EFFECTIVE TASK BINDING IN WORK STEALING RUNTIMES FOR NUMA MULTI-CORE PROCESSORS

B. Vikranth
CVR College Of Engineering, Hyderabad,
501510, India
b.vikranth@gmail.com
http://it.cvr.ac.in/home/faculty

Rajeev Wankar and C. Raghavendra Rao
School Of Computer and Information Sciences, University Of Hyderabad,
Hyderabad, India
wankarcs@uohyd.ernet.in , crrcs@uohyd.ernet.in
http://scis.uohyd.ac.in

Abstract Modern server processors in high performance computing consist of multiple integrated memory controllers on-chip and behave as NUMA in nature. Many user level runtime systems like Open MP, Cilk and TBB provide task construct for programming multi core processors. Task body may define the code that can access task local data and also shared data. Most of the shared data is scattered across virtual memory pages. These virtual pages may be mapped to various memory banks of sockets due to first touch policy of Linux. The user level run-time environments must ensure that the tasks are mapped at a minimum possible distant memory bank where the shared data is located. At the same time, the runtime systems must ensure the load balancing among the multiple cores. Many of the user level runtime systems apply work stealing technique for balancing the load among the cores. Often, there is a tradeoff between these two requirements: object locality and load balancing. In this paper, we address the issue of shared object binding onto NUMA nodes, propose an adaptive solution for task-stealing runtime systems. The proposed solution is in the form of hints to the runtime system and the runtime system manages the object binding effectively so that shared object locality and load balancing are satisfied. These hints are compatible with newly introduced environment variables of OpenMP 4.0 specification and can be accommodated in future implementations. Experiment results show that this policy can improve the performance of standard bench marks.

Keywords: Locality; NUMA; User level runtimes; Task; Shared Object; Work stealing; Stealing Domain

1. Introduction

Modern task based run time systems relieve the programmer from the burden of worrying about the hardware details such as processor topology and memory hierarchy. They provide easy constructs for parallelizing various parts of the code to explore fine grained concurrency at loop level and coarse grained concurrency using tasks. For example, the task construct was added in Open MP 3.0 onwards which allows the programmer to write code in block that can be executed as an individual entity. The runtime system maintains a pool of native threads of operating system called worker threads and associated with each worker thread, there is a queue or double ended queue (Chase, 2005.). When the programmer invokes a task, the API adds the tasks to one of the queues. The order in which these tasks are executed by the worker threads depends on the scheduling policy of the runtime system. The runtime also follows a load balancing strategy such as work stealing to maintain load balance among the workers. Work stealing strategy is popular in many runtime systems such as Cilk, TBB and few implementations of Open MP. In these runtime systems, whenever a worker queue is empty, it attempts to steal one or more tasks from other queue (victim) and executes those stolen tasks to maintain load balancing among worker threads.

Modern many core processors with integrated memory controllers (IMC) (Ziakas, 2010)on chip supporting multiple memory banks introduce a new challenge for work stealing runtimes. Stealing tasks from a worker belonging to a core of remote socket may obviously increase memory latency thereby effecting the overall performance (Vikranth B, 2013) of the application. During the initialization of run-time, only master thread runs as part of all user level runtimes. According to first touch policy, when master thread allocates memory for objects on heap, they may be only stored on the memory node where thread that initializes the object is pinned to. As a result, an issue may arise where object bound to one node may be accessed by one or more tasks

belonging to other NUMA node. Hence minimizing memory latency and maintain load balancing are conflicting goals. Since memory latencies are high compared to processor speed, this situation has great impact on overall performance of workloads. As an illustration, when memory latency checker program was run on experimental setup, latencies from nodes to access local and remote memory are presented in Table1:

Table 1: Memory Latencies of dual socket Xeon E5- series prococessor

	Memory access latency in ns	
Node	0	1
0	82.4	145.8
1	145.7	80.7

This paper proposes a method to bind shared objects accessed by tasks across the memory banks to mitigate these memory latencies. The method proposed can be applied for work stealing based runtimes in particular. This paper is an extension of the work presented in (Vikranth B, 2013). The rest of the paper is organized as follows: Section 2 explains our previous contributions to task stealing runtime systems. Section 3 describes the mathematical model and implementation of our proposed strategy and Section 4 describes the experimental evaluation of our proposal along with comparison to GOMP implementation of benchmarks.

