Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 379 – 388

Procedia Computer Science

2013 International Conference on Computational Science

Topology Aware Task stealing for On-Chip NUMA Multi-Core Processors

B.Vikranth, Rajeev Wankar[1], C.Raghavendra Rao

b.vikranth@cvr.ac.in, CVR College Of Engineering, Hyderabad, India
wankarcs@uohyd.ernet.in, School Of Computers and Information Sciences, University Of Hyderabad, Hyderabad, India
crrcs@uohyd.ernet.in, School Of Computers and Information Sciences, Hyderabad, India

Abstract

"The On Chip NUMA Architectures (OCNA) introduce a new challenge namely memory-latency to the scheduling methods. The language run-times and libraries try to explore the processing power of these multiple cores by mapping the user-created tasks on to these cores by using suitable scheduling algorithms with load balancing support to improve throughput. The popular load balancing techniques used are work-sharing and work-stealing and many run-time systems such as Cilk, TBB and wool implement task stealing algorithm to schedule the tasks on to the cores by multiplexing the program generated tasks on to the native worker threads supported by the operating system. But the task stealing strategy applied in present run-time systems assumes the sharing the last level cache (LLC) and common shared bus among all cores on Chip Multi Processor. It tries to optimize the utilization without considering the presence of multiple On Die DRAM controllers and their topological arrangements. Current task stealing technique also suffers from problem of randomly choosing the victim worker queue. In this paper we address these issues and propose a solution for these problems by suggesting few optimizations. Our proposed task stealing strategy dynamically analyzes the topology of the underlying hardware connections and models the group of cores and connections as a logical topology tree. This logical tree is translated into multiple worker pools called stealing domains. By restricting the task stealing within these domains, this strategy is implemented and shows an average of 1.24 times better performance on NAS Parallel Benchmark programs compared to popular runtimes Cilk and OpenMP.

Keywords: multi-core, task stealing, work sharing, worker queue, load balancing, On Chip NUMA Multi Core.

1.Introduction

The modern trend in multi-core processors is to pack multiple groups of cores onto chip with point-to-point links using the latest communication technology such as QPI[11] links from Intel or Hyper Transport links from AMD. These groups of processors also integrate multiple on chip memory controllers to reduce the memory bandwidth problem and support scalability. So, modern many-core processors can be treated as On-Chip-NUMA Architectures (OCNA) where the memory address space is divided among multiple dies and

[1] * Corresponding author. Tel.: +91-9440-679-869; fax: +9140-23010780.
E-mail address: wankarcs@uohyd.ernet.in.

doi:10.1016/j.procs.2013.05.201

sockets. In this context, it is a common observation that the work load distribution and data distribution between the various processors is often not uniform. A particular processor may get heavily loaded while the others are idle. To avoid this kind of non uniform work load distribution, a load balancing algorithm is commonly employed. The sender initiated algorithms generally employ a centralized dispatcher which is responsible for dispatching various processes to the available processors. This phenomenon is called as work-sharing. All sender initiated strategies work with this phenomenon.

On the other hand each of the processors would be maintaining their individual local queues of tasks assigned to them by the central dispatcher. If this queue gets exhausted, i.e. if a processor completes execution of all of its assigned tasks then it is in an idle state becomes a thief. These architectures which are following receiver initiated algorithms would implement a technique where the thief-worker would steal tasks from the victim worker and executes those tasks while making a decision on which processor to choose and which task to choose. This is called as task stealing. The method of choosing the victim is randomly done in the present run time systems. Randomly choosing the victim worker may cause two types of delays namely:

Steal miss: choosing the random victim worker queue may result a success or a failure. A failure may be resulted if the randomly chosen queue is also under loaded or empty.

Remote steal: The second possibility is the randomly chosen run-queue that may belong to the other socket or other group of cores DRAM in which case, the stealing of task results task migration and data migration. In this paper, we analyze and address these issues and propose a new method to improve it.

Nomenclature

A	Worker: is a native thread which is part of worker pool. The number of workers generally is equal to the number of cores in the system.
B	Thief: A worker becomes a thief when it has no work to do. Therefore, its run-queue is empty. It is eligible to steal task from other run queues.
C	Victim: A worker becomes a victim when its task queue reaches a threshold value level.

1.1.Structure

The rest of the paper is organized in the following manner. In section 2 the related work is covered and in section 3 the problems identified in existing task stealing strategy are discussed. Section 4 contains the detailed description of the proposed algorithm and the result analysis is presented in section 5.

2. Motivation

Task stealing algorithm has been enhanced by various optimization strategies. [9, 12, 13] address the issues based on profiling data which require the tools to capture it. Theses profile based strategies do not consider the topology of core and memory interconnections. In contrast to this, we propose a topological aware, non-profile and dynamic task stealing strategy. This proposed strategy is based on capturing the topology of underlying hardware which does not cause any overhead since building topology tree is one time process and that too during the initialization of the library. This optimized task stealing strategy can be directly opted into user-level run-time libraries such as Cilk[4], OpenMP and TBB[14]. To the best of our knowledge, the work proposed in the paper provides better solution for On Chip NUMA Multi Core runtime environments.

3. Existing Task stealing Technique: Issues

The task-stealing strategy implemented in most runtime systems such as Cilk, OpenMP and TBB create *n* number of native worker threads onto which the programmer created tasks are mapped. Associated with each worker thread is a queue or a double ended queue with both ends open. In a OCNA architecture, the task-queues may be physically allocated on different DRAM chips. Hence accessing a remote task queue is not same as accessing local task-queue. The worker thread accesses the queue from front end and rear end is open for stealing by other worker thread. When the task queue associated with a worker thread becomes empty, it randomly chooses one victim queue and steals a task. The stolen task is added to the thief's run queue. This randomization has following drawbacks when applied to OCNA:

- The random victim worker queue is also under loaded in which case the thief worker may wait for some amount of time or attempt repeatedly until victim with large enough run queue is found. This may involve considerable amount of delays [9]. If the terms T_e and T_r denote the time taken to make the task queue empty and the time involved in each random steal attempt respectively, then the time involved in each steal attempt T_s (the time duration of task stealing from non trivial queue length state) is given as:

  $$T_s = T_e + \sum_{i=1}^{k} T_r \text{ .... (1)}$$

  In the above equation, T_e represents the time a queue spends between minimum threshold state and empty state. T_r may represent the back-off time or the random number computation time and modulo operation on random number. This worker thread will be idle for the time $\sum_{i=1}^{k} T_r$
- The random victim worker is bound to a core on a different die or socket where stealing a task is not just enough but even the data required by the task has to be migrated which is costly in terms of memory latency and page-fault. These remote task migrations introduce 30% additional delays [8].
- The task stolen from a remote node may result *performance isolation problem* [12, 19]. Since the most run-times maintain only related tasks in their local task queues and remotely chosen task may be an odd one in terms of shared cache. This may result eviction of cache lines which are used by a group of related tasks and cause false sharing.

4.Topology aware task stealing

Our work focuses upon implementation of a Topology Aware Task Library (TATL) at user-level which multiplexes user created tasks to the native threads of the operating system. As part of this library, co-operative stealing domain based worker-pools are maintained. Each stealing domain implements a separate worker pool with few modifications to existing method of task stealing. Stealing domains are responsible for minimizing the cross chip task steals. Though we implemented our own task stealing library, we adapted the flexible macro based API provided in wool [5]. This API allows us a flexible method of spawning the tasks with variable number of arguments.

Our proposed task stealing thereof would perform the following functionalities:

4.1 Topology of the architecture

During the initialization of our run-time library, we parse the proc file system of the kernel by looking at the directory structure of /proc/cpuinfo. There are some automated tools such as *lstopo* from hwlock[15] using which we can convert the /proc/cpuinfo into topology objects. But for simplicity reason, we have not used any library but wrote an explicit parser which converts the existing architecture into a tree structure. For illustration of this conversion, the following example from Intel architecture is taken [10, 11].

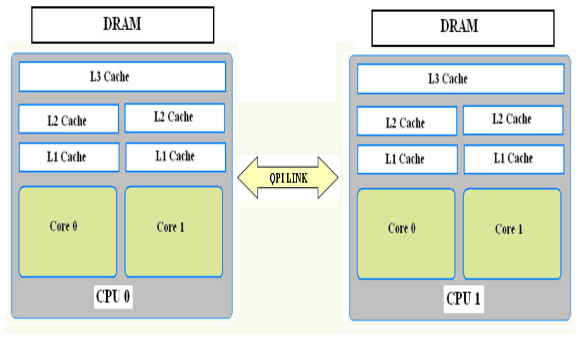

Fig.1. An example On Chip NUMA architecture [10,11]

The above topology is translated into the following tree form for identifying the core groups

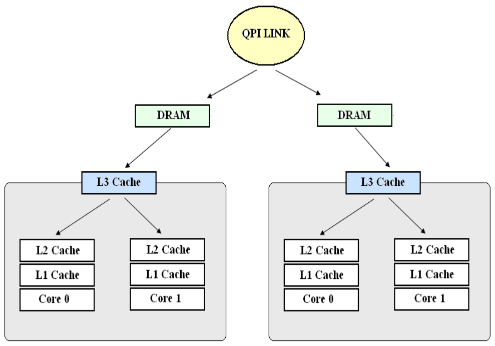

Fig.2. Logical tree of the topology in Fig.1

It can be observed from the logical tree in Fig. 2 that cores belonging to the same group share the last-level-cache (either L2 or L3). Many of the tasks that are similar in program code but differ only in terms of data can perform better if two tasks with same code and different data are bound to such group. We call such group of cores as a *stealing domain*.

During the initialization of the runtime, the topology tree shown in Fig.2 is partitioned into two sub trees. An array of worker pools is created per sub tree where each worker pool represents a stealing domain. As a principle, if there are *M* sockets or dies with separate memory controller per each, and there are *N* cores per socket, then *M* worker pools are created where each pool contains *N* worker threads there by grouping the total worker threads 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 *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.

4.2 **Avoiding the random victim selection**

The method of selecting the victim worker queue is not done by random method in our implementation as proposed in many popular run times. The individual worker is responsible for advertising itself whenever its queue length reaches the threshold value.

The term $\sum_{i=1}^{k} T_r$ of equation (1) of section 3 can be eliminated if the victim worker announces itself during the time interval T_e.

The values of minimum-threshold and maximum-threshold are computed using simple inventory model equations.

$$S = C - \lambda T_{push} + \mu T_{pop} \quad \ldots\ldots\ldots\ldots\ldots\ldots(2)$$

$$s = \mu T_{pop} \quad \ldots\ldots\ldots\ldots\ldots\ldots(3)$$

Where S and s denotes the maximum and minimum threshold values for queue size; C is the queue capacity; T_{push} and T_{pop} denote the times taken to perform push and pop on to double ended queue; λ and μ represent the arrival and processing rates of task queue respectively.

This self announcing strategy will not only reduce the time to find a suitable victim but also reduces the delay involved in computing the random numbers. While all the processors do maintain their local queues for the respective tasks allocated to them, there might be some processors which have completed all the tasks in their queues and are waiting to execute the tasks of other queues. In this situation, the run-queue of the worker thread enters into *THIEF* state. The thief worker first searches the list of workers whose state is already *VICTIM* in the same stealing domain. In our implementation, we added a status bit to each worker queue to represent either *THIEF* or *VICTIM*. The thief worker searches only the run-queues with status *VICTIM*. This solves the problem of randomly choosing the victim and failure to find a queue with enough number of tasks. When a thief worker tries to search for a victim worker, it can find the victim easily by looking at the bit. Hence the delays involved in repeated attempts are removed.

4.3 The Proposed Algorithm

As part of any task stealing library, each worker thread invokes a function on the creation of the worker during the worker pool initialization process and this function is responsible for the activities of task running and task stealing. We present the simplified version pseudo code of such function where, the locking and synchronization steps are hidden.

In the algorithm(given in pseudo code form), the function call `searchForVictimQueue()` searches for the run queues whose status is already set to VICTIM. The values of THRESHOLD_MAX_SIZE and THRESHOLD_MIN_SIZE are computer using the *S* and *s* variables from equations (2) and (3) from section 4.2.

```
Algorithm: workerRun
Inputs: thisStealingDomain
if ( localTaskQueue.size == THRESHOLD_MAX_SIZE  )
then
  this.status = VICTIM;
endif
if ( ! isEmpty(localTaskQueue)  )
then
  run:
     popAtFront(&localTaskQueue, &task);
     execute task;
     if (localTaskQueue.size == THRESHOLD_MIN_SIZE )
     then
        this.status = THIEF;
     endif
else
    this.status = THIEF;
end if
taskQueue= searchForVictimQueue ( thisStealingDomain );
popAtRear(&taskQueue, &task);
if ( task )
then
     pushAtRear(&localTaskQueue , task );
     goto run;
else
  runQueue= searchForVictimQueue ( remoteStealingDomain);
  popAtRear(&taskQueue, &task);
  if ( task )
  then
     pushAtRear(&localTaskQueue , task );
     goto run;
  endif
end if
```

5.Result analysis

To analyze the problem of random victim selection, we executed MatMul[17] benchmark using existing task-stealing runtime. Since the victim is randomly selected, the count of attempts to task queues associated with the worker which may not be full enough is computed by running the benchmark number of times. Table 1 gives the results obtained where steal miss ratio is the proportion of selecting a victim which is not potential. It is the ratio of the number of failure attempts to non-full- enough with respect to total number of steal attempts.

While calculating the steal miss ratios, it is assumed that all the worker-threads are not grouped into stealing domains. The experimental results revealed that this steal miss ratio as 94%. Hence it can be observed that huge percentage of the stealing attempts choose a wrong victim and cause repeated computation of random numbers and recursive steal attempts.

Table 1. Local Steal-misses due to random victim selection

Numberof Worker Threads	Steal Miss Ratios
4	0.934
8	0.949
16	0.954

The proposed task stealing library tries to almost eliminate the local and remote steal misses since the victim is of local stealing domain.

Another experiment has been carried out to understand and analyze the impact of core-topology by measuring the number of remote task stealing attempts in OCNA architecture. We executed MatMul[17] benchmark using existing task-stealing runtime. For calculating the number of remote misses we used 8 and 16 worker threads and these workers are grouped into two and four stealing domains respectively. It can be observed from the Table 2 that, on an average 36% of randomly chosen victims access a remote worker. Accessing a remote worker may involve migration of task causing additional 30% delays.

Table 2. Remote Steal-misses due to random victim selection

Numberof Worker Threads	Remote Steal Miss Ratios
8	0.379746835
16	0.355805243

When victim worker announces itself in the proposed task stealing technique, the local and remote steal miss ratios are very near to 0 and are presented in Table 3.

Table 3. Remote Steal-misses in TATL

Number of Worker Threads	Remote Steal Miss Ratios
8	0.000001
16	0.000004

Though the implemented runtime system is meant for finer grained tasks, we chose NAS Parallel Benchmark 3.3[20], which contains some applications with coarse grained tasks too. As part of NPB 3.3, we added our task spawning directives to CG, EP and IS benchmarks. NPB 3.3 pack contains a OpenMP version and serial version of the benchmarks. CG program is an implementation of conjugate gradient method that is used to compute an approximation to the smallest Eigen value of a large, sparse, symmetric positive definite matrix. EP is an “embarrassingly parallel” kernel which does pseudorandom number generation at the beginning and collection of results at the end. EP program does not require any inter-processor communication. There are some challenges in computing required intrinsic functions (which, according to specified rules, must be done using vendor-supplied library functions) at a rapid rate. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication. The parameter and values of these benchmarks are given in Table 4.

Table 4. NPB 3.3 benchmark parameters

Benchmark	Parameter	Class S	Class W	Class A	Class B	Class C	Class D
CG	no. of rows	1400	7000	14000	75000	150000	1500000
	no. of nonzero	7	8	11	13	15	21
	no. of iterations	15	15	15	75	75	100
	Eigen value shift	10	12	20	60	110	500
EP	no. of random-number pairs	224	225	228	230	232	236

The results shown in Table 5 present the execution times obtained using of our proposed library TATL compare to the most popular run time systems like Cilk and OpenMP implementation of NPB Benchmark applications. The performance improvement obtained using TATL is due to minimizing the latencies involved in selection of random victims (94% as stated in Table 1) and remote victim selection (36% as stated in Table 2). In case remote victim selection, additional overheads are involved, and contribute to remote memory access and task migration delays. The basic reason for delays in existing task stealing technique, used in Cilk and OpenMP could be due to the assumption of shared memory paradigm. With the proposed optimization in TATL the improvement resulted in 24% improvement in performance. The results presented here are obtained on dual socket Nehalem based Xeon 8-core machine with 16 threads. Though the performance improvement is primitive, it may scale well on many-core machines.

Table 5. Execution time comparison (seconds) of NPB 3.3 benchmarks

Benchmark Name	CG			EP		
Benchmark Class	W	A	B	W	A	B
OpenMP	0.71	2.49	123.32	2.07	17.27	66.92
Cilk	0.69	2.39	119.15	2.01	16.72	63.22
TATL	0.55	1.91	96.11	1.63	13.4	51
Serial Version	1.24	4.77	221.0	7.16	57.1	228.23

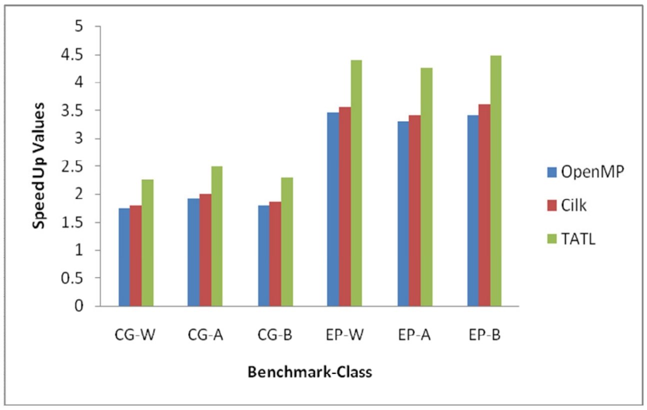

Fig.3. Speedup ratio comparison of TATL over OpenMP and Cilk

6.Conclusions

In this paper we analyzed the drawbacks of existing task stealing technique and presented an improved version of task stealing technique suitable for runtimes on Chip NUMA multi core architectures. We also presented the performance results of the proposed strategy on NPB 3.3 bench mark programs and compared results with run times such as Cilk and OpenMP which implement existing task stealing method. The results show that our proposed optimizations to existing task stealing strategy give consistent improvement in performance.

Acknowledgements

We are thankful to NASA Advanced Super Computing division for providing the NPB 3.3 benchmark applications for free of cost.

References

[1] Robert D. Blumofe,Charles E. Leiserson "Scheduling multithreaded computations by work stealing" Journal of the ACM (JACM)JACM Volume 46 Issue 5, Sept. 1999, p. 720 - 748

[2] Nimar S. Arora, Robert D. Blumofe,C. Greg Plaxton 1998 "Thread scheduling for multiprogrammed multiprocessors", SPAA'98 Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures, p. 119 - 129

[3] Bailey, D.H. Barszcz, E,Barton, J.T,Browning, D.S.,Carter, R.L. , Dagum, L. , Fatoohi, R.A. , Frederickson, P.O. , Lasinski, T.A. , Schreiber, R.S., Simon, H.D., Venkatakrishnan.V., Weeratunga, S.K. "Numerical Aerodynamic Simulation (NAS) Syst" Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, 1991. Supercomputing '91, p. 158 - 165

[4] Matteo Frigo,Charles E. Leiserson,Keith H. Randall "The implementation of the Cilk-5 multithreaded language" ACM SIGPLAN 1998 conference on Programming language design and implementation Volume 33 Issue 5, May 1998, p. 212 - 223

[5] Karl-Filip Faxén 2008 "Wool-A work stealing library" ACM SIGARCH Computer Architecture News archive Volume 36 Issue 5, December 2008, p. 93-100

[6] Molina da Cruz, E.H.Zanata Alves, M.A,Carissimi,Navaux, P.O.A,Ribeiro, C.P,Mehaut, J.-F.2011 "Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms" IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), p. 551 – 558

[7] S Blagodurov, A Fedorova 2011 "User-level scheduling on NUMA multicore systems under Linux" Proc. of Linux Symposium, 2011 - kernel.org

[8] Z Majo, TR Gross - ACM SIGPLAN Notices, 2011 "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead" ACM SIGPLAN Notices - ISMM '11 Volume 46 Issue 11, November 2011 p. 11-20

[9] S Lu, Q Li "Improving the Task Stealing in Intel Threading Building Blocks" International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on Date of Conference: 10-12 p. 343 – 346

[10] Ziakas, D. 2010 "Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures" IEEE 18th Annual Symposium on High Performance Interconnects (HOTI), 2010 p. 1 - 6

[11] “Intel Quick Path Interconnect” http://www.intel.com/technology/itj/2011/v15i1/pdfs/Intel-Technology-Journal-Volume-15-Issue-1-2011.pdf

[12] Sergey Zhuravlev,Sergey Blagodurov,Alexandra Fedorova "Addressing shared resource contention in multicore processors via scheduling" ACM SIGPLAN Notices - ASPLOS '10 Volume 45 Issue 3, March 2010 p. 129-142

[13] David Tam, Reza Azimi, Michael Stumm "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors" ACM SIGOPS Operating Systems Review - EuroSys'07 Conference Proceedings Vol 41 Issue 3, p.47-58

[14] Yi Guo,Jisheng Zhao,Cave, V.,Sarkar, V. "SLAW: A scalable locality-aware adaptive work-stealing scheduler" IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010 p. 1 – 12

[15] Alexey Kukanov,Michael J. Voss 2007 "The Foundations for Scalable Multi-Core Software in Intel® Threading Building Blocks" Intel Technology Journal. http://www.intel.com/technology/itj/2007/

[16] Broquedis, F., LaBRI, Univ. of Bordeaux, Talence, France "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications " 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010

[17] PARKBENCH Committe (assembled by R. Hockney and M. Berry). PARKBENCH report - 1: Public international benchmarks for parallel computers. Scientific Programming, 3(2):101{146, 1994. ttp://www.netlib.org/parkbench.
[18] (S,s) Inventory policies in general equilibrium http://www.richmondfed.org/publications/research/working_papers/1997/wp_97-7.cfm
[19] J.K. Rai, A.Negi, R. Wankar K.D. Nayak "On Prediction Accuracy of Machine Learning Algorithms for Characterizing Shared L2 Cache Behavior of Programs on Multicore Processors" International Conference on Computational Intelligence, Communication Systems and Networks" 2009 p. 213-219.
[20] "NAS Parallel Benchmarks", *www.nas.nasa.gov/publications/npb.html*