr>
<tr>
<td>ReP Comm (%)</td>
<td>n/a</td>
<td>9.3</td>
</tr>
<tr>
<td>ReP local (%)</td>
<td>n/a</td>
<td>1.8</td>
</tr>
<tr>
<td>Avg ReP Local (%)</td>
<td>n/a</td>
<td>1.1</td>
</tr>
</tbody>
</table>

seconds. The above mentioned components are represented as ratios to $T_{segv}$.

For IS class A, as shown in Table 6.8, $T_{segv}$ of the original CLOMP raises with increasing number of processes on both the GigE and DDR IB networks, as we expect. $T_{segv}$ of the ReP techniques is roughly constant on the GigE network. On the other hand, $T_{segv}$ of RePs on DDR IB increases with number of processes, and
becomes more than that of the original CLOMP when the number of processes is larger than 4. This is mainly due to the relative higher software overhead introduced by ReP, which is $\sim$4% of $T_{\text{segv}}$ on average, whereas that number on GigE is only $\sim$1%. According to Table 5.2, the ReP prefetch coverage of IS increased from $\sim$50% to $\sim$70% when the number of processes increased from 2 to 8. This reflects the increasing $RePComm$ in Table 6.8.

For IS class C, as shown in Table 6.9, $T_{\text{segv}}$ of the original CLOMP raises with an increasing number of processes on both GigE and DDR IB networks. While $T_{\text{segv}}$ of RePs does not show a clear trend, the ratio of page fault handling costs reduced is much higher than what we observed for the class A IS benchmark on both networks. On average, the local software overhead introduced by RePs is $\sim$2% on both the GigE and DDR IB networks. The prefetch coverage of IS class C increased from $\sim$51% (2 processes) to $\sim$62% (8 processes), which reflects an increasing “ReP Comm” percentage for the GigE network in Table 6.9. Due to the much higher data transfer rate provided by DDR IB, there is no clear trend of “ReP Comm” movement with increasing number of processes.

In addition, for both classes, time spent on communication dominates the $T_{\text{segv}}$. With the deployment of the ReP techniques, the overall communication time (“TMK Comm” and “ReP Comm”) is around 40% less than that of the original CLOMP (“TMK Comm”) for all cases, except that of IS class A on DDR IB. The overall local cost of ReP enhanced CLOMP (“TMK local” and “ReP local”) is roughly the same as that of the original CLOMP (“TMK local”).

### 6.3.3 LINPACK Benchmarks

The two LINPACK benchmarks described in Section 5.1.1 are also used to evaluate and compare ReP enhanced CLOMP with the original CLOMP. A small matrix ($N = 4096$) and a large matrix ($N = 8192$) are deployed for the naive and optimized LINPACK benchmarks respectively with the same blocking factor ($NB = 64$).\(^3\)

The sequential elapsed time for both LINPACK benchmarks is shown in Table 6.10:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size ($N$)</th>
<th>Block Size ($NB$)</th>
<th>Elapsed Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLPK</td>
<td>4096</td>
<td>64</td>
<td>8.8</td>
</tr>
<tr>
<td>oLPK</td>
<td>8192</td>
<td>64</td>
<td>102.5</td>
</tr>
</tbody>
</table>

\(^3\)Due to huge number of page faults and memory usage, the naive LINPACK benchmark with the large matrix ($N = 8192$) exceed the maximum $mmap$ count allowed on NCI XE system. Therefore, the smaller matrix ($N = 4096$) is used instead.
### 6.3 Performance Evaluation of the ReP Enhanced CLOMP

Table 6.11: Page fault handling costs comparison for LINPACK benchmarks among the original CLOMP, the theoretical and the ReP techniques enhanced CLOMP. The computation part of elapsed time is common to all compared items. The page fault handling costs of the original CLOMP is presented in second, and that of others are presented as a reduction ratio (e.g. \( \frac{\text{Orig} - \text{TReP}}{\text{Orig}} \)).

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Network</th>
<th>Nprocs</th>
<th>Computation (sec)</th>
<th>Page Fault Handling Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Orig (sec)</td>
</tr>
<tr>
<td>iLPK</td>
<td>GigE</td>
<td>2</td>
<td>5.3</td>
<td>73.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.2</td>
<td>112.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1.9</td>
<td>133.1</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td></td>
<td>61.6</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>2</td>
<td>5.3</td>
<td>17.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.2</td>
<td>26.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>1.9</td>
<td>31.8</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td></td>
<td>45.8</td>
</tr>
<tr>
<td>oLPK</td>
<td>GigE</td>
<td>2</td>
<td>48.9</td>
<td>30.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>40.5</td>
<td>23.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>20.2</td>
<td>34.7</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td></td>
<td>51.6</td>
</tr>
<tr>
<td></td>
<td>DDR IB</td>
<td>2</td>
<td>48.9</td>
<td>21.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>40.5</td>
<td>12.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>20.2</td>
<td>19.3</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td></td>
<td>74.1</td>
</tr>
</tbody>
</table>

Table 6.10. The broken down elapsed time, when LINPACK benchmarks are run in parallel, for the original CLOMP, the calculated theoretical cost based on Equations (6.5) and (6.6), and the ReP enhanced CLOMP, is shown in Table 6.11. Similar to NPB-OMP benchmarks, the metrics used for theoretical speedup calculation is shown in Appendix B.

In Table 6.11, the computation and the page fault handling costs measured for the original CLOMP are shown in seconds. For the calculated theoretical lowest page fault handling costs and RePs enhanced CLOMP, page fault handling costs are presented as a overhead reduction rate to that of the original CLOMP. For the naive LINPACK benchmark, all ReP techniques achieve good page fault handling costs reduction rate. On the average, only around 6% less overhead reduction rate is observed compared to that of the theoretical lowest page fault handling costs for both GigE and DDR IB networks. On the contrary, similar to the offline simulation results in Section 5.6, DReP is the only ReP techniques that perform well for the optimized LINPACK benchmark, achieving ~15% less page fault handling costs reduction compared to the theoretical overhead reduction rate. Both TReP
and HReP techniques increase the page fault handling costs slightly due to the additional prefetch overhead. Since HReP issues more useless prefetches for the optimized LINPACK benchmark, the additional cost is slightly higher than that of TReP.

6.3.3.1 The Naive Benchmark

Although the page fault handling costs can be significantly reduced by prefetch techniques for the naive LINPACK benchmark, this benchmark still does not scale, as shown in Figure 6.11, on both the GigE and DDR IB networks.

In theory, this method of parallelism is not scalable for CLOMP. There are two major reasons. Firstly, according to the memory access and page fault pattern analyzed in Section 5.1.1 for the naive LINPACK benchmark, it exhibits extremely poor data locality and load balance. In different region-executions, the writers of the same page are never the same, and process 0 will always occur around $p$ times more page misses, where $p$ is total number of processes. Secondly, due to the lazy released memory consistency model [52, 38] used in CLOMP, the memory consistency is resolved post global synchronization operations and upon request. This means that all other processes are idling for the majority of the time waiting for process 0 to complete its memory consistency work.

However, despite the poor scalability, all the ReP techniques achieve roughly the same performance improvement as that predicted in theory on GigE, and a similar performance improvement on DDR IB.

![RePs VS. Original CLOMP: speedup for the naive LINPACK benchmark (N=4096, NB=64) via GigE](image1)

![RePs VS. Original CLOMP: speedup for the naive LINPACK benchmark (N=4096, NB=64) via IB](image2)

**Figure 6.11**: RePs VS. Original CLOMP: the naive LINPACK evaluation results comparison using $N \times N$ matrix ($N = 4096$) with blocking factor $NB = 64$ via both GigE and IB.
6.3 Performance Evaluation of the ReP Enhanced CLOMP

6.3.3.2 The Optimized Benchmark

CLOMP achieves greater scalability for the optimized LINPACK program. This is due to the improved memory access and page miss patterns shown in Figure 5.4 for the optimized LINPACK benchmark. In particular, much fewer page faults are caused, and the load imbalance problem is largely reduced.

Figure 6.12 illustrates the speedup of the ReP enhanced CLOMP compared to the original CLOMP using the optimized LINPACK benchmark. The optimized LINPACK benchmark scales slightly on the original CLOMP with the peak \( \sim 1.8 \) and \( \sim 2.4 \) speedup on GigE and DDR IB respectively.

In the theory, the speedup can be improved by \( \sim 61\% \) and \( \sim 75\% \) on GigE and DDR IB respectively. As a comparison, DReP achieves similar performance improvement as that in the theory, with \( \sim 50\% \) via GigE and \( \sim 58\% \) via DDR IB. TReP and HReP slow the CLOMP down slightly for the optimized LINPACK due to the useless prefetch issues and the extra software overhead introduced by prefetch techniques.

![Figure 6.12: RePs VS. Original CLOMP: the optimized LINPACK evaluation results comparison using \( N \times N \) matrix (\( N = 8192 \)) with blocking factor \( NB = 64 \) via both GigE and IB.](image)

These observations reflect the page fault handling costs reduction rates demonstrated for ReP techniques in Table 6.11. Since both the TReP and HReP techniques assume that the future page misses for the same region have already occurred in the previous executions, neither of them are able to predict the page fault pattern of the optimized LINPACK benchmark. DReP, on the other hand, is able to achieve good prefetch performance.
6.3.4 ReP Techniques with Multiple Threads per Process

The processes-threads parallel model feature provided by CLOMP is also inherited by ReP enhanced CLOMP. Since DReP is the most robust ReP technique, the optimized LINPACK benchmark with $N = 8192$ and $NB = 64$ and the DReP technique is used to evaluate multiple threads performance of the enhanced CLOMP system.

**Table 6.12:** Page faults handling cost comparison between DReP and the original CLOMP for the optimized LINPACK benchmark with multiple threads per process. “SEGV” represents the ratio of page faults handling cost to the corresponding elapsed time; “SEGV Lock” in turn represents a ratio of pthread mutex cost within “SEGV”.

<table>
<thead>
<tr>
<th>interconnects</th>
<th>$p \times t$</th>
<th>Elapsed (sec)</th>
<th>Orig SEGV (%)</th>
<th>Orig SEGV Lock (%)</th>
<th>DReP Elapsed (sec)</th>
<th>DReP SEGV (%)</th>
<th>DReP SEGV Lock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GigE</strong></td>
<td>2x2</td>
<td>64.4</td>
<td>43.6</td>
<td>37.8</td>
<td>51.7</td>
<td>29.8</td>
<td>17.9</td>
</tr>
<tr>
<td></td>
<td>4x2</td>
<td>53.1</td>
<td>58.6</td>
<td>41.3</td>
<td>38.0</td>
<td>42.1</td>
<td>21.9</td>
</tr>
<tr>
<td></td>
<td>8x2</td>
<td>61.4</td>
<td>55.9</td>
<td>30.9</td>
<td>50.0</td>
<td>45.8</td>
<td>20.8</td>
</tr>
<tr>
<td></td>
<td>2x4</td>
<td>54.0</td>
<td>58.0</td>
<td>63.9</td>
<td>34.2</td>
<td>33.7</td>
<td>39.0</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>55.6</td>
<td>56.4</td>
<td>62.8</td>
<td>35.5</td>
<td>31.7</td>
<td>39.3</td>
</tr>
<tr>
<td></td>
<td>8x4</td>
<td>56.7</td>
<td>56.0</td>
<td>62.5</td>
<td>37.9</td>
<td>34.1</td>
<td>41.8</td>
</tr>
<tr>
<td><strong>DDR IB</strong></td>
<td>2x2</td>
<td>44.3</td>
<td>34.0</td>
<td>31.0</td>
<td>36.5</td>
<td>19.7</td>
<td>14.7</td>
</tr>
<tr>
<td></td>
<td>4x2</td>
<td>35.5</td>
<td>44.2</td>
<td>21.7</td>
<td>29.0</td>
<td>31.6</td>
<td>6.9</td>
</tr>
<tr>
<td></td>
<td>8x2</td>
<td>36.5</td>
<td>44.6</td>
<td>20.5</td>
<td>31.1</td>
<td>35.1</td>
<td>9.2</td>
</tr>
<tr>
<td></td>
<td>2x4</td>
<td>42.7</td>
<td>43.7</td>
<td>51.8</td>
<td>34.8</td>
<td>31.0</td>
<td>36.1</td>
</tr>
<tr>
<td></td>
<td>4x4</td>
<td>39.0</td>
<td>46.2</td>
<td>52.1</td>
<td>29.3</td>
<td>28.4</td>
<td>32.8</td>
</tr>
<tr>
<td></td>
<td>8x4</td>
<td>51.4</td>
<td>45.6</td>
<td>56.6</td>
<td>35.9</td>
<td>22.1</td>
<td>31.3</td>
</tr>
</tbody>
</table>

CLOMP uses pthread mutex to maintain the correctness of diffs/pages update when multiple threads are deployed in the same process. As shown in Table 6.12, the page fault handling cost (“SEGV”) is presented as a ratio to the corresponding elapsed time, and the time spent on locking within “SEGV” cost is labeled as “SEGV LOCK”, and in turn represented as a ratio to “SEGV”.

Like in the single thread cases, DReP effectively reduces the page fault handling costs. The DReP technique also reduces the locking time for all cases, as shown in Table 6.12. This is mainly because DReP prefetches at the beginning of each region-execution, and prefetched pages are immediately updated and their state is changed to prefetched_page or prefetched_diff before any computation, which eliminates most locking contention between threads. As a result, the elapsed time is reduced by DReP.

The scalability of DReP enhanced CLOMP and the original CLOMP is shown in Figure 6.13. DReP has improved the scalability of CLOMP on both the GigE and
6.4 Summary

In this chapter, the ReP page prefetch techniques are implemented in to CLOMP runtime. Some implementation issues are discussed as follows.

- Due to memory usage concerns, the proposed stride-augmented run-length encoding method has been utilized as a common strategy for all ReP techniques.
- The flush filter mentioned in Chapter 5 is implemented for all ReP techniques to remove the flushed shared page from the issued prefetch page list.

Figure 6.13: DReP vs Original CLOMP: the optimized LINPACK benchmark \((N = 8192\) and \(NB = 64\)) results comparison with multiple threads per process via both GigE and IB. (a) 2 threads per process, (b) 4 threads per process.

DDR IB networks. DReP achieves more speedup improvement when more threads are used within the same process. This is mainly due to the reduction of cost for locking contention.

6.4 Summary
Chapter 6: Implementation and Evaluation

- As Intel compilers are closed source and OpenMP runtime does not provide sufficient information about start of a region, a new region notification interface is implemented to allow the user to directly notify the ReP predictor the start of a region.

- Two new page states, prefetched\_diff and prefetched\_page, are introduced in order to keep track of invalid page accesses, and two sub-states, pref\_diff\_locked and pref\_page\_locked, are introduced to prevent redundant page prefetches between threads within the same process.

The memory consistency cost of the RePs enhanced CLOMP runtime are measured with MCBENCH, and the performance is evaluated using the NPB-OMP benchmarks and the two LINPACK benchmarks. Due to the different memory access and page fault pattern MCBENCH explores by varying chunk size, the performance of ReP techniques varies significantly. On average, when 4 byte chunks are used, the memory consistency cost of CLOMP has been significantly reduced by all ReP techniques to \( \sim 15\% \) and \( \sim 22\% \) on GigE and DDR IB respectively, and the more cost reduction observed for a larger number of accessed shared pages. For 2048 bytes chunk size, all ReP techniques performs as well as that of 4 bytes chunk with 2 processes, and no obvious improvement is observed with larger number of processes. For the 4096 bytes chunk size, DReP significantly reduced the memory consistency cost of CLOMP by \( \sim 40\% \) on the GigE and \( \sim 33\% \) on the DDR IB networks.

To verify the ReP enhanced CLOMP system, the theoretical peak performance, which can be achieved with an ideal prefetch technique, is calculated based on the measured number of page and diff requests along the critical path and the peak bandwidth provided by the communication layer of CLOMP. The measured performance of ReP enhanced CLOMP is compared with both the original and theoretical performance for NPB-OMP and the two LINPACK benchmarks.

For the NPB-OMP benchmarks, except CG, all ReP techniques effectively reduced the page fault handling costs of CLOMP. An average overhead reduction rate for benchmarks with larger elapsed time, e.g. BT, LU and SP, is \( \sim 60\% \) on GigE and \( \sim 38\% \) on DDR IB. This represents \( \sim 10\% \) and \( \sim 18\% \) lower overhead reduction rate compared to that in theory. Since the total elapsed time for IS and FT is relatively small (several seconds), the software cost brought by prefetching becomes more apparent. This in turn results in \( \sim 34\% \) and \( \sim 15\% \) of the page fault servicing costs reduction rate on GigE and DDR IB respectively.

For the naive LINPACK benchmark, all ReP techniques can reduce the page fault handling costs significantly (\( \sim 55\% \) on GigE and \( \sim 40\% \) on DDR IB), which represents less than 5% difference with that in theory. However, the naive
LINPACK benchmark does not scale with either the RePs enhanced CLOMP or that with the theoretical optimal prefetch technique. On the other hand, for the optimized LINPACK benchmark, only the DReP technique effectively reduces the page fault handling costs, and results in an noticeable improvement for speedup, 50% on the GigE and 58% on the DDR IB networks.

Finally, ReP enhanced CLOMP are also evaluated with multiple threads per process using the optimized LINPACK benchmark. Similar to what has already observed for the single thread cases, DReP again significantly reduces the page fault handling costs as well as the associated locking cost. This results in better scalability.

To sum up, the ReP page prefetch techniques effectively reduced the system overhead compared to the original CLOMP system. The larger the problem size is, the more overhead is reduced by using RePs. The software overhead introduced by ReP is relatively small compared to the memory consistency overhead of CLOMP, and it becomes negligible with increasing problem sizes.
Part 

IV

Conclusions and Future Work
Chapter 7

Conclusions and Future Work

Contents

7.1 Conclusions ...................................................... 150
  7.1.1 Performance Evaluation of CLOMP ...................... 150
  7.1.2 SIGSEGV Driven Performance Models .................. 152
  7.1.3 Performance Enhancement by RePs ..................... 152

7.2 Future Directions ............................................... 155
  7.2.1 Performance Evaluation ................................. 156
  7.2.2 Performance Optimizations .............................. 156
  7.2.3 Adapting ReP Techniques to the Latest Technologies .. 156
  7.2.4 Potential Use of sRLE .................................. 157
Cluster OpenMP systems extend OpenMP program on to clusters. Consequently, such systems significantly reduce the programming costs of parallel applications. However, they suffer from a high system overhead for applications exhibiting poor data locality, fine-grained granularity, and frequent synchronizations.

In this thesis, we have evaluated, modeled and enhanced the performance of Intel Cluster OpenMP (CLOMP), a representative of page-based cluster OpenMP systems. We have successfully demonstrated that our region-based techniques are able to accurately model and effectively enhance performance of cluster OpenMP systems.

In this chapter, we draw conclusions in three sections corresponding to the different contributions. Future directions are described after conclusions.

7.1 Conclusions

Through a quantitative approach, we have successfully demonstrated that the major overhead for cluster OpenMP system is the page fault servicing costs also known as memory consistency costs. With identifying parallel and sequential regions of OpenMP programs, we have developed two region-based SIGSEGV driven performance (SDP) models which successfully rationalized the types and numbers of different page faults to the performance of cluster OpenMP systems. Furthermore, the most expensive type of overhead has been identified as the cost associated with servicing page faults that involve inter-process data transfers. Moreover, our three region-based prefetch techniques effectively reduced the major overhead of cluster OpenMP systems. Detailed conclusions for each aspect are discussed in following three sections.

7.1.1 Performance Evaluation of CLOMP

The system overhead of CLOMP is quantitatively analyzed, and MCBENCH is the first micro-benchmark that measures memory consistency cost of OpenMP implementations. This research contributes towards an understanding of the performance of page-based cluster OpenMP systems.

CLOMP provides different parallelism configurations that allows utilizing physical shared memory within a process and virtual global shared memory among processes. We have evaluated all these configurations using the NAS Parallel Benchmark OpenMP (NPB-OMP) suite via a quantitative approach.
7.1 Conclusions

According to NPB-OMP evaluation results, when single process with multiple threads configuration is used, we conclude that CLOMP shows comparable performance (less than 2% difference) to native Intel OpenMP on a symmetric multiprocessing (SMP) compute node when the compute cores are not fully subscribed.

When single thread per process and multiple processes are used, CLOMP suffers from a high memory consistency overhead. Except for EP, it is not able to achieve scalability on all other NPB-OMP benchmarks via Gigabit Ethernet. Only a little speedup is achieved via faster interconnects, such as DDR and QDR InfiniBand. On the average, \( \sim 75\% \) and \( \sim 55\% \) of the total elapsed time is spent on memory consistency cost for NPB-OMP benchmarks on GigE and DDR IB respectively. A linear speedup is observed for EP, because it is an embarrassingly parallel application that exhibits minimal synchronizations, good data locality and coarse granularity.

When multiple threads per process and multiple processes are used, CLOMP still does not show scalability for NPB-OMP benchmarks, and there is no clear effect on the memory consistency costs. This is due to an extra locking cost, which is introduced to maintain the correctness of diffs/pages update between threads within the same process. This locking cost increases with both increasing number of processes and threads. From these evaluation results, we conclude that the memory consistency costs still dominate the performance of CLOMP with multiple threads deployed in the same process.

Since all existing OpenMP benchmarks only measure individual OpenMP operations, we have developed a micro-benchmark, MCBENCH which is able to measure the memory consistency costs of OpenMP implementations. In the thesis, we use MCBENCH to characterise the memory consistency costs of cluster OpenMP systems. From the MCBENCH evaluation results, we conclude that the memory consistency cost of CLOMP is proportional to the total number of shared pages and the number of writers to the same shared page.

Based on above conclusions, on multiple compute nodes, CLOMP is able to achieve good performance for the application with minimal synchronizations, good data locality, and coarse-grained granularity. On the other hand, CLOMP struggles with the application exhibiting complex synchronization and memory accessing patterns.
7.1.2 SIGSEGV Driven Performance Models

Based on identifying parallel and sequential regions of OpenMP programs, two region-based SIGSEGV driven performance (SDP) models were developed to quantitatively rationalise the performance of CLOMP to the numbers and types of different page faults with their associated costs. This research contributes towards the performance modeling of page-based cluster OpenMP systems. To our knowledge, it is the first time when performance models have been developed in the context of OpenMP regions for software distributed shared memory systems.

SDP models assume that page faults occurring on different processes can be fully overlapped, the overheads associated with the creation of a parallel region are small and negligible in comparison to the memory consistency cost, and OpenMP applications are perfectly load-balanced. The SEGVprof profiling tool provided by Intel is used to report the page misses numbers of regions.

The SDP models enhance our understanding in the performance of page-based cluster OpenMP implementations. According to measured cost of different page fault types, a FETCH fault is much more expensive compared to a WRITE fault by more than an order of magnitude. Therefore, we conclude that servicing the FETCH fault is the most expensive overhead of CLOMP.

The NPB-OMP benchmarks are used to validate the SDP models. The validation results of the two performance models show that for CLOMP the model based on critical path analysis is slightly more accurate than the simpler one based on aggregate page fault counts. For applications where the page fault overhead from different processes is highly overlapped, the models are generally accurate to within 10%, and when this is not the case, the models are optimistic.

The limitations of the SDP model have been resolved by reformulating the critical path model. The local page fault servicing costs (WRITE faults and local components of FETCH faults) and the computation cost are represented as a unified local cost. The communication component of FETCH faults is represented as a communication cost. These two replaced items can be profiled, and can address all the limitations introduced by the assumptions. The revised model has been used to calculate the theoretical peak performance for enhanced CLOMP.

7.1.3 Performance Enhancement by RePs

We believe that this research contributes three effective page prefetch techniques and a novel application of run-length encoding method, which are not limited to cluster OpenMP context. To our knowledge, it is the first time that parallel and
7.1 Conclusions

Sequential regions of OpenMP programs are identified and utilized in page prefetch techniques. Additionally, it is the first time that a run-length encoding method has been augmented with stride of consecutive data points and used multiple times to reconstruct page miss record that facilitates more accurate page miss prediction.

We developed three region-based page prefetch (ReP) techniques to improve existing page prefetch techniques for sDSM systems. Based on the memory access and page miss patterns observed from different OpenMP programs. The details of these three ReP techniques are summarised as follows.

- **Temporal ReP (TReP) technique** examines the temporal locality of pages accesses between previous two consecutive region-executions. If the temporal locality is experienced in the previous two consecutive region-executions, it prefetches all missed pages in the most recent region-execution.

- **Hybrid ReP (HReP) technique** combines TReP with the Adaptive++ techniques that prefetch a limited number of pages each time a page fault occurs. HReP considers the temporal locality between consecutive region-executions, and the spatial locality within a region-execution.

- **We developed a novel stride-augmented run-length encoding (sRLE) method to reconstruct page miss records which facilitate much accurate and efficient analysis for dynamic memory access patterns. Based on sRLE, the third ReP technique, a dynamic ReP (DReP) technique, is developed and successfully addressed dynamic paging behaviour between executions of a region.**

The prefetch efficiency and coverage of the proposed ReP techniques were compared to that of some well known page prefetch techniques including Adaptive++, and third order finite context method (TODFCM) via offline simulations. The simulations are conducted based on the page faults record generated by the NPB-OMP benchmarks, and the two LINPACK benchmarks, the naive and the optimised implementations. All ReP techniques achieve good prefetch efficiency and coverage for all NPB-OMP and the naive LINPACK benchmarks. Only DReP achieves both good efficiency and coverage for the optimised LINPACK benchmark. On average, TReP, HReP and DReP effectively reduced page misses by 54%, 62% and 64% for all benchmarks respectively. This represents an improvement of 45% and 28% (TReP), and 52% and 35% (HReP), and 54% and 37% (DReP) on Adaptive++ and TODFCM respectively. In terms of efficiency, TReP showed the best efficiency overall, followed closely by TODFCM. The main difference in effective page miss reduction was however due to coverage, with DReP achieving 2% and 13% better coverage than HReP and TReP, largely due to its capability of handling dynamic page miss patterns of the optimised LINPACK benchmark.
where other prefetch technique can not. Moreover, HReP achieves 11% better coverage than TReP, which is because of its exploitation of spatial locality. TReP in turn achieved 35% and 46% better coverage than TODFCM and Adaptive++ respectively.

Furthermore, the ReP techniques were implemented into the CLOMP runtime library effectively to maintain:

- minimal memory usage by utilizing the stride-augmented run length encoding method and inheriting the garbage collect mechanism of CLOMP;
- correct page state transition by introducing two additional page state, prefetched_diff and prefetched_page;
- correct transaction sequence by offload all prefetch request handling to a sDSM daemon thread.

There are two major source of useless and redundant prefetch issues. The first is caused by OpenMP non-global synchronization operations. The second is caused when different threads within the same process issues prefetch for the same page. The ReP implementation successfully addressed these by:

- removing the page misses record caused by non-global synchronization operations such as flush and lock;
- introducing two page sub-state, pref_diff_locked and pref_page_locked to block the pages that are currently under prefetching.

We evaluated the ReP enhanced CLOMP runtime using MCBENCH, NPB-OMP and LINPACK. The results are compared among the theoretical best performance that can be achieved by prefetch technique, the ReP enhanced CLOMP, and the original CLOMP. The elapsed time measured for different benchmarks were broken down to show effect of ReP techniques. According to the evaluation results, the ReP techniques have significantly reduced the memory consistency overhead of CLOMP. For the NPB-OMP benchmarks with large elapsed time, e.g. BT, LU and SP, this overhead is reduced effectively by \( \sim 60\% \) and \( \sim 38\% \) on the GigE and DDR IB networks respectively. For IS and FT, the ReP techniques reduce the overhead on GigE by 34% and on DDR IB by 15%. The benefit on DDR IB is less because the software overhead brought by prefetching becomes more apparent. In general, the introduced software overhead by ReP is relatively small compared to the memory consistency overhead of CLOMP, and it becomes negligible with increasing problem sizes. For the LINPACK benchmarks, with the assistance of sRLE, DReP significantly outperforms the other ReP techniques with effectively
reducing 50% and 58% of page fault handling costs on the GigE and DDR IB networks respectively.

We conclude that the ReP techniques significantly reduce the memory consistency cost of CLOMP. This results in an obvious performance improvement for the most tested benchmarks. DReP is more robust as its assumptions cover more memory access patterns.

7.2 Future Directions

With the enhancement of region-based prefetch techniques, CLOMP still does not show good performance for some applications in terms of speedup scalability. After the in depth research on cluster OpenMP systems, we find that there are two major reasons.

The first is the communication layer of CLOMP (CAL) is not efficient. The measure peak bandwidth on both GigE and IB networks are much lower than what is measured with MPI (see Appendix E for detailed performance data).

The second is the limitation of software distributed shared memory systems. The attempt to hide explicit data exchange control from programmer is very expensive, especially when this attempt is for general purposes. Cluster OpenMP is possibly more feasible when faster interconnects that can provide a network bandwidth at the same level of memory bandwidth become available.

We think a better approach of simplifying parallel programming is to expose necessary data exchange to programmers rather than hide all transactions. Recently, both industries and researchers have expressed high enthusiasm to discover alternative parallel programming models and environments, such as PGAS languages and single-system-image hardware virtualization technique. In terms of performance, it would be benefit to consider hybrid programming models, such as OpenMP and MPI hybrid, or sDSM and MPI hybrid.

However, we also feel that more work could be done on CLOMP to better understand and further optimize it. Additionally, our region-based techniques and proposed stride-augmented run-length (sRLE) encoding method can be adapted to the latest technologies. These future directions are described in the rest of this section.
Chapter 7: Conclusions and Future Work

7.2.1 Performance Evaluation

The performance evaluation in this thesis is mainly focusing on identifying the major system overhead of CLOMP. However, a performance evaluation of each individual part of CLOMP will be interesting as well, such as flush, lock, barrier OpenMP operations. Some existing OpenMP benchmarks, such as EPCC, can be used for this purpose.

7.2.2 Performance Optimizations

As mentioned above, the CAL layer of CLOMP is not as efficient as other communication libraries, such as OpenMPI. Optimization of CAL may improve performance of CLOMP. Additionally, utilising the multi-rail techniques, as shown in Appendix D, can significantly improve the network bandwidth.

Additionally, the current implementation of RePs require the programmer to insert calls to notify the runtime of the start of a region and its type (parallel or sequential). It will be ideal to have the compiler or OpenMP runtime to handle this task. In order to achieve this goal, we need to obtain access to the Intel compiler source code for a deeper understanding of the compilation of OpenMP directives to CLOMP runtime.

7.2.3 Adapting ReP Techniques to the Latest Technologies

ScaleMP, a single-system-image virtualized SMP machine, is a state-of-art infrastructure that extends shared memory programs on to distributed memory architectures. Since it is similar to a sDSM in that all inter-process memory transactions is hidden from the programmer and handled by ScaleMP directly, there is a potential use of the ReP prefetch techniques to improve its performance. In the future, RePs can be used in this systems.

Some PGAS languages, such as UPC, allows programmer directly access remote partitioned global space. In order to achieve good performance with such language, good data locality is required. However, this condition usually is hard to be satisfied. Therefore, the ReP techniques could be used to prefetch remote data into local partition before the actual access.
7.2 Future Directions

7.2.4 Potential Use of sRLE

The run-length encoding method is widely used in data compression, image processing, and pattern recognition area. Our proposed sRLE has the ability to compress a series of individual points into a number of rectangles, which can then be analysed for shifting patterns. In the future, the sRLE can be utilized recursively to increase the level of compression. We are looking for applying sRLE in various application areas.
Part

Appendices
Appendix

A

Algorithms Used in DReP

Contents

A.1 Stride-augmented Run-length Encoding Algorithms . . . . . 162
  A.1.1 Algorithm 1: Page Fault Record Reconstruction Step (a) . . 162
  A.1.2 Algorithm 2: Page Fault Record Reconstruction Step (b) . . 162
  A.1.3 Algorithm 3: Page Fault Record Reconstruction Step (c) . . 163
A.2 Algorithm 4: DReP Predictor . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
Appendix A: Algorithms Used in DReP

A.1 Stride-augmented Run-length Encoding Algorithms

A.1.1 Algorithm 1: Page Fault Record Reconstruction Step (a)

**Input:** int fpage_list[N]

**Output:** int sub_arrays[], int n_arrays

\[\text{sorted_list}[N] \leftarrow \text{sort}(fpage_list[N]);\]

\[\text{for } i = 0 \rightarrow N - 1 \text{ do}\]

\[\text{if } (\text{sorted_list}[i] - \text{sorted_list}[i - 1] \neq \text{sorted_list}[i - 1] - \text{sorted_list}[i - 2])\]

\[\text{AND } (i \geq 2) \text{ then}\]

\[\text{Create a new sub_array;}\]

\[\text{end}\]

\[\text{Append sorted_list}[i] \text{ to current sub_array;}\]

\[\text{end}\]

**Algorithm 1:** Page Fault Record Reconstruction: the original record to sorted sub-arrays (Figure 5.7 (a)).

A.1.2 Algorithm 2: Page Fault Record Reconstruction Step (b)

**Input:** int sub_arrays[n_arrays]

**Output:** struct 1st_encoding_fpage 1st_lvl_lists[n_arrays]

\[\text{foreach sub-arrays do}\]

\[\text{foreach elements in sub-array do}\]

\[1st_lvl_lists[n_lvl1].start_page \leftarrow \text{first element in the sub-array;}\]

\[1st_lvl_lists[n_lvl1].stride \leftarrow \text{the common stride of the sub-array;}\]

\[1st_lvl_lists[n_lvl1].run_length \leftarrow \text{length of the sub-array;}\]

\[\text{end}\]

\[\text{end}\]

**Algorithm 2:** Page Fault Record Reconstruction: the sorted sub-arrays compressed into the first level sRLE record (Figure 5.7 (b)).
A.2 Algorithm 4: DReP Predictor

A.1.3 Algorithm 3: Page Fault Record Reconstruction Step (c)

**Input:** struct 1st_encoding_fpage 1st_lvl_lists[n_arrays], int n_arrays

**Output:** struct 2nd_encoding_fpage 2nd_lvl_lists[n_lvl2], int n_lvl2

\[
\text{start_lvl}_1 \leftarrow 1st\_lvl\_lists[0]; \\
n_{lvl2} = 0;
\]

for \( i = 1 \rightarrow n_{lvl1} \) do

\[
n_{lvl2} \leftarrow 0;
\]

if \( 1st\_lvl\_lists[i].stride == 1st\_lvl\_lists[i - 1].stride \) AND

\[
1st\_lvl\_lists[i].run\_length == 1st\_lvl\_lists[i - 1].run\_length \]

then

\[
\text{if } i == 1 \text{ then } \]

\[
\text{run\_length} \leftarrow \text{run\_length} + 1; \\
\text{stride} \leftarrow 1st\_lvl\_lists[i].start\_page - 1st\_lvl\_lists[i - 1].start\_page;
\]

else if \( 1st\_lvl\_lists[i].start\_page - 1st\_lvl\_lists[i - 1].start\_page \) AND

\[
1st\_lvl\_lists[i - 1].start\_page - 1st\_lvl\_lists[i - 2].start\_page \]

then

\[
\text{run\_length} \leftarrow \text{run\_length} + 1; \\
\text{stride} \leftarrow 1st\_lvl\_lists[i].start\_page - 1st\_lvl\_lists[i - 1].start\_page;
\]

else

\[
\text{new}_{lvl2} \leftarrow 1;
\]

end

else

\[
\text{new}_{lvl2} \leftarrow 1;
\]

end

\[
2nd\_lvl\_lists[n_{lvl2}].start\_lvl_1 \leftarrow \text{start\_lvl\_1}; \\
2nd\_lvl\_lists[n_{lvl2}].stride \leftarrow \text{stride}; \\
2nd\_lvl\_lists[n_{lvl2}].run\_length \leftarrow \text{run\_length};
\]

if \( \text{new}_{lvl2} == 1 \) then

\[
n_{lvl2} \leftarrow n_{lvl2} + 1;
\]

end

end

Algorithm 3: Page Fault Record Reconstruction: the the 1st level sRLE records compressed into the 2nd level sRLE records (Figure 5.7 (c)).

A.2 Algorithm 4: DReP Predictor
Appendix A: Algorithms Used in DReP

Algorithm 4: DReP predictor to predict pages to be prefetched.
Appendix

B

$T_{segv,local}$ and $N^f_{total}$ for Theoretical ReP Speedup Calculation

Contents

B.1 NPB-OMP Benchmarks Datasheet 166
B.2 LINPACK Benchmarks Datasheet 166
Appendix B: $T_{segv,local}$ and $N^f_{total}$ for Theoretical ReP Speedup Calculation

B.1 NPB-OMP Benchmarks Datasheet

In this section, the parameters, $T_{segv,local}$ and $N^f_{total}$, used to calculate the theoretical performance with improvement of prefetch techniques for NPB-OMP benchmarks, which is illustrated in Chapter 6, are presented in Table B.1 and B.2.

$T_{segv,local}$ is presented in seconds. $N^f_{total}$ consists of the numbers of both diff and page requests, and it is presented in multiple of thousands ($\times 1000$)

Table B.1: $T_{segv,local}$ (sec) for some NPB-OMP benchmarks with different number of processes.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>nprocs</th>
<th>class A</th>
<th>class B</th>
<th>class C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>2</td>
<td>6.9</td>
<td>26.7</td>
<td>122.4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.3</td>
<td>22.0</td>
<td>87.4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3.9</td>
<td>13.7</td>
<td>50.7</td>
</tr>
<tr>
<td>IS</td>
<td>2</td>
<td>0.1</td>
<td>0.5</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.3</td>
<td>1.0</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.3</td>
<td>1.1</td>
<td>4.3</td>
</tr>
<tr>
<td>FT</td>
<td>2</td>
<td>2.5</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.5</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.7</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>LU</td>
<td>2</td>
<td>23.3</td>
<td>86.8</td>
<td>382.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>19.1</td>
<td>65.5</td>
<td>244.2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16.6</td>
<td>45.3</td>
<td>150.4</td>
</tr>
<tr>
<td>SP</td>
<td>2</td>
<td>11.4</td>
<td>44.6</td>
<td>170.9</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11.7</td>
<td>28.3</td>
<td>130.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.5</td>
<td>25.9</td>
<td>103.5</td>
</tr>
<tr>
<td>CG</td>
<td>2</td>
<td>0.3</td>
<td>3.2</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.4</td>
<td>3.6</td>
<td>6.6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.5</td>
<td>4.7</td>
<td>7.7</td>
</tr>
</tbody>
</table>

B.2 LINPACK Benchmarks Datasheet

In this section, the parameters, $T_{segv,local}$ and $N^f_{total}$, used to calculate the theoretical performance with improvement of prefetch techniques for two LINPACK benchmarks, which is illustrated in Chapter 6, are presented in Table B.1 and B.2.

$T_{segv,local}$ is presented in seconds. $N^f_{total}$ consists of the numbers of both diff and page requests, and it is presented in multiple of thousands ($\times 1000$)
### Table B.2: $N_{total}^t$ for some NPB-OMP benchmarks with different number of processes.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>nprocs</th>
<th>class A</th>
<th></th>
<th>class B</th>
<th></th>
<th>class C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>diffs</td>
<td>pages</td>
<td></td>
<td>diffs</td>
<td>pages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(×1000)</td>
<td></td>
<td>(×1000)</td>
<td></td>
<td>(×1000)</td>
</tr>
<tr>
<td>BT</td>
<td>2</td>
<td>910</td>
<td>108 (×1000)</td>
<td>2784</td>
<td>1058 (×1000)</td>
<td>10576</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>870</td>
<td>70 (×1000)</td>
<td>2480</td>
<td>836 (×1000)</td>
<td>8808</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>740</td>
<td>88 (×1000)</td>
<td>1980</td>
<td>786 (×1000)</td>
<td>3848</td>
</tr>
<tr>
<td>IS</td>
<td>2</td>
<td>14</td>
<td>0 (×1000)</td>
<td>50</td>
<td>6 (×1000)</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>0 (×1000)</td>
<td>56</td>
<td>12 (×1000)</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>20</td>
<td>2 (×1000)</td>
<td>66</td>
<td>14 (×1000)</td>
<td>178</td>
</tr>
<tr>
<td>FT</td>
<td>2</td>
<td>112</td>
<td>198 (×1000)</td>
<td>n/a</td>
<td>n/a (×1000)</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>102</td>
<td>134 (×1000)</td>
<td>n/a</td>
<td>n/a (×1000)</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>82</td>
<td>62 (×1000)</td>
<td>n/a</td>
<td>n/a (×1000)</td>
<td>n/a</td>
</tr>
<tr>
<td>LU</td>
<td>2</td>
<td>2712</td>
<td>596 (×1000)</td>
<td>6206</td>
<td>5978 (×1000)</td>
<td>13860</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2742</td>
<td>468 (×1000)</td>
<td>6036</td>
<td>5398 (×1000)</td>
<td>14452</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2624</td>
<td>274 (×1000)</td>
<td>5134</td>
<td>3106 (×1000)</td>
<td>11832</td>
</tr>
<tr>
<td>SP</td>
<td>2</td>
<td>1570</td>
<td>178 (×1000)</td>
<td>4146</td>
<td>2094 (×1000)</td>
<td>15228</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1564</td>
<td>144 (×1000)</td>
<td>4056</td>
<td>1658 (×1000)</td>
<td>13384</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1506</td>
<td>154 (×1000)</td>
<td>3660</td>
<td>1456 (×1000)</td>
<td>8480</td>
</tr>
<tr>
<td>CG</td>
<td>2</td>
<td>50</td>
<td>0 (×1000)</td>
<td>458</td>
<td>46 (×1000)</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>82</td>
<td>0 (×1000)</td>
<td>740</td>
<td>24 (×1000)</td>
<td>1146</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>194</td>
<td>0 (×1000)</td>
<td>1340</td>
<td>18 (×1000)</td>
<td>1832</td>
</tr>
</tbody>
</table>

### Table B.3: $t_{segv}^{avg}$ (sec) for LINPACK benchmarks with different number of processes.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>nprocs</th>
<th>$t_{segv}^{avg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>oLPK (N=8192, NB=64)</td>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.0</td>
</tr>
<tr>
<td>iLPK (N=4096, NB=64)</td>
<td>2</td>
<td>5.8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10.4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>11.6</td>
</tr>
</tbody>
</table>

### Table B.4: $N_{total}^f$ for LINPACK benchmarks with different number of processes.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>nprocs</th>
<th>diffs (×1000)</th>
<th>pages (×1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>oLPK (N=8192, NB=64)</td>
<td>2</td>
<td>16</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>40</td>
<td>242</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>72</td>
<td>250</td>
</tr>
<tr>
<td>iLPK (N=4096, NB=64)</td>
<td>2</td>
<td>378</td>
<td>360</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>628</td>
<td>482</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>852</td>
<td>444</td>
</tr>
</tbody>
</table>

167
Appendix B: $T_{seg,local}$ and $N_{total}^f$ for Theoretical ReP Speedup Calculation
Appendix

C

TReP and DReP Performance
Results of the NPB-OMP
benchmarks on a 4-node Intel Cluster

Contents

C.1 Experimental Setup ............................ 170
C.2 Sequential Elapsed Time ......................... 170
C.3 TReP and DReP Evaluation ...................... 170
  C.3.1 Elapsed Time over Gigabit Ethernet ......... 170
  C.3.2 Elapsed Time over DDR InfiniBand .......... 173
### C.1 Experimental Setup

The implementations of DReP and TReP are evaluated on a Intel cluster consisting of four Sun Ultra24 workstations. Each node contains an Intel Core 2 Quad Q6600 CPU and 4GB DDR2 memory. Both Gigabit Ethernet and DDR InfiniBand are deployed to connect workstations, which deliver peak MPI bandwidth at 103.2 MB/s and 1750.8 MB/s respectively (measured by using OSU-MPI benchmark [61]).

The maximum number of prefetched pages in a single request is set to 128 in our experiments. Some benchmarks from NPB-OMP suite are used to evaluate the prefetch efficiency, coverage and latency effects of utilizing TReP and DReP. These benchmarks are run with 1 to 4 processes on the Sun Ultra24 4-node cluster.

In the rest of this section, we will present the elapsed time of both TReP and DReP enhanced CLOMP. The corresponding elapsed time on both Gigabit Ethernet and DDR InfiniBand for the original CLOMP, TReP and DReP will be presented in the following sections.

### C.2 Sequential Elapsed Time

The elapsed time of different benchmarks on one thread is listed in Table C.1.

### C.3 TReP and DReP Evaluation

#### C.3.1 Elapsed Time over Gigabit Ethernet

As shown in Table C.1, the single thread elapsed time for different benchmarks varies in the same order of magnitude as their total number of page misses. Class B IS benchmark poses shortest single thread elapsed time (4.8 seconds).
C.3 TReP and DReP Evaluation

Therefore, we might not be able to achieve much performance benefit by introducing prefetch techniques. The detailed performance comparison among original CLOMP, CLOMP with TReP and CLOMP with DReP over Gigabit Ethernet is shown in Figure C.1 and C.2 in term of speedup.

Corresponding to Figure C.1 and C.2, we see that the original CLOMP does not scale for almost every NPB-OMP benchmarks, except BT and CG class C. Both

\[\text{Figure C.1: Speedup of the BT and CG benchmarks over Gigabit Ethernet.}\]
TReP and DReP improve the performance based on that of original CLOMP. For BT class B, both TReP and DReP shows a little speedup, while TReP reaches 1.4 at 4 threads and DReP reaches $\sim 1.2$ at 4 threads. Both TReP and DReP prefetch techniques save significant elapsed time compared to original CLOMP ($\sim 51\%$ for TReP and $\sim 41\%$ for DReP). An even better scalability is observed for TReP and DReP with BT class C. The original CLOMP shows $\sim 1.8$ speedup at 4 threads,
C.3 TReP and DReP Evaluation

while both TReP and DReP shows \(~3.8\) speedup at 4 threads. It is equivalent to \(~53\%\) less the elapsed time compared to original CLOMP. Similarly, for CG benchmark, the original CLOMP is not scalable for class B, and shows a little speedup (\(~1.5\)) for class C. TReP and DReP shows a very little speedup for class B, and \(~2.2\) speedup for class C. In average, TReP and DReP save \(~29\%\) and \(~50\%\) in elapsed time for class B and C CG benchmarks respectively.

Moreover, for IS and LU benchmarks, none of the original CLOMP, TReP and DReP enabled CLOMP show speedup for class B. Since IS class B only last for a few second, utilizing TReP and DReP is not sufficient to make this program scale. For elapsed time, TReP and DReP does show \(~30\%\) less elapsed time. For the LU benchmark, utilizing TReP and DReP does significantly reduced the elapsed time by \(~40\%\), but this reduction is not sufficient to allow a speedup yet. On the contrary, with increasing problem size from class B to C, both TReP and DReP show scalability for IS and LU benchmarks. TReP and DReP both achieve \(~1.8\) speedup for IS class C on 4 threads, which stands for around \(~40\%\) less elapsed time. TReP and DReP show \(~1.5\) and \(~1.3\) speedup on 4 threads for LU class C, which is stands for \(~48\%\) and \(~39\%\) less elapsed time respectively. Once again, the scalability of prefetch techniques reflects the coverage and efficiency discussed in the previous section.

To summary the performance results observed for Gigabit Ethernet, for all NPB-OMP benchmarks, TReP and DReP significantly improve the scalability of the CLOMP system, and the larger improvement is achieved for the larger problem size. DReP shows comparable or better performance compared to TReP except BT and LU class C on 2 and 4 threads respectively.

C.3.2 Elapsed Time over DDR InfiniBand

DDR InfiniBand is around \(17x\) faster than Gigabit Ethernet in terms of bandwidth. Therefore, the benefit of data aggregation brought by utilizing TReP and DReP prefetch techniques is weakened. We would expect less performance improvement when DDR InfiniBand is used. Similar to the Gigabit Ethernet results, performance data is presented in terms of speedup in Figure C.3 and C.4.

For NPB-OMP benchmarks, the original CLOMP scales on most benchmarks except LU. Furthermore, we observe that TReP and DReP performs a little better than the original CLOMP for BT, CG, LU and IS class C benchmarks. For IS class B benchmark, TReP and DReP scale but not as good as the original CLOMP. This is because the introduced overhead for TReP and DReP is significant compared to the such small single thread elapsed time (4.8s).
To sum up, the performance results observed for DDR InfiniBand, TReP and DReP improves less performance compared to Gigabit Ethernet case. Moreover, for IS class B, TReP and DReP shows less speedup compared to the original CLOMP. DReP shows comparable or better performance compared to TReP except BT class C on 2 threads.
Figure C.4: Speedup of IS and LU benchmarks over DDR InfiniBand.
Appendix

D

MultiRail Networks
Optimization for the
Communication Layer

Contents

D.1 Introduction .................................................. 178
D.2 Micro-Benchmarks ............................................. 178
  D.2.1 Design Issues ........................................... 178
  D.2.2 Single-Rail Benchmark ............................ 179
  D.2.3 Multirail Benchmark ................................. 180
D.3 Bandwidth and Latency Experiments ...................... 181
  D.3.1 Experimental Setup .................................. 182
  D.3.2 Latency .............................................. 184
  D.3.3 Uni-directional Bandwidth ....................... 185
  D.3.4 Bi-directional Bandwidth ......................... 186
  D.3.5 Elapsed Time Breakdown ......................... 188
D.4 Related Work on Multirail InfiniBand Network ........ 190
D.5 Challenge and Conclusion ................................. 191
Appendix D: MultiRail Networks Optimization for the Communication Layer

D.1 Introduction

This chapter investigates multirail technique [20, 62, 94] over InfiniBand [46] interconnects with uDAPL [57, 22]. Two approaches, threaded and non-threaded, are proposed. uDAPL bandwidth benchmarks are developed for the proposed approaches, and are evaluated on an InfiniBand connected Intel cluster.

D.2 Micro-Benchmarks

Since the improvement on bandwidth is the major advantage brought by “multirail networks”, we have developed two different approaches, threaded and non-threaded, to achieve the bandwidth improvement on the multirail networks. The corresponding benchmarks have been implemented for each approach. RDMA operations are utilized to transfer the primary data in the benchmarks. Design issues related to these benchmarks and the details of their implementation will be discussed in the following sub-sections.

D.2.1 Design Issues

DAPL recognizes different ports as different DAT InfiniBand devices, even if those ports are physically located in the same adapter. Therefore, we can utilize one Interface Adapter (IA) object for each device. The benchmark for the single-rail configuration will just open a single IA per node, while the benchmarks for the multirail configurations will need to open multiple IAs per node.

Correspondingly, other objects, such as protection zone (PZ), endpoint (EP), public service point (PSP), event dispatcher (EVD) and local/remote memory region (LMR/RMR), will need to be created for each IA. LMRs/RMRs for different IAs can be registered to the same memory region, which allows fast data storing and avoids data moving operations at the remote site after a multirail RDMA write operation.

Due to the event-driven mechanism and data transfer operation mechanism utilized in uDAPL, the timed section would include a phase of waiting for notification events. Moreover, to explore the peak bandwidth of using uDAPL over multirail InfiniBand networks, we RDMA-transferred a chunk of data multiple times. The completion event suppress flag is utilized to avoid multiple RDMA DTO completion events. The benchmarks only waits for the events from the notification messages sending after all RDMA DTOs. The details of implementation for the different benchmarks will be described and discussed in the rest of this section.

1The source code of these benchmarks are available at http://ccnuma.anu.edu.au/dsm/multirail.
D.2 Micro-Benchmarks

D.2.2 Single-Rail Benchmark

A single-rail bandwidth benchmark was implemented to measure the base line bandwidth. In the benchmark, a single IA and a single set of leaf objects (EP, PZ, LMR/RMR) are created at both the server (local) and the client (remote) sides. A single PSP object is created at the server side for connection establishment.

As showed in D.1, the benchmark contains a major loop which is looping for different data sizes (N) from 0 bytes to 2MB. Within the loop, the server side RDMA writes data 20 times followed by a message sent to the remote side. Then the server issues a receive operation to receive an acknowledgment back from the client, and a event-wait to wait for the completion event of this receive operation. The capture of the event indicates the RDMA-written data is visible at the client side.

At the client side, for each different data sizes, the client needs to issue a message receive operations to match the message send at the server side, and wait for an event indicating the completion of this receive operation. Then the client issues an acknowledgment to the server side informing that the data is visible at user's buffer.

Server (local) Side:
Loop for data size (N) from 0 bytes to 2 Mbytes
Start Timer (t1)
Loop from 0 to 20
    dat_ep_post_rdma_write() with suppress flag
End Loop
dat_ep_post_send() with suppress flag
dat_ep_post_recv() matching client post send
dat_evd_wait() wait for event of above recv completion
Stop Timer (t2)
Bandwidth_N = 20*N/(t2-t1)
End Loop

Client (remote) Side:
Loop for data size (N) from 0 bytes to 2 Mbytes
dat_ep_post_recv() with default flag
dat_evd_wait() wait for event for above recv completion
dat_ep_post_send() with suppress flag to indicate server RDMA data is visible on the client side.
End Loop

Figure D.1: Single-rail bandwidth benchmark

A timer has been inserted on the server side. If the time spent on all above operations on server side is $\Delta T$, then we have the bandwidth for RDMA write of data of length N as $20N/\Delta T$ on a single-rail InfiniBand network.
D.2.3 Multirail Benchmark

The basic idea to implement the multirail benchmark is to split the data to be RDMA transferred into two even parts, with the RDMA sending one part through one rail and another part through another rail simultaneously.

According to our previous discussion in section D.2.1, different IAs need to be opened for different DAT InfiniBand devices. For each IA, we need to create an EP, a PSP, a PZ and a LMR/RMR for it. However, uDAPL supports that the LMR/RMR for different IAs can be registered to the same memory region, in which the send/recv buffer has been allocated, as shown in figure D.2.

![Multirail communication memory access pattern](image)

**Figure D.2:** Multirail communication memory access pattern.

The top half of the send/recv buffer is handled by rail 0, and the bottom half of send/recv buffer is handled by rail 1.

The multirail benchmarks have been implemented with two different approaches. The first approach utilizes a single process at both the server and the client sides to handle the communication through both rails, while the second
D.3 Bandwidth and Latency Experiments

approach utilizes one thread per rail, and explores the full parallelism of data sending through different rails.

D.2.3.1 Non-Threaded Approach

By inserting loops over different rail IDs for every uDAPL communication and event wait operations at both the server and the client side of the single-rail benchmark, we create the the non-threaded multirail bandwidth benchmark, as shown in figure D.3.

The communications on both rails are through the loops to achieve the “simultaneous” data transfer. The term “simultaneous” does not mean the data transfer through both rail happened at exactly the same time. The RDMA function calls are actually serialized; however, the real data transfer operations are maximally overlapped.

Although the overlapping of communication on different rails boosts the benchmark performance, the waiting for DTO completion events are still fully serialized and cannot be overlapped. This is the major overhead for the non-threaded approach. This problem will be solved by utilizing the threaded approach as discussed in next section.

D.2.3.2 Threaded Approach

To avoid the overhead introduced by the non-threaded approach, we use OpenMP threads to fully parallelize the communication and event waiting on the different rails. The details are shown in figure D.4.

Bi-directional benchmarks are created based on above uni-directional benchmark with data sending happened at both direction (server→client and client→server).

D.3 Bandwidth and Latency Experiments

In this section, we present performance results of our uDAPL non-threaded and threaded benchmarks on an InfiniBand cluster with different multirail configurations. Both the uni-directional and the bi-directional bandwidth benchmarks are employed. The bi-directional benchmarks required trivial extensions based on the uni-directional benchmarks. The latency results were obtained by running the uni-directional benchmark with short message sizes. These benchmarks will
Appendix D: MultiRail Networks Optimization for the Communication Layer

Server (local) Side:

Loop for data size (N) from 0 bytes to 2 Mbytes

Start Timer (t1)

Loop from 0 to 20
   Loop for different rails from 0 to 1
      rdma_write() N/2 data with suppress flag
   End Loop
End Loop

Loop for different rails from 0 to 1
dat_ep_post_send() with suppress flag
End Loop

Loop for different rails from 0 to 1
dat_ep_post_recv() matching client post send
dat_evd_wait() wait for event of above recv completion
End Loop

Stop Timer (t2)
Bandwidth_N = 20*N/(t2-t1)
End Loop

-----------------------------------------------

Client (remote) Side:

Loop for data size (N) from 0 bytes to 2 Mbytes

Loop for different rails from 0 to 1
dat_ep_post_recv() with default flag
dat_evd_wait() wait for completion event for recv
End Loop

Loop for different rails from 0 to 1
dat_ep_post_send() with suppress flag to indicate server RDMA data is visible on the client side.
End Loop

End Loop

Figure D.3: Non-threaded multirail bandwidth benchmark

be evaluated on the multirail network configurations (a) and (b), as described in section D.4.

D.3.1 Experimental Setup

Our experimental InfiniBand cluster consists of four Sun Ultra24 workstations, which contain one Intel Core 2 Quad Q6600 CPU with 4GB DDR2 memory on each.
D.3 Bandwidth and Latency Experiments

![Threaded multirail benchmark design](image)

Figure D.4: Threaded multirail benchmark design.

Each Sun Ultra24 workstation contains two x16 PCI-e Gen2 slots, which supports up to 4GB/s bandwidth for one x8 PCI-e 2.0 connectors. Additionally, the Sun Ultra24 workstation supplies two DDR2 memory channels, connected to the two x16 PCI-e Gen2 slots separately.

The Mellanox ConnectX MHGH28-XTC dual-port HCA has a host bus speed at PCIe 1.1 speed, which is 2.0GB/s. The InfiniBand speed of each port in MHGH28-XTC HCA is 4x DDR InfiniBand, which is 2.0GB/s (peak).

Based on equations (D.1) and (D.2), the theoretical peak uni-bandwidth on the multi-HCA configured InfiniBand cluster is 4GB/s, and the theoretical peak uni-bandwidth on the multi-port configured InfiniBand cluster is 2.0GB/s. By contrast, the peak uni-directional bandwidth for the single-rail network is limited by the InfiniBand speed (or HCA PCI speed), which is 2.0GB/s.

Moreover, for the threaded benchmark, the two different threads has been set to have affinity to cores 0 and 3 respectively on each node.
Appendix D: MultiRail Networks Optimization for the Communication Layer

D.3.2 Latency

As we have mentioned in the beginning of this section, the uni-bandwidth benchmarks are utilized to collect the short message (from 8 Bytes to 4KB) latency data on different network configurations. Figure D.5 shows the RDMA write latency ($\mu s$) for the non-threaded and the threaded benchmarks on three different network configurations: single-rail, multi-port and multi-HCA. However, the latency on the single-rail network is only measured with the non-threaded benchmark.

![Short Message Latency for RDMA Write](image)

**Figure D.5:** RDMA write latency comparison.

In figure D.5, the latencies for the threaded benchmark on the multi-HCA and the multi-port configurations are quite comparable with the single-rail latency, in the range of 2$\mu s$ to 4$\mu s$. The latencies for the multi-port threaded benchmark are slightly higher than single-rail latencies, which maybe due to hardware contention on the single HCA processor. On the contrary, when the message size is smaller than 128 bytes, the latencies for the multi-HCA threaded benchmark are a little higher ($\sim 0.5\mu s$) than the single-rail latency. When the message size is larger than 128 bytes, the latencies of the multi-HCA threaded benchmark are less than the latency of the single-rail case. The threaded benchmark with the multi-HCA configuration achieved 2.7$\mu s$ for a 4KB message, which is $\sim 0.62\%$ of the single-rail latency.

The non-threaded benchmark shows significant higher latencies ($>10\mu s$) for both the multi-port and the multi-HCA configurations. This is due to the overhead brought by serializing RDMA function calls, and event waiting. Hardware
D.3 Bandwidth and Latency Experiments

Contention affects are more marked for the non-threaded approach.

In terms of performance improvement, the threaded approach starts showing distinct improvement when the data size larger than 256 bytes on the multi-HCA configuration and 1KB on the multi-port configuration. The non-threaded approach does not show any improvement on the small messages with both configurations.

D.3.3 Uni-directional Bandwidth

Figures D.6 and D.7 show the uni-directional bandwidth comparison between the non-threaded, the threaded and the single-rail bandwidth on the multi-port and the multi-HCA configurations. The bandwidth results are measured from 8 bytes to 2MB. On the single-rail InfiniBand network, the uni-directional bandwidth peaks at $\sim 1350$MB/s.

![Uni-Directional RDMA Write Two Ports Bandwidth](image)

**Figure D.6:** Uni-directional multi-port bandwidth.

As the theoretical bandwidth of the multi-port configuration is 2.0GB/s, which is the same as the single-rail bandwidth limit. The non-threaded multirail benchmark shows around the same peak bandwidth on the multi-port configured InfiniBand network as on the single-rail, see figure D.6. However, when the data size is less than 2MB, the non-threaded multirail benchmark shows lower bandwidth than the single-rail. This is due to hardware and software contention with a single thread to service two different network interface communications.
Appendix D: MultiRail Networks Optimization for the Communication Layer

On the contrary, the threaded benchmark with the multi-port configuration starts achieving the bandwidth improvement when message size is larger than 4KB, and shows the peak bandwidth at \( \sim 1800 \text{MB/s} \) between 32KB and 2MB. An \( \sim 33\% \) improvement on bandwidth is achieved.

![Uni-Directional RDMA Write Two HCAs Bandwidth](image)

**Figure D.7:** Uni-directional multi-HCA bandwidth.

In figure D.7, on the multi-HCA configured InfiniBand cluster, the non-threaded approach starts achieving a bandwidth improvement from 8KB, and the threaded approach starts achieving bandwidth improvement from 512 bytes. Moreover, the non-threaded approach reaches the peak uni-directional bandwidth at \( \sim 2600 \text{MB/s} \) and the threaded approach reaches the peak uni-directional bandwidth at \( \sim 3350 \text{MB/s} \). The non-threaded approach improves the uni-directional bandwidth by \( \sim 90\% \), and surprisingly an improvement of \( \sim 148\% \) is achieved with the threaded approach.

By comparing two different multirail configurations, the non-threaded approach achieves \( \sim 90\% \) higher bandwidth on the multi-HCA network than on the multi-port network, and the threaded approach achieves \( \sim 89\% \) higher bandwidth on the multi-HCA network than on the multi-port network.

### D.3.4 Bi-directional Bandwidth

Figures D.8 and D.9 show the bi-directional bandwidth comparison between the non-threaded and the threaded approach on the multi-port and the multi-HCA configurations. Similar to the uni-directional bandwidth tests, the bandwidth is
D.3 Bandwidth and Latency Experiments

measured between 8 bytes and 2MB data sizes. The bi-directional bandwidth on single-rail InfiniBand cluster reaches the peak at $\sim 2350$MB/s with 2MB data size.

![Figure D.8: Bi-directional multi-port bandwidth.](image)

A similar trend has been observed in figure D.8 for the bi-directional multi-port bandwidth as the uni-directional multi-port bandwidth results shown in figure D.6. The non-threaded approach does not show any improvement; however, the threaded approach shows around 34% improvement for the peak bi-directional bandwidth at 3150MB/s reached at 2MB data size.

In figure D.9, both the non-threaded and the threaded approaches achieve bandwidth improvement on the multi-HCA configuration when the data size is larger than 32KB and 1KB respectively. The non-threaded approach achieved the peak bi-directional bandwidth at $\sim 3950$MB/s and $\sim 68\%$ improvement when the data size is larger than 32KB. By contrast, the threaded approach achieved the peak bi-directional bandwidth at $\sim 5045$MB/s and an improvement $\sim 114\%$ at a data size of 128KB. Then the bandwidth decreased to 4450MB/s for 2MB message with some fluctuation.

According to figures D.8 and D.9, around 68% more bi-directional bandwidth was obtained on the multi-HCA network by the non-threaded approach compared to the multi-port network, and around 60% more bi-directional bandwidth was achieved by the threaded approach for the multi-HCA configuration as well.
Appendix D: MultiRail Networks Optimization for the Communication Layer

Bi-Directional RDMA Write Two HCAs Bandwidth

![Graph showing bandwidth vs data size for Single-rail, Non-threaded multi-HCA, and Threaded multi-HCA.]

Figure D.9: Bi-directional multi-HCA bandwidth.

D.3.5 Elapsed Time Breakdown

To understand the different behaviours of the non-threaded and the threaded approach, we break the elapsed time for running the uni-directional benchmarks on the multi-HCA configured cluster down into two different parts: the time for RDMA write function return, and the time for busy waiting on DTO completion events. Here, the elapsed time is the total time spent on transferring a fixed sized message 20 times with notifications. Two different data sizes, 512 bytes and 4KB, are used for the following demonstration.

![Bar chart showing elapsed time for Single-rail, threaded, and non-threaded approaches.]

Figure D.10: Benchmarks elapsed time breakdown for 512bytes message.
D.3 Bandwidth and Latency Experiments

Figure D.10 shows the breakdown for the elapsed time on transferring a 512 bytes message 20 times. The time spent on RDMA function return is 10µs and 7µs for the single-rail and the threaded approach respectively, which are comparable. The non-threaded approach spends ~15µs on RDMA function return, which is due to the serialized RDMA function calling procedure utilized in non-threaded benchmark design.

36µs and 35µs are spent on waiting event for the single-rail and the threaded approach respectively. For any message smaller than 512 bytes, the threaded approach does not show much improvement on performance, due to the overheads on joining threads and ending a parallel region. Moreover, 184µs is spent on waiting for events for the non-threaded approach, which is around four times more than the single-rail. This is because that the event waiting on different rails is fully serialized in non-threaded design, although some portion of data transferring is overlapped.

![Bar Chart](image)

**Figure D.11**: Benchmarks elapsed time breakdown for 4KB message.

Figure D.11 shows the breakdown for the elapsed time on transferring a 4KB message 20 times. The threaded approach spends 4µs less on waiting for RDMA return than the single-rail (9µs), while the non-threaded approach spends 14µs on waiting RDMA return. This is quite similar to the 512 bytes case.

However, for the 4KB message, the single-rail test spends 78µs on waiting for DTO completion events. As the data transfer is fully parallelized, the threaded approach spends 45µs on event waiting. On the contrast, the non-threaded approach spends 197µs on event waiting, which is more than twice that of the single-rail, and this ratio is much less than the 512 bytes case. This is because the larger data size allows more communication overlap.
D.4 Related Work on Multirail InfiniBand Network

Multirail refers to a network containing multiple physical connections or “rails” [20]. Utilizing multirail for cluster communications can significantly improve network bandwidth and communication efficiency.

As some current InfiniBand host channel adapters (HCA) contain multiple ports, the InfiniBand multirail network can be configured in different ways, as shown in figure D.12 [62].

![Diagram](image.png)

**Figure D.12:** Different ways to configure a InfiniBand multirail network [62].

As illustrated in figure D.12, there are three different configurations of InfiniBand multirail networks. There are multiple HCAs employed for each node at configuration (a). For configuration (b) a single HCA with multiple ports (SHMP) is employed for each node. Configuration (c) establishes a multirail network at an abstract level via software design. For configurations (a) and (b), physical “rails” are actually established, which is the real “multirail”. For configuration (c), only abstraction sub-channels are established on a single physical connection.

There are different theoretical bandwidth upper bounds for different multirail configurations. In general, the theoretical bandwidth for configuration (a) can be formulated as follows:

\[
B_a = N \times \min(B_{IB}, B_{HCA}, B_{host})
\]  

(D.1)
D.5 Challenge and Conclusion

In equation (D.1), $B_a$ stands for theoretical bandwidth for configuration (a), $N_r$ stands for number of rails, $B_{IB}$ stands for InfiniBand speed, $B_{HCA}$ stands for HCA PCI speed, and $B_{host}$ stands for host machine PCI speed. For example, if we use two 4x DDR x8 PCI-e 1.0 InfiniBand HCAs with a node equipped with PCI-e 2.0 slots, $B_{IB} = 2.0$GB/s [66], $B_{HCA} = 2.0$GB/s, and $B_{host} = 4$GB/s. Hence, $B_a = 4$GB/s$^2$.

For configuration (b), the theoretical bandwidth could be formulated as equation (D.2).

$$B_b = \min(N_r B_{IB}, B_{HCA}, B_{host})$$  \hspace{1cm} (D.2)

Taking the same example as above, the theoretical bandwidth for configuration (b) is limited by HCA PCI speed, which is 2.0GB/s.

For configuration (c), the theoretical bandwidth is shown in equation (D.3).

$$B_c = \min(B_{IB}, B_{HCA}, B_{host})$$  \hspace{1cm} (D.3)

Taking the same example as above, the theoretical bandwidth for configuration (c) is limited by either $B_{IB}$ or $B_{HCA}$, which is 2.0GB/s.

Based on the above analysis, generally, configuration (c) has the lowest theoretical peak bandwidth. When $N_r B_{IB}$ is larger than $B_{HCA}$, $B_{HCA}$ becomes the limitation. In this case, although the theoretical peak bandwidth of the configuration (c) may be as the same as the configuration (b), software and hardware contention on configuration (c) has more effect [62]. Therefore, in this appendix, we will only consider the first and second configurations of InfiniBand multirail networks.

D.5 Challenge and Conclusion

In this Appendix, we have developed the non-threaded and the threaded approaches with uDAPL to explore the bandwidth improvement on multirail networks. The results of the non-threaded approach are quite similar with MVAPICH2 InfiniBand Verb multirail implementation on the multi-HCA configurations. As some different InfiniBand HCAs were used, MVAPICH2 achieved some improvement on the non-threaded approach with the multi-port configuration as well [62, 94]. However, in our setup, utilizing a portable communication library does not prevent the performance improvement over multirail networks.

$^{2}$8B/10B encoding has been used for both PCI-e 2.0 or lower and InfiniBand data transmission.
Appendix D: MultiRail Networks Optimization for the Communication Layer

In summary, an \( \sim 33\% \) and \( \sim 148\% \) improvement for the large message uni-directional bandwidth is achieved by the threaded approach on the multi-port and the multi-HCA configured cluster respectively. No improvement and 90\% improvement for the large message uni-directional bandwidth is achieved by the non-threaded approach on the multi-port and the multi-HCA configured cluster respectively. A similar pattern of improvement has been achieved for the bi-directional bandwidth tests for both the threaded and the non-threaded approaches.

Since the threaded approach fully parallelizes the data transfer on the different rails, it shows significantly better improvement than the non-threaded approach. As utilizing the multiple HCAs increases the theoretical peak system bandwidth, both the threaded and the non-threaded approaches achieved the better improvement than on the multiple port configured network, in our hardware setup. Therefore, the best way to achieve the bandwidth improvement is to use the threaded approach over a multi-HCA configured network.

The multirail techniques can be used to leverage higher bandwidth provided by high performance interconnects, which can be potentially further improve the performance of cluster OpenMP systems. This requires significantly design and development effort to change the whole communication (CAL) layer of CLOMP into multirail aware to maximize the communication overlapping.
Appendix

E

Performance of CAL

Contents

E.1 Bandwidth and Latency of CAL . . . . . . . . . . . . . . . . 194
E.2 Comparison Between OpenMPI and CAL . . . . . . . . . 194
Table E.1: Complete bandwidth and latency measured by the communication layer (CAL) of CLOMP on XE.

<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>XE GigE Bandwidth (MB/s)</th>
<th>Latency (µs)</th>
<th>XE DDR IB Bandwidth (MB/s)</th>
<th>Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.6</td>
<td>87.7</td>
<td>5.4</td>
<td>11.3</td>
</tr>
<tr>
<td>128</td>
<td>1.4</td>
<td>88.5</td>
<td>10.6</td>
<td>11.5</td>
</tr>
<tr>
<td>256</td>
<td>2.6</td>
<td>91.4</td>
<td>20.8</td>
<td>11.8</td>
</tr>
<tr>
<td>512</td>
<td>5.2</td>
<td>92.6</td>
<td>38.4</td>
<td>12.7</td>
</tr>
<tr>
<td>1024</td>
<td>8.8</td>
<td>111.6</td>
<td>70.4</td>
<td>13.9</td>
</tr>
<tr>
<td>2048</td>
<td>18.6</td>
<td>105.2</td>
<td>124.2</td>
<td>15.8</td>
</tr>
<tr>
<td>4096</td>
<td>33.0</td>
<td>118.6</td>
<td>198.2</td>
<td>19.7</td>
</tr>
<tr>
<td>8192</td>
<td>45.4</td>
<td>171.9</td>
<td>263.2</td>
<td>29.7</td>
</tr>
<tr>
<td>16384</td>
<td>42.6</td>
<td>366.7</td>
<td>347.2</td>
<td>45.0</td>
</tr>
<tr>
<td>32768</td>
<td>61.4</td>
<td>509.3</td>
<td>379.4</td>
<td>82.4</td>
</tr>
<tr>
<td>65536</td>
<td>68.8</td>
<td>907.4</td>
<td>484.8</td>
<td>129.0</td>
</tr>
<tr>
<td>131072</td>
<td>80.6</td>
<td>1551.2</td>
<td>479.0</td>
<td>261.0</td>
</tr>
<tr>
<td>262144</td>
<td>89.8</td>
<td>2783.3</td>
<td>446.4</td>
<td>560.0</td>
</tr>
<tr>
<td>524288</td>
<td>93.0</td>
<td>5377.2</td>
<td>521.8</td>
<td>958.2</td>
</tr>
<tr>
<td>1048576</td>
<td>95.2</td>
<td>10513.7</td>
<td>483.2</td>
<td>2069.9</td>
</tr>
<tr>
<td>2097152</td>
<td>94.6</td>
<td>20885.0</td>
<td>479.0</td>
<td>4178.5</td>
</tr>
</tbody>
</table>

E.1 Bandwidth and Latency of CAL

In this section, we measured bandwidth and latency of CAL using a pingpong test program packaged within Intel CAL library source code. The test program iterates through different message sizes on NCI NF XE cluster, which deploys both Gigabit Ethernet and double data rate InfiniBand networks, and the corresponding performance data is shown in Table E.1.

E.2 Comparison Between OpenMPI and CAL

This section compares performance, bandwidth and latency, between OpenMPI version 1.4.3 and CAL on the XE cluster. Table E.2 presents the comparison bandwidth and latency between CAL and OpenMPI on the GigE network, and Table E.3 presents that on DDR IB. The OpenMPI bandwidth and latency are measured using a pingpong MPI program.

Under similar experiments, CAL is around 18% less efficient on GigE than OpenMPI in terms of bandwidth. This number grows to around 60% on DDR IB. Unlike MPI for which the allocated buffer can be used for data transfer, CAL maintains number of arenas. All data to be transferred need to be copied into these arenas first. To send a message, CAL firstly packs it into a data transfer descriptor (DTD), which contains the raw data, destination, sequence number and
E.2 Comparison Between OpenMPI and CAL

<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>CAL GigE</th>
<th>OpenMPI GigE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bandwidth (MB/s)</td>
<td>Latency ($\mu s$)</td>
</tr>
<tr>
<td>64</td>
<td>0.6</td>
<td>87.7</td>
</tr>
<tr>
<td>128</td>
<td>1.4</td>
<td>88.5</td>
</tr>
<tr>
<td>256</td>
<td>2.6</td>
<td>91.4</td>
</tr>
<tr>
<td>512</td>
<td>5.2</td>
<td>92.6</td>
</tr>
<tr>
<td>1024</td>
<td>8.8</td>
<td>111.6</td>
</tr>
<tr>
<td>2048</td>
<td>18.6</td>
<td>105.2</td>
</tr>
<tr>
<td>4096</td>
<td>33.0</td>
<td>118.6</td>
</tr>
<tr>
<td>8192</td>
<td>45.4</td>
<td>171.9</td>
</tr>
<tr>
<td>16384</td>
<td>42.6</td>
<td>366.7</td>
</tr>
<tr>
<td>32768</td>
<td>61.4</td>
<td>599.3</td>
</tr>
<tr>
<td>65536</td>
<td>68.8</td>
<td>907.4</td>
</tr>
<tr>
<td>131072</td>
<td>80.6</td>
<td>1551.2</td>
</tr>
<tr>
<td>262144</td>
<td>89.8</td>
<td>2783.3</td>
</tr>
<tr>
<td>524288</td>
<td>93.0</td>
<td>5377.2</td>
</tr>
<tr>
<td>1048576</td>
<td>95.2</td>
<td>10513.7</td>
</tr>
<tr>
<td>2097152</td>
<td>94.6</td>
<td>20885.0</td>
</tr>
</tbody>
</table>

Table E.3: Comparison of CAL and OpenMPI: bandwidth and latency measured on XE via DDR IB.

<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>CAL DDR IB</th>
<th>OpenMPI DDR IB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bandwidth (MB/s)</td>
<td>Latency ($\mu s$)</td>
</tr>
<tr>
<td>64</td>
<td>5.4</td>
<td>11.3</td>
</tr>
<tr>
<td>128</td>
<td>10.6</td>
<td>11.5</td>
</tr>
<tr>
<td>256</td>
<td>20.8</td>
<td>11.8</td>
</tr>
<tr>
<td>512</td>
<td>38.4</td>
<td>12.7</td>
</tr>
<tr>
<td>1024</td>
<td>70.4</td>
<td>13.9</td>
</tr>
<tr>
<td>2048</td>
<td>124.2</td>
<td>15.8</td>
</tr>
<tr>
<td>4096</td>
<td>198.2</td>
<td>19.7</td>
</tr>
<tr>
<td>8192</td>
<td>263.2</td>
<td>29.7</td>
</tr>
<tr>
<td>16384</td>
<td>347.2</td>
<td>45.0</td>
</tr>
<tr>
<td>32768</td>
<td>379.4</td>
<td>82.4</td>
</tr>
<tr>
<td>65536</td>
<td>484.8</td>
<td>129.0</td>
</tr>
<tr>
<td>131072</td>
<td>479.0</td>
<td>261.0</td>
</tr>
<tr>
<td>262144</td>
<td>446.4</td>
<td>560.0</td>
</tr>
<tr>
<td>524288</td>
<td>521.8</td>
<td>958.2</td>
</tr>
<tr>
<td>1048576</td>
<td>483.2</td>
<td>2069.9</td>
</tr>
<tr>
<td>2097152</td>
<td>479.0</td>
<td>4175.5</td>
</tr>
</tbody>
</table>

other information. Secondly, this descriptor is sent to the destination. At the
destination, CAL reads the incoming DTD to find the initiator of the message, and
builds an response DTD and sends the DTD back to the initiator to acknowledge
the reception of the message.


Bibliography


[34] Hiroshi Harada, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Toshiyuki Takahashi, and Yutaka Ishikawa. SCASH: Software DSM using high performance network on commodity hardware and software. In *Eighth


[74] Open MPI. *Open MPI v1.4.1 Documentation*, 2009.


BIBLIOGRAPHY