2. Background

OMPi [5]is a light weight compiler of Open MP 3.1specification 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 gives flexibility to the programmer to link any thread library with minimal implementation of headers and start up functions with prototype ort_xxx().

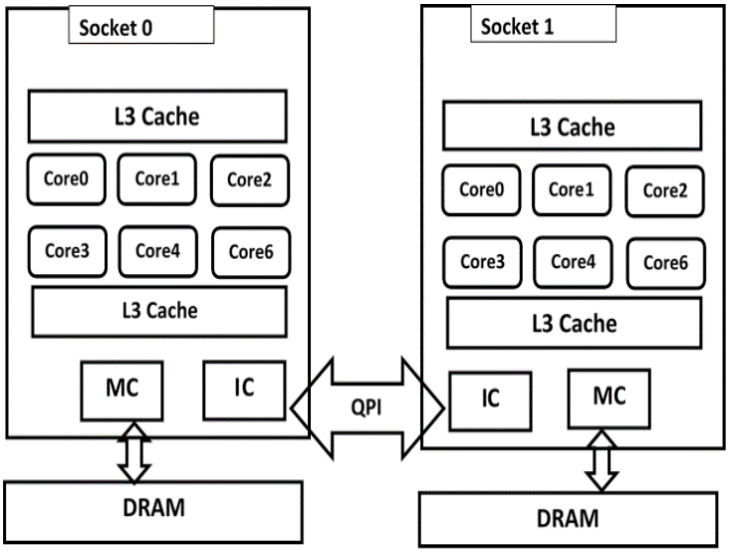

Fig. 1: Dual Socket Xeon E5 2620 architecture

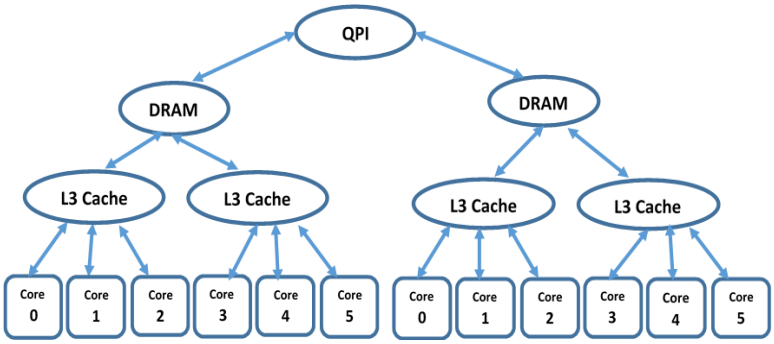

Fig. 2: Topology tree of Xeon E5 2620 processor

2.1. *Topology Aware Task Stealing Library (TATL)*

TATL (Vikranth B, 2013)is a small work stealing runtime with few extensions of adaptation to NUMA multi core environment. During the initialization of the runtime, NUMA topology of the hardware is translated into a tree structure by parsing the proc file system CPU information. The tree structure simplifies the understanding of the topology and locality with respect to shared caches and node distances.The runtime creates a pool of worker threads using pthreads based on the number of cores of all sockets and these workers form groups called stealing domains. Each child of the root of the tree represents a stealing domain. A stealing domain is a restriction for minimizing the cross node task stealing attempts. As an illustration the hardware of our experimental setup with dual socket Xeon E5-2420 and Xeon E5-2620 processors (Figure1) and its corresponding tree topology is presented in Figure 2. Newly added facility in Open MP 4.0 by setting the value for OMP_PLACES environment variable almost does the same grouping as that of stealing domains introduced. But stealing domains are particularly meant for work stealing run time systems which is not strictly mentioned in Open MP specification. If there are M sockets and N cores per each socket, then the number of sub trees in level 1 are M. Level 2 represents the next level shared locations such as L2 or L3 cache.

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($type_1$ arg_1, $type_2$ arg_2,……$type_n$ arg_n);

The shared objects such as global and heap objects are accessed by these tasks with default privilege without mentioning them in their prototypes. But few environments like Open MP provide few additional constructs such as shared, barrier, critical, atomic etc. Recent specification of Open MP 4.0 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. In most cases, the allocation of shared objects is done prior to creation of the tasks and all are treated as heap objects. In Linux, these shared objects are allocated according the default first-touch policy (Z Majo, 2011). The key idea in our proposed work is that these keywords can play important role in task binding and object binding too. 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. In this paper, we propose a method which maps the user created tasks on to the processor cores to improve object locality by taking the hints given by the programmers as directives.

3. Proposed Work

3.1. *Mathematical Model*

In this section, we try 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 randomly. 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. Similarly let there be T number of tasks present in a system snapshot. These tasks may try to access these shared data objects and let D_T denote such mapping.

Hence, at a given instance, there are D shared data objects involved in the system where D is given in Eq. (1)

$$D = D_N \cup D_T \tag{1}$$

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 defined as follows:

$$S_{ij} = similarity \quad (D_{T_i}, D_{N_j}) \tag{2}$$

$$S_{ij} = \frac{\left| D_{T_i} \cap D_{N_j} \right|}{\left| D \right|} \tag{3}$$

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, many of the tasks can access the shared objects at minimal access time there by reducing the memory latency.

3.2. *Implementation*

Though the mathematical model involves sets and join operations, implementing them as they are at runtime may be an overhead. Hence the concept of similarity between shared object mapping onto nodes and task using shared objects is simplified with the help of *libnuma* API (Kleen, 2005)and hash maps. In summary, the tasks accessing a particular shared object are mapped to a hash index of the memory node. This additional

functionality is taken care by an existing module within the architecture called task-locality-dispatcher (TLD). TLD takes responsibility of dispatching the newly created tasks on to the worker threads that are pinned to the processor cores belonging to a memory node where the required shared objects are bound to.

By the time TLD begins its activity, all the worker threads have been created and pinned to various cores on different nodes. If object initialization is done according to the locality of worker threads, the effect of first touch policy can be mitigated and tasks can find the objects at nearer places. The object binding mechanism is according to the default operating system NUMA features. Conventionally all task run-times provide task construct and allow the programmer implement the 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 Open MP, task body is a block with few additional directives to mention what objects they are. In the work proposed, these directives are used to take a decision of how to map the tasks to which memory nodes.

The task library we created allows the programmer to specify the arguments used by the task with the following syntax.

```
void createTask( type1 arg1, type2  arg2,……typen argn ){
        shared(obj1,thisTask() );  shared(obj2, thisTask());
        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 heap 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 when 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 directive shared gives a clue to the run time the necessary node or memory binding clues. In the implementation of the runtime system, the directives are just the macros which get expanded to runtime system function calls. These function calls help the 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 the following code sequence.

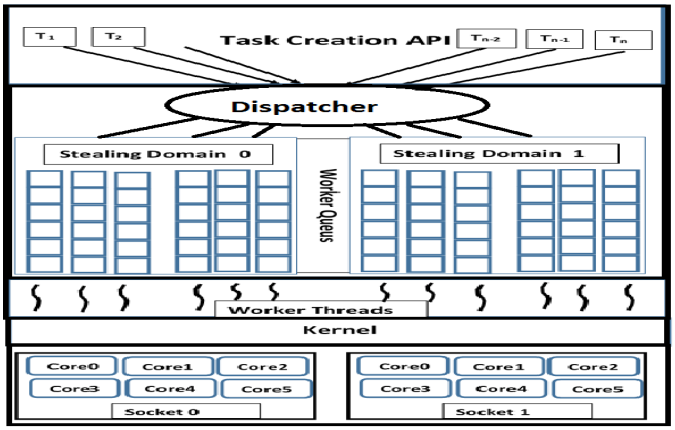

Fig. 3. Dispatcher introduced in existing work-stealing architecture

```
Algorithm: creatTask
Input: Task t
{
 if ( t does not use any shared object )
    add t to the queue of core which runs with load below s;
 else if(t uses a sharedobject o and o bound to corei )
 {
   sdx = stealing domain of corei
   if (    corei.load   <  S )
      add t to the queue of corei;
```

```
  else
  {
    find queue q within same SD with load < S ;
    addQueue ( q, t);
  }
 }
 else if( t uses a shared object but object is unbound )
 {
  select corex with load < S belonging to  SD;
  mbind ( shared object , core x );
  addQueue( t , x );
 }
 else if ( t uses multiple shared objects )
 {
  Use similarity index to decide binding core;
 }
}
```

In the above algorithm, the values s and S are computed lower and upper threshold values of the task queue loads (Vikranth B, 2013). 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.

4. Experimental Evaluation

To evaluate our proposed shared object locality model, we implemented a small task stealing library called Topology Aware Task stealing Library (TATL) (Vikranth B, 2013) which supports task stealing using pthreads pool. The worker threads of this pool are grouped as stealing domains. The functionality of TLD in this architecture is the main contribution of this paper. The architecture of TATL is presented in Fig.3.TATL is linked to a source to source compiler and runtime called OMPi (V.V. Dimakopoulos, 2003) which has support for work stealing at runtime level. To make the job of running benchmarks easier, we modified the TATL library calls such that they fit into the execution environment of OMPi. The additional facility added to the OMPi runtime by linking with TATL is the stealing domains and the TLD. TLD maintains the hash map with the number of buckets equal to the total number of shared objects. Since the shared objects are explicitly specified using the directives, they can be counted even before the compilation begins. 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 all scheduled on to the worker threads which are bound to the same stealing domain or same socket. Stealing domains ensure that the task stealing is limited within the local socket to avoid task migration from remote socket.

Open MP implementation of NAS Parallel Benchmark 3.3 version (NPB3.3) (Bailey, 1991.) is used to evaluate the performance of proposed technique. NPB suit consists of irregular parallel programs implemented using Open MP to evaluate features of modern runtimes and multi core processors. BT and LU programs of NPB suit exhibit huge amount of remote memory access (Z Majo, 2011). BT benchmark with 11-35%of shared object access (Z Majo, 2011) is considered in our work for evaluating the proposed method.

BT (Block Tri-diagonal) solves a synthetic system of nonlinear partial differential equation (PDE)s using three different algorithms involving block tri-diagonal, scalar penta-diagonal and symmetric successive over-relaxation (SSOR) solver kernels, respectively. Procedure followed in BT is solving these systems using a multi-partition scheme, e.g. 5X5 blocks. Multi-partition approach is opted because it provides good load balance among processors and uses coarse grained communication.

Table 2:NAS Parallel benchmark BT problem sizes

Benchmark class	Description	Problem size
Class S	Small for quick test purposes	12^3
Class W	Workstation size	24^3
Class A	Standard test problem-small	64^3
Class B	Standard test problem-medium	102^3
Class C	Standard test problem-high	162^3

The programs are executed on dual socket server with 12 core (6 cores per socket) Xeon E5-2620 with Linux Kernel version 3.16 environment. BT benchmark programs are executed with 12 worker threads (hyper threading disabled i.e. one worker per core) with work stealing enabling feature provided by OMPi. All workers are mapped to cores using identity affinity with the help of likwid tools (Treibig, 2010).

The speed ups of BT benchmark is presented for the purpose of analyzing speed up achieved using GNU Open MP and OMPi with TATL library is presented in Figure 4.

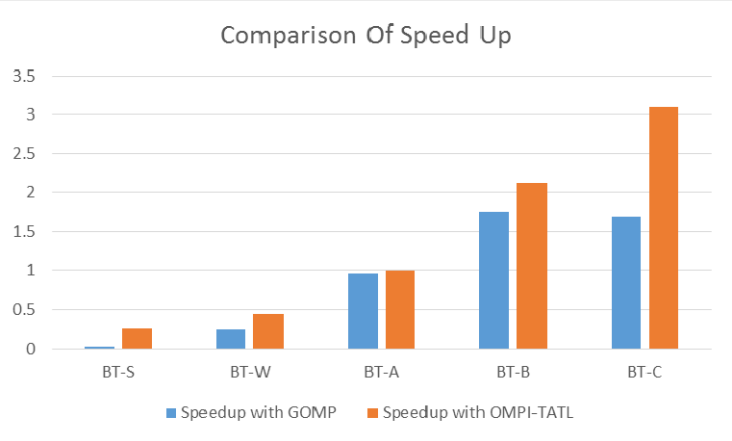

Fig. 4. Speedups achieved when various classes of BT benchmark are executed with GOMP and OMPi-TATL

It can be observed from the chart that for bigger sized benchmark classes of BT-B and BT-C, since the data is of bigger size and scattered across two nodes of the processor, GOMP could not give much speed up. The proposed method gave better performance in these cases because of its nearest object task binding strategy. The speed up improvement obtained by our proposed method for smaller size classes BT-S, BT-W is not much compared to GNU Open MP since the shared data is not much scattered across nodes. Because of effective mapping of tasks on to the worker threads with load balancing our proposed method could yield only little improvement.

NPB benchmark programs also obtain micro-architectural parameters how many number of operations are executed per thread in Million operations per sec per thread unit. This is a good measure how tasks are equally balanced among the worker threads. Comparison between GOMP and OMPi-TATL is given in the Table1.

Table 3:Comparison of Million operations per thread per second in GOMP and OMPi-TATL

Benchmark class	**BT-S**	**BT-W**	**BT-A**	**BT-B**	**BT-C**
GOMP	140.883	305.06	284.70	281.826	324.122
OMPI-TATL	326.834	748.190	929.278	560.247	1102.23

When paired *t* test is performed on throughput values of both approaches, the one-tailed *t* values obtained are presented on Table 2. The possible reasons for the difference is GOMP run-time spends most of its time in accessing memory objects scattered across NUMA nodes since tasks are not mapped keeping affinity of tasks there by many of the processor cycles are wasted in fetching the data from memory. In case of OMPi-TATL the throughput is high since tasks are mapped to suitable worker threads with proximity of shared objects. Hence most of the time is spent on computations on available data than wasting time in memory cycles. The proposed method showing better performance bigger problem sizes with huge data access above certain threshold level.

Table 4:Paired t test resluts of proposed method

Benchmark class	Paired t Test result(*t* value)
Class S	-15.254116
Class W	17.933782
Class A	25.931156
Class B	35.159314
Class C	86.400929

To investigate the reasons of the performance in the proposed strategy, using likwid performance counter tools, we also measured the average amount of remote memory data access to evaluate the effectiveness our proposed method. For smaller benchmark classes such as S, W and A the amount of remote data access is not significant. For bigger problem sized benchmark classes B and C, the amount of remote data access is high and the data is allocated on remote node memory because of Linux first touch policy. These remote memory accesses contribute significant performance degrade in plain Open MP run-time.

Table 5:Average remote data accesses sizes of BT benchmark classes

	Remote Data access size in GB	
Benchmark class	GOMP	OMPi-TATL
BT-S	0.0027	0.0038
BT-W	0.0235	0.0218
BT-A	9.4741	5.5784
BT-B	58.8179	36.1399
BT-C	253.6485	205.4133

5. RELATED WORK

Deciding which objects are shared among which tasks is a pure runtime issue. So many proposals are made to provide a solution to this problem but many of them are static in nature (Tam, 2007.) or profile based (Robert D. Blumofe, Sept, 1999) . Static solutions try to analyze the source code and detect the object access among parallel entities. Profile based solutions (da Cruz, 2011, May.) require the program to be run for first time to know the access patterns of the threads in the form of profiles. These profiles are used to decide which threads are mapped to which worker threads in the user level runtime. The proposed work in this paper is neither completely static nor pure dynamic. It is a hybrid mechanism based on the fact that parallel directives give enough information to the runtime on shared objects and task mapping is done dynamically based on locality of the shared objects. The programmer knows better during programming that what shared objects are required by that task. To the best of our knowledge this work is a unique approach based on the compiler hints and can be easily added in Open MP 4.0 compatible implementations.

6. References

[1] C. E. L. Robert D. Blumofe, "Scheduling multithreaded computations by work stealing", Journal of the ACM (JACM) JACM , vol. 46, no. 5, pp. 720-748, Sept, 1999.

[2] D. Ziakas, "Intel Quick Path Interconnect Architectural Features Supporting Scalable System Architectures", IEEE 18th Annual Symposium on High Performance Interconnects (HOTI), pp. 1-6, 2010.

[3] T. G.-. Z Majo, "Memory Management in NUMA multicore systems: Tapped between cache contention and interconnect overhead", ACM SIGPLAN Notices-ISMM'11, vol. 46, no. 11, pp. 11-20, 2011.

[4] Rajeev. Wankar. a. C. Raghavendra Rao. Vikranth B, "Topology aware task stealing for on-chip NUMA multi-core processors", Procedia Computer Science (ICCS'13), pp. 379-388, 2013.

[5] E. L. a. G. T. V.V. Dimakopoulos, "A portable C compiler for OpenMP V.2.0", in in Proc. EWOMP , 5th European Workshop on OpenMP, pp. 5–11, Aachen, Germany, Sept. 2003, 2003.

[6] E. A. M. C. A. N. P. R. C. a. M. J. da Cruz, " Using memory access traces to map threads and data on hierarchical multi-core platforms.", in Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium, pp. 551-558, 2011, May..

[7] S. a. F. A. Blagