Adaptive Work Stealing Run-times for NUMA Multi-core Architecture

A dissertation submitted to the University of Hyderabad in partial fulfilment of the requirements for the award of degree of

Doctor of Philosophy

in

Computer Science

by

B. Vikranth
08MCPC07

School of Computer and Information Sciences
University of Hyderabad
Hyderabad-500046

June, 2018

Acknowledgement

It gives me immense pleasure to express my deepest sense of gratitude and sincere thanks to highly respected and esteemed supervisors **Prof. Rajeev Wankar** and **Prof. C. Raghavendra Rao** for their continuous guidance, support and encouragement in carrying out this work. I consider myself highly privileged to be mentored by them. Their vast knowledge and experience has immensely benefited me.

I am grateful to the DRC members Prof. Arun Agarwal and Prof. Alok Singh for directing me in research with their valuable suggestions in meetings.

I am grateful to the Dean, SCIS for providing excellent computing facilities and a congenial atmosphere for progressing with my research work.

I express my sincere gratitude to the management of CVR College of Engineering for their support in carrying out my research work.

I would like to express my sincere thanks to all my friends, lab colleagues and others who have helped directly or indirectly during this dissertation work. I also express my thanks to my family members without whose support I could continue my research.

B. Vikranth
(08MCPC07)

Contents

List of Figures

List of Tables

Abbreviations

ARMCI	Aggregate Remote Memory Copy Interface
BOTS	Barcelona OpenMP Task Suit
CFS	Completely Fair Scheduler
CSS	Chunk Self Scheduling
DAG	Directed Acyclic Graph
DRAM	Dynamic Random Access Memory
FSB	Front Side Bus
GSS	Guided Self Scheduling
HT	Hyper Threading
HT-links	Hyper Transport Links
LLC	Last Level Cache
IMC	Integrated Memory Controller
IPC	Instructions Per Cycle
KLT	Kernel Level Threads
LWP	Light Weight Process
McRT	Multi-core Run-Time
MPI	Message Passing Interface
MSMC	Multi-Socket Multi-Core Architecture
MRC	Miss Rate Curves
NUMA	Non Uniform Memory Architecture
PGAS	Partitioned Global Address Space
QPI	Quick Path Interconnect
RTID	Related Thread Identifiers
SMP	Symmetric Multi Processing
TATL	Topology Aware Task-stealing Library
TLB	Translation Look aside Buffer
TLD	Task Level Dispatcher
ULT	User Level Thread
UMA	Uniform Memory Architecture
VM	Virtual Memory, Virtual Machine

Chapter 1

Introduction

1.1 Motivation

The work done in this thesis was motivated by the developments in processor architecture, performance studies and the functionality of user-level runtime systems such as Cilk [3] and TBB [4]. These run-times were developed to support task based parallelism. These run- times support work stealing strategy for load balancing. The features of these run-times are targeted for first generation multi-core processors with common bus interface to memory. The strategies proposed in Chapter 3 aim to improve the performance of work stealing run-time for such an architecture. But the next generation server processors started supporting on-chip-interconnect technology such as Quick Path Interconnect links from Intel (QPI) [5] and Hyper Transport links [6] from AMD. These systems typically behave as non uniform memory architecture (NUMA) but the task based run-times (Cilk,TBB etc.) continue to support the same shared memory paradigm in parallel programming. The locality issues to support process based and thread based programming in such an environment were well addressed at operating system level and process based programming in [7] [1] [8]. This work induced new research avenues and allowed our study on user level run-times in general and task based run-times in particular. In this thesis, we proposed strategies to adapt user level run-times in NUMA multi-core architectures. Our contributions are mainly in load balancing and locality aspects of work-stealing infrastructure as depicted in the figure 1.1.

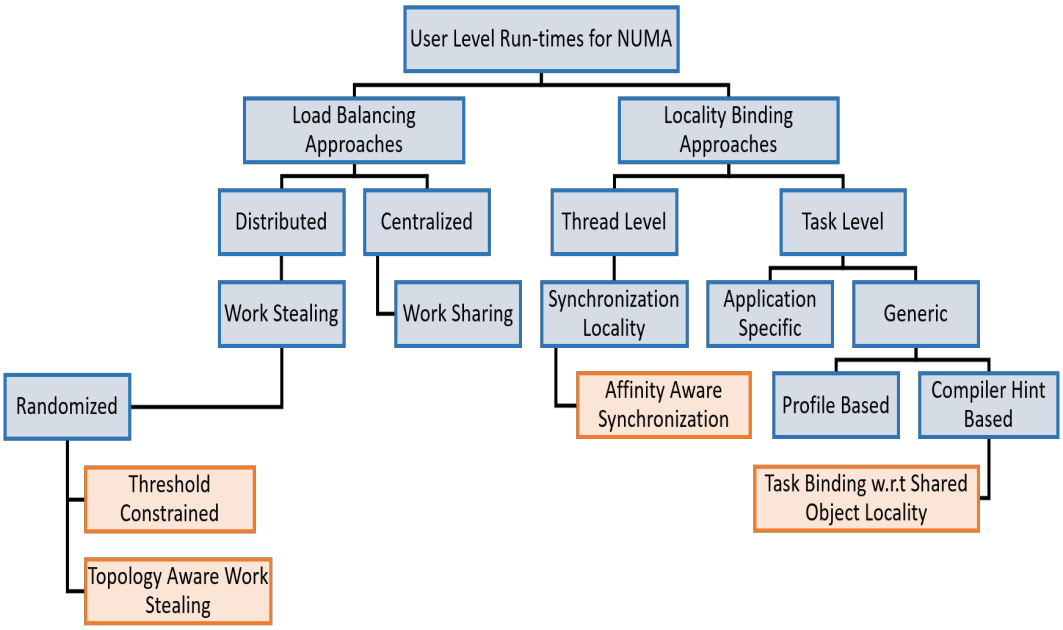

Figure 1.1: Thesis taxonomy

1.2 Parallel Programming Paradigms

In this section, we explain the reason why our work is focused on task-based parallelism briefing the parallel programming paradigms.

- **Process based:** Splitting the given job into multiple processes using $fork()$ like call. MPI programming is process based parallel programming paradigm where the communication among processes is taken care using active messages.

- **Thread based:** Within a single process, split the job into multiple threads using pthread API or Java threads. Threads live within a single address space and take the advantage of shared memory communication avoiding the message passing delays. Each thread can access the shared address space of the same process.

- **Task based:** Since creation of thread is also of some overhead, new run-time entities called tasks are introduced at user level run-times. Tasks are considered even lighter than threads. Tasks are in turn mapped to threads or virtual processors which are created during the initialization of user level run-time systems. Run-time systems of paral-

lel programming environments like OpenMP, Cilk, TBB are few examples of task based run-time system. Our focus in this thesis is mainly on task based user-level run-times.

1.3 Task based run-times

Since thread creation is of some overhead due to its resource consumption, a group of threads equivalent to the number of hardware threads is kept ready during the initialization of the run-time and behave as a set of virtual processors to execute parallel execution entities, we call it a thread pool. *"A thread pool is a set of pre-instantiated, idle threads which stand ready to be given work"*. Creating a task on fly for each job is a preferred approach over instantiating a new thread on fly when there is a large number of short jobs to be done rather than a small number of long ones.

`Task` is a unit of execution which is considered far lighter than `thread`. A task is about 20 times lighter than thread on Linux environment and 100 times lighter on Windows. Task programming model can be effectively applied for the code with less amount of blocking statements. Task based run-times have gained much importance in multi-core era. Multi- threaded programming is more effective when the number of logical threads in thread-pool is equal to the number of processors (cores or hardware threads). Though multi-threading is a traditional way of programming shared memory, the following disadvantages are observed in it.

- Undersubscription: occurs when the number of program instantiated logical threads is less than the number of cores at hardware level.
- Over subscription: occurs when the number of program created logical threads are more than available hardware threads. In this case, logical threads follow some kind of multiplexing such as time slicing. These create overhead of context switching. Hence, thread scheduling is dependent on operating system's scheduling strategy. In case of Linux, threads are scheduled guaranteeing *fairness*. Fairness can be a hindrance for performance. If tasks are used, intelligent scheduling and load balancing strategies to guarantee performance can be applied.
- Heaviness: Common resources of threads are instruction pointer, copy of registers and stack. On Linux, a thread is also created using *clone()* system call. Hence thread needs additional resources depending on

implementation making them heavy though they are of less weight than process.

Hence task based programming can perform well because they are purely implemented at user level implementations and can do better in the following areas [9]:

- Matching parallelism to available execution resources
- Less time consumption during task-start-up and task-shutdown
- Improved evaluation order
- Effective load balancing
- Higher level thinking

Having realized the importance of task based parallel programming environments, our initial focus was to explore the popular load balancing techniques like *work sharing* and *work stealing*. In the work sharing approach, all the application instantiated tasks are managed at centralized storage and dispatched to one of the available free processors. Hence, it is a centralized approach of load balancing. Worker threads of the thread pool execute the tasks. In the work sharing approach, worker threads contend for accessing the centralized task-queue every time a task is to be executed. To overcome the contention involved in centralized approach, a distributed approach called ***work-stealing*** strategy has been proposed where every worker has its own queue of tasks. A worker thread becomes a *thief* when it's own task queue is empty and tries to *steal* tasks from other queue called *victim*. In case of work stealing approach, *a worker thread needs to access other worker's queue only if it has no tasks in it's own queue*. Hence it is a distributed approach with less contention. This popular approach has been implemented in many task based run-time systems such as Cilk [3], TBB [4] and few implementations of OpenMP. The study of these run-times and rapid changes in hardware prompted us to contribute few extensions to work stealing infrastructures.

During the course of analyzing work stealing strategy and exploring the source code of the run-times, few gaps were identified related to victim selection method. This analysis initiated us to contribute extensions to present work-stealing approach. The method followed in these run-times is *randomized work stealing*. We could identify gaps in randomized approach that impair the overall performance of a task based application. In chapter 3,

we proposed a metric called *false steal count* that keeps track of those additional delays. Our proposed strategy in chapter 3, **threshold constrained work stealing** attempts to mitigate the effects of those delays in randomized stealing approach.

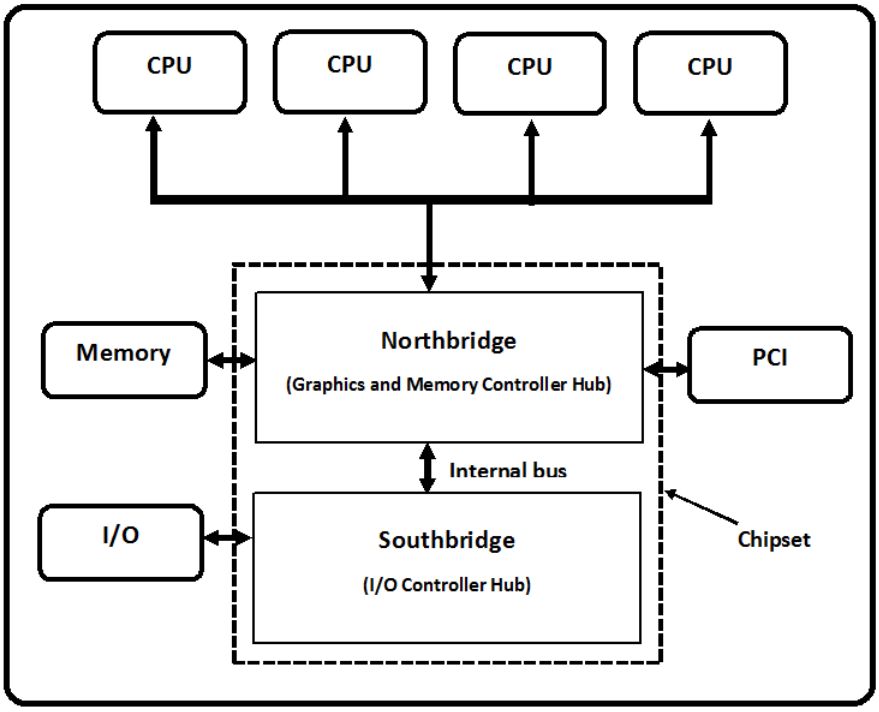

Figure 1.2: Multi-core Architecture with common bus interconnect

While proposing this idea, our assumption about the underlying architecture was *"Multi-core architecture with shared common bus among the cores"* as illustrated in the figure 1.2. Shared common bus was the prominent micro-architectural trend in multi-core design when the number of cores on chip was a small number. But such architectures suffer from *memory wall problem* which means high speed cores interfacing with low speed memory. Hence, the processor designers started introducing on-chip memory controllers and separate memory connected to each integrated memory controller (IMC). The processors following this new micro architecture are called *NUMA multi-core processors* or *On-Chip NUMA multi-core processors*. Nehalem[10] and later architectures from Intel and Opteron series from AMD started supporting multiple integrated memory controllers on chip during the year 2010 and later. These architectures behave as *NUMA in single machine*. But the distributed programming models such as message passing (MPI) or Partitioned

Global Address Space(PGAS) can't be applied to these architectures since the interconnections among cores are high speed links such as quick path interconnect (QPI) or Hyper Transport Links. The same shared memory programming paradigm is followed at user level run-time layers only with the difference being that the logical address space of a single process is now spanned among different physical memory nodes. The figure 1.3 illustrates such an example Nehalem micro architecture. A commercial NUMA system today is available on single board in the form of multi-socket machine. In a NUMA system, memory is classified into two or more NUMA nodes. A typical high-performance server today contains two or more sockets and will, therefore, have more than one NUMA nodes in a single machine. Memory access latency within a node is approximately 100 ns and memory access of all cores on same chip exhibit same access speeds. Memory access latency to remote node memory is more than local memory access latency by 50-75 percent. The processing elements (cores) of the same NUMA node can access memory with the best performance since they are locally attached. Memory is said to be *node local* if it was allocated on the NUMA node which is nearest to the processor. Processing elements of other node suffer from access delays (NUMA penalty) when they access data from other NUMA nodes.

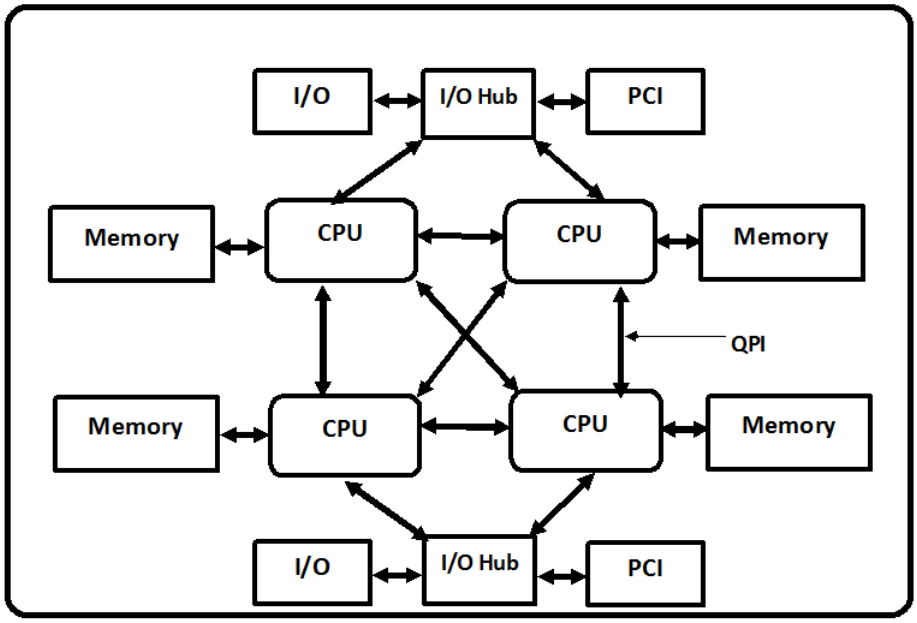

Figure 1.3: An example NUMA Multi-core architecture [1]

The *same shared memory model with on-chip-NUMA-multi-core hardware environment* attracted our focus towards further improving randomized-

work-stealing strategy. Work stealing environment supports *identity affinity policy* (as a default policy or as a configurable feature) which means that there is one to one mapping between worker threads and processors. If randomized work stealing is applied in NUMA multi-core environment, the randomly selected victim may refer a worker pinned to a core belonging to other node. This additional requirement motivated us to add **topology awareness of NUMA architecture** in work stealing environment which is presented in the chapter 4. In chapter 4, we also introduce a metric *remote steal count* that can measure how many randomly chosen attempts lead to remote victim reference. An important finding of this chapter is the relationship between remote steal count and the execution time of an application in the work stealing run-time environment. A strategy called stealing domains is introduced to limit the stealing activity mostly within a node. Topology aware work stealing strategy with stealing domains could show significant performance gain when compared to threshold constrained work staling-strategy.
NUMA multi-core environment also introduces locality issues in user level run-times. In Linux kernel, approach of assigning memory to a process in the system from the available NUMA nodes is called NUMA placement policy[11]. As placement policy can influence only performance and not the correctness of the code, heuristic approaches in memory placement can yield acceptable gain in performance. There are various categories in which operating systems handle for the management of NUMA: accepting the performance mismatch, hardware memory striping, heuristic memory placement, static NUMA configurations, and application-controlled NUMA placement.

Our focus in this thesis is mostly on the Linux environment since it has refined NUMA facilities at kernel level and most performance-critical environments use Linux today. Though the operating systems address the NUMA related locality issues in the form of tools such as *taskset* and *numactl* [12], the programmer needs to understand the topology of hardware to apply these tools. These tools allow to explicitly map the process to certain cores belonging to different memory nodes provided the programmer is familiar with the numbering scheme of the processors and nodes. Our intention in the chapter 5 is to relieve the programmer from being aware of hardware topology while creating tasks. The extended proposal to the existing work-stealing infrastructure takes hints from the compiler directives and maps the user created tasks based on what data objects the task is accessing. In this contribution, our proposed strategy depends on kernel supported tool *libnuma* [12] to automatically detect where the objects are mapped to. Using this metadata about the locality of objects and the compiler hints, tasks are added to respective queues near to object binding location. We consider Linux environment with kernel version 3.16 for all our experiments. In Linux

kernel, the processes and threads are created using *clone()* system call with minor differences. For the kernel, process and thread are the instances of `task_struct`. Linux supports first-touch policy for memory binding of objects. According to this policy, the objects are bound to the memory node where the thread accessed it for the first time. In user level run-times, all the objects are initialized by master thread and tasks are spawned using *fork-join* model programming. In this case, the first touch policy of Linux may result in a situation where all shared objects are bound to single node where master thread is pinned to. As a result, all the tasks executed by worker threads pinned to a remote node suffer from NUMA delays. Hence it is a good practice in parallel programming to invoke initialization of objects in parallel. This ensures the whole-object or parts of a larger objects scatter among the available nodes. If the run-time ensures task mapping to the worker queues based on where their shared object binding, it can automatically guarantee locality. This also relieves the programmer from explicitly mapping tasks on to cores. This idea motivated us to propose a strategy in the chapter 5 which is based on the compiler hints. We considered OpenMP run-time that supports source-to-source translation to implement this policy. It could show significant performance gain on standard benchmark programs.
The effort put in mapping tasks near to shared objects also motivated us to work at a layer beneath the work stealing infrastructure. Work stealing run-times extensively use native threads as worker threads and native synchronization objects to implement virtual processors. Worker threads frequently depend on mutex locks to know the task arrival into their respective queues. This motivated us to work on locality of synchronization objects at native thread level. Recently implementation of synchronization objects in NUMA multi-core environment is gaining much importance and issues related to locality are being addressed by researchers. Lock cohorting [13][14] is one of such attempts towards porting existing locks on to NUMA multi-core environment. Influenced by these developments, we proposed **Affinity awareness in work-stealing** domains in chapter 6. This idean could show little performance gain over the previous contributions.
This thesis is an effort towards bringing task based programming paradigm suitable to NUMA multi-core environment. The benchmark programs for illustrating performance comparison are also task based parallel programs from Barcelona OpenMP Task Suit [15].

Our goal is to minimize the cost of accessing memory by achieving affinity between tasks and data, and as a result minimizing overall execution time of the application program.

1.4 Main contributions of the thesis

Main contributions of this thesis are:

- Study of existing user level run-time systems for multi-core processors.
- Study of modern multi socket multi core processor architecture emphasizing on multiple memory controllers and on-chip NUMA features.
- Identifying the problems in adapting existing user level run-time systems to NUMA multi-core processors.
- Approaches in the form of algorithms to adapt existing work-stealing technique to NUMA multi core processors.
- Approaches to improve shared object locality in multi-socket multi-core processor architectures.
- Approaches to improve the internal performance of work stealing runtime by applying affinity aware locks.

1.4.1 Proposed Architecture

The architecture presented in the figure 1.4 represents all hardware components and chapter-wise contributions at software layer collaborating within our proposed user level work stealing runtime. This architecture is compatible with existing work-stealing infrastructure and easily adaptable.

1.5 Organization of the Thesis

This thesis is organized into seven chapters as mentioned below. Though the chapters are interlinked with each other, we tried to present each chapter in self-contained manner to the extent possible, to ease the sequential reading of thesis document. Pictorial representation of the thesis organization is presented in the figure 1.5.

- **Chapter 1: Introduction.** This chapter gives an overview of evolution of processor architecture and associated user-level run-times which motivated us to attempt the work carried out. This chapter also discusses the research objectives.
- **Chapter 2: Literature Review.** This chapter provides a brief review of previous contributions in the area of multi-core user level runtimes.

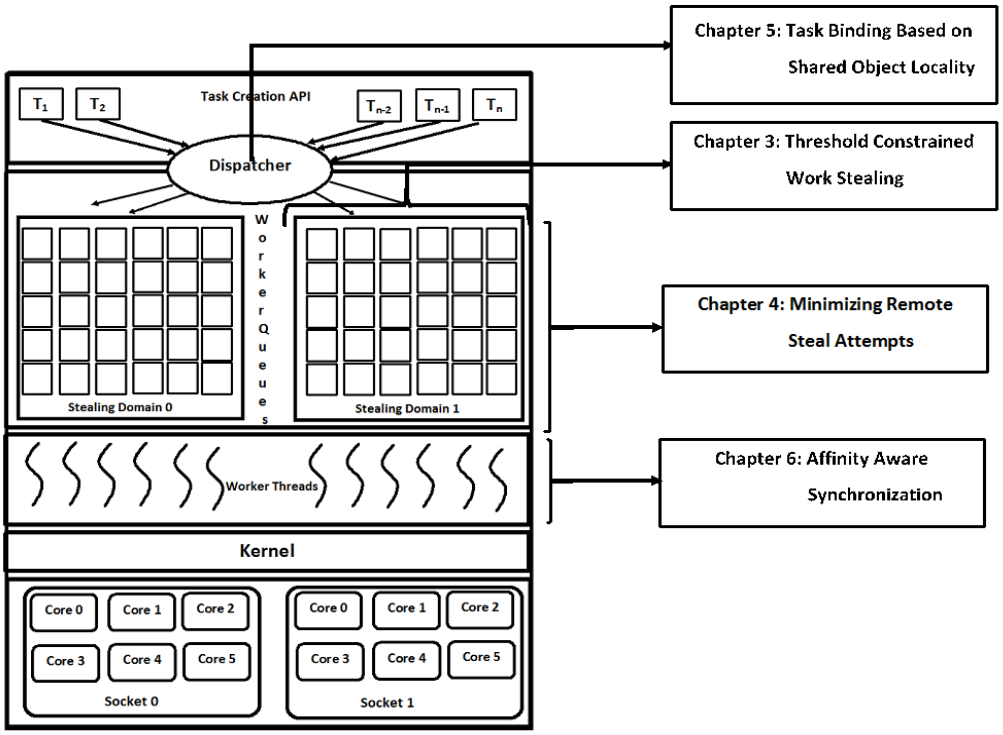

Figure 1.4: Architecture of Task based Run-times

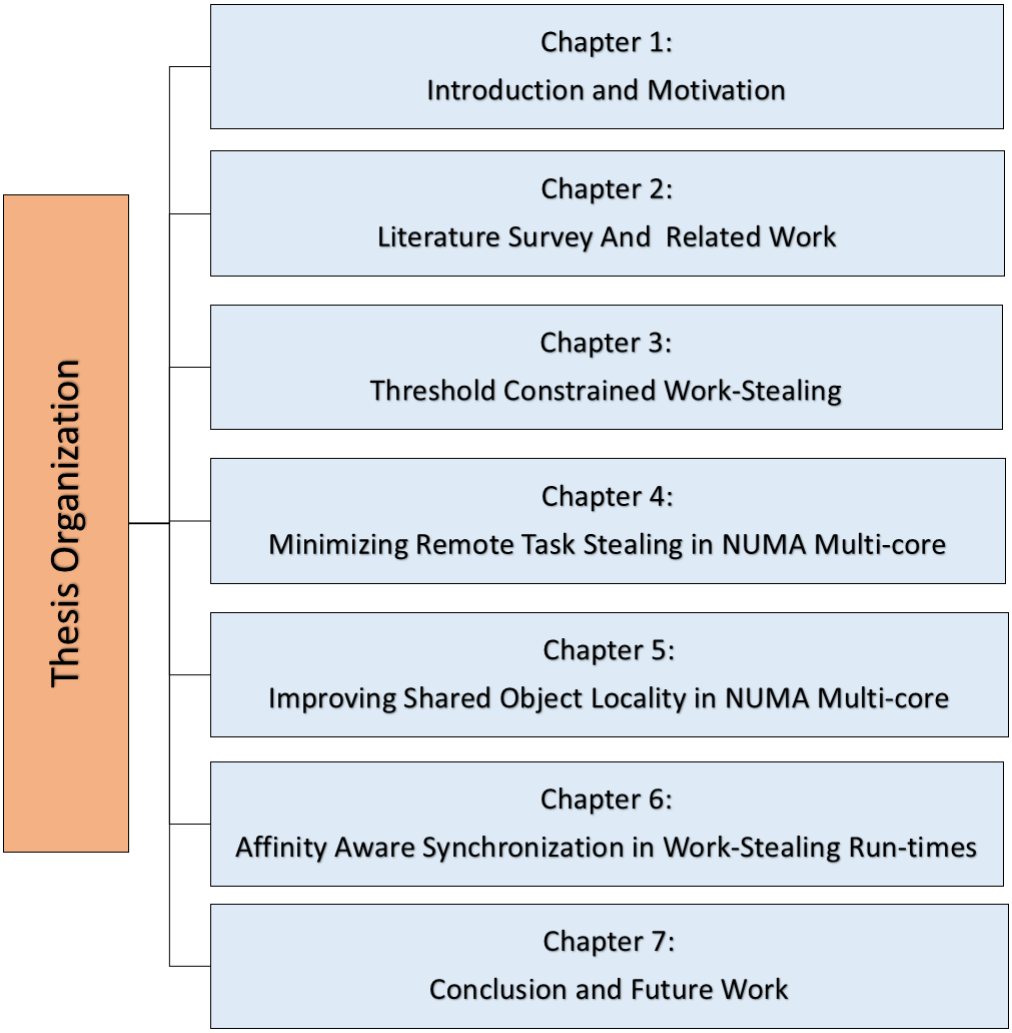

Figure 1.5: Organization of the thesis

- **Chapter 3: Threshold Constrained Work-stealing.** Prior to 2010 the multi-core processors were single socket. Our contribution in this chapter is to improve the performance of randomized work stealing applicable to both single socket and multi-socket multi-core environments.
- **Chapter 4: Topology Aware Work-stealing.** NUMA multi-core processors became popular in server environment from 2010 onwards. The analysis of randomized work stealing for these architectures is studied in this chapter. Our contribution to minimize remote stealing attempts is discussed in this chapter.
- **Chapter 5: Improving Shared Object Locality.** This chapter is aimed to improve shared object and task proximity in work staling runtimes for NUMA architectures.
- **Chapter 6: Affinity Aware Synchronization.** This chapter proposes a strategy to improve synchronization in NUMA multi-core architectures suitable to work-stealing environment.
- **Chapter 7: Conclusion and Future work.** In this chapter, we propose our future work plan and conclude the thesis.

Chapter 2

Literature Survey and Related Work

In this chapter, an overview of various scheduling and load-balancing strategies for multi-core architectures proposed during 2005-2016 are summarized. As the processor, memory and interconnect connection technologies evolve during last two decades, various methods of performance improvements are proposed in previous studies. These studies identified, analyzed and proposed various strategies at operating system level, middle-ware runtime level, and user-level. These studies address the performance issues assuming the unit of scheduling in a parallel environment:

- Processes
- Threads
- Tasks

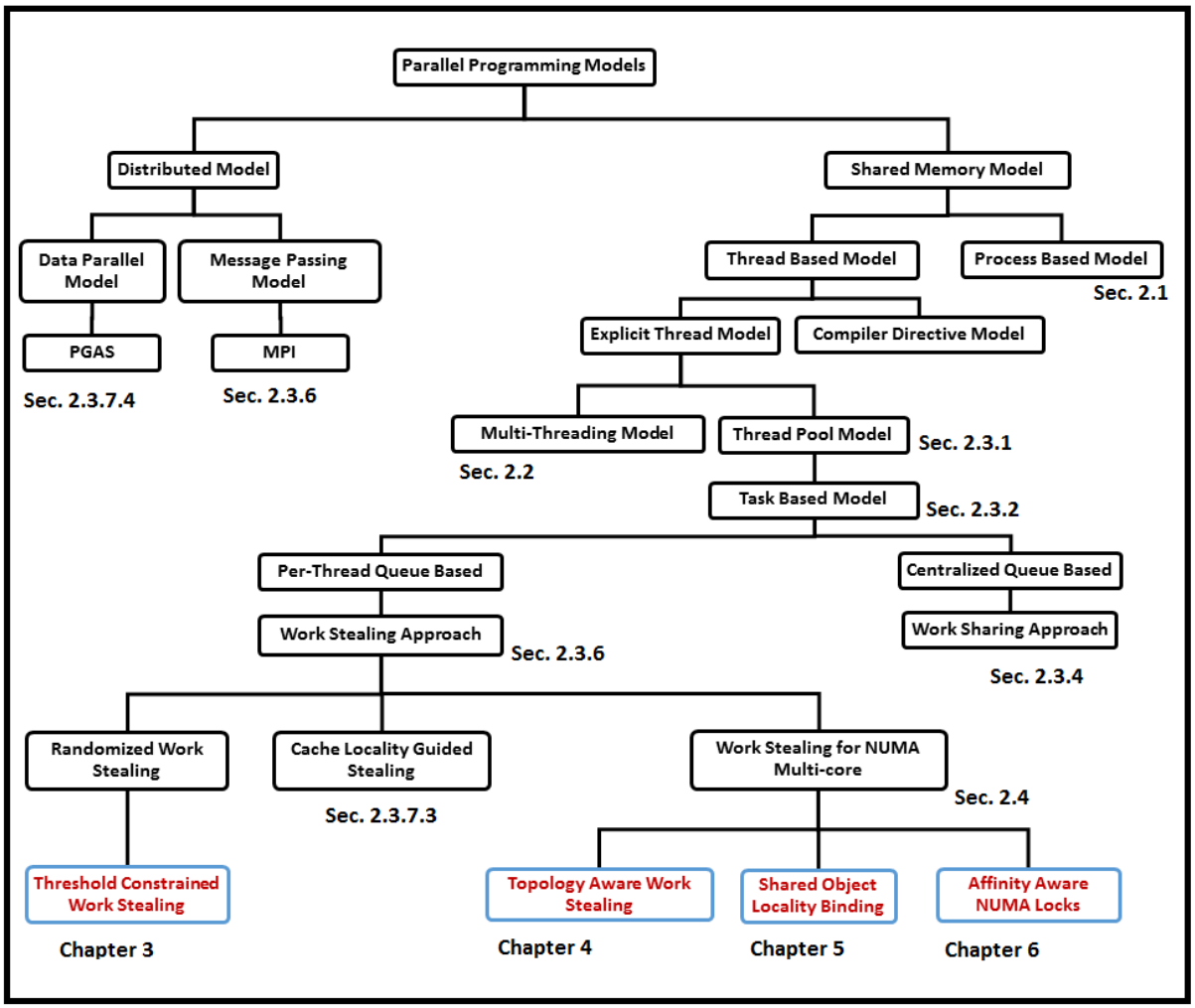

Figure 2.1: Related work at various levels

2.1 Process Level

2.1.1 The need of operating system scheduling to address multi-core architectures

The trend of multi-processors prevailed in previous decades in the form of loosely coupled clusters and tightly coupled symmetric multi processors (SMP). The operating systems were developed targeting these processors mostly. In SMP based architecture, machines with more than one processor on board can access memory using the common bus with uniform access latencies. Multi-core processors are different than SMP machines.

Siddha Suresh et al.(2007) in their work[16], studied the importance of Operating system in detecting the topology of multi-core architectures. They elaborated the differences between Symmetric Multi Processing (SMP) and Multi-core processors. The main difference between SMP and multicore processors is the shared resources such as last level cache (LLC), common bus interface, pre-fetchers in the case of old multi-core processors and shared interconnect and DRAM controllers in case of multi-socket multi-core processors (MSMC). If threads that share data are bound to a group of cores sharing LLC, they can take the full advantage of cache locality. They proposed how scheduling domains of Linux can effectively be applied to improve the throughput of processes in the multi-core environment. The challenges to be addressed in their work are:

- The importance of operating system's awareness about underlying hardware topology.
- The operating system has to address the shared resources like cache hierarchy within chip and the interconnect across CPU sockets.
- Contention for shared resources by execution entities like processes and threads must be minimized.
- Power saving issues and dynamic acceleration technology of Intel's multi-core processors.

This work leads us to keep topological parameters into consideration and propose a similar model at user-level stealing domains. Many of the challenges posed in their work can be addressed at user-level in our work rather than addressing them at the kernel level.

2.1.2 Meta schedulers

Meta scheduler does not introduce a new scheduling policy but it tries to map processes on to the cores based on heuristics obtained from dynamic execution parameters of a process. J K Rai (2009) proposed a meta-scheduler [17] which uses solo-run-L2-cache-stress as parameter metric to evaluate the impact of co-running processes on the last level cache. Taking these parameters as input, machine learning approach is followed to decide how the process mapping has to be done. The two options for mapping processes are:

- Bind the processes onto neighboring cores to take advantage of the shared last level cache.
- Map processes on to different chips with no shared cache.

In their work, the unit of execution is a process and hence not applicable for modern shared memory programming run-times such as Open MP, Cilk and TBB. These run-times target shared memory programming paradigm on SMP and multicore processors. Our work considers a task as a parallel execution unit.

2.1.3 The main difference between SMP and Multicore

Zhuravlev et al. (2012) conducted a survey on process and thread scheduling strategies quoting the key differences between SMP and multicore processors[18]. They illustrate the flaws with present operating systems in considering each core on a multicore chip as an isolated processor on SMP. Kernel level

scheduling policies do not consider the cache and other shared resources in taking wise decisions. If one of the thread pre-fetches the data needed by other thread into the shared cache, other threads of same process address space can take the advantage of locality of reference on these multicore processors. This concept is termed as *co-operative scheduling* and can be implemented at operating system layer. Their work addressed the following points related to multicore processors:

- Cache miss rates experienced by a thread is influenced by a thread running on neighboring core (co-runner) of same chip.
- Threads belonging to a different process may impede the performance of each other by contending for shared resources if mapped onto cores on the same chip.
- Threads of a single process can take advantage of shared resources if mapped onto the cores of same chip with shared cache.
- The current DRAM controllers are built based on single threaded applications. Hence the need for multiple memory controllers on the chip with high speed interconnect among chips is required.
- Time slice based priority scheduling followed by present operating systems doesn't guarantee threads to effectively utilize the time-slice due to the contention of shared resources.
- To reduce the cache contention, additional cache levels can be introduced where every two cores on the same die have common L2 cache and two such dies can have common L3 cache. This can mitigate the contention at the same time, can take advantage of the co-operative running of threads.
- The goal of contention aware schedulers is to decide which combinatorial mapping of threads to cores yields best performance and which mapping doesn't.

They also conducted an excellent survey on process based scheduling. Our focus is on task based programming where tasks are even lighter than threads and obviously than processes which can be adapted for user level run-time systems.

2.2 Native Thread Level

To achieve parallelism within a single address space, a new thread can be created by the programmer on the fly. This model follows fork-join parallelism. Threads are created whenever there is need and hence called *thread-per-request architecture*. Every thread owns its primary context consisting of the instruction pointer, registers and stack depending on at which level(user-level-thread(ULT), kernel-level-thread (KLT) or light-weight-process(LWP)) it is implemented. POSIX threads, Solaris user level threads are examples of this model.

2.2.1 Based on run-time characteristics given by performance monitoring units

Multi-core processor manufacturers started adding new features such as performance counters, and performance monitoring units (PMU) event registers at the hardware level. These counters can capture the details such as per thread execution progress, cache misses, throughput and energy characteristics. These values can be used as input to influence the decisions on thread mapping and scheduling. David Tam et al.(2007) in their work [19], proposed a method of clustering the running threads within a process which share common data. They proposed three step method to co-schedule related threads on to symmetric multi processor architecture or multicore architecture.

- The data access pattern of the threads is determined using fine-grained hardware level performance monitoring unit (PMU).

- Signature of data access regions is maintained in a per-thread storage called *shMap*.
- These shMaps are analyzed using dot product of access-patterns and the threads with access pattern above a certain level are migrated on to same chip and scheduled by operating system on to the nearest CPUs possible.

This study was limited to symmetric multi-processors (SMP) and multicore processors with common bus to access common memory. The approach is not based on profile based learning and gave us hint for compiler directed hints in our proposed work.

2.2.1.1 Stack Distance Profile

Some contention-aware approaches for mapping threads onto cores is based on stack distance competition of two threads. Stack distance profile of a program is the summary of its cache-line reuse patterns.This approach to some extent can tell the run-time the pattern of cache access and memory access. Chandra et al.(2005) in their work [20] analyzed the effect of last level cache sharing among threads that are co-scheduled on neighboring cores. They proposed stack distance based model to predict the impact of L2 cache sharing on co-running threads. This model takes the isolated L2 cache stack distance or circular sequence profile as an input of each thread. This model attempts to estimate the number of additional LLC misses caused by sharing cache, compared to solo-run of the thread (i.e. without sharing the L2 cache with other co-runner). They used a cycle-accurate simulator for a dual-core CMP architecture.
Constructing Miss Rate Curves (MRC) (2009)[21] is also one of such methods using stack distance algorithm. Here, memory address traces are taken as inputs. These address traces can be obtained in one of the following two ways:

- Running an application on the simulator for the first time and obtain simulator memory traces.

- Use instrumentation tools which dynamically get the address traces.

These methods require the application to be executed two times:

- In the first run, the trace log is constructed using stack distances.
- In the second run, prediction models are applied.

We tried to depict the common approach proposed by previous studies on profile based methods in the Fig:2.2. These methods are well applicable for study of architecture, but they can't fit into a parallel run-time system since an application has to be executed for more than once. Our approach takes compiler directive based clues for improving the shared objects locality and controlling the affinity of tasks accessing these shared objects .

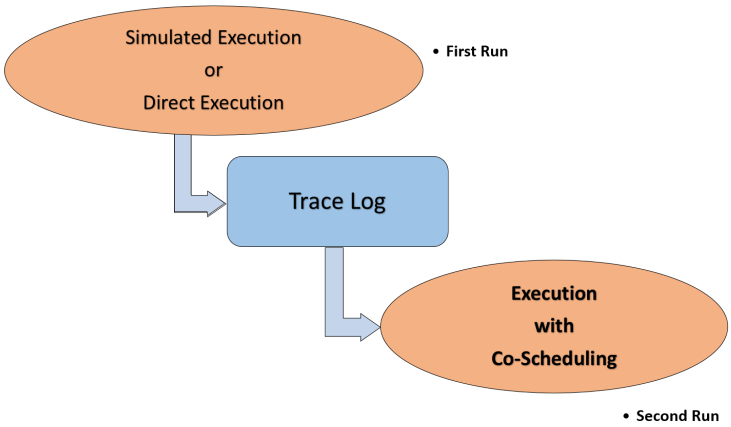

Figure 2.2: Common approach followed by profile based strategies

2.2.2 Work Stealing support at thread level

Gautier (2007) proposed thread based run-time library called KAAPI [22] that supports thread level programming for global address spaces in clusters. The thread API is provided as an extension to POSIX threads with work stealing support. This interface supports $M : N$ model of mapping threads where M represents the number of user level threads and N represents the number of actual processors at hardware level. This work has guided us

to propose the M:N model for task based run-times where M is far greater than N. Whenever a processor finds no KAAPI threads in its own queue, the processor can steal a KAAPI thread from another processor. In our proposed work stealing model, the unit of work is a task which is lighter than thread. Inspired by this work, the number of worker threads in our proposed model are kept always equal to the number of processors N. M tasks are multiplexed on to these N worker threads bound onto N cores or hardware threads where $M >>> N$.

2.2.3 Dynamic re-scheduling

As the processor manufacturers added new features like performance counters, and performance monitoring units (PMU), the schedulers can keep track of different counter values and can evaluate the progress of execution. Dynamic re-scheduling is one such technique, the scheduler captures the run-time characteristics of a process. Zedlewski [23](2010) proposed a solution to re-schedule threads during run-time based on the negative effect of co-scheduling those threads onto a particular core. After sensing that the thread-group is not performing well using performance counters, the run-time takes measures not to co-schedule the thread-group on a single processor chip. The essence of this work is to conclude that only those threads which are sharing common data have to be co-scheduled on to processor domain. This work gave us clues on proposing user level task binding onto worker threads that share data during compilation time itself so that the run-time need not suffer until the negative effects are realized.

2.2.4 Operating system level

Rajagopalan et al.(2007) proposed a framework in [24] for multi-core processors called *McRT* which helps the programmer to experiment with user guided scheduling and user guided placement of threads on to cores. He also proposed gang scheduling of threads using a group of related thread identifiers (RTID). The concept of thread grouping guided us to group worker threads in our proposed run-time.

Fedorova et al.(2007) in their work [25], addressed the performance penalty of binding threads on to the cores of the same chip that with common last level cache (LLC). This effect is called *performance isolation*. She proposed *cache-fair algorithm* to alleviate this effect by adjusting more time quantum to a thread that suffers from performance isolation. Their proposal guarantees fairness in terms of instructions per cycle (IPC) among threads running on multi core processors.
Matthias Diene et al.(2010) [26] studied the impact of thread placement on shared cache multi-core processors. In their simulation study, they analyzed thread data sharing effect if operating system default scheduler is used. As the first step, a communication matrix is generated from the simulation. Then applied two approaches of thread placement namely, exhaustive search where communication matrix is generated for all combinations of threads and heuristic approach where communication matrix is sorted as per sharing pattern and place threads according to sharing pattern. In their experimental results, it is analyzed that if two threads are placed on same core, performance degrades due to thread interleaving. Threads with shared data when placed on to cores sharing LLC, performance is better due to locality of reference to both threads. This work has guided us to create separate worker threads per each core maintaining their own queues of tasks.

2.3 Task Based Programming Model

2.3.1 Importance of Thread Pools

Multi threading is a better choice than process based model for programming Symmetric Multi Processing (SMP) and multi-core environments. But creating and destroying threads on the fly also incurs some amount of overhead (stack space allocation and context creation overhead). These overheads result in slowdown of system when there is sudden creation of many. To mitigate these creation overheads, a pool of threads can be pre-spawned one time and can be reused as on workloads arrive. Each pre-spawned thread can be considered as a virtual CPU and can wait for work to be allotted.

The optimal size of a thread pool is studied by (Yibei Ling 2000)[27] and concluded that the performance of thread pool is optimal if the number of pre-instantiated threads in pool is equal to the number of physical processors at hardware. In our work, we dynamically count the number of physical cores parsing Linux pseudo file system */proc/cpuinfo* and create worker threads equal to the number of cores present.
Belkin (2003)[28] proposes maintaining more than one thread pool depending on the type of work loads. When a job belongs to a particular category, that job is executed by one of the threads of a particular pool of that category. When job execution is done the thread re-joins it's pool back again getting ready for next job of that kind. In our proposed run-time, we maintain multiple worker thread pools

2.3.2 Task or Future based parallel programming

In the era of multi-core processors, programming using native threads adds an overhead of creating user-level or kernel level thread context depending on what type of threads(Kernel Level Threads or User Level Threads) are taken. There were many proposals to support a lighter weight construct for parallel programming than process and thread. The *task* or *future* construct is a coarse grained parallel entity which is lighter than threads. The main difference between a task and future is that a task does not return any value where as a future may have a return value. In programmer's view, a task can be a small block of statements or a function. The task based run-times allow the programmer to create and join tasks where ever parallelism is needed. These task units are mapped to already created threads for completion during run-time. User level run-time environments such as Cilk, Intel Threading Building Blocks (TBB) and OpenMP 3.0 have support for these tasks.

2.3.3 Resolving Task Dependencies

Josep M.(2008) [29] proposed automatic parallelism applying compiler directives in their compiler called SMPS/CellS for the old generation SMP

processors and dual core processors. It supports task based parallelism with task dependency graphs automatically constructed. Among all threads created during initialization, the role of the main thread is to analyze the task dependencies and maintain a task dependency graph. Scheduling strategy followed is depth first approach to resolve data dependencies among tasks. This work also proposes task scheduling my maintaining two separate lists:

- tasks of high priority list are looked up by worker threads without any data locality consideration.
- normal priority tasks are executed by worker threads when they are free.

2.3.4 Work Sharing Approach

In work shared load balancing method, whenever a new parallel execution entity is created, the master processor searches for free processors available, and tries to schedule it on one of those free processors. The parallel entity here can be a thread or a process. This approach is centralized approach where one processor is dedicated to monitor what is the load on the individual processor and distributes the executable entity on to the available processors evenly. But it suffers from the problems of centralized approaches such as contention on single queue.

2.3.5 Self Scheduling Approach

It is also a dynamic load balanced scheduling approach with minor difference to work sharing approach. In self scheduling [30] approach, whenever a worker thread or a process becomes idle, it fetches a task from a centralized queue and executes it. This approach is well suited for implementation of loop-level-parallelism where, the chunk of statements within an iteration is taken as parallel unit. Such tasks are added to a centralized queue from where the worker thread or process can take a task out and execute it. The

difference between work sharing and self scheduling is that that in work sharing approach a dedicated process/thread has to monitor the balance where as in self scheduling, individual workers will access the central queue of tasks. There are two variants of self secheduling [31]:

- **Chunk Self Scheduling (CSS)** where the chunk size is fixed for all iterations.
- **Guided Self Scheduling (GSS)** where the chunk size decreases as execution proceeds.

The main disadvantage of self-scheduling approach is the cost of synchronization in implementation of centralized task-queue. All workers share access to the queue and hence it needs to be synchronized. If more than one workers have become idle at the same instance, they contend to access the central queue to get a task and only one worker wins the race. The other contending worker threads have to waste their execution cycles for winning the synchronization.

2.3.6 Hybrid Programming on clusters with multicore nodes

Georg Hager (2007)[32] proposed a hybrid approach of MPI and OpenMP for programming in the cluster environment where each node can be either SMP or multicore architecture. The disadvantages of inter-node communication of MPI can be minimized by applying this hybrid programming model. The objective of their work is to improve performance of high-performance scientific applications to take full advantage of cluster environment with multiprocessing nodes. Our work focuses on NUMA multicore architectures.

2.3.7 Work Stealing Run-times

RD Blumofe [33] proposed a distributed load balancing technique which is purely distributed. It is distributed in nature because an individual processor can monitor its own load rather than one dedicated processor monitoring

work load of all processors. If it is under-loaded with work, it is eligible to steal work from other processors. The processor from which the work is stolen is called a *victim* and the processor which steals the work is called *thief*. There are two approaches followed in stealing work from victim work queues based on how many number of units of work is stolen:

- **Steal-one Approach**: A thief steals only one unit of work from the victim.
- **Steal-half Approach**: A thief steals half of the units of work from the victim's queue[34].

In the first versions of work stealing implementation, randomized stealing was followed where a thief selects the victim randomly with the seed value equal to the number of processors. This approach has become so popular in the multicore processor era and much research is done to enhance or adapt this technique to various architectures. Our entire work is focused on adapting work stealing for NUMA multi-core processors.

2.3.7.1 Dynamic resource allocation based on feedback

K.Agrawal (2008) proposed [35] an adaptive work-stealing (A-STEAL)as an enhancement to plain work-stealing. This strategy is analyzed for space-shared processor scheduling using "trim analysis". The algorithm gives regular feedback to the scheduler on how much parallelism is obtained by a job. The scheduler can alter the amount of processing to that job based on the feedback. In this theoretical model, the parallel unit of execution is assumed as a job which may take one or more processors during run-time. A task in our work is executed only on one processor or core.

2.3.7.2 Work first and Help first approaches

Yi Guo (2009) [36]analyzed new enhancements in work stealing technique for different kinds of workloads.

- Work first policy: the worker thread executes the newly spawned task first and leaves the parent task to be stolen. This policy is suitable for recursive task spawning application.
- Help first policy: the worker thread executes the parent task first and leaves newly created child task eligible to be stolen. This policy is well suited for iterative based work loads where number of steals is high. They also proposed [37](2010) a new scheduling enhancement when to choose which policy based on parameters such as stack pressure and double ended queue size. When stack pressure is greater than certain threshold help first policy is followed and if local double ended queue size reaches above certain threshold level, work fist policy is set by the scheduler automatically. Our work is not to propose a new such policy. Default work first policy is assumed in our run-time.

2.3.7.3 Cache Awareness in Work Stealing

Chen et al.(2012) in their work [38], adds cache awareness for multi-core processors. When a workload is run for the first time, hardware performance monitoring unit gets the task cache usage characteristics. Using these characteristics, a directed acyclic graph (DAG) is constructed. This DAG is partitioned in such a way that all tasks using the same data are grouped together. When the workload is run for the second time, tasks are mapped to the cores using this partitioned graph. The hardware assumption here is multi-socket-multi-core processors(MSMC) with last level shared cache. The limitation of this approach is, it is profile based task partitioning where an application is run for the first time for gathering hardware counter data and the second time, the cache awareness is applied.

2.3.7.4 Work stealing on Partitioned Global Address Space (PGAS)

James Dinan (2009) proposed randomized work stealing based load balancing for partitioned global address space (PGAS) model[39]. It is meant for irreg-

ular parallel tasks in distributed cluster environment. Their run-time system is built on top of Aggregate Remote Memory Copy Interface (ARMCI) which is meant for PGAS parallel model. In this model, each worker is a process rather than a thread. The goal of this work is to minimize migration of threads across nodes in a cluster. Our work focuses on work stealing enhancement for chip NUMA multi-core environment with shared memory and the worker being a thread rather than a process.

2.3.7.5 Task-based Model in PGAS

Hartmut Kaiser (2014) proposed task-based model HPX [40] as an extension to C++ 11 to support adaptive resource management at run-time. It comes with the features such as scalability, active message passing across nodes called parcels, fixed data and moving worker to the data.

2.4 NUMA Multicore processors

From 2009, the multi-core processor manufacturers introduced new technology to mitigate the effect of the *memory-wall problem*. In this trend of high-performance server processors, more than one memory controllers are kept on-chip along with processor and separate DRAM chips are connected to these integrated memory controllers(IMC). The processor-cores on the die along with integrated memory controller behaves like a node in non-uniform-memory-architecture (NUMA). These memory controllers is an addition to the existing shared resource list when compared to old generation multicore processors. In essence, modern server processors consist of more than one such nodes on chip. The components of these chips are interconnected by high speed links such as Quick Path Inter Connect (QPI) from Intel or Hyper Transport (HT) links from AMD. Starting from Nehalem architecture of Intel, all modern architectures continue supporting this on-chip-NUMA feature. These multicore processors introduce new challenges to parallel programming environment. Though the architecture follows NUMA, it offers the

programmer with same shared memory programming paradigm as in SMP and first generation multicore architectures. The memory locality policies applicable to distributed memory NUMA may not be applicable to these architectures. In this section, a brief survey of previous contributions suitable for these modern architectures is presented.

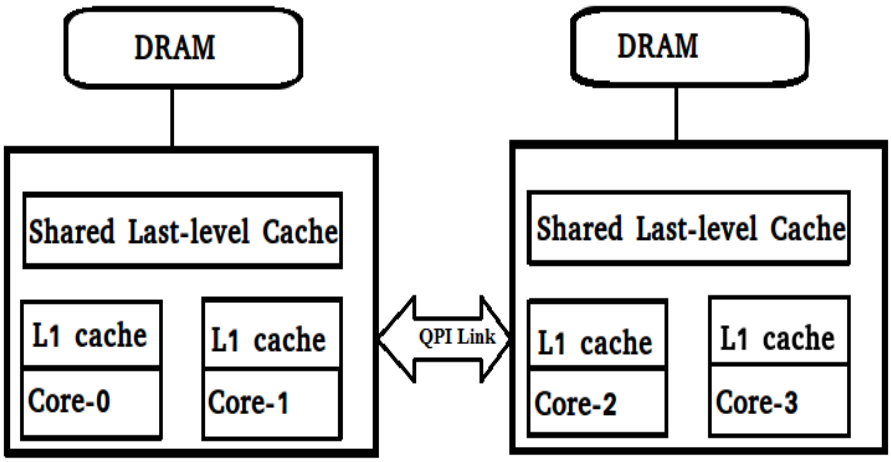

Figure 2.3: NUMA multi-core architecture with 2 nodes and 2 cores per node [1]

2.4.1 Operating System issues for NUMA multi-core systems

Memory management is an important issue for on chip NUMA architectures. Because the architecture can not be considered as distributed NUMA as in the case of clusters. A process address space can span across different memory modules on-chip and behave like shared memory architectures. The same policies that are applicable for distributed NUMA can't be applied here. Majo (2011) studied the effects of memory subsystems for on-chip-interconnect based architectures in [41]. In his other work [1], he studied the effects of operating system's memory management related to NUMA multi-core environment and emphasizes its effect on compilers and run-time systems. Their

work is focused on process vs data mapping on to the nodes.

- Contention for the resources shared on the single chip which was discussed in section 2.2.
- Improving data locality for processes.

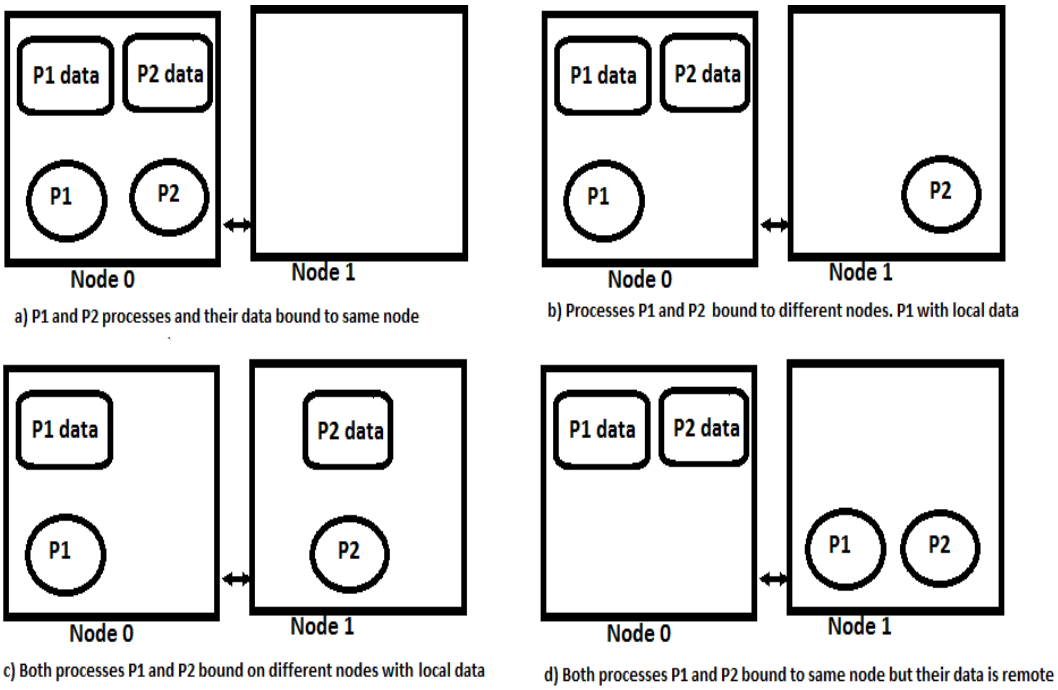

Figure 2.4: Possibilities of Data mapping for processes P1 and P2 across two nodes [1]

The effect of process and data mapping across all combinations of nodes presented in Fig:2.4. The combination presented in case (d), represents the worst data locality scenario. When P1 and P2 processes are bound to the same node, they suffer from memory controller, LLC and common bus contention. Case (b), represents the scenario where one of the processes gets the advantage of data locality and the other suffers from remote data access. Case (c) is the best case scenario where both processes find their local data on respective nodes of processes and there is no contention for memory access and other resources such as LLC and memory bus. The present operating system schedulers do not address the data locality issue. If the scheduler tries

to improve data locality, it can give negative effect on overall performance. They proposed an extension to fair-share-scheduler of Linux called N-MASS which sorts the process list in descending order of cache pressure and NUMA penalty. This work is based on process level scheduling at kernel level. Our work has taken inputs such as NUMA effects from this contribution but we focus on user-level run-times at task level parallelism.

2.4.2 Data Locality improvement at Thread level

Broquedis (2009)[42] identified the data object locality discrepancy in thread based parallelism because of first-touch policy of the operating systems. Their study claims that the threads which allocate the data object is not always the consumer of the object. If other threads of the application access the objects initialized by threads pinned to different node. The consumer threads will suffer from remote access penalty. To resolve this issue, they proposed multi-level scheduling policy. According to this policy, work-staling is limited to cores on the same node. If threads access the data object which got allocated on a different node, that thread along with other data is migrated to that node. This has to be done by the hints provided by the programmer while creation of threads. To apply this strategy, the programmer needs to have deep understanding of topology of hardware so that he can give hints to the run-time about threads and the data binding.
Majo(2012) [43], proposed a program-level transformation method to improve data locality. They provided an API as an extension to gcc compiler which has the ability to transforms the parallel looping constructs to improve data locality.

2.4.3 Data Locality improvement in Task-Level Stealing run-times

Stephen Oliver (2013) [44] analyzed the causes of performance delays in task based run-time systems. The total execution time of thread involves three

types of delays:

- *work-time* during which the threads execute the tasks.
- *idle-time* during which threads remain idle due to imbalance in work assigned to them.
- *overhead-time* is the overhead involved in implementation of task construction and synchronization.
- *work-time-inflation* is the additional time taken by a multi-threaded program than a sequential program. Operating system's locality policy is one of the causes of this overhead.

Their work also proposed an extension API to OpenMP library to improve data locality in NUMA multicore. They introduced identifiable *locality domains* in ROSE OpenMP compiler [45]. Locality domain concept does almost similar functionality which we proposed in chapter 4. To minimize the idle-time component, we proposed a threshold based task stealing in 3 and to minimize the overhead time, we propose lock and mutex locality issue in 6.

Muddukrishna (2013) proposed locality improvement in work-stealing run-times for NUMA architectures in OpenMP. Their proposed strategy [46] tries to improve locality by the hints provided by malloc calls for shared data. These hints provide the task data footprints in terms of kernel level pages. Based on this information their proposed scheduling strategy optimizes data locality. They conducted experiments with default first-touch policy of Linux and conducted the comparison with the results of explicit data placement using *numactl* tool.
Inspired by our work on topology aware task stealing in [47], Chen (2014) proposed locality aware work stealing in his work [48] which limits the work stealing within a socket of a multi-socket-multicore architectures. Lifflande (2014) proposed constrained work-stealing for improving data locality in coarse-grained fork-join tasks. In their approach[49], the programmer gives his priorities of locality in the form of a steal-tree before spawning a new task.

This approach allows the programmer to specify hints to the run-time using special syntax of Cilk language. For locality improvement of tasks, three types of improvements to existing work stealing scheduling are proposed.

- **Strict Order Work Stealing**: where the run-time follows the programmer specified order.
- **Strict Unordered Work Stealing**: a greedy schedule is followed in addition to specified order of tasks.
- Relaxed Work Stealing: the scheduler need not follow the order specified by the programmer.

a technique called dynamic coarsening optimization is applied on steal-tree input. Dynamic coarsening is also capable of adjusting the task size.
Inspired by compiler hints approach we also proposed data locality improvement strategy which only depends on existing OpenMP directives related to shared objects.

2.4.4 Explicite Task mapping

The execution of parallel entities such as tasks and futures is taken care by the user level scheduler as part of the runtime system. Some previous contributions suggest the programmer to have control over pinning of these entities to respective processor cores. The key reason behind this proposal is the programmer's awareness about the topology of the underlying architecture. The programmer can effectively utilize the hardware resources such as shared cache, interconnections and NUMA features if he is given the freedom of explicit scheduling. The tools of explicit affinity control are provided at various levels.

- **Process Affinity Conrol Commands** affinity of processes can be explicitly specified by command line utilities like `taskset`.
- **Thread Affinity Control API** the affinity of a thread can be set and modified by `sched_setaffinity()` during the runtime by using

GNU standard preprocessor macros such as *CPU_SET* in Linux environment.

- **Node Mapping API** tools such as `numactl` and `likwid-pin` support the feature of binding processes, threads to specific cores or nodes within a NUMA system.

Wang(2015) proposed explicit task mapping for work-stealing based runtime to improve the performance of matrix multiplication benchmark. Their work [31], attempted to improve Winograd algorithm for matrix multiplication on NUMA multi-core architectures. In their proposed method, matrix is partitioned and allocated on to different nodes of the architecture. They proposed a hierarchical task stealing approach to Winograd algorithm for load balanced scheduling. Lower level stealing is applied among the cores within a socket(node) to improve the locality of matrix units. Their proposal aims at minimizing task stealing by using a threshold limited queue length. This proposal was limited to one matrix multiplication technique. We tried to propose a generic strategy to improve data locality for various workloads.

Lee et al.(2016) proposed a programmer controlled affinity for the task construct in OpenMP [50]. They provided an extension to GNU OpenMP library, *libgomp* where a programmer can explicitly specify the data and task mapping. As a first step, the programmer has to distribute the data across nodes using API such as *libnuma*. In the next step, a new compiler clause $node_bind$ can be used by the programmer to map the tasks to the nodes where their data is located. We also propose a similar strategy which can improve the locality of shared objects in OpenMP but we do not introduce a new clause such as $node_bind$ but by utilizing the existing OpenMP clause *shared*.

Chapter 3

Threshold Constrained Work Stealing Queues

3.1 Introduction

Programming models provide APIs or compiler directives to create parallelism in the application. These parallel entities can be one of the following:

- Data Parallel
- Task Parallel

User level runtime systems such Cilk, TBB, Charm++ targeting SMP or multi core architectures implement work stealing as load balancing. All these run-times support task as a primitive construct for supporting asynchronous parallel entity. Almost all these run-times follow similar architecture presented in the figure 3.1.

User-level runtimes support APIs to the programmer for creation and managing tasks. Common practice by all runtimes is to follow the most popular *fork-join* model. We also consider this model throughout the thesis. As the programmer creates the tasks on the fly, these tasks have to be mapped to the native threads called worker threads. During the initialization of these runtime systems, native threads are created to form a thread

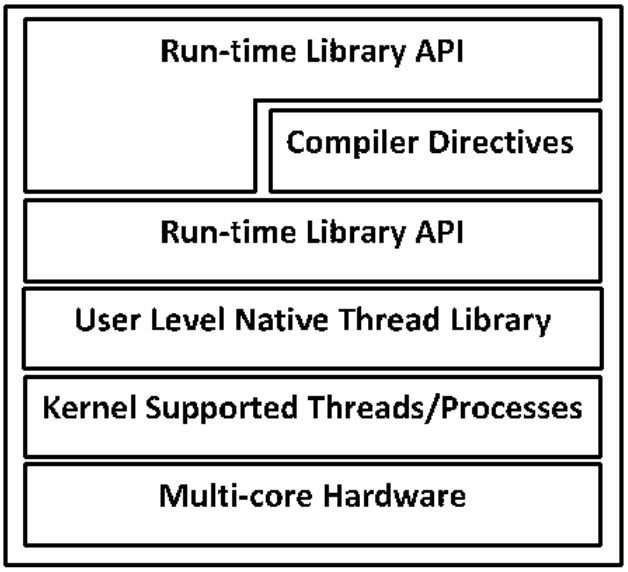

Figure 3.1: Generic Architecture of Userlevel Runtimes

pool. These threads remain alive till the end of application's lifetime or the run-time is explicitly shutdown by the programmer. The common practice is that the total number of worker threads is equivalent to the number of processor cores at hardware level. The runtime implementation also may follow *identity affinity* where worker threads are pinned to individual cores or hardware threads (in case of hyper threading enabled at hardware level). In OpenMP, the number of cores at hardware level can be explicitly controlled using environment variable $OMP\_NUM\_THREADS$ or using an API call $omp\_set\_num\_threads()$. In Cilk, the number of processors can be controlled using the command line option `--nproc`. The runtime also maintains a list of *task queues* one task queue per worker thread. These task queues are preferably double ended queues [51]. As the programmer instantiates tasks using task creation API, the task objects get added at the top end of the queue. Tasks are popped out for execution by the worker thread(also called virtual processor). Hence a worker thread view of the double ended queue is a stack and worker thread mimics sequential execution since it works locally[52]. The detailed version of figure 3.1 is presented in the figure 3.2.

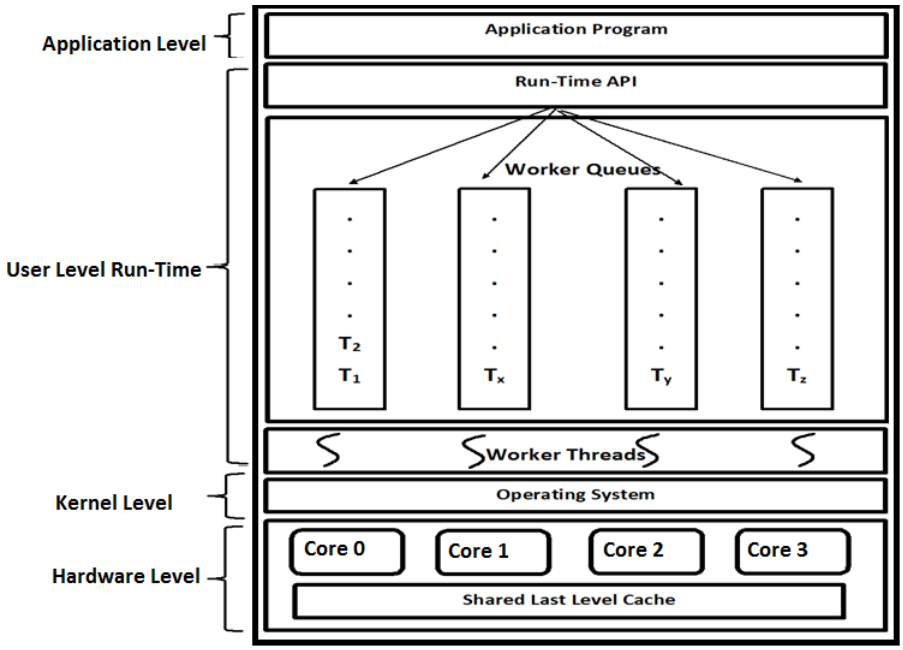

Figure 3.2: Userlevel Task based runtime detailed architecture

Execution of tasks may lead to imbalance of load among the worker threads due to:

- The variable length of execution time of tasks.
- Operating system scheduling decisions on worker thread execution.

When one of the queues becomes empty because of imbalance in load distribution, the worker thread associated with the queue becomes a *thief* and attempts to steal one or more tasks from the queues associated with other queues. The worker from which the task is stolen is called a *victim*. In other words, "when worker thread has enough number of jobs in its own dequeue, it operates locally and when it has no jobs for execution, it operates globally" [**?**]. The plain work stealing algorithm is also called randomized work stealing [53]. In this approach, the selection of a victim worker is as follows.

- The thief worker-thread generates a random number within the range $[0..(n-1)]$ where n represents the total number of worker threads created during the initialization of the runtime. The random number

generated $victimID$ is the index of victim within the list of available task queues.

- There is a possibility that randomly generated $victimID$ may represent an empty queue which also may be trying to become another thief. In this scenario, thief thread waits for an amount of time using back off technique and attempts again to randomly select a victim queue. This scenario is depicted in figure 3.3.

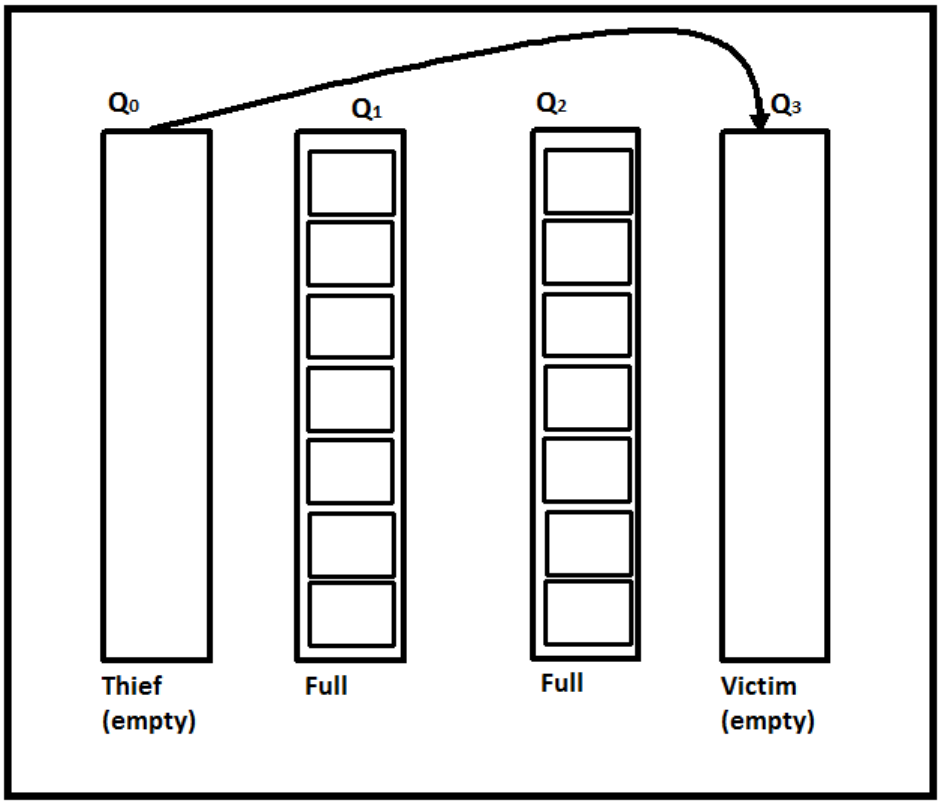

Figure 3.3: A thief worker thread randomly selecting an empty victim

In this chapter, an attempt is made to analyze the existing work stealing algorithms and an improvement policy is proposed as an enhancement to existing work-stealing technique.

3.2 Problem identified in Randomized Work-stealing

When we tried to analyze the randomized work stealing experimentally, the following drawbacks are identified resulting an ineffective load balancing:

- A thief worker thread tries to steal tasks from other workers only after it becomes completely empty. In repeated attempts of finding a right victim, the thief worker has to starve thereby affecting the overall performance of the parallel application.
- Random selection of a victim thread may fail for number of times either by selecting an empty victim or a queue which is about to become empty.
- A worker is unaware of its status to announce its eligibility to become a victim. It has to wait for a thief to detect it and steal tasks from it.

It can be realized that we can mitigate the delays involved in repeated victim selection using an appropriate methodology. The proposed strategy is based on the worker queue metadata which ensures that a thief worker can find the right victim with minimal attempts. The metadata information of worker queue is taken in the form of flags that indicate the *stealability status* of respective worker queue. (s, S) [54] inventory model is adapted to our domain for implementation of our proposed strategy. Every worker queue sets minimum and maximum threshold levels based on (s, S) inventory model[54] where s and S represent the minimum and maximum threshold levels on size of the queue. The proposed improvements are:

- A worker thread need not wait until it becomes completely empty to become a thief. It can start attempting to steal tasks after reaching minimum threshold level s.
- All the worker thread queues which have reached certain maximum threshold size have to announce themselves that they are eligible to

become victims. The delays involved in repeated attempts of empty victims can be minimized by marking the queue whenever its size reaches S state. A thief will attempt to select a victim only from the list of queues which are really overloaded.

3.3 Implementation of Work Stealing Runtime

To analyze the existing work stealing strategy, we implemented a small user level work stealing run time.

Worker Thread Implementation: using the native pthread library on Linux. We followed identity-affinity principle while implementation i.e. if there are n cores at hardware, n worker threads are created. Identity affinity ensures that threads are bound to respective processors at hardware level (Thread 0 to CPU 0, Thread 1 to CPU1 .....). Linux's non-portable pthread affinity control library functions are used to achieve this.

Worker Queue Implementation:There are two approaches in the literature for implementation of worker queues:

- **Concurrent Shared Double ended Queue**: This approach allows multiple worker threads to access on the same dequeue at the same time. Implementation of concurrent dequeues require strong memory consistency.

- **Non-concurrent Private Dequeue**: This approach does not allow multiple worker thread access at the same instance. The possibility of accessing by multiple workers is less since a dequeue has top and bottom ends open. The bottom end can be dedicated for access by local worker thread and the top end can be dedicated for global access by other worker threads only in case of stealing activity.

Concurrent shared dequeues suffer from memory fencing issues and can't be suitable for non-divide and conquer based tasks implementation. Non concurrent private dequeue needs synchronized protection only in the instance

where there is only one task remains in the dequeue and owner worker thread and thief worker are attempting to pop a task from the queue. Because of our threshold constrained stealing strategy, this case never arises. Hence, we followed non-concurrent private double ended queue(dequeue) approach. Such a dequeue is associated with each worker thread. The runtime system architecture we implemented resembles the architecture presented in the Fig. 3.2.

Task Implementation: Our library provides API for primitive task management functionality like *task_create*(), *task_join*(). We implemented task construct as a simple C structure with function pointer to task body, arguments and other task attributes. Whenever a call to task creation API is made, a new task object is constructed and is added to one of the queues associated with cores. As a first step towards load balancing, round-robin approach is followed to distribute these task objects among all the worker queues. All these tasks are multiplexed to the worker threads. i.e. the worker threads which are created during the initialization of runtime wait for the tasks to arrive using the condition waits. As the queue is added with one or more task objects, the associated worker thread comes out of barrier and start popping out the tasks for execution. The worker thread comes out of condition wait and starts popping out the task object and invokes the task body using the function pointer. The task is allowed to run till its completion.

3.3.1 Handling Failures in Randomized Stealing

When a worker queue becomes empty, it randomly selects one of the other queues in range $[0..n-1]$ where n represents the total number of cores(worker threads). If randomly selected victim queue is empty, it is called *false steal attempt*. Whenever such false steal attempt occurs, the worker can apply one of the following techniques till it succeeds in stealing a task from other queue.

- Repeat generating random index till success in stealing.

- Make the thief worker wait for certain amount of time and start again to select a random victim.

In our experimental work-stealing runtime, we followed the second approach i.e. whenever a worker thread fails in selecting a victim for the first time, it waits for an amount of back off time. This back off time is doubled on every repeated successive failures.

3.3.2 MATMUL Becnhmark

To analyze the false steal attempts in randomized work-stealing, we implemented matrix multiplication benchmark. Two bigger sized square matrices $A_{n\times n}$ and $B_{n\times n}$ are multiplied to produce the result matrix $C_{n\times n}$ using regular matrix multiplication approach.

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} * B_{kj} \qquad \forall i = 0..(n-1) \qquad \forall j = 0..(n-1) \qquad (3.1)$$

Computation of C_{ij} is considered as single task body which involves one loop iteration with $O(n)$ parallel time complexity. Since the sizes of multiplicand and multiplier matrices are $n \times n$, the result matrix $C_{n\times n}$ must also contain $n \times n$ elements. Since each task has to compute one element of the result matrix, there are $n \times n$ tasks in the benchmark. Since the goal of running the benchmark is to analyze task stealing, we took the matrices of size 8192 and 16384 which can fill the worker queues very soon, cause load imbalance and result in substantial number of steal attempts.
MATMUL benchmark was executed on dual socket Xeon E5-2620 series processor running Linux kernel 3.16. In each thread pool we introduced two counters called *falseStealCount* and *successStealCount*.

- **falseStealCount** is incremented if a random steal causes a failure.
- **successStealCount** is incremented on the event of a successful steal operation that yields a valid task to the thief worker.

False steal operations and successful steal operations are counted for different number of worker threads. The results presented in the table 3.1 are the average steal counts after running each experiment for 10 times. We introduce

Table 3.1: False Vs Success Steal attempts Analysis w.r.t. number of worker threads

Number of Worker threads	Randomized Work Stealing	
	False Steal Count	Success Steal Count
2	1	34
4	38	72
6	46	82
8	62	84
10	87	68
12	221	191
14	373	243
16	561	595
18	499	888
20	322	846
22	136	546
24	172	101

a metric, *steal-miss ratio* for measuring the efficiency of victim selection in work stealing strategy given by:

$$Stealmissratio = \frac{Number\ of\ steal\ attempts\ that\ choose\ an\ empty\ queue}{Total\ number\ of\ attempts\ to\ select\ a\ victim}$$

MATMUL benchmark was run 25 times and an average steal miss ratio was measured using the counters introduced. The experiment was run with different number of worker threads configuration for its effectiveness. The table 3.2 gives the average stealing ratios obtained with respect to various number of worker threads.

The following facts are stated after observing the steal miss ratios:

- There is no considerable relationship between the number of worker threads and steal miss values.
- On an average, 40 percent steal attempts are leading to failures in

Table 3.2: Steal miss ratios of MATMUL benchmark

Number Of Worker Threads	Average Steal Miss ratio
2	0.03
4	0.35
6	0.36
8	0.42
10	0.56
12	0.54
14	0.61
16	0.48
18	0.36
20	0.28
22	0.20
24	0.63

finding a victim worker queue with enough load. These failure attempts cause repeated random attempts for selection of victim workers.

3.3.3 Mathematical representation of delays

The delays involved in various stages of work stealing can be modeled mathematically. Let T_E denote the time a worker thread spends between minimum threshold state to reach an empty state. Let T_{R_i} denote the delay involved in i^th contiguous attempt for selecting a victim randomly and back off time operation. Then the time involved in each steal attempt T_S in case of a worker queue with non-trivial queue length is given by the following equation

$$T_S = T_E + \sum_{i=1}^{k} T_{R_i} \tag{3.2}$$

In the equation 3.2, the term $\sum_{i=1}^{k} T_{R_i}$ indicates the delays involved in k repeated attempts. The proposed method attempts to minimize T_E and $\sum_{i=1}^{k} T_{R_i}$.

3.4 Setting minimum and maximum threshold levels for worker queues

The key idea of our proposal is that a worker monitors its own state and announces its eligibility to become a victim and a thief.

- A worker thread will be eligible to become a victim if the number of task objects in its associated worker queue reaches a certain threshold limit by setting *stealability flag*.
- A worker thread will start attempting to steal tasks if the number of task objects in its associated worker queue reaches to minimum threshold.

The values of minimum and maximum threshold levels for worker queues are set by using the inventory model proposed in [54]. Let the capacity of the worker queue is C that follows a poisons distribution with task object arrival rate λ and task execution rate by the worker thread μ. If T_{Push} and T_{Pop} denote the time to perform push and pop operations on double ended queue respectively, maximum and minimum threshold level values of each worker queue can be computed by the following equations:

$$S = C - \lambda T_{Push} \tag{3.3}$$

$$s = \mu T_{Pop} \tag{3.4}$$

Computation of S and s values does not involve any computational intensive operations. Each worker thread maintains a counter of how many tasks are currently present on its worker queue and average task execution can be computed using time library calls. In our implementation, the values of S and s are computed only once when all task objects in a queue have finished execution and these threshold values can be used for further usage. Every worker thread can monitor these S and s levels and update the *stealabilitybitmask* vector. If the number of tasks in an ith worker queue

reaches S level, the worker thread sets ith bit of the bit mask to 1 announcing that it is eligible to become a victim. The thief worker can steal tasks only from victims whose stealability status is set to 1 thereby minimizing the random attempts. This scenario is depicted in the figure 3.4

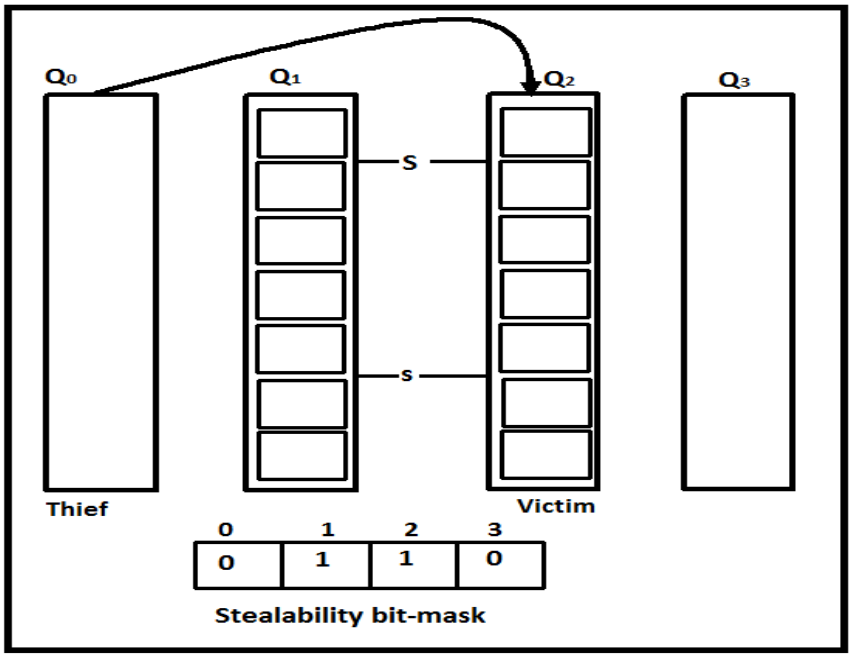

Figure 3.4: Threshold Constrained Victim Selection based on S and s method

3.5 Algorithm

In the algorithm **Worker Run 1**(given in pseudo code form), the function call searchForVictimQueue() searches for the run queues whose status is already set to VICTIM. The values of THRESHOLDMAXSIZE and THRESHOLDMINSIZE are computed using the S and s variables from equations 3.3 and 3.4 from section 3.3.3.

Algorithm 1: Worker Run 1: Input $ptrWorker$

```
1  if (localTaskQueue.size == THRESHOLDMAXSIZE) then
2      stealabilityStatus = VICTIM
3  if (!isEmpty(localTaskQueue)) then
4      run:
5      popAtFront(localTaskQueue, task);
6      execute task;
7  if (size(localTaskQueue) == THRESHOLDMINSIZE) then
8      stealabilityStatus = THIEF;
9  else
10     stealabilityStatus = THIEF;
```

The **Worker Run 1** algorithm is part of run method of every worker thread. The role of the worker thread is to run the tasks created by the programmer. The object localTaskQueue refers to the double ended queue associated with each worker. As the tasks are added to the queue, the size of the queue increases. When the size of the double ended queue reaches THRESHOLDMAXSIZE, the worker announces itself as eligible to allow stealing from its queue. This status is set by setting the status bit to VICTIM. Similarly when the value of localTaskQueue.size reaches a minimum threshold level THRESHOLDMINSIZE, a worker thread announces its readiness to become a thief by setting the status to THIEF. By incorporating these optimizations into randomized work stealing, it is possible to minimize T_E and $\sum_{i=1}^{k} T_{R_i}$ components of the equation 3.2. The values of THRESHOLDMAXSIZE and THRESHOLDMINSIZE are computed using the equations 3.3 and 3.4. Table 3.3 shows the effect of steal miss ratio between randomized work stealing and threshold constrained work stealings. By introducing threshold limit on queue size for selection of a victim, the steal miss ratio is reduced to 13% on an average for less number of threads. It can also be observed that for higher number of threads, the false steals are almost zero.

Table 3.3: Steal miss ratios of MATMUL benchmark

Number of Worker Threads	Average Steal Miss Ratios	
	Randomized Work Stealing	Threshold Constrained Work Stealing
2	0.03	0.48
4	0.35	0.20
6	0.36	0.28
8	0.42	0.12
10	0.56	0.12
12	0.54	0.12
14	0.61	0.12
16	0.48	0.08
18	0.36	0.08
20	0.28	0.00
22	0.20	0.00
24	0.63	0.00

3.6 Benchmarks

To test the performance of proposed threshold constrained work-stealing strategy, we had to choose task based benchmark programs that can explore the features of work stealing run-time. We considered Barcelona OpenMP Task Suit(BOTS) [15] among the available benchmarks. BOTS consists set of programs that explore task parallelism where one of the subsets is the iterative task based programs and another subset is recursive task based programs.

Strassen Kernel: The Strassen benchmark calculates the multiplication of dense matrices using the Strassen's algorithm [15]. The algorithm splits the number of matrix multiplication operations by dividing each matrix into 4 equal sized sub-matrix chunks. The output matrix C is split into 4 sub-matrices in the first invocation of matrix multiplication function. Each sub-matrix computation is done in parallel using independent tasks. Each task further splits the sub-matrix into 4 smaller matrix chunks and generates tasks to handle them. It is guaranteed in this parallel task generation that

the child tasks always compute the output elements which are allocated in the parent task. We executed strassen benchmark with matrix input sizes 2048×2048, 4096×4096 and 8192×8192.
Sort: benchmark is a recursive task implementation of merge-sort algorithm. Instead of dividing the random list into two equal sizes, BOTS sort divides the list into 4 equal parts[15]. Recursive tasking is applied on each sub-array in parallel. When the sub-array size reaches the base condition minimum size in successive recursion, one of the serial sorting methods is applied. When sub-array size is big enough, serial quick sort is applied on it. If sub-array chunk is too small insertion sort is applied to avoid the overhead of recursion in quick-sort. sub-list for sorting. In the merge phase, parallel merge operation is done on first two sub-arrays and second two sub-arrays. In our experimental environment, we considered the default array size and cut off values specified in BOTS. The input array size, N is taken 33554432:Sequential quick sort cut off value, Q is taken as 2048: Sequential insertion sort cut off size, I is taken as 20:

SparseLU: benchmark calculates the LU decomposition of a sparse matrix [15]. The input is a 2D array of which each element is the memory pointer to the submatrix. SparseLU allocates a submatrix to the locations where the problem matrix has non-zero values. The LU decomposition is carried out to the non-NULL submatrices. BOTS contains two versions of this benchmark

- In `Sparse-LU-for` version, tasks are implemented using `parallel for` construct of OpenMP
- In `Sparse-LU-single` version, tasks are implemented using `single` construct of OpenMP

3.6.1 Experimental Evaluation and Results

All the benchmark programs are executed on dual socket Xeon E5-2620 machine, running Linux kernel version 3.16. Each benchmark is run for 10 times

and their execution times are recorded. The parallel version programs are run with work stealing supported OpenMP run-time where one implementation supports randomized work stealing and the other supporting threshold constrained work-stealing. The execution times are presented in table 3.4.

Based on the execution times presented in table 3.4, we computed the speed

Table 3.4: Execution Time Comparison of Randomized Vs Threshold Constrained Work-stealing strategies

Benchmark	Execution time in seconds		
Name-Size	Randomized Work Stealing	Threshold Constrained Work Stealing	Serial
Strassen-2048	0.306036	0.287889	1.81542
Strassen-4096	1.848	1.781	13.139
Strassen-8192	12.045	11.742	92.7439
Sort-33554432	0.7114	0.669	6.696
SparseLU-single	1.0023	1.0355	11.1524
SparseLU-for	0.9901	0.9875	10.842

up values of both versions and are presented in 3.5.

Table 3.5: Performance Comparison of Randomized Vs Threshold Constrained Work-stealing strategies

Benchmark	$Speedup = \frac{ExecutionTime_{serial}}{ExecutionTime_{parallel}}$	
Name-Size	Randomized WS Work Stealing	Threshold Constrained Work Stealing
Strassen-2048	5.9320472101	6.3059720934
Strassen-4096	7.1098484848	7.3773161145
Strassen-8192	7.6997841428	7.8984755578
Sort-33554432	9.4124262019	10.0089686099
SparseLU-single	11.1268083408	10.7700627716
SparseLU-for	10.9504090496	10.9792405063

It can be observed from the figure 3.5 that the proposed threshold constrained work-stealing strategy is showing little performance gain over randomized work- stealing. The performance of Strassen-2048 is improved by 6.3%, Strassen-4096 is improved by 3.76% and Strassen-8192 is improved

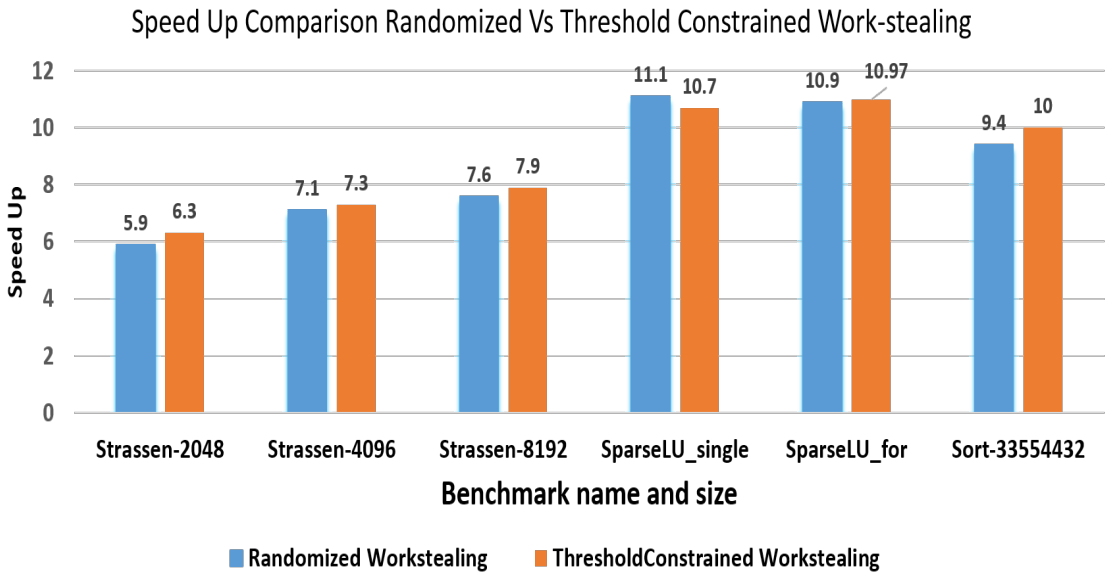

Figure 3.5: Performance comparison of Randomized Vs Threshold Constrained Work stealing on BOTS benchmark

by 2.5%. By close observation, it can be realized that for bigger problem sizes, the performance is not as expected. The performance improvement contributed by threshold constrained work stealing is dominated by memory overheads. The performance improvement in `Sort` benchmark is 6%. Randomized work-stealing strategy is outperforming our proposed strategy by 3% in case of `Sparse-LU-single` version benchmark . Though we could mitigate the delays involved in randomized task stealing, the memory overhead of larger sized benchmarks is dominating the little performance improvement. When statistical *paired t-test* was done on the speed up comparison of randomized and threshold constrained strategies, we obtained the following results:

$$Mean : 0.26$$

$$\mu = 0$$

$$S^2 = \frac{SS}{df} = \frac{1.24}{(6-1)} = 0.25$$

$$S_M{}^2 = \frac{S^2}{N} = \frac{0.25}{6} = 0.04$$

$$S_M = \sqrt{S2M} = \sqrt{0.04} = 0.20$$

T-value Calculation:

$$t = \frac{(M - \mu)}{/} S_M = \frac{(0.26 - 0)}{0.20} = 1.29$$

The value of t is 1.294729. The value of p is 0.125988. The result is not significant at $p \leq 0.05$.

Table 3.6: Steal Miss ratios of BOTS benchmark programs

Benchmark Name-Size	Success Steal Count	Steal Attempt Count	Steal Miss Ratio
Strassen-2048	534	4797195	0.999888685
Strassen-4096	786	19155228	0.9999589668
Strassen-8192	1063	76063619	0.9999860249
Sort-33554432	3601	26600561	0.9998646269
Sparse-LU-single	1292	106656	0.9878862886
Sparse-LU-for	1324	132329	0.9899946346

3.7 Conclusion

In this chapter, we proposed a threshold constrained work-stealing strategy which mitigates the delays involved in random task stealing from other worker thread queues. The proposed strategy is tested and little gain in performance over randomized stealing is observed. Our assumption in proposing this strategy is the uniform memory organization i.e. memory is at equal distance for all the cores. But the proposed strategy is failing to show performance gain for larger problem classes. The reason here is, we did not consider the memory latencies involved in these benchmarks. The next generation server processors also support non uniform memory access (NUMA) on chip. In bigger sized classes of benchmark programs, the data accessed may span into multiple virtual pages, and their respective physical pages may be bound to different nodes on NUMA architecture. The applications will suffer from

remote memory latencies in such cases. The next chapter details how to adapt work stealing for such architectures.

Chapter 4

Minimizing the remote task stealing attempts in NUMA multi-core processors

4.1 Introduction

Modern high performance processors consist of more than one integrated memory controllers (IMC) on CPU chip to fill the gap between fast growing speeds of CPU and stable data delivery rates of memory units. Introducing more than one IMC serves the data needs of threads pinned to cores belonging to different chips simultaneously. These multiple DRAM controllers increase the memory bandwidth and reduce contention for single memory controller hub. The processors are grouped and deployed in a socket. Server processors such as Intel Xeon or AMD Opteron the processors are connected with high speed links such as Quick Path Interconnect (QPI) [55] links from Intel or Hyper transport links from AMD. These links allow more than one socket to be deployed on high performance servers. The presence of multiple memory controllers makes these processors to behave like Non Uniform Memory Architecture (NUMA). A two socket Xeon E5-2620 series processor architecture is presented in the figure **??**.

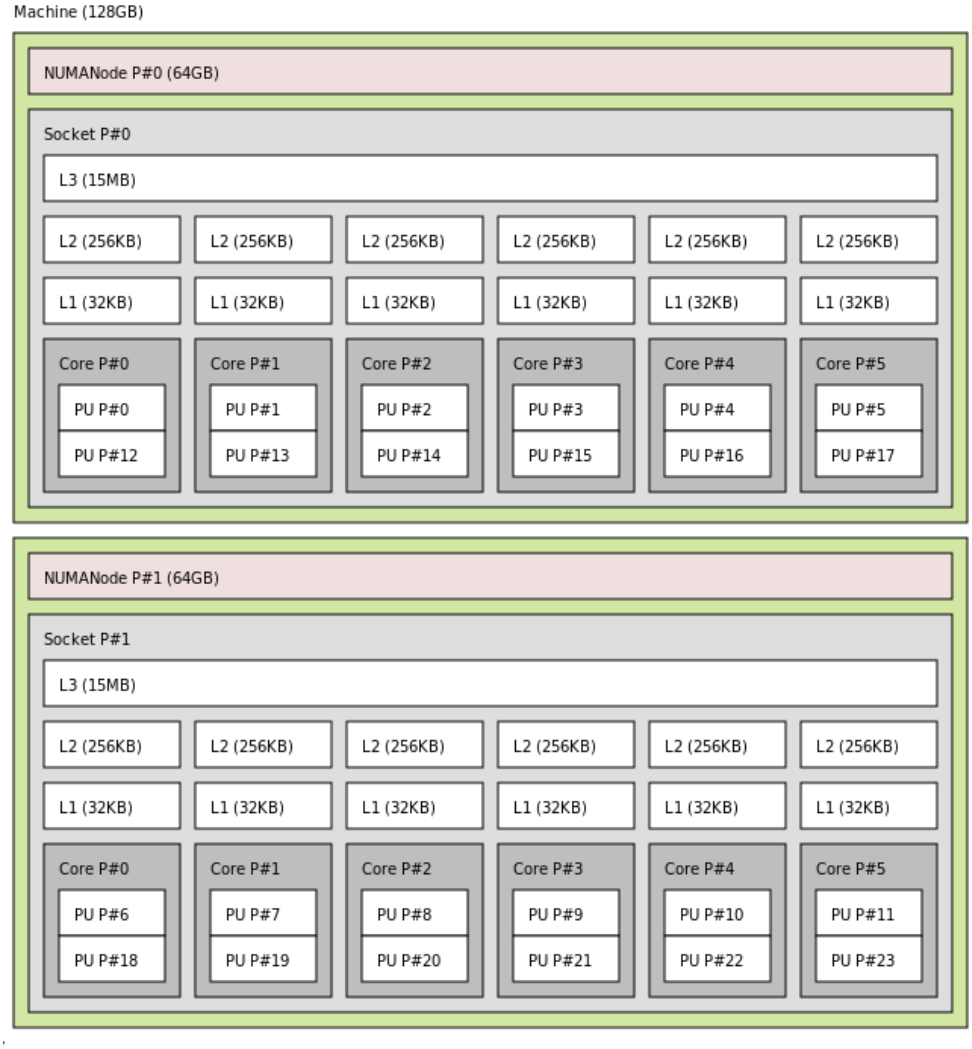

Figure 4.1: Dual Socket Xeon E5-2620 processor architecture

It can be observed from the figure 4.1 that it is a two-node NUMA architecture since, separate memory controllers(MC) are attached to each socket. A thread running on a core can access data from a memory bank connected to its local controller at a faster rate than that of a remote memory module belonging to different chip. The ratio of the remote memory access latency to the local memory access latency is called *NUMA ratio* (R_{NUMA}) and is given by the following equation

$$R_{NUMA} = \frac{T_{remoteaccess}}{T_{localaccess}} \tag{4.1}$$

This new improvement in hardware of high performance processors introduces new challenge for user level runtime systems. The ratio of memory access latencies can be in the order of 2. It can be measured by using the command:

```
$numactl --hardware
```

We could measure the accurate memory access latencies of our target architecture (Dual socket Xeon E5-2620) using the Memory Latency Checker (MLC)[56]. Memory latencies on this two node machine are given in the table 4.1. From the table 4.1, NUMA Ratio for our experimental setup is 1.625.

Table 4.1: Local vs Remote memory access latencies on Dual socket Xeon E5-2620 series processor

NUMA Node	0	1
0	77.3 ns	124.7 ns
1	122.8 ns	75.0 ns

4.1.1 NUMA Effects on Work Stealing Run-time Systems

The goal of work stealing is to balance the load among the available processor cores. The main assumption in the implementation of work stealing run-time

in the chapter 4 is the memory is uniformly accessible by all processor cores. When chips with multiple memory controllers are available on single die, the concept of locality of memory with respect to processor cores is an important aspect in performance.

- As discussed in chapter 4, if identity affinity has to be guaranteed, individual worker thread is pinned to processor core belonging to a node.

- Associated with each worker thread, there is a worker queue. If underlying hardware is NUMA, the memory locality of these worker queues is important since it being the frequently accessed by the associated worker thread. In other words, the worker queue must be bound to the memory node(socket) where the worker thread is pinned to. Worker threads access these task queues in almost fully distributed way except in the instance of stealing occurrence i.e. regular job of the worker thread is to pop tasks from its own queue and execute the job on its processor. If locality of these queues is not considered, and if the worker thread and its task queue are mapped to different nodes due to the default first-touch policy of Linux the overall performance may be affected due to increased remote memory access.

- If a thief and victim are pinned to two different cores belonging to different sockets, the delays involved in stealing introduce additional performance issues.

In this chapter, the concept of remote stealing is analyzed and a solution is proposed in the form of stealing domains that makes the work stealing runtimes adaptable on to these NUMA multi-core processors.

- In randomized work stealing strategy, whenever a worker thread finds no tasks in its own task queue, it becomes a thief and it can randomly choose a worker queue as a victim for stealing tasks. But if randomly chosen worker thread is pinned to a core belonging to a different node, it is a *remote steal attempt*. This scenario is depicted in figure 4.2.

- These remote task stealing attempts may introduce additional delays such as task migration and TLB (translation look aside buffer) misses for data. Remote memory access page faults cause up to 30% degraded performance[1].

- As a result of random stealing, unrelated tasks stolen from other workers brought to execution on local worker thread may result in performance isolation problems [57].

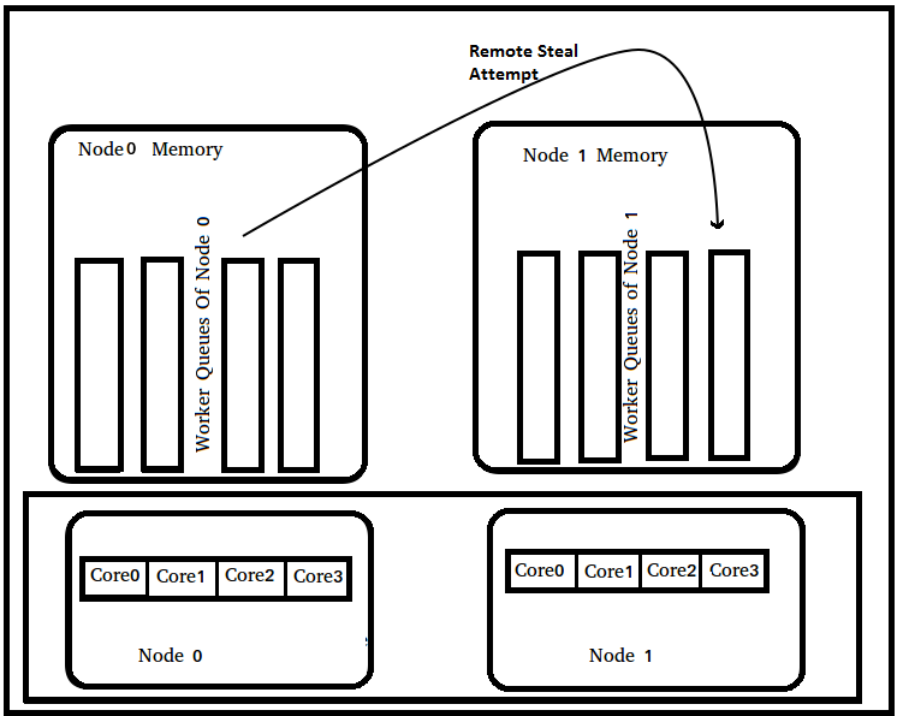

Figure 4.2: Remote task stealing attempts in Randomized work-stealing on two-node architecture

4.2 Analysis of remote steal misses

To analyze the remote work stealing attempts, MATMUL benchmark implemented using our work stealing runtime is executed on our experimental platform. The size of matrix is taken as 8192×8192 for the following reasons:

- The size of matrix should cross the kernel supported virtual memory page size, so that the data section of the program occupies multiple virtual pages. Consideration of a large matrix also increases the possibility of pages mapping across multiple nodes of NUMA architecture.

- Huge number of tasks must be generated quickly so that the run-time overwhelms the worker queues with tasks thereby causing considerable number of task stealing attempts by the worker threads.

The experiments are run using randomized work stealing policy with configurable implementation of thread to CPU pinning policy. Remote steal count is measured for varying number of worker threads. While running the experiments we ensured that the number of worker threads is even such that two different halves of worker threads are bound to two different nodes. For instance, the first entry of the table 4.2 represents the case with only two worker threads. In such a case, we explicitly bound one thread on to a core belonging one node and the other worker thread onto a different core belonging to different node. These explicit affinity control is done using affinity control functions of Linux environment.
While analyzing the remote steal attempts, special variables have to be maintained to find the values of the following interested counters:

- **Remote Steal Attempts**: counts how many randomly generated victim indexes are leading to refer a worker thread on remote nodes.

- **Remote False Steals** counts how many randomly generated victim indexes are leading to a failure to select a proper victim.

The common approach followed in randomized work stealing to choose a victim is:

$$victimID \ = \ random(\ seed\ )\ \%\ ncpus \tag{4.2}$$

Where *ncpus* represents the number of processors or hardware-threads in the machine. On a NUMA multicore machine, n is the count of cpus on all nodes. For instance, if the machine is with 2 nodes where each node has 6 processors, the victim index generated may be in range [0..11]. All

the victim indexes generated in range [6..11] will cause a remote worker access. The counter *Remote Steal Attempts* is used to count such remote event occurrences. It can be observed from the table 4.2 that, 50% of the steal attempts are remote. We already proposed technique to minimize the value of *false steal count* in the chapter 4. If same technique is applied for NUMA multi-core runtime, the second counter *Remote False Steals* can be minimized.

Table 4.2: Remote steal miss ratios in Randomized work stealing

Number Of Worker Threads	Average Remote Steal Count	Average Local Steal Count
2	23	21
4	41	49
6	46	54
8	56	55
10	66	56
12	61	45
14	123	88
16	243	235
18	334	401
20	354	473
22	226	312
24	102	77

4.3 Proposed method

To effectively map the worker threads on to cores belonging to memory nodes, it is necessary to understand the hardware topology. Tools such as `numactl` [12] or `likwid`[58] can be used to easily obtain this information. Our interest is to identify the number of cores per node. Hence, we depend on `numactl` and `/proc/cpuinfo` commands output to group the cores belonging to a node. These features are added to the simple threshold constrained work stealing library (TC-WS) which was implemented in the chapter 4 and call

it Topology Aware Task-stealing Library (TATL). As part of this runtime, we propose, co-operative stealing domain based worker-pools. The pool of worker threads are grouped into *stealing domains* based on locality with respect to last level cache, and DRAM. Addition of the new feature stealing domain restricts the work stealing activity within a memory node and it is responsible for minimizing the cross node task steal attempts. To analyze the remote stealing effects, a task stealing library was implemented using flexible macro based API provided in wool [59]. This API allows us a flexible method of spawning the tasks with variable number of arguments. For running the benchmark programs, the proposed strategy was applied in the run-time layer of OpenMP.

4.3.1 Topology of the architecture

During the initialization of our run-time library, the runtime divides the available nodes and their associated processors into a tree like data structure called topology tree. The construction of topology tree [47] is not so complex as that of hwlock [60]. For illustration of topology tree construction, we took our experimental architecture (dual socket Xeon E5-2620) of the figure 4.3 into consideration. This architecture is translated into the topology tree of the figure 4.4. Once the topology tree is constructed, identifying the processor core grouping can be done based on memory locality wise or last level cache locality wise.

It can be observed from the topology tree in figure 4.4 that cores belonging to the same memory node also give the advantage of sharing the last level-cache (either L2 or L3). Most of the tasks in a parallel program constitute similar task body and operate on different data. Overall performance of such applications can be enhanced if such related tasks are bound to such group. Stealing domain strategy is proposed to serve this purpose.
During the initialization of the run-time, the topology tree shown in figure 4.4 is partitioned into two sub trees. A worker pool array is created per sub tree where each worker pool represents a stealing domain. This strategy can be scaled easily and generalized: if there are M nodes and each node has

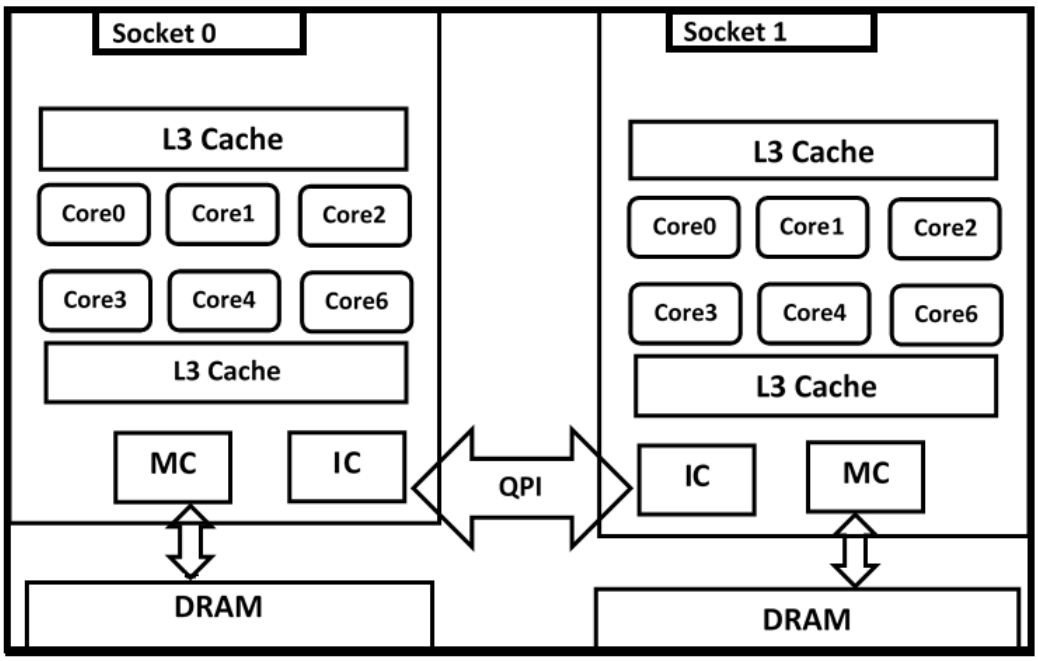

Figure 4.3: Schematic View of Dual Socket Architecture

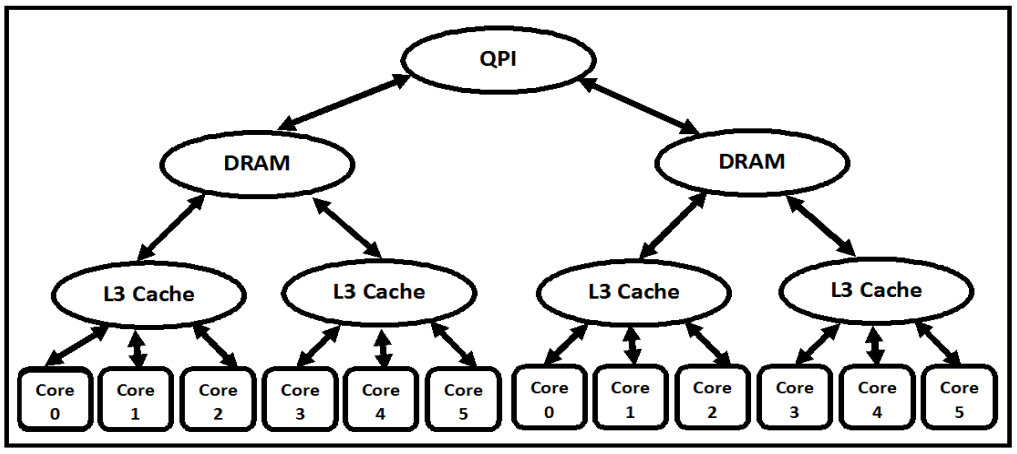

Figure 4.4: Topology tree representation of Dual Socket architecture of ??

N cores, then M worker pools are created where each pool has N worker threads. The total worker threads are grouped into M stealing domains. By restricting the task stealing within the same domain, the number of cross chip references and remote cache misses are reduced. Task stealing from a remote domain is allowed only when a thief worker is unable to find a victim worker in its local domain. Grouping a total of MN worker threads in M domains of N workers each gives the advantage of flexible implementation and does not cause any overhead. The stealing domains also allow the run-time to be easily scalable. During the initialization of the run-time, the run time system can parse the processor topology and detect the number of NUMA nodes creating the equivalent number of stealing domains. The number of worker threads in each stealing domain will be equivalent to the number of cores or hardware threads depending on hyper threading feature enabled or disabled.

4.4 The proposed algorithm

In a task stealing run-time implementation, it is a common practice that each worker thread invokes an outline function during the worker pool initialization process. This outline function is responsible for the activities of task running and task stealing. We present the simplified version pseudo code of such an outline function. For ease of readability, the synchronization steps are not presented in the pseudo code. The algorithm **Worker Run 2** presented here is a combined approach of stealing domain strategy (proposed in this chapter) and steps introduced in threshold constrained work stealing strategy of the chapter 3.

Threshold constrained work stealing strategy added with stealing domains strategy could improve the efficiency by minimizing the remote socket steals. This method puts the best effort to constrain a worker to select a victim within a stealing domain. Remote stealing attempts are done only if all workers of the local stealing domain are under loaded. We maintained separate counters to the stealing attempts in debugging mode and gathered remote steal attempts at the end of each run. The MATMUL bench mark

Algorithm 2: Worker Run 2

```
   input : Pointer to current worker context *this
 1 if (this.localTaskQueue.size == THRESHOLDMAXSIZE) then
 2     this.status = VICTIM;
 3 if (!isEmpty(localTaskQueue)) then
 4     run:
 5     popAtFront(localTaskQueue, task);
 6     execute task;
 7     if (localTaskQueue.size == THRESHOLDMINSIZE) then
 8         this.status = THIEF;
 9 else
10     this.status = THIEF;
11     taskQueue= searchForVictimQueue ( thisStealingDomain );
12     popAtRear(taskQueue, task);
13     if (task) then
14         pushAtRear(localTaskQueue , task );
15         goto run;
16     else
17         if ( node_stealability_i > 0) then
18             taskQueue = searchVictim( remoteStealingDomain_i );
19             popAtRear ( taskQueue, task );
20             if ( task ) then
21                 pushAtRear(localTaskQueue , task );
22                 goto run;
```

Table 4.3: Remote steal miss ratios after stealing domains

Number Of Worker Threads	Remote Steal Steal Count	Local Steal Count
2	0	0
4	0	72
6	0	1001
8	0	85
10	0	69
12	0	271
14	0	251
16	0	638
18	0	835
20	0	823
22	0	522
24	0	24

was run 10 times and average remote stealing miss ratio is computed. The results are presented in the table 4.3. It can be observed that remote steal miss ratios are almost zeros when compared to the results presented in the table 4.2.

4.4.1 Impact of Remote Steals on Performance

To investigate the effect of remote steal attempts on the performance of the workload, we ran the MATMUL benchmark for 50 times. Unlike the tables 4.2 and 4.3 we fixed the number of threads equivalent to the number of cores to follow identity affinity and collected data on remote tasks steal attempts and respective execution times. The relationship is presented in the figure 4.5. When we applied Pearson Correlation Coefficient Calculator on the obtained data, the value of R was 0.9123 and it is a strong positive relationship between remote steals and execution time.

$$T_E = 1.609907157N_R + 1723.235601 \tag{4.3}$$

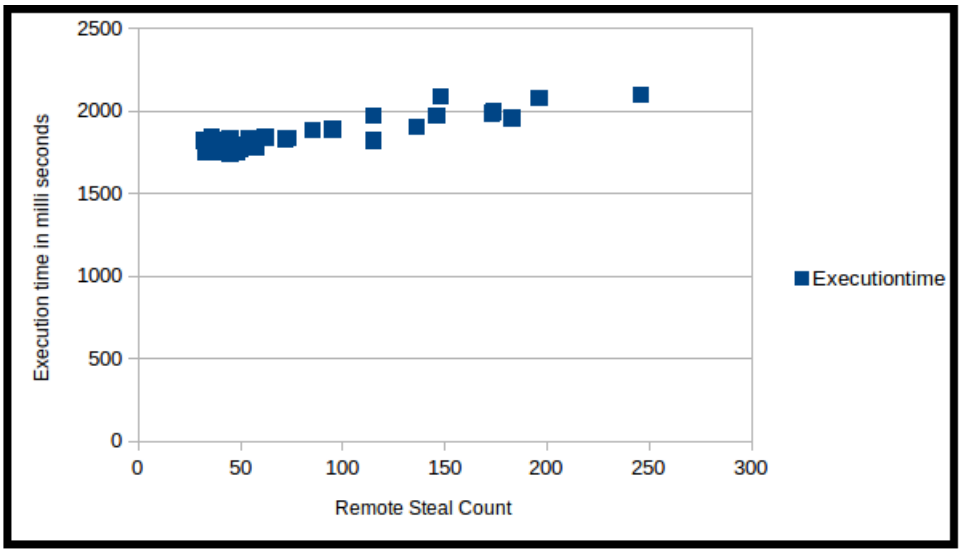

Figure 4.5: Effect of Remote Stealing on the Execution Time

The equation 4.3 shows the linear relationship between remote steal count and execution time of the program. T_E represents the time taken for execution and N_R represents the remote steal count.
Correlation Coefficient: $r = 9.122574337 \times 10^{(} - 1)$
Residual Sum of Squares: $rss = 66779.39655$
Coefficient of Determination: $R2 = 0.8322136254$

4.5 Implementation

The concept of stealing domains from this chapter and threshold constrained work stealing from the chapter 4 are implemented as part of OMPi OpenMP runtime. OMPi is an open source implementation of OpenMP runtime with work stealing support. We added the new features by changing the code of the runtime to adapt it for NUMA multicore processors. The number of worker threads created in this runtime can be configured by setting the environment variable $OMP\_NUM\_THREADS$ or using API call $omp\_get\_num\_threads()$. To implement the concept of stealing domains, we

divided the total number of threads into number of stealing domains which is equal to the number of nodes based on memory topology. The selection procedure of victim worker is modified in such a way that work stealing is confined mostly within the stealing domain. This can be done by modifying the equation 4.4.

$$victimID \ = offset + \ random(\ seed\ )\ \%\ ncpus_{SD} \tag{4.4}$$

The term $ncpus_{SD}$ reprsents the number of CPUs within the stealing domain(SD) and the offset indicates the index of first worker thread within the stealing domain. The modulo operator guarantees that the generated $victimID$ is within the range [0 .. $ncpus_{SD}$]. Though the physical CPU numbers configured at hardware may not be contiguous numbers, the worker thread numbers allotted at runtime level can be considered for grouping of worker threads into stealing domains. Implementation of stealing domain for a two node architecture with 4 cores per node is depicted in the figure 4.6. The figure presents the worker threads belonging to a stealing domain attempting to steal tasks from other worker queue of the same stealing domain.

An additional feature to minimize the remote false steal attempts is added to the runtime. A worker thread pinned to a socket will attempt to steal a task only if one or more worker queues of remote node have stealability status set to one. Stealability status of a node is the sum of stealability status of worker queues belonging to that node. The equation 4.5 gives the stealability status of a node.

$$node\_stealability \ = \sum_{i=1}^{k} stealability_i \tag{4.5}$$

where, k represents the number of worker threads within the node.

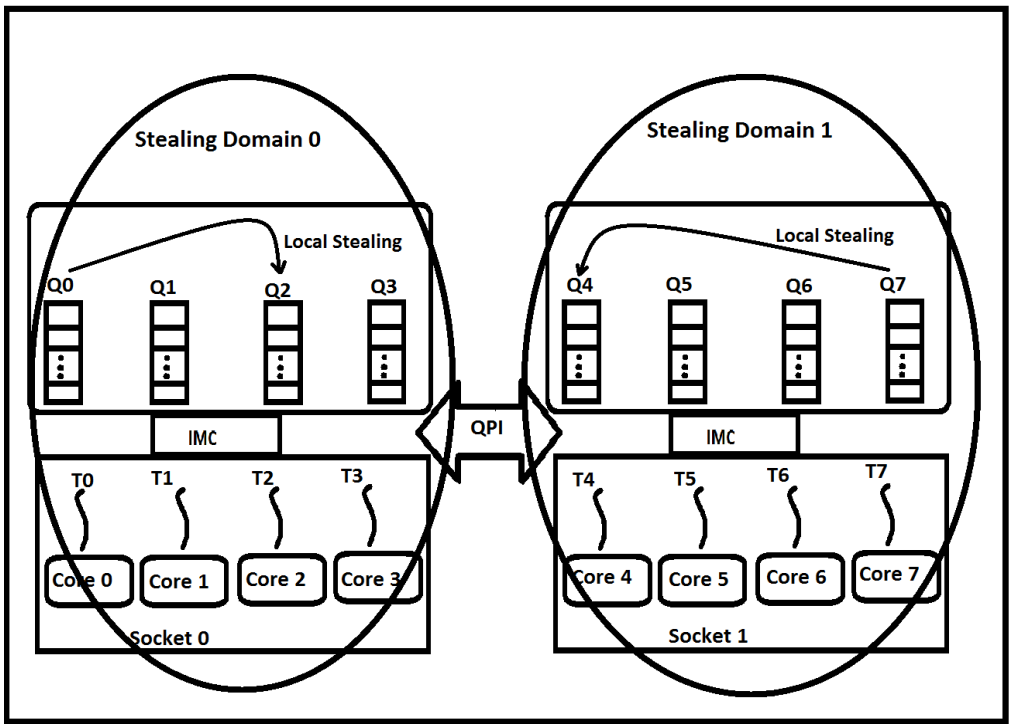

Figure 4.6: Stealing Domain Implementation for two node architecture

4.6 Benchmarks and Results

Barcelona OpenMP Task Suit(BOTS) [15] consists of set of task based benchmarks. These task parallel programs spawn large number of tasks. Proportionate to the problem size, these programs also exhibit NUMA sensitiveness. We executed the same benchmarks as in the chapter 4 for assessing the effect on performance.

The **Strassen benchmark** calculates the multiplication of dense matrices using the Strassen's algorithm. The algorithm divides the number of multiplication operations by splitting each matrix into 4 equally divided sub-matrix chunks. The output array C is split into 4 sub-matrices in the first matrix multiplication function call. Each sub-matrix is calculated in parallel using independent tasks. Each task splits the submatrix into 4 smaller submatrices and generates tasks to handle them. This recursive computation guarantees that the child tasks always compute the output elements which are allocated in the parent task. If parent task and child tasks belong to the same node,

these tasks can take the advantage of data locality. The tasks should not be migrated to a different node because of stealing action initiated at other node. We distributed the array elements explicitly by using the OpenMP parallel construct and libnuma APIs. First, *aligned_alloc*() is used to allocate the output array with a page boundary alignment. The starting index of the corresponding submatrix is calculated in a parallel region. A thread is selected for each NUMA node in the parallel region. Then the thread calls *numa_setlocal_memory*() to migrate memory pages to the local NUMA node.
The **Sort** benchmark is a recursive task implementation of merge-sort algorithm. Since sorting involves permuting the elements of vector, it leads to remote memory access. Instead of dividing the random list into two equal sizes, BOTS sort divides the list into 4 equal parts[15]. Recursive tasking is applied on each sub-array in parallel. These recursive subtasks are the children of the parent task and mostly access the array data. If task is stolen by a worker of remote node, it has to depend on remote memory access. Again in the merge phase, if the four sub arrays are located on the same node, merging subtasks can take the locality of reference advantage. In our experimental environment, we considered the default array size and cut off values specified in BOTS. The input array size, N is taken 33554432.

SparseLU benchmark calculates the LU decomposition of a sparse matrix [15]. The input is a 2D array of which each element is the memory pointer to the submatrix. For larger matrix size, SparseLU allocates a submatrix to the locations where the problem matrix has non-zero values possibly on different NUMA node. The LU decomposition is carried out to the non-NULL submatrices. BOTS contains two versions of this benchmark

- In `Sparse-LU-for` version, tasks are implemented using `parallel for` construct of OpenMP
- In `Sparse-LU-single` version, tasks are implemented using `single` construct of OpenMP

The results shown in Table 4.5 present the execution times obtained using

our proposed library TATL compared to randomized OpenMP implementation of BOTS Benchmark applications. The performance improvement obtained using TATL is due to minimizing the latencies involved in selection of random victims (94% as stated in the table 4.2) and remote victim selection (50 % as stated in Table 4.2). In case of remote victim selection, additional overheads are involved, and contribute to remote memory access latencies while accessing remote worker queues and stealing tasks from remote worker queues. To assess the remote access effect on performance, we measured remote data volume of each benchmark program using performance counters. The reason for delays in randomized task stealing technique may be due to

Table 4.4: Remote data volumes accessed by benchmark programs

Benchmark Name - Size	Remote Data Volume Accessed by benchmark in GB
Strassen-2048	0.3423
Strassen-4096	1.1157
Strassen-8192	2.8416
Sort-33554432	1.6365
SparseLU-single	0.0675
SparseLU-for	0.0613

the assumption of shared memory paradigm in its implementation. The proposed strategy resulted in 22% improvement in overall performance. The results presented here are obtained on our experimental platform with dual socket Xeon E5-2620 12-core machine with 24 worker threads(hyper threading enabled). Though the performance improvement appears to be primitive, it may scale well on machines with more than two nodes.

In case of Strassen-2048 benchmark, TATL has shown 23% performance gain over randomized work stealing and it is little improvement over threshold constrained work stealing. In case of Strassen-4096 benchmark, TATL could yield 20% speedup gain over randomized work stealing and in case of Strassen-8192, the speed up gain is 18%. It can be observed that the speedup gain is diminishing as the problem input size is doubled.
Sort-33554432 benchmark could achieve speedup gain of 33% over randomized work stealing which is significant.

Table 4.5: Execution Time Comparison of Randomized Vs Topology aware Work-stealing strategies

Benchmark Name-Size	Execution time in seconds		
	Randomized Work Stealing	Topology Aware Work Stealing	Serial
Strassen-2048	0.306	0.247600539	1.81542
Strassen-4096	1.848	1.5349547822	13.139
Strassen-8192	12.045	10.12196554	92.7439
Sort-33554432	0.7114	0.533518016	6.696
Sparse-LU-single	1.0023	0.82665136	11.1524
Sparse-LU-for	0.9901	0.860222	10.842

`SparseLU-single` has shown a speedup gain of 21% where threshold constrained work stealing could not show any improvement over randomized work stealing.
`SparseLU-for` benchmark could give only 15% speedup gain over randomized work stealing. Threshold constrained work stealing of 4. The speedup improvement is illustrated in the figure 4.8

The run times illustrated for performance comparison are GNU OpenMP and OMPi [61]. GNU OpenMP supports task level parallelism in addition to loop level parallelism but does not implement work stealing at runtime level. OMPi runtime is a work stealing based implementation of OpenMP 4.0 standard specification. Within OMPi runtime, stealing domain feature was added with minor changes to source code. It can be observed from the figure 4.8 that the speed up achieved by Topology aware task stealing is better than both Randomized work stealing and threshold constrained work stealing strategy proposed in the chapter 4. Threshold constrained work stealing could improve the performance to little extent in case of certain benchmarks by minimizing the false steal attempts but it could not localize the task steals within a memory node. The task stealing activity from remote memory is costly in terms of memory cycles. Introducing topology awareness in work stealing activity confines the task steals within the memory locality. The reason for significance in the improvement is, memory cycles are far slower compared to CPU cycles and stealing domain strategy could reduce the

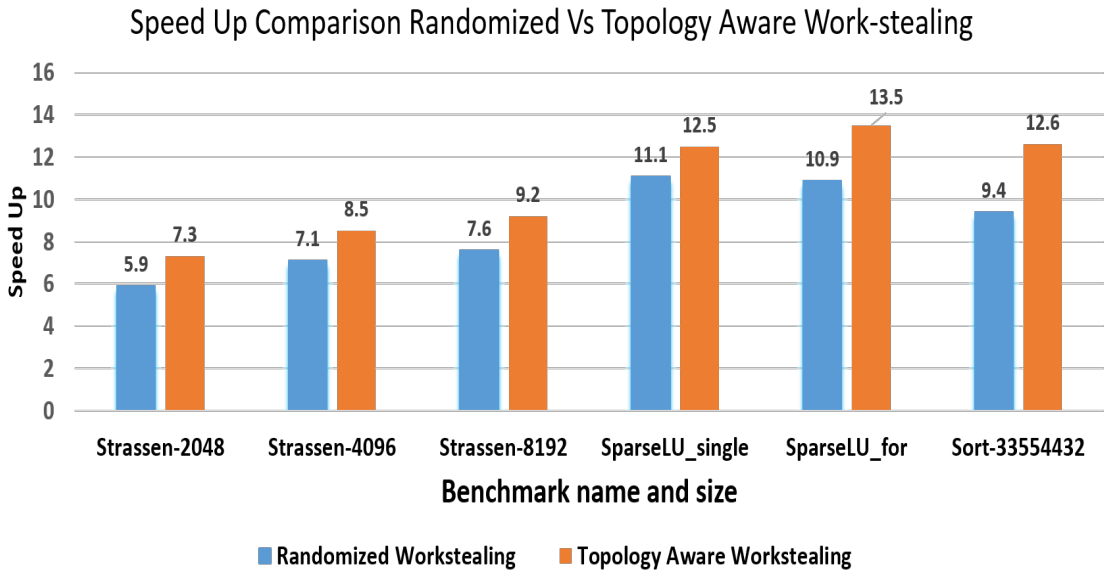

Figure 4.7: Performance comparison of OpenMP vs Topology Aware Task stealing Strategy

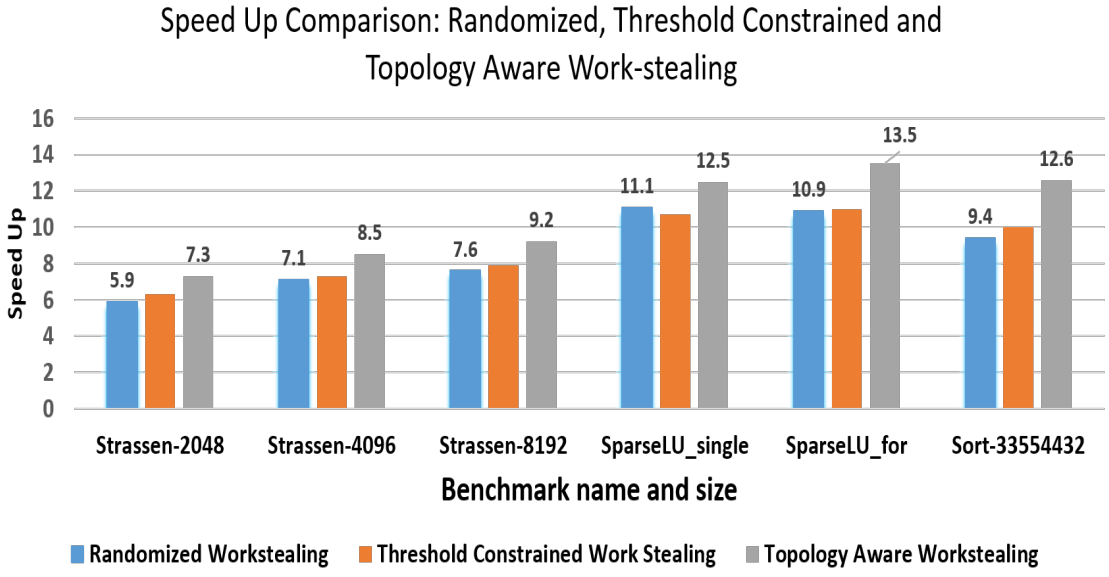

Figure 4.8: Speedup comparison of Randomized, Threshold constrained and Topology-aware work stealing techniques

remote memory task steal cycles and hence could contribute to performance gains.

4.7 Conclusion

This chapter focuses on analyzing the effect of on chip NUMA multicore processor architectures on user level work stealing run-time systems. This study emphasizes on porting work stealing runtime systems on to machines with on chip NUMA architecture where memory domain of a process is scattered across more than one node. The contributions proposed in this chapter are aimed to provide better performance by limiting the task stealing activity within memory locality. The concept of stealing domains is introduced in this chapter keeping memory locality as an assumption. This strategy can be extended to even last level caches provided, the cache topology parameters can be obtained by using sophisticated tools such as `hwloc`[60].

Chapter 5

Improving Shared Object locality in NUMA multi core processors

5.1 Introduction

Modern HPC server processors are containable of multiple cores and integrated memory controllers (IMC) [5] on chip. These processors support the feature of connecting separate memory bank to individual memory controller. Hence the system can be visualized as an On chip NUMA multi-core architecture. These architectures introduce a new challenge for work stealing run-times. Stealing tasks from a worker belonging to a remote socket core may increase memory latency thereby affecting the overall performance. This issue was addressed in the chapter 4. User level work stealing runtime systems like Cilk and TBB provide task construct for programming multi core processors. Starting from version 3.0 specification, OpenMP also started supporting task construct. OpenMP task support in two forms:

- **Implicit Tasks:** The code written as part of *parallel* is automatically translated to a task body by the runtime. In this case, synchronization is done at barriers.

- **Explicit Tasks:** The tasks created by the programmer using *task* construct are also considered as tasks at the runtime level. Explicit tasks also allow task to spawn child tasks and synchronize with them using *taskwait* construct.

The task body contains a set of statements that define the piece of parallel work. The execution statements of a task also can access task local data and also shared data. The shared objects are of two types:

- Global objects which are created prior to parallel region or task creation.
- Heap objects which are allocated using dynamic memory allocation.

The virtual memory mechanism of Linux allows the programmer to visualize the data organization as continuous storage. Shared data is also divided into virtual pages and these pages are logically contiguous in virtual address space. But when these virtual pages are mapped to physical memory, the physical memory frames need not be allocated contiguously on physical memory (Dynamic RAM). In particular, in a Non Uniform Memory Architecture(NUMA), these physical pages may be scattered across the available memory nodes to mitigate the memory bus bottleneck problem. In particular, large sized (more than a page size) shared data objects accessed by tasks experience this scattered physical allocation in parallel environment. As a result, the memory of a large shared object is scattered across physical memory pages belonging to different nodes in a multi socket multi core architecture. To minimize the remote memory latencies, the user level work stealing run-time must maintain proximity between shared objects and the tasks that access these objects. In other words, the runtime must guarantee task-mapping at a minimum possible distant to the physical memory bank where the shared data is located. At the same time, the user-level runtime system must ensure load balancing among the multiple cores.
In user level run-time systems, there is a trade off between ensuring object locality and load balancing. In a work staling runtime, a task belonging to a worker queue on one socket may access an object bound to other NUMA node.

- If work stealing run-times are developed only with the goal of load balancing, the object locality may be compromised. Due to work stealing strategy, a stolen task may be migrated to a memory node where it's shared data is not physically mapped to.

- If the runtime system is built only keeping goal of object locality, load balancing may be compromised. If all tasks accessing shared objects are grouped on to single socket, those worker threads experience high load and load-balancing can not be guaranteed.

In this chapter, we address the issue of shared object binding onto NUMA nodes, propose an adaptive solution for work-stealing runtime systems. A strategy to bind the shared objects across the memory banks to mitigate memory latencies is proposed particularly for work stealing run-times. This proposal is an extension of the work presented in [47]. The proposed solution is in the form of hints to the runtime system and the runtime system dynamically manages the object binding effectively so that shared object locality and load balancing are not compromised. These hints are compatible with newly introduced environment variables of OpenMP4.0 specification and are easily adaptable in future implementations. Experiment results show that this policy can improve the performance of standard benchmarks.

5.2 The effect of First Touch Policy

The default behavior of Linux kernel to address memory locality is to allocate an object nearest to the thread which first attempts to access it. This policy is not applied during allocation of virtual pages for the object but applied when a thread accesses the object for the first time. This strategy is called *first touch policy*. The common method of creating heap objects is invoking one of the standard C library functions: `malloc(), calloc()` or `realloc()`. These function calls do not allocate any physical memory immediately after the call. Later, when one of the threads belonging to the process tries to access this object, Linux first finds to which processor core the thread is pinned to and to which memory node the core belongs to. Based on the thread affinity,

physical memory is allocated on the memory bank to which the processor core is attached via the integrated memory controller. Such shared objects defined in standard workloads are dynamically allocated. These objects may be accessed by threads as a whole or in a part.

- **Whole Object Access**: Objects whose size is less than a page size and aligned to page boundary can be accessed by multiple threads. If all threads are pinned to the cores of same node, they can access it locally. Otherwise, remote memory access is done by few of the threads based on which node they are bound to.

- **Part Of Object Access**: In case of large 1D arrays or 2D arrays, different parts of the array may be accessed due to loop level parallelization. If such arrays are initialized by master thread (In fork join model like OpenMP, master thread starts first), due to first touch policy, all virtual pages may be mapped on single node as depicted in the figure 5.1. In such a scenario, all threads belonging to other socket experience remote access latencies.

If the initialization of 1D or 2D array is done by multiple threads, physical pages are allocated based on the affinity of initializing thread. In such a case, the memory allocation of a process is distributed among the nodes. All threads pinned to the node containing the memory page find local access whereas the threads pinned to processors of other node find it a remote access. This scenario is depicted in the figure 5.2.

5.2.1 Analysis

To analyze the impact of work stealing environment in NUMA multi-core environment, we executed matrix multiplication benchmark program on our experimental server with dual socket Xeon E5-2620. By instrumenting the code with libnuma API calls, we tried to find the total number of accesses to the base address of each row in a matrix. We did not consider the individual element address since all the remaining elements of the row are stored in contiguous locations after the first element and caches guarantee locality of

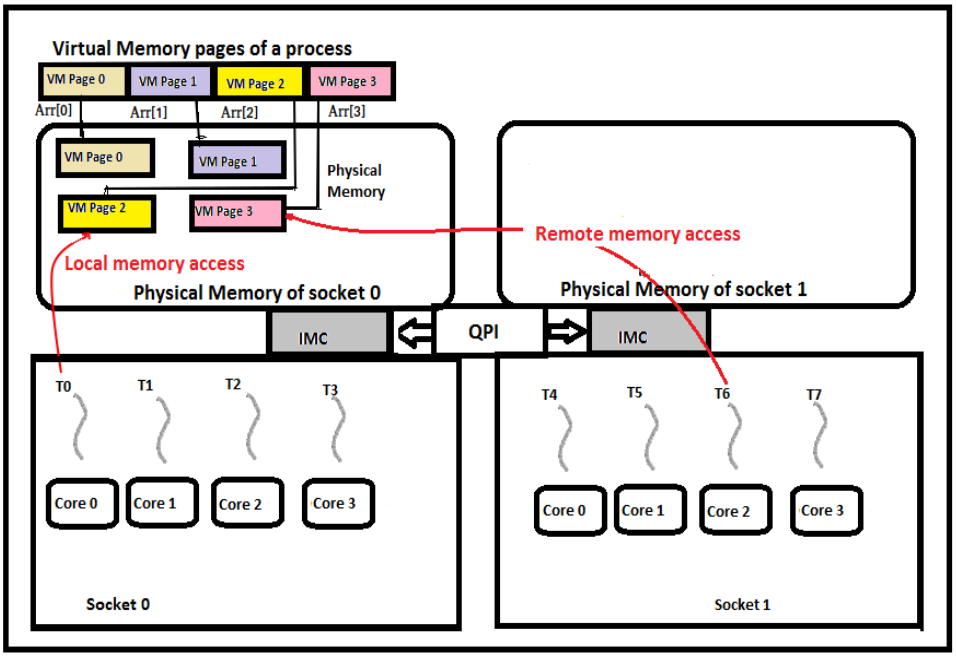

Figure 5.1: Poor memory allocation due to first touch policy

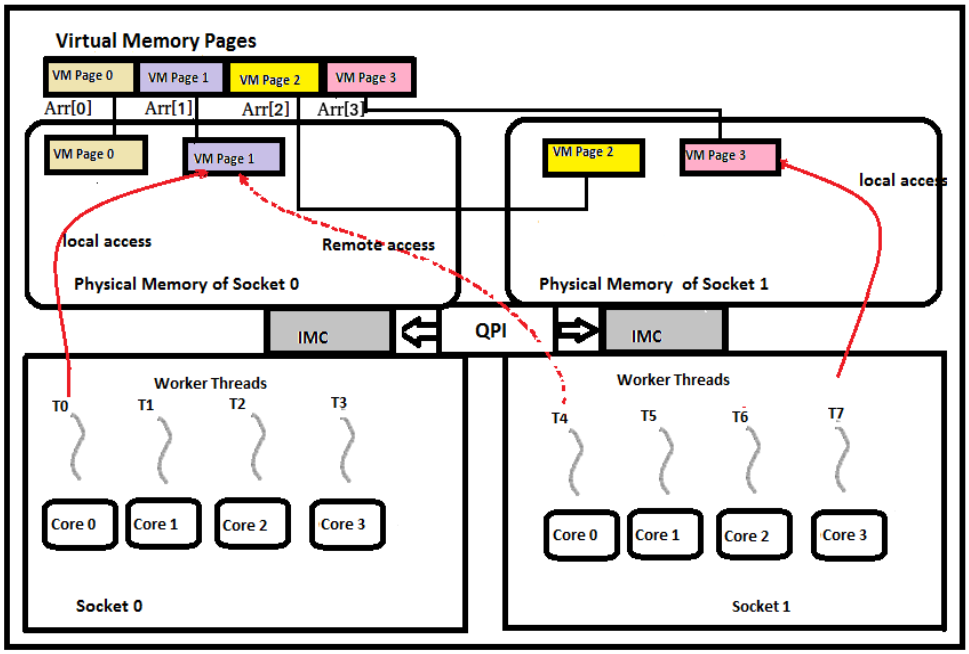

Figure 5.2: Better First-touch policy

reference for the rest of elements in same row. It can be observed from

Table 5.1: Analysis of scattered data among NUMA nodes

Matrix Size	Row-wise Access Count	
	Node 0	Node 1
512	512	0
1024	1024	0
2048	254	1794
4096	3873	223

the table 5.1 that for smaller sized matrices 512×512 and 1024×1024, all the elements are stored on single node. The reason is entire matrix row did not cross a page size (4096 bytes in our experimental setup). All the row elements of smaller sized matrix are mapped to a single node. In this scenario, the threads pinned to cores belonging to Node 1 will experience remote memory access latency as depicted in the figure 5.1. But for the larger sized matrices, the physical pages are scattered across the nodes. In case of 2048×2048 matrix, more physical pages are mapped to the Node 1 whereas for 4096×4096 matrix, more number of physical pages are mapped to Node 0. In this scenario, depending on the allocation of physical pages, few threads pinned to minor allocated node will experience remote memory access latency. For larger sized matrices, the matrix is split across the nodes. In turn, each row of the matrix may also scatter across multiple pages when an entire row can not fit into a single page. A single array being mapped to multiple nodes is depicted in the figure 5.2.

5.3 Tasks and Shared Objects

When call to task creation API is made during the runtime, the newly created task is added to the worker queue. The task creation APIs generally allow the programmer to pass the local arguments to the task. The syntax of task creation function in most runtimes looks like

```
void creatTask( type1 arg1, type2 arg2,typen argn );
```

The shared objects such as global and heap objects are accessed by these tasks with default privileges without mentioning them in their prototypes. But few environments like OpenMP provide few additional constructs such as `shared, critical, atomic` etc. Recent specification of OpenMP 4.0 [62] adds few more features how the tasks are to be pinned to the processor group keeping NUMA multi-core processors. These constructs give hints to the runtime to manage the object state. For example, the keyword `shared` gives hint to the runtime that the object is accessed by multiple tasks and suitable locking and synchronization care need to be taken. Setting the value for OMP_PLACES environment variable in OpenMP 4.0 [62] almost does the same grouping as that of stealing domains introduced in the chapter 4. But the stealing domain strategy we proposed is particularly meant for work stealing run time systems which is not strictly mentioned in OpenMP specification [62].

In a task based parallel runtime, the allocation of shared objects is done prior to creation of the tasks and all these objects are treated as heap objects. These shared objects are allocated according the default first-touch policy [1] in Linux environment. The first-touch policy is the main cause of object scattering across different memory nodes. The key idea in our proposed work is, the keywords specified while programming tasks can play important role in task binding and object binding. This helps the runtime to become more dynamic and the programmer need not depend on explicit management of task affinity using environment variables like KMP_AFFINITY from intel compilers [63]. The proposed strategy puts its best effort to map the user created tasks automatically on to the processor cores to improve object locality. To achieve this, the runtime depends on the hints given by the programmers in the form of directives and clauses.

5.4 Mathematical Model

In a work stealing runtime, task is a unit of work that is executed asynchronously by a worker thread. Each task has its own data and whole or

part of the data may be shared with other tasks. We made an attempt to model the relationship between tasks and the objects accessed by the tasks. Let there be N number of memory nodes in the system and shared data objects are bound to these nodes according to first-touch policy of Linux. Each memory node may consist of zero or more number of shared data objects and let D_N denote such mapping of shared data objects onto these memory nodes. We assume that there are T number of tasks created in the system. These set of task tasks may try to access the set of shared data objects and let D_T denote such mapping. Hence, at a given instance, there are a total of D shared data objects involved in the system where D is given by the equation 5.1.

$$D = D_T \cup D_N \tag{5.1}$$

In this scenario, a page fault can occur on a system whenever $D_T - D_N = \Phi$ The vector cross product, $D_N \times D_T$ consists of all the possible ordered quadruples between node-object and task-object mappings. The factor which is used to decide effective task to shared data object binding is called *similarity index* and is modeled in the equation 5.2

$$S_{ij} = \frac{|D_{T_i} \cap D_{N_j}|}{|D|} \tag{5.2}$$

The similarity index indicates the relationship between shared object and the tasks that try to access the object. If objects can be mapped on to the memory nodes such that the S_{ij} value is high, most of the tasks can access the shared objects at minimal access delays there by reducing overall memory latency.

5.4.1 An Example

Let the set of tasks in a run-time system are t_0, t_1, t_2, t_3, t_4 and t_5 and let the set of shared objects be s_0, s_1. Let us assume that the tasks t_0, t_1, t_3 and t_5 access shared object s_0 and the tasks t_1, t_2 and t_4 access shared object s_1 respectively.

Then the mapping of tasks and shared objects can be represented using

$$D_T = \{(t_0, s_0), (t_1, s_0), (t_2, s_1), (t_3, s_0), (t_4, s_1), (t_5, s_0)\}$$

where t_i, s_j values of each ordered pair (t_i, s_j) represent ith task and s_j jth shared object respectively. The ordered pair is based on access relationship between t_i and s_j.
The mapping of shared objects to nodes is given by:

$$D_S = \{(s_0, n_0), (s_1, n_1)\}$$

The ordered pair (s_j, n_k) means that shared object s_j is mapped to memory node n_k.
The mathermatical relations, D_T and D_S can be transformed into matrix form for the ease of implementation. The matrix D_T represents the mapping of tasks to shared objects where s_j represents jth shared object. The matrix D_S represents object to memory node mapping where, s_j represents jth shared object and n_k represents kth node to which s_j is bound to in each ordered pair (s_j, n_k).

$$D_T = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The matrix D_S represents the mapping of shared objects to nodes.

$$D_S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The matrix $D_T \times D_S$ represents the task mapping matrix on to the nodes.

$$D_T \times D_S = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The example considered here is a trivial one and each element of the matrix is significant. But in a real scenario, where the number of tasks are in thousands and the count of shared objects is high, most of the matrix entries may be *null* resulting in a sparse matrix. While implementing the concept of similarity, hash tables were used to avoid storage overhead of sparse matrix representation.

5.5 Implementation

Though the mathematical model involves sets and join operations, implementing them as they appear at runtime level may introduce additional space and performance overheads. Hence the concept of similarity between shared object mapping onto nodes and task using shared objects is simplified with the help of libnuma API [12] and hash tables. The API of libnuma allows the runtime to find the node where the shared object is bound to. Each shared object is represented by an index in the hash table and the tasks accessing that shared object are mapped to that hash index. Hash index directly hints the runtime to identify which tasks access a particular shared object. The runtime can easily derive memory node mapping of shared object using libnuma API calls. Indirectly, the runtime could achieve task clustering based on shared objects access. Since a stealing domain at runtime level represents a memory node at physical level, task mapping can be done based on locality. Then the work-stealing runtime maps the tasks to respective worker queues of specific stealing domain according to object locality.

5.5.1 Task Level Dispatcher

Maintenance of *object-locality* is taken care by introducing an additional module within the runtime architecture called *task-locality-dispatcher (TLD)*. The main job of TLD is to dispatch the newly created tasks on to the appropriate worker threads. A task is mapped to a worker thread pinned to a core of a socket (memory node) on which shared object is bound to.

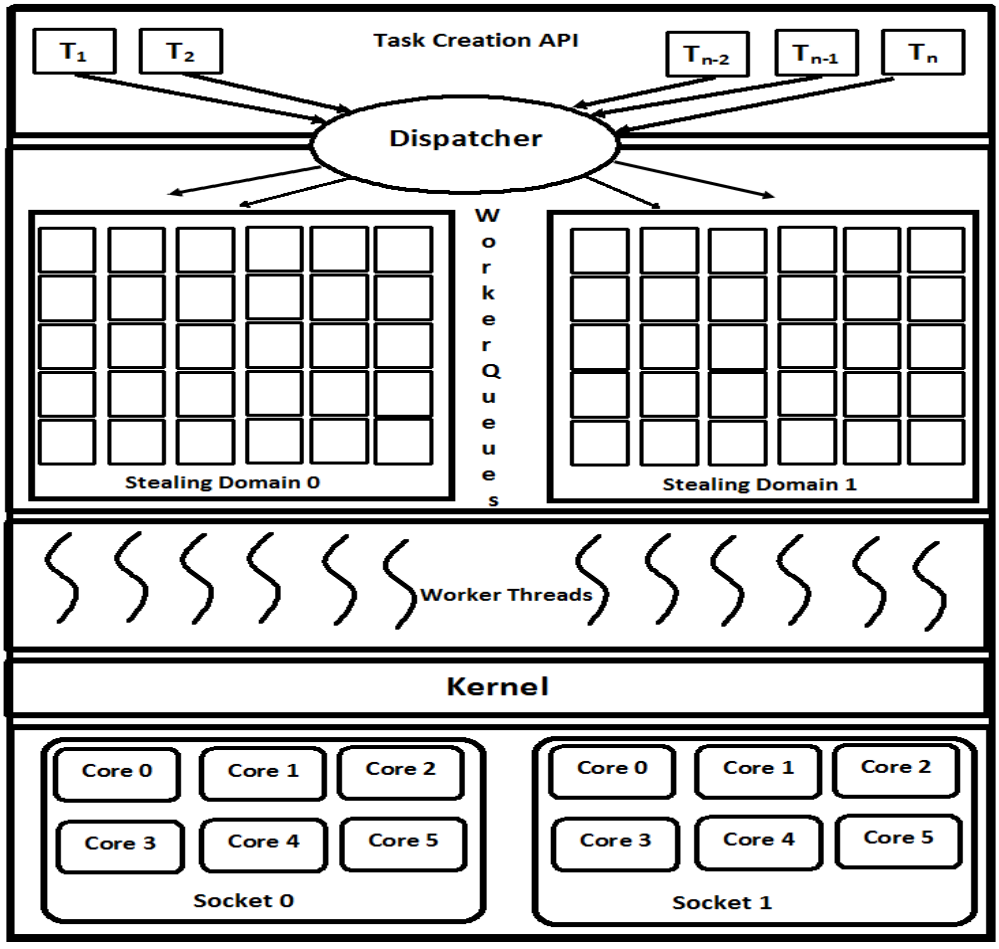

Figure 5.3: Dispather added to existing work stealing run-time for dual socket 12-core Xeon-2620 architecture [2]

5.5.2 Generic Approach to Handle Shared Objects

Conventionally all task based run-times provide task construct and allow the programmer to implement task body as a block or a function. The arguments required by the task are sent as formal parameters to the function. In case of directive based languages like OpenMP, task body is translated to an outline function by the compiler. This outline function consists a block of code with an additional information about the object. This additional information is all about the object is a value type or a reference type. Since all shared objects are translated as reference types, the memory node mapping of the object can be obtained using libnuma API calls. TLD depends on this information to take a decision on mapping tasks to respective workers belonging to memory nodes.

The common syntax followed by C/OpenMP translators to create a task is given below listing. This syntax allows the programmer to specify the arguments used by the task. Our runtime adds some additional information such as pointer to the task object ($thisTask()$)

```
void createTask( type1  arg_1, type2 arg_2, ...type_n arg_n  ){
    shared(obj1, thisTask() );
    shared(obj2, thisTask());
    /*Task body */
    .
    .
    .
}
```

In the above task definition, $arg1, arg2, ...$ $argn$ are the task local arguments used by the task for processing. $obj1$ and $obj2$ are the reference objects that may be shared by more than one task. These shared objects might be allocated on one of the memory nodes according to first-touch policy of operating system. In NUMA multi core environments, memory allocation is done based

on default first touch policy of Linux if NUMA feature is enabled. That is, memory allocation of objects is done on the memory bank to which the worker thread is bound to. The clause `shared` gives a clue to the run time the necessary memory node binding information. The directives or clauses are just the macros which get expanded to runtime system function calls. These function calls guide TLD to map the task to the memory node where the object is bound to. The code snippet given in the example above is translated by the preprocessor into equivalent C code presented in the algorithm 3.

Algorithm 3: Add Task

input : void * obj, Task t

1 $node_j$ = getNUMALocality(obj);
2 sd_i = getMemoryStealingDomain ($node_j$);
3 enqueue(t , $sd_i.workerqueue$);

TLD's job is to find out to which node the shared object is bound and add the task to the worker queue which belongs to the stealing domain associated with that node. TLD executes the following procedure taking libnuma API help. If the programmer specifies a shared object as part of task syntax, the algorithm **Create Task** tries to add the newly created task to a worker queue belonging to the stealing domain where shared object is bound. It also tries to add the task to a queue with less load (chapter 4) within the stealing domain. This algorithm is invoked during task creation.

In the algorithm **Create Task**, the values s and S are computed lower and upper threshold values of the task queue loads [47]. By introducing these minimum and maximum threshold values to the queue levels, the selection of a wrong victim in task- stealing can be avoided. The algorithm ensures two purposes:

- Newly created task is bound to the same stealing domain where the shared objects are bound to.
- The task is added to only such queue with less work load.

Algorithm 4: Create Task

```
   input : task parameter list
1  initialize task object;
2  if ( task t does not use any shared object ) then
3      add t to the queue with load below s;
4  else if (t uses a sharedobject o ) then
5      node_x = getNUMALocality( o );
6      sd_x = getMemoryStealingDomain( node_x )
7      i = getWorker(sd_x);
8      if ( taskCount(que_i) < S ) then
9          enqueue(t , que_i);
10     else
11         find queue que within sd_x with load < S ;
12         enqueue( que, t);
13 else if ( t uses a shared object but object is unbound ) then
14     find que such that taskCount(que) < S of stealing domain sd_y;
15     enqueue( t , que );
```

5.6 Experimental Evaluation

To evaluate our proposed shared object locality strategy, we added the proposed task stealing functionality library with topology awareness [47] to an existing OpenMP runtime. The worker threads created during initialization of the runtime are grouped as stealing domains discussed in the chapter 4. TLD strategy proposed in this chapter is implemented as a module and embedded as part of the runtime. The entire architecture of topology aware task stealing along with TLD is presented in the figure 5.3. Since, we have been using OMPi[61] [64] runtime for all our previous chapters, the TLD module is implemented as an add on module to this runtime.

TLD maintains the hash map with the number of buckets equal to the total number of shared objects. Shared objects in an OpenMP program are explicitly specified using directives and clauses. These shared objects can be identified even before the compilation begins or during preprocessing phase

of C compiler. As the tasks are created during, they are added to the hash map based on the shared objects they depend on. TLD ensures that all the tasks that use a particular object are pinned on to the worker threads which are bound to the same stealing domain or same socket. Though TLD is guaranteeing locality of the objects, task implementation model we followed is a *untied model* so that the task can be allowed for stealing without restriction. Stealing domains ensure that the task stealing is limited within the local socket to avoid task migration from remote socket.

OMPi[61] is a light weight compiler of OpenMP 3.1 specification which does source to source translation of OpenMP directives to a specific thread library. It also supplies an implementation of runtime system with work stealing strategy for load balancing. The runtime is built with support of pthreads as worker threads from native pthread library of Linux. It also supports other threads such as psthread [65] library. It gives flexibility to the programmer to link any thread library with minimal implementation of headers and start up functions with prototype *ort_xxx()* where *xxx* can be replaced by thread library implementation.

In the following C/OpenMP code snippet, two tasks have been defined as part of parallel region. Hence a total of $2 \times OMP\_NUM\_THREADS$ tasks are created. These tasks are scheduled on $OMP\_NUM\_THREADS$ worker threads. The integer object *shObj* is given with `shared` clause and is accessed by all tasks.

```
#pragma omp parallel
  {
    int shObj = 0;
    #pragma omp task shared(shObj)
    {
      shObj++;
    }
  }
```

```
    .
    .
    .
```

OMPi source to source translator generates the following code:

```
 static void * _taskFunc0_(void * __arg)
{
  struct __taskenv__ {
      int (* shObj);
    };
  struct __taskenv__ * _tenv = (struct __taskenv__ *) __arg;

  /* byref variables */
  int (* shObj) = _tenv->shObj;
    .
    .
    .
  {
    (*shObj)++;
    CANCEL_task_10 :
      ;
  }
  ort_taskenv_free(_tenv, _taskFunc0_);
  return ((void *) 0);
}
    .
    .
```

The node binding of `tenv->shObj` can be known using *libnuma* [12] API calls. The task is mapped onto the worker thread of that particular stealing domain where the object is physically bound to as explained in the algorithm**??**. The detailed description of threshold constrained, locality awareness and shared object locality algorithm is explained in the algorithm

Add Task.

5.6.1 Benchmark Programs

Barcelona OpenMP Task Suit(BOTS) [15] consists of set of task based benchmarks. These task parallel programs spawn large number of tasks. Proportionate to the problem size, these programs also exhibit NUMA sensitiveness. We executed the same benchmarks as in the chapter 4 for assessing the effect on performance.

The **Strassen benchmark** calculates the multiplication of dense matrices using the Strassen's algorithm. The algorithm divides the number of multiplication operations by splitting each matrix into 4 equally divided submatrix chunks. The output array C is split into 4 sub-matrices in the first matrix multiplication function call. Each sub-matrix is calculated in parallel using independent tasks. Each task splits the submatrix into 4 smaller submatrices and generates tasks to handle them. This recursive computation guarantees that the child tasks always compute the output elements which are allocated in the parent task. If parent task and child tasks belong to the same node, these tasks can take the advantage of data locality. The tasks should not be migrated to a different node because of stealing action initiated at other node.

We parallelized the initialization of matrix in `init_matrix()` function such that the matrix elements are interleaved across the available nodes. This parallelization mitigates the poor first touch policy effects as depicted in the figure 5.1. *aligned_alloc*() library function is used to allocate the output array with a page boundary alignment. The starting index of the corresponding sub-matrix is calculated in a parallel region. A thread is selected for each NUMA node in the parallel region. Then the thread calls *numa_setlocal_memory*() to migrate memory pages to the local NUMA node.

The **Sort** benchmark is a recursive task implementation of merge- sort

[15]. Since sorting involves permuting the elements of vector, it leads to remote memory access. The array size (N) we considered while running the program is, 33554432 which is the default input defined by the benchmark. The page size supported in our experimental setup is 4K. Hence, the array occupies 32,768 pages of virtual memory and has to be mapped on to physical memory. The program consists of a function `fill_array()` which initializes the array sequentially. If this initialization is done by single thread, all the array data may be bound to a single node where the master thread is pinned to as depicted in the figure 5.1. Hence, we modified the benchmark code in `sort.c` file such that, the array elements are initialized in parallel. Parallel initialization ensures that those threads which initialized part of the array can find that part of the array physical pages in local node. During sorting phase, recursive tasking is applied on each sub-array in parallel. These recursive subtasks are the children of the parent task and mostly access the array data. If a task is stolen by a worker pinned on remote node, it results in a remote memory access. Such remote steals are minimized because of stealing domains. Again in the merge phase of sorting, if the four sub arrays are located on the same node, merging subtasks can take the locality of reference advantage within the same node. In our experimental environment, we considered the default array size and cut off values specified in BOTS.

SparseLU benchmark calculates the LU decomposition of a sparse matrix [15]. The input is a 2D array of which each element is the memory pointer to the submatrix. For larger matrix size, SparseLU allocates a submatrix to the locations where the problem matrix has non-zero values possibly on different NUMA node. The initialization of matrix code in function `genmat()` was made parallel such that the matrix data is scattered across the available nodes. BOTS contains two versions of this benchmark:

- In `Sparse-LU-for` version, tasks are implemented using `parallel for` construct of OpenMP
- In `Sparse-LU-single` version, tasks are implemented using `single` construct of OpenMP

The programs are executed on server with dual socket 24 thread (6 cores per socket and two hyper threads per core) Xeon E5-2620 with Linux Kernel version 3.11 environment. All workers are mapped to cores using identity affinity with the help of likwid tools [58].

Table 5.2: Execution Time Comparison Topology aware Work-stealing Vs Locality binding strategies

Benchmark Name - Size	Execution time in seconds		
	Topology aware Work Stealing	NUMA aware Locality Binding	Serial
Strassen-2048	0.2476	0.182148	1.8142
Strassen-4096	1.53495	1.091279	13.139
Strassen-8192	10.1219	7.010121	92.7439
Sort-33554432	0.5335	0.496975	6.696
SparseLU-single	0.8266	0.719509	11.1524
SparseLU-for	0.8602	0.641538	10.842

Strassen benchmark code doesn't contain any task code with `shared` clause, but by using the source to source translator of OMPi, we added the statements to improve the locality of a matrix row by the tasks which access it. The speed up gain in case of 2048, 4096 and 8192 is significant.
Sort benchmark is computation intensive benchmark and doesn't contain any task definition with `shared` clause. In this benchmark, the number of remote accesses during merge phase are dependent on the array element values. The list is initialized with random element values, and based on these values and element permutation is done accordingly. We could notice little performance gain. The possible speedup gain was achieved only by threshold constrained and topology aware work-stealing strategies of chapters 3 and 4.
SparseLU is again a matrix based program and is bandwidth(data) intensive. This program makes use of shared objects among tasks. Despite the load balancing optimization effort put in chapters 3 and 4, it could not yield better speed up gain. But locality object binding strategy allows the tasks to bind to the nodes, with proximity of data. Hence significant gain in performance can be noticed both `SparseLU_for` and `SparseLU_single` versions.
To investigate the reasoning for achieved speed-up, we compared the total

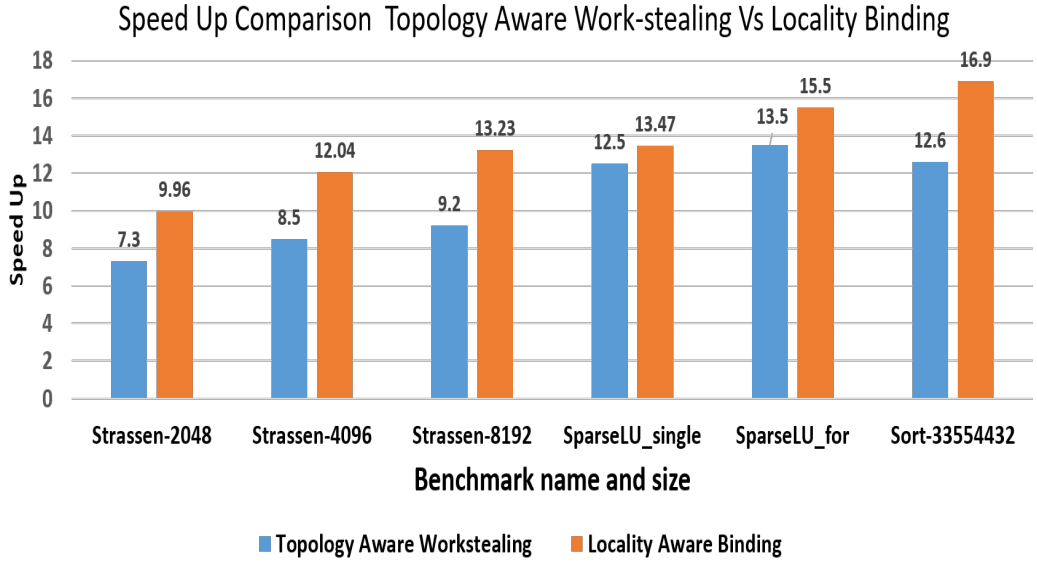

Figure 5.4: SpeedUp comparison of topology aware workstealing with locality object binding strategy

remote data volume accessed by each benchmark. This comparison is done between load balancing strategy proposed in the chapter 4 and locality object binding strategy proposed in this chapter. We could measure remote data volumes of each program using command line performance counters tool likwid [58]. The remote data volumes accessed by each program are given in the table 5.3.

Table 5.3: Remote Data volume access Comparison: Topology aware Workstealing Vs Locality binding strategies

Benchmark Name - Size	Remote data volume access by in GB	
	Topology Aware Work Stealing	NUMA Aware Locality Binding
Strassen-2048	0.9986	0.7836
Strassen-4096	5.2197	4.1961
Strassen-8192	39.3668	30.7298
Sort-33554432	3.4797	2.4709
SparseLU-single	5.2117	3.705
SparseLU-for	3.4587	1.996

It can be observed that in Strassen benchmark, the remote data volume is proportionate to the problem size. The speed-up improvement is on par with decreased remote data volume. Sort benchmark is 1D array based program and the percentage of random remote data access is less due to problem splitting and sequential sorts are applied on smaller sized data. When base condition is reached, sequential quick-sort and insertion sort routines mostly find locality of reference except the starting element. Sparse-LU versions is highly susceptible to problem size and remote data volume. Though the input matrix size was 128×128, since the memory allocation and accessing of matrix elements is highly random, the impact on performance is very high.

5.7 Conclusion

Deciding which objects are shared among which tasks is a pure runtime issue. Several proposals are made to provide a solution to this problem but many of them are static in nature or profile based[8] [66] [67] . Static solutions try to analyze the source code and detect the object access among parallel entities. Profile based solutions require the program to be run for first time to know the access patterns of the threads in the form of mamory access profiles. These profiles are used to decide which threads can be mapped to which worker threads in the user level runtime. The work proposed in this chapter is neither completely static nor dynamic. It is a hybrid approach based on the fact that parallel directives give enough information to the runtime. The programmer knows better during programming that what shared objects are required by that task. The proposed strategy is an approach based on the compiler and runtime hints and can be easily added in OpenMP 4.0 [62] compatible implementations.

Chapter 6

Affinity aware synchronization in Work Stealing runtimes

6.1 Introduction

Task based programming environments like Cilk [3] and TBB [4] implement work stealing based load balancing in their user-level runtime implementation. These run-time systems assume uniform memory access in multi-core processors, as the earlier multi-core architectures were behaving like symmetric multi processors (SMP). Recent multi-socket multi-core processors support multiple memory modules thereby resulting in a Non Uniform Memory Access (NUMA) architecture. Work stealing run-times can be more effective if they are aware of the underlying NUMA topology. Work stealing run-times typically rely on lock-based synchronization to guarantee the coherency of shared mutable state among the worker threads. Synchronization constructs such as mutex locks, condition variables and barriers are extensively used in the implementation of these run-times. The worker threads or virtual processors of these run-times are implemented using user level threads (ULT) such as *pthreads*. Hence, the synchronization constructs provided by respective ULT libraries are directly adopted for synchronization implementation. The locality of these synchronization constructs in NUMA multi-core processors has considerable impact on the performance of these run-time systems [68]

[69] [70]. This chapter studies the effect of locality of these synchronization constructs and proposes NUMA awareness to them. The proposed methodology is implemented using a source to source translator of OpenMP run-time and evaluated using OpenMP micro-benchmark programs.

In multi-socket multi-core processors, each socket has a separate integrated memory controller(IMC) interfacing with separate memory module to minimize overall memory latency. The processors(cores) on each socket are connected via an interconnection network such as Quick Path Interconnect (QPI) [5] in case of Intel processors and Hyper Transport link [6] in case of AMD processors. The processors within a socket can access the memory locations at faster rate via IMC. But when a processor of one socket tries to access a memory location attached to other socket, the accessing latency is more than local memory access latency. This is a common phenomena in NUMA architectures. The fraction of the time taken to access a remote memory location to the time taken to access a local memory location is called *NUMA ratio* [71] (R_{NUMA}) and is given by the following equation

$$R_{NUMA} = \frac{T_{remoteaccess}}{T_{localaccess}} \tag{6.1}$$

If the operating systems are aware of the interconnection topology, kernel itself can manage proximity between processing elements and storage elements. It will allow the user applications to access the data from memory in such a way that the memory latencies are minimized. But the kernel's memory management unit can address locality issue at process level but not at thread level. Linux kernel uses the same system call *clone()* for creation of process and thread, thread and process are treated as same at kernel level using the same `task_struct` object [72][73]. Linux kernel follows first touch policy[41]. The first-touch policy ensures the data locality to threads based on first write accesses to the memory location. It is not based on allocation affinity i.e. physical memory for an object is not allocated on node when a thread requests heap memory allocation rather when actually thread ini-

tializes(write) the object for the first time. This is applicable to all types of dynamically allocated objects.
Synchronization constructs such as spin lock, mutex, condition variables are not an exception for this first touch policy. Implementation of primitive synchronization constructs is done using either busy waiting or blocking and follows generic code given below.

```
int   acquire_lock ( lock_t  *lock ){
  while ( ! cmp_and_swap(lock, 0, 1 ) )
    pause();
}

int  release_lock ( lock_t  *lock){
   *lock = 0;
}
```

To acquire a lock, a thread repeatedly reads the value of *lock* variable as part of `while` loop in `acquire_lock()` function. Repetition for testing the lock value is done $(n-1)$ times and thread could succeed in nth attempt of *cmp_and_swap* operation. Hence, there are a total of $n-1$ read attempts for testing the *lock* object value and one write attempt on acquiring and updating its value. These lock variables can not take the advantage of cache locality since they need frequent update (`lock` is defined `volatile` in implementation). If these lock objects are all initialized by a single thread (the master thread in fork-join model), there is a possibility that in addition to busy wait overhead, threads also suffer from remote memory access latency in multi-socket architectures. Figure 6.1 depicts the scenario where the main thread whose affinity is on node 0 initializes a lock object. The threads pinned on to the cores of other node suffer from remote memory latency to access. The delay is in multiplied by number of attempts of this remote latency in the process of acquiring the lock.

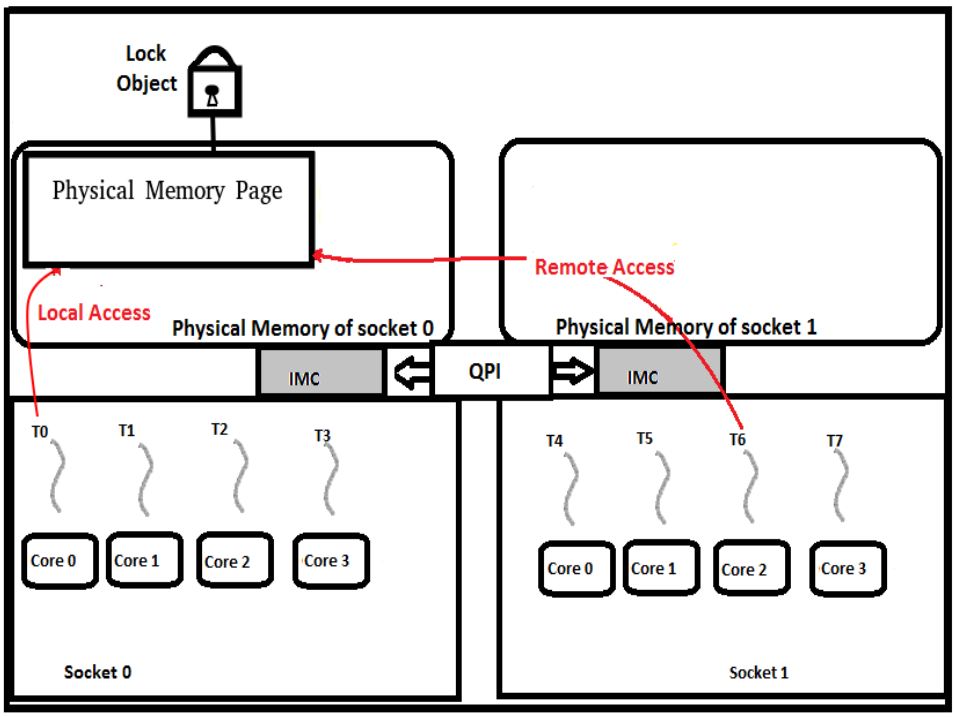

Figure 6.1: All virtual pages physically mapped to single node

6.1.1 User Level Work Stealing Run-time Systems

Because of the creation overhead of process and thread constructs, task based run-times are moved to user level. The parallel run-times such as Cilk, TBB and OpenMP operate at user level maintaining a pool of threads using the native thread library support of kernel. Instead of creating a thread for every parallel activity, these run-times provide a lighter construct than thread called *task*. During the initialization, the master thread of these run-times, creates a pool of native threads called *worker threads* whose life time is till the end of parallel program. After creation, these worker threads wait for user created tasks, waiting on a condition variable associated with a mutex lock. These tasks are generally kept in a queue. The run-times of Cilk, TBB and few implementations of OpenMP extend the functionality of thread pool concept as work stealing pools where each worker thread maintains a separate double ended queue [51]. One end of the queue is accessible to worker thread to pop and execute the task body. The other end of the queue is accessed by other workers which are idle with empty task queue[53]. The worker thread which attempts to steal a task from other worker queue is called *thief* and the worker thread from which a task is stolen is called a *victim*.
In a work-stealing based run-time, there is a need of synchronization protection to avoid race conditions among worker threads in the following cases:

- All worker threads wait for tasks waiting on its own condition variable and associated mutex lock.
- Each worker queue maintains a lock to inform that no more tasks can be added further until there is enough space in it.
- A thief worker has to acquire a lock before stealing a tasks from other queue so that no other thief attempts to steal on the same queue.
- A barrier is maintained so that if a worker thread has finished all its work, it waits until all other workers finish.

The affinity of worker threads can be controlled explicitly using *numactl* [12] or *likwid* tools [58], or by setting environment variables $KMP\_AFFINITY$

from Intel compilers or OMP_PLACES in OpenMP 4.x specification [62]. But these are static settings and can be specified while starting program execution. The affinity can't be changed during run-time of the application using external tools. All the synchronization constructs such as lock variables, condition variables and barriers are created and initialized by the master thread of a work stealing run-time. If these objects are allocated by the master thread, the physical memory of these objects may be allocated on the node where master thread is executing. As a result, all primitive synchronization operations of the worker threads which are pinned on different node processors suffer from remote access latency. Hence, the study of lock variables and synchronization techniques at user-level implementation has regained its importance because of NUMA multi-core processors [13][14]. The purpose of this chapter is not to propose a new locking methodology, but to suggest how to design and implement existing locking techniques in a work stealing run-time for NUMA multi-core environment. We study the effect of lock object's affinity and propose NUMA extensions to pthread lock API particularly for work stealing run-times and evaluate the proposed method using micro-benchmark [74] [75] [76]programs.

6.2 Effect of Synchronization constructs Locality in Work Stealing

The important component in work-staling run-times is a thread pool. The run-time starts its execution with a master thread. This master thread is responsible for initializing all threads and data structures. Master thread of a work stealing runtime does the following actions before any new fork-join activity takes place.

- A pool of worker threads is created during the initialization of the run-time. The total number of worker threads (including the master thread) is equal to the number of processing elements at hardware level.
- Memory allocation for the worker queues and the associated mutex

locks is done during initialization.

When a worker thread is created, it does not start execution immediately since it's worker queue is empty. Hence it waits on a condition variable until task level dispatcher (TLD) adds newly created tasks to its queue. The worker thread is engaged in busy waiting state using system level atomic primitive such as CAS(compare-and-swap). If the worker thread attempts n times to read the condition variable's value which is allocated on local node, the busy waiting time can be given by the following equation where T_{lock} represents the time spent on *acquire_lock*() operation, $T_{localaccess}$ the local memory access latency time and T_{CAS} the time spent of compare and swap operation at machine instruction level.

$$T_{lock_{local}} = \left(\sum_{i=1}^{n} T_{localaccess} \right) + T_{CAS} \tag{6.2}$$

Since the master thread initialized memory for lock objects, all physical memory for such locks is done on the node where master thread runs. But the worker threads are pinned to different cores on different nodes. If the condition variable and the associated mutex lock variables are stored on a remote node, the delay involved in busy waiting can even be greater than that of equation 6.2. Because on every primitive CAS operation, it has to access remote memory location to compare the lock variable value. In this scenario, the busy waiting time can be given by the following equation

$$T_{lock_{remote}} = \left(\sum_{i=1}^{n} T_{remoteaccess} \right) + T_{CAS} \tag{6.3}$$

If synchronization objects are not NUMA-aware, they are susceptible to NUMA effects. These effects not only result performance mismatch between cores but also cause starvation or even live-lock. By the time lock is available and its status in known to the thread on other node, a local thread on same node where lock object is located may grab the lock. The effect of locality of spin locks is studied in[77]. These circumstances may cause the thread on remote node to starve. Optimizing the placement of shared lock objects across

cores of different sockets minimizes NUMA effects. If shared lock is accessed by a group of worker threads on a single socket, and if lock objects resides on same socket, the workers can take advantage of locality of reference.

On the experimental set up of dual socket Intel's Xeon-E5 series running Linux kernel, Memory Latency Checker program [56] results presented in the table 6.1. which yields average NUMA ratio $R_{NUMA} = 1.625$.

Table 6.1: Memory latency values on dual socket Xeon E5-2620 series processor

NUMA Node	0	1
0	77.3 ns	124.7 ns
1	122.8 ns	75.0 ns

Hence it can be observed from the equations 6.1, 6.2 and 6.3 that the additional delay involved in testing a remote condition variable and lock is about a factor of $(R_{NUMA} - 1) \sum_{i=1}^{n} T_{localaccess}$. This leads to a performance penalty of $\approx R_{NUMA}$ times.

The delays explained in the equations 6.1, 6.2 and 6.3 are also applicable to worker thread barriers. If barrier variable is allocated on a different node, the worker threads have to experience the same delays while joining the worker threads. To evaluate the above theoretical concept, a simple spin lock based program was run on dual socket Xeon E5-2620 Linux machine. Average starvation time of spin lock access by two threads pinned on same socket cores versus pinned on cores of different sockets is presented in the table 6.2.

Table 6.2: Comparison of spin lock access times local vs remote

Scenario	Average lock access time
Contending threads on same socket	15.435 ns
Contending threads on different sockets	38.428 ns

6.3 Thread Affinity Lock API for NUMA multi-core processors

The possible improvements that can be done for work stealing run-times in NUMA are:

- Using NUMA aware locks.
- Allow worker threads to access local locks whenever there is contention among threads within a node.
- Access remote lock only when a worker thread has to steal tasks from other node worker queues.

Recently, lock cohorting [74] [75] [76] was proposed as a generalized methodology for bringing NUMA awareness in lock implementation. Lock cohorting approach is based on a combination of two locks: one used as a global lock and another used as local locks (there is one global lock for all nodes and one local lock per NUMA node). In the work staling run-time we group all worker threads pinned to the cores of single socket as stealing-domain (chapter 5). As a general rule, if there are M sockets with N cores on each socket, at runtime level, there are N stealing domains. If work stealing run-time is implemented using the proposed local locks along with lock cohorting, all N worker threads belonging to a single socket can depend on local lock for synchronization within the socket. When one of these N workers has to access a remote node for task stealing among M nodes (in rare situations), it can depend on global lock.

POSIX standard pthread library defines `pthread_xxx_init()` for initializing lock object; `pthread_xxx_lock()` for acquiring a lock; and `pthread_xxx_unlock()` for releasing a lock where xxx may be either mutex or spin. But POSIX standard locks are specifically meant for symmetric multi processing (SMP) and not aware of NUMA environment. Hence, we propose NUMA and first-touch policy aware API with common syntax `pthread_NUMA_xxx_lock()` and `pthread_NUMA_unlock()`. Simlarly, for condition variables, the proposed

syntax is `pthread_NUMA_cond_init()`, `pthread_NUMA_cond_wait()` and `pthread_NUMA _cond_signal()`. These proposed API calls can collect the thread information to which node the thread is pinned to using *libnuma* API calls [12]. The common steps in pseudo code form of the proposed API calls are presented in `pthread_NUMA _xxx_init( )` function.

```
int  pthread_NUMA_xxx_init( pthread_xxx_t *var)
{
  tid=pthread_self();
  cpu_id=sched_getcpu(tid);
  node=numa_node_of_cpu(cpu_id);
  var=numa_alloc_onnode(sizeof(*var),node);
}
```

6.3.1 Affinity aware Worker Threads Implementation

In our proposed system, the number of nodes and CPUs per node information is collected during initialization of run-time. This information can be obtained with the help of *libnuma* [12] API calls. Associated with each core(CPU), a new worker thread is created during initialization of the run-time. The group of worker threads that are pinned to the cores belonging to a single socket is called a stealing-domain [47]. The concept of stealing-domain is introduced to put best efforts of allowing the worker threads to steal tasks only from worker queues belonging to the same node. It is an improvement to plain work stealing technique which selects the victim worker randomly. Applying stealing domain concept doesn't completely avoid remote node task stealing but minimizes the remote node steal attempts to maximum extent. The concept of affinity awareness API described in the section 6.4 can be applied to the stealing-domains so that the worker threads belonging to same domain also operate on mutexes and condition variables belonging to same node. Then all worker threads of same stealing domain can access to synchronization constructs locally. The architecture of stealing domains along with lock cohorting is presented in the figure 6.2. Even the barriers can be restricted to the worker threads belonging to same stealing-domain.

Algorithm 5: WorkerRun3

```
   input : Pointer to current worker
 1 if (localTaskQueue.size == THRESHOLDMAXSIZE) then
 2     this.status = VICTIM;
 3 if (!isEmpty(localTaskQueue)) then
 4     run:
 5     popAtFront(localTaskQueue, task);
 6     execute task;
 7     if (localTaskQueue.size == THRESHOLDMINSIZE) then
 8         this.status = THIEF;
 9 else
10     this.status = THIEF;
11     if (global lock G is acquired by localnode) then
12         /*Stealing a task from local node */
13         localStealingDomain.acquireLock();
14         taskQueue= searchForVictimQueue ( thisStealingDomain );
15         popAtRear(taskQueue, task);
16         localStealingDomain.releaseLock();
17     if (task) then
18         pushAtRear(localTaskQueue , task );
19         goto run;
20     else
21         /* Stealing a task from remote node*/
22         if (global lock G is acquired by remote node_i) then
23             stealingDomain_i.acquireLock();
24             runQueue = searchForVictimQueue (
               remoteStealingDomain_i );
25             popAtRear ( taskQueue, task );
26             stealingDomain_i.releaseLock();
27         if ( task ) then
28             pushAtRear(localTaskQueue , task );
29             goto run;
```

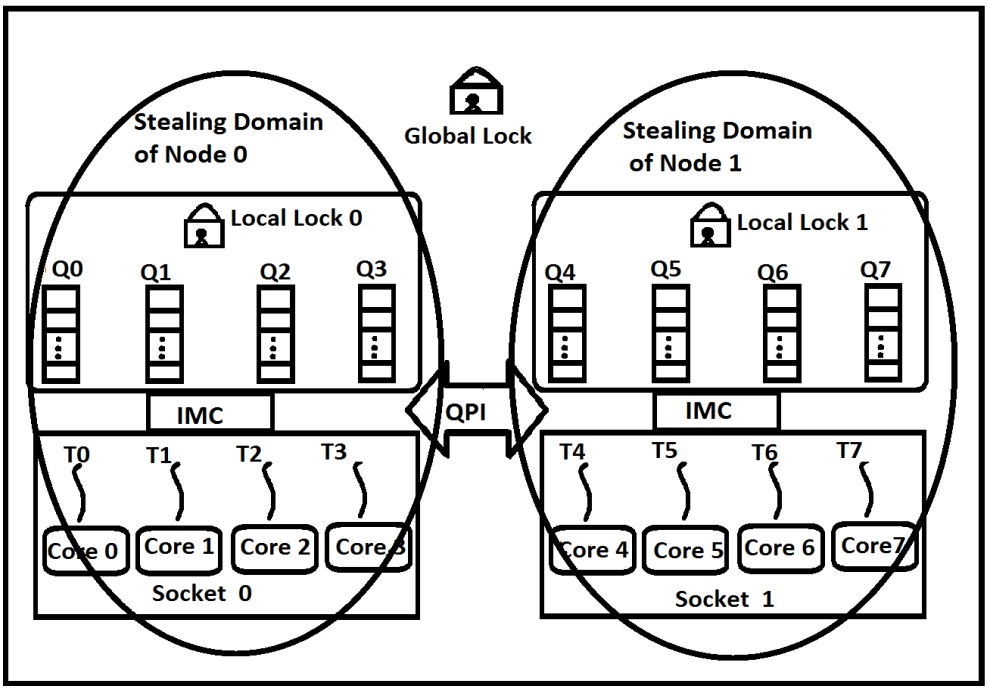

Figure 6.2: Stealing Domain and Lock cohorting Collaboration

The following two improvements are done to existing work stealing technique proposed in [47].

- The condition variable on which the worker thread waits is located on the same node where the worker thread is pinned. Hence, the $T_{remoteaccess}$ component of the equation 6.2 (from the Section 6.2) in each attempt is minimized.

- Whenever a worker thread finds no tasks in its queue, it becomes a thief and it attempts to steal a task from other queue belonging to the same domain. Before stealing, the thief worker has to acquire a lock of the victim's queue. Since the queues and locks of workers are all located on the same node the thief does not attempt any remote lock access.

- If all the queues belonging to same stealing domain are with less load, an attempt is made to steal tasks from other stealing-domain. Only in this case, the worker thread has to poll at a lock on remote node and

Table 6.3: Remote steal miss ratios after stealing domains

Number Of Worker threads	Remote Steal Miss ratio
12	0.000001
24	0.0000002

suffers from remote memory access. The probability of remote stealing is very minimal due to the implementation of stealing domains[47].

6.4 Results and Analysis

To evaluate the results of the proposed Affinity Aware NUMA lock library, open source OpenMP run-time OMPi [78] [61] was taken which supports work stealing load balancing among the worker threads. OMPi is a open source implementation of OpenMP runtime. At runtime level, it supports processes, pthreads, or any other native threads as worker threads. We configured this runtime only to support pthread library. As part of this run-time, we replaced the lock initialization primitives in `othr.c` file of `ee_pthreads` module. The function `othr_init_lock(othr_lock_t *lock, int type)` is modified using affinity aware lock primitives and OpenMP micro bench mark programs [74] [75] [76]are run to evaluate the proposal.

6.4.1 Micro Benchmarks

OpenMP micro benchmark [74] [75] [76] is a set of programs to assess the implementation overheads of synchronization at runtime level. It gives the overhead details of OpenMP directives such as parallel for, arrays loop and scheduling constructs offered by OpenMP specification. The set of programs include Array based programs of various sizes, scheduling benchmark whose purpose is to evaluate the scheduling overhead in OpenMP implementation. Since our intention is to evaluate the NUMA aware synchronization constructs such as locks and mutexes, we ran `syncbench` on our experimental setup. The experimental environment is a dual socket Xeon E5 2620 series

processor running Linux kernel 3.10 version.
After running the experiments for 10 times, average overhead of various synchronization constructs is collected from the output of the benchmark.

Table 6.4: Comparison of benchmark Synchronization Overheads

OpenMP construct	Average Overheads of Implementation in ms	
	NUMA Oblivious Synchronization	Affinity aware Synchronization
`parallel`	10.812	8.661
`parallel for`	6.893	6.09
`barrier`	2.219	2.092
`critical`	0.252	0.2438
`lock/unlock`	0.291	0.267
`atomic`	0.365	0.158
`reduction`	8.805	6.839

- It can be observed from the figure 6.3 that the overhead of OpenMP constructs such as `parallel` and `parallel for` is significant. This is due to the worker threads waiting on condition variable located on remote node for the chunks of work.

- `barrier` construct is also a kind of lock where all the worker threads wait for others to finish. Locality of barrier object contributes to the difference in performance.

- `lock/unlock` and `atomic` constructs of OpenMP are internally translated to native mutex locks and the difference in performance is due to the locality of lock variable across the nodes.

- `reduction` overhead is due to the critical section code placed at the end of `parallel for` in its expanded form uses again locks for protection of reduction variable. The `sync` benchmark contains many `parallel for` with `reduction` clause. Hence the difference of overheads is more.

To crosscheck whether the overhead observed is due to the remote memory access delays or for some other reason, likwid tools[58] are used to measure

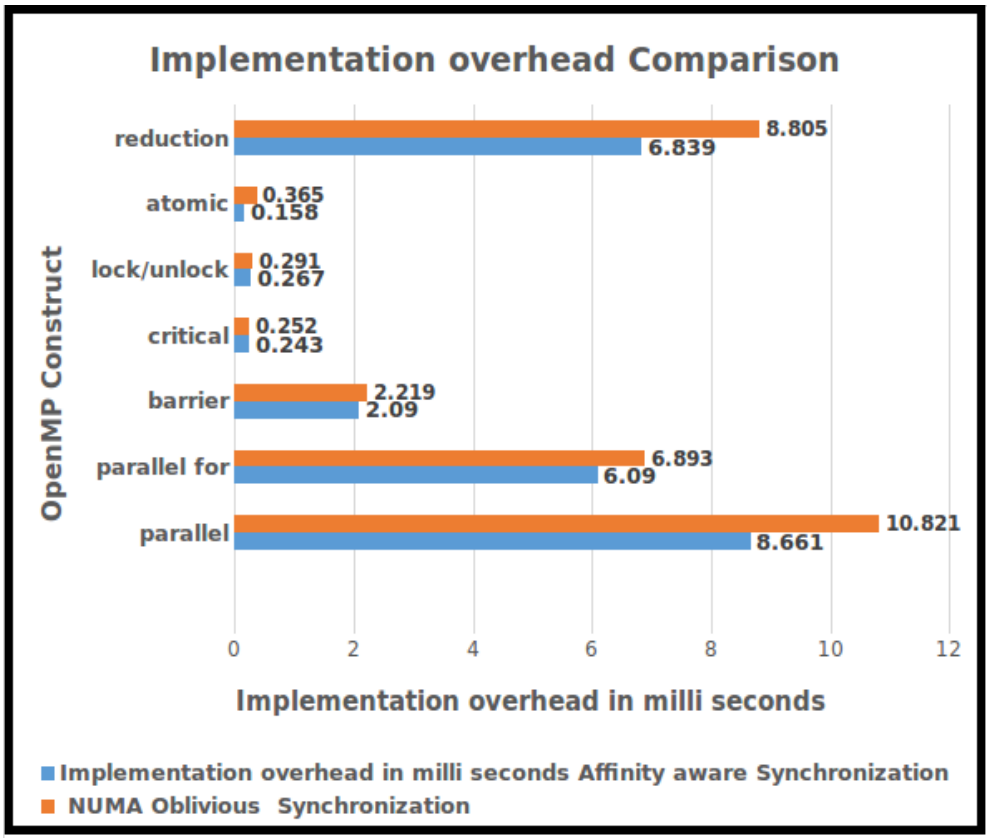

Figure 6.3: Implementation overhead comparison of synchronization constructs

performance counter values related to uncore NUMA events. The sum of remote data volumes are presented in table 6.5. It can be easily concluded that the excess remote data volume in the default approach is the cause of additional overhead observed in Table 6.4.

Table 6.5: Comparison of Remote Data volumes in GB

Synchronization method	
Affinity aware	NUMA Oblivious
0.0420 GB	0.0772 GB

6.4.2 BOTS Benchmarks

To assess the impact of affinity aware synchronization on overall performance on work stealing runtime, we continued to execute the same benchmark programs as done in previous chapters. The speed up comparison is done be-

Table 6.6: Execution Time Comparison NUMA Oblivious Vs Affinity aware synchronization

BenchmarkName-Size	Execution time in seconds		
	NUMA Oblivious Synchronization	Affinity aware Synchronization	Serial
Strassen-2048	0.182148	0.171838	1.8142
Strassen-4096	1.091279	0.983134	13.139
Strassen-8192	7.010121	6.259036	92.7439
Sort-33554432	0.496975	0.447725	6.696
SparseLU-single	0.719509	0.741776	11.1524
SparseLU-for	0.641538	0.675303	10.842

tween the best of our previous contributions and affinity aware lock implementation. The figure 6.4 depicts little improvement case of Strassen-2048, Strassen-4096, Strassen-8192 and Sort benchmarks. All these programs are exclusively meant for computational intensive and the number of tasks and other OpenMP directives used in these programs are more. Hence there is significant performance improvement (11%). Among the benchmark classes,

`single` and `for` implementations of Sparse LU program are bandwidth intensive and could not take the advantage of minute improvement contributed by affinity aware synchronization.

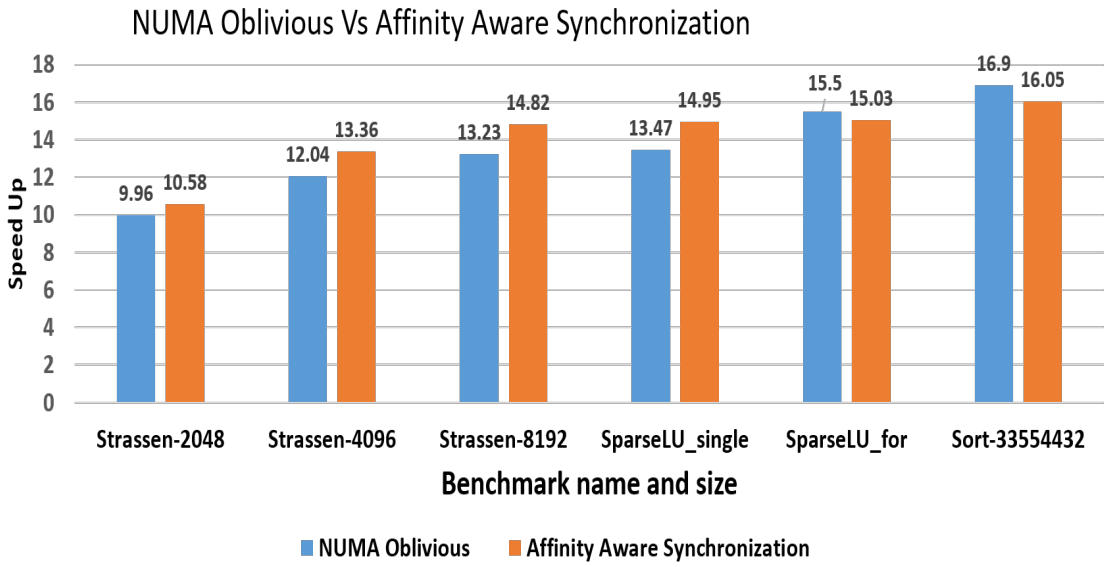

Figure 6.4: Speed-up comparison of NUMA Oblivious vs Affinity aware lock implementations

6.5 Conclusion

This chapter is an effort put to analyze the importance of locality of synchronization constructs in NUMA multi-core processors. The analysis is helpful in adapting existing work stealing based run-times to NUMA multi-core which use native thread library and synchronization constructs. If NUMA aware locks are used in these run-times, the performance of run-time and target application can be improved. These locality aware constructs can be implemented at run-time layer and do not effect the source code of the application.

Chapter 7

Conclusion and Future Work

The thesis contributes strategies for adapting the existing task stealing based run-time systems to NUMA multi-core architecture environment. While analyzing the randomized work stealing strategy, few metrics such as false steals and remote steals are introduced to analyze the effectiveness of victim selection. The proposed strategies were presented in mathematical and implementation form and were applied on standard task based benchmark programs for evaluation. The proposed strategies are intended to improve the overall performance of user level task based run-time systems by reducing the false steals and remote steal attempts, improving task-data proximity and synchronization affinity. The first three contributions of the thesis in chapters 3, 4 and 5 are proposed and implemented at the work stealing run-time layer and last contribution in the chapter 6 is proposed at native thread layer located below the work-stealing run-time layer.

7.1 Summary of Contributions

The first contribution, *Threshold constrained work-stealing* strategy proposed in the chapter 3 aimed to improve randomized victim selection in standard work-stealing run-times assuming uniform memory access in multi-core architecture. This strategy is based on queuing theory inventory model and applies min-max constraints on the task queues. This strategy could show

little performance gain on certain applications which are computational intensive but it was observed that the performance gain is insignificant in case of data intensive parallel applications. The results presented in the chapter 3 also are statistically insignificant though there is little performance gain.
Topology aware task stealing strategy proposed in the chapter 4 considers the topological features of modern NUMA multi-core architectures to further improve work-stealing run-times. This strategy is proposed to minimize the remote task stealing attempts which are specific to NUMA multi-core. Stealing domain concept introduced in this chapter is a best effort approach to minimize the remote task stealing actions which impact the performance of data intensive applications. The proposed strategy could show significant performance gain since it also addresses the remote task steal attempts in work stealing environment.
Shared object binding strategy proposed in the chapter 5 is an approach to improve task-data affinity in work-stealing environment. It is an extension to the contribution in chapter 4 to further improve the performance of data intensive applications in NUMA multi-core environment. This strategy is based on the compiler hints and guides the task dispatcher module of the run-time to map the tasks on to the same memory nodes where the dependent data objects are mapped to. This strategy relieves the programmer from explicitly mapping tasks based on object binding. The proposed strategy could show additional performance gain in case of data intensive applications with shared objects larger than page size.
In the chapter 6, we extend the concept of locality to native thread synchronization objects in NUMA multi-core environment. *Affinity aware synchronization constructs* are proposed at worker thread layer which lies below the work stealing layer. These synchronization constructs add an additional performance gain to all the previous contributions.
All the contributions of this thesis are tested on dual socket NUMA multi-core platform with Linux environment. The user level task based run-time considered for experimentation is OMPi source to source translator. Barcelona OpenMP Task Suit benchmark programs used for illustration in all contributed chapters are all task based programs composed with computational

and data intensive applications. The strategies proposed in this thesis can also be adapted in Cilk, TBB and other work stealing run-time environments.

7.2 Future Work

Our contribution in chapter 6 is an effort to address locality issues at user level thread layer. This work gave us new avenues in synchronization object locality and motivated us to further address locality issues at kernel level. The proposed strategy in chapter 6 was tested only for mutex objects. We wish to apply these strategies to all other synchronization constructs such as condition variables and semaphores and come out with NUMA aware native thread synchronization library.

If more than one virtual machine is running on a NUMA multi-core environment, the data locality of one VM is going to impact the performance of other VM. Scheduling strategies such as Completely Fair Scheduler(CFS) at hypervisor layer are not aware of the NUMA topology. We wish to contribute our future work on locality aware VM scheduling for NUMA multi-core architecture based on hardware topology.

The present high performance servers follow Integrated Many Core architecture. In these machines, hundreds of co processors are attached on single unit along with main processor. Xeon Phi(KNL) is an example for such processor, where the main processor is a dual socket Xeon (discussed in our thesis) and associated co-processors with floating point computational abilities. We wish to extend the task based work stealing infrastructure for these integrated multi-core architectures as part of our future work.

References

[1] Zoltan Majo and Thomas R Gross. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *ACM SIGPLAN Notices*, volume 46, pages 11–20. ACM, 2011.

[2] Intel®. Xeon E5 2620v3 processor architecture, 2014. `https://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2.40-GHz`

[3] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[4] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[5] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 1–6. IEEE, 2010.

[6] Pat Conway and Bill Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2), 2007.

[7] Iakovos Panourgias. Numa effects on multicore, multi socket systems. *The University of Edinburgh*, 2011.

[8] Zoltan Majo and Thomas R Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 12. ACM, 2011.

[9] Intel. *Intel Threading Building Blocks (IntelTBB) Developer Guide*. Intel, 2016.

[10] Trent Rolf. Cache organization and memory management of the intel nehalem computer architecture. *University of Utah Computer Engineering*, 2009.

[11] Martin J Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on NUMA systems. In *Proceedings of the Linux Symposium*, volume 1, pages 89–102, 2004.

[12] Andi Kleen. A NUMA api for linux. *Novel Inc*, 2005.

[13] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *ACM SIGPLAN Notices*, volume 47, pages 247–256. ACM, 2012.

[14] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing*, 1(2):13, 2015.

[15] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 124–131. IEEE, 2009.

[16] Tong Li, Dan Baumberger, David A Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 53. ACM, 2007.

[17] Jitendra Kumar Rai, Atul Negi, and Rajeev Wankar. Using machine learning techniques for performance prediction on multi-cores. In *Applications and Developments in Grid, Cloud, and High Performance Computing*, pages 259–273. IGI Global, 2013.

[18] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.

[19] Michael Stumm David Tam, Reza Azimi. Thread clustering:sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Workload Characterization, 2008. IISWC 2008. EuroSys'07*, pages 47–58. ACM, 2007.

[20] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

[21] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 121–132. ACM, 2009.

[22] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.

[23] John R Zedlewski and Carl A Waldspurger. Mechanism for scheduling execution of threads for fair resource allocation in a multi-threaded and/or multi-core processing system, April 27 2010. US Patent 7,707,578.

[24] Mohan Rajagopalan, Brian T Lewis, and Todd A Anderson. Thread scheduling for multi-core platforms. In *HotOS*, 2007.

[25] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38. IEEE Computer Society, 2007.

[26] Matthias Diener, Felipe Madruga, Eduardo Rodrigues, Marco Alves, Jorg Schneider, Philippe Navaux, and Hans-Ulrich Heiss. Evaluating thread placement based on memory access patterns for multi-core processors. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 491–496. IEEE, 2010.

[27] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review*, 34(2):42–55, 2000.

[28] Ruslan Belkin and Viswanath Ramachandran. Mechanism for implementing multiple thread pools in a computer system to optimize system performance, April 1 2003. US Patent 6,542,920.

[29] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[30] Burton J Smith et al. Architecture and applications of the hep multiprocessor computer system. In *SPIE*, volume 298, page 4, 1981.

[31] Yizhuo Wang, Weixing Ji, Xu Chen, and Sensen Hu. Task parallel implementation of matrix multiplication on multi-socket multi-core architectures. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 93–104. Springer, 2015.

[32] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[33] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[34] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289. ACM, 2002.

[35] Kunal Agrawal, Yuxiong He, and Charles E Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 112–120. ACM, 2007.

[36] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[37] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, volume 45, pages 341–342. ACM, 2010.

[38] Quan Chen, Minyi Guo, and Zhiyi Huang. Cats: cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 163–172. ACM, 2012.

[39] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[40] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[41] Zoltan Majo and Thomas R Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 12. ACM, 2011.

[42] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over NUMA architectures: An openmp runtime perspective. *IWOMP*, 9:79–92, 2009.

[43] Zoltan Majo and Thomas R Gross. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 230–241. ACM, 2012.

[44] Stephen L Olivier, Bronis R De Supinski, Martin Schulz, and Jan F Prins. Characterizing and mitigating work time inflation in task parallel programs. *Scientific Programming*, 21(3-4):123–136, 2013.

[45] Dan Quinlan, Chunhua Liao, Justin Too, Robb P Matzke, and Markus Schordan. Rose compiler infrastructure, 2012.

[46] Ananya Muddukrishna, Peter A Jonsson, Vladimir Vlassov, and Mats Brorsson. Locality-aware task scheduling and data distribution on NUMA systems. In *International Workshop on OpenMP*, pages 156–170. Springer, 2013.

[47] B Vikranth, Rajeev Wankar, and C Raghavendra Rao. Topology aware task stealing for on-chip NUMA multi-core processors. *Procedia Computer Science*, 18:379–388, 2013.

[48] Quan Chen, Minyi Guo, and Haibing Guan. Laws: locality-aware work-stealing for multi-socket multi-core architectures. In *Proceedings of*

the 28th ACM international conference on Supercomputing, pages 3–12. ACM, 2014.

[49] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. Optimizing data locality for fork/join programs using constrained work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 857–868. IEEE Press, 2014.

[50] Jinpil Lee, Keisuke Tsugane, Hitoshi Murai, and Mitsuhisa Sato. Openmp extension for explicit task allocation on NUMA architecture. In *International Workshop on OpenMP*, pages 89–101. Springer, 2016.

[51] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[52] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Notices*, volume 48, pages 219–228. ACM, 2013.

[53] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[54] S Stidham Jr and NU Prabhu. Optimal control of queueing systems. In *Mathematical methods in queueing theory*, pages 263–294. Springer, 1974.

[55] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 261–270. IEEE, 2009.

[56] V Viswanathan, Karthik Kumar, and T Willhalm. Intel memory latency checker.

[57] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance implications of cache affinity on multicore processors. In *European Conference on Parallel Processing*, pages 151–161. Springer, 2008.

[58] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010*

39th International Conference on Parallel Processing Workshops, pages 207–216. IEEE, 2010.

[59] Karl-Filip Faxén. Wool-a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2008.

[60] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[61] Panagiotis E Hadjidoukas and Vassilios V Dimakopoulos. Nested parallelism in the ompi openmp/c compiler. In *European Conference on Parallel Processing*, pages 662–671. Springer, 2007.

[62] ARB OpenMP. Openmp 4.0 specification, june 2013, 2013.

[63] Karl W Schulz, Rhys Ulerich, Nicholas Malaya, Paul T Bauman, Roy Stogner, and Chris Simmons. Early experiences porting scientific applications to the many integrated core (mic) platform. In *TACC-Intel Highly Parallel Computing Symposium, Austin, Texas*, volume 44, 2012.

[64] Spiros N Agathos, Panagiotis E Hadjidoukas, and Vassilios V Dimakopoulos. Design and implementation of openmp tasks in the ompi compiler. In *Informatics (PCI), 2011 15th Panhellenic Conference on*, pages 265–269. IEEE, 2011.

[65] Panagiotis E Hadjidoukas, G Ch Philos, and VV Dimakopoulos. Exploiting fine-grain thread parallelism on multicore architectures. *Scientific Programming*, 17(4):309–323, 2009.

[66] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over NUMA architectures: an openmp runtime perspective. In *International Workshop on OpenMP*, pages 79–92. Springer, 2009.

[67] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66. IEEE, 2008.

[68] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[69] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[70] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97. Springer, 2013.

[71] Henrik Löf and Sverker Holmgren. affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-NUMA system. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 387–392. ACM, 2005.

[72] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware.* " O'Reilly Media, Inc.", 2005.

[73] Robert Love. *Linux kernel development.* Pearson Education, 2010.

[74] J Mark Bull, Fiona Reid, and Nicola McDonnell. A microbenchmark suite for openmp tasks. In *International Workshop on OpenMP*, pages 271–274. Springer, 2012.

[75] J Mark Bull and Darragh O'Neill. A microbenchmark suite for openmp 2.0. *ACM SIGARCH Computer Architecture News*, 29(5):41–48, 2001.

[76] J Mark Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49. Citeseer, 1999.

[77] Samy Al Bahra. Nonblocking algorithms and scalable multicore programming. *Queue*, 11(5):40, 2013.

[78] Vassilios V Dimakopoulos, Elias Leontiadis, and George Tzoumas. A portable c compiler for openmp v. 2.0. In *Proc. EWOMP*, pages 5–11, 2003.

[79] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[80] Paweł Gepner, David L Fraser, and Michał F Kowalik. Second generation quad-core intel xeon processors bring 45 nm technology and a new level of performance to hpc applications. In *International Conference on Computational Science*, pages 417–426. Springer, 2008.

[81] James Laudon. Performance/watt: the new server focus. *ACM SIGARCH Computer Architecture News*, 33(4):5–13, 2005.

[82] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[83] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[84] Jean-Loup Baer. *Microprocessor architecture: from simple pipelines to chip multiprocessors*. Cambridge University Press, 2009.

[85] Wolfgang Mauerer. Linux kernel architecture. *Wrox*, 2008.

[86] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013.

Adaptive Work-Stealing Run-Times for NUMA Multi-core Architecture

by B Vikranth

Adaptive Work-Stealing Run-Times for NUMA Multi-core Architecture

ORIGINALITY REPORT

7%	3%	6%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	"OpenMP: Memory, Devices, and Tasks", Springer Nature, 2016 Publication	2%
2	Vikranth, B., Rajeev Wankar, and C. Raghavendra Rao. "Topology Aware Task Stealing for On-chip NUMA Multi-core Processors", Procedia Computer Science, 2013. Publication	2%
3	Quan Chen, Minyi Guo. "Task Scheduling for Multi-core and Parallel Architectures", Springer Nature, 2017 Publication	1%
4	queue.acm.org Internet Source	<1%
5	Matthias Diener, Eduardo HM Cruz, Philippe OA Navaux. "Modeling memory access behavior for data mapping", The International Journal of High Performance Computing Applications, 2016 Publication	<1%

6	link.springer.com Internet Source	<1%
7	Lecture Notes in Computer Science, 2015. Publication	<1%
8	mirror.ufs.ac.za Internet Source	<1%
9	iconline.ipleiria.pt Internet Source	<1%
10	www.comp.nus.edu.sg Internet Source	<1%
11	documents.mx Internet Source	<1%
12	www.cs.ucf.edu Internet Source	<1%
13	www.research-collection.ethz.ch Internet Source	<1%
14	reference.kfupm.edu.sa Internet Source	<1%
15	www.doxtop.com Internet Source	<1%
16	Submitted to Maastricht School of Management Student Paper	<1%

17	Lecture Notes in Computer Science, 2003. Publication	<1%
18	amsdottorato.unibo.it Internet Source	<1%
19	www.imit.kth.se Internet Source	<1%
20	www.usenix.org Internet Source	<1%
21	Zheng Li, Olivier Certner, Jose Duato, Olivier Temam. "Scalable hardware support for conditional parallelization", Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10, 2010 Publication	<1%
22	etheses.whiterose.ac.uk Internet Source	<1%
23	cs.uoi.gr Internet Source	<1%
24	Varisteas, Georgios, and Mats Brorsson. "Palirria: accurate on-line parallelism estimation for adaptive work-stealing : PALIRRIA: ACCURATE ON-LINE PARALLELISM ESTIMATION FOR ADAPTIVE WORK-STEALING", Concurrency and	<1%

Computation Practice and Experience, 2015.
Publication

25	ethesis.nitrkl.ac.in Internet Source	<1%
26	linknovate.com Internet Source	<1%
27	Yaqiong Peng, Song Wu, Hai Jin. "Towards Efficient Work-Stealing in Virtualized Environments", 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015 Publication	<1%
28	Submitted to Higher Education Commission Pakistan Student Paper	<1%
29	supertech.lcs.mit.edu Internet Source	<1%
30	Lecture Notes in Computer Science, 2010. Publication	<1%
31	ChunYi Su, Dong Li, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Bronis R. de Supinski, Edgar A. Leon. "Model-based, memory-centric performance and power optimization on NUMA multiprocessors", 2012 IEEE International Symposium on Workload Characterization (IISWC), 2012	<1%

Publication

32	Michihiro Horie, Hiroshi Horii, Kazunori Ogata, Tamiya Onodera. "Balanced double queues for GC work-stealing on weak memory models", Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management - ISMM 2018, 2018 Publication	<1%
33	"Euro-Par 2017: Parallel Processing Workshops", Springer Nature, 2018 Publication	<1%
34	classes.cs.uoregon.edu Internet Source	<1%
35	Rai, Jitendra Kumar, Atul Negi, Rajeev Wankar, and K.D. Nayak. "Performance Prediction on Multi-core Processors", 2010 International Conference on Computational Intelligence and Communication Networks, 2010. Publication	<1%
36	ivythesis.typepad.com Internet Source	<1%
37	repository.ntu.edu.sg Internet Source	<1%
38	researchspace.auckland.ac.nz Internet Source	<1%

39	www.dtic.mil Internet Source	<1%
40	Lecture Notes in Computer Science, 2011. Publication	<1%
41	ix.cs.uoregon.edu Internet Source	<1%
42	Yan Cui, Weida Zhang, Yu Chen, Yuanchun Shi. "A Scheduling Method for Avoiding Kernel Lock Thrashing on Multi-cores", 2010 IEEE 16th International Conference on Parallel and Distributed Systems, 2010 Publication	<1%
43	www.inderscience.com Internet Source	<1%
44	brage.bibsys.no Internet Source	<1%
45	cs.brown.edu Internet Source	<1%
46	www.soberit.hut.fi Internet Source	<1%
47	140.113.2.129 Internet Source	<1%
48	hal.inria.fr Internet Source	<1%

49	cadl.iisc.ernet.in Internet Source	<1%
50	eprints.maynoothuniversity.ie Internet Source	<1%
51	digitalassets.lib.berkeley.edu Internet Source	<1%
52	kopa.ir Internet Source	<1%

Exclude quotes	On	Exclude matches	< 10 words
Exclude bibliography	On		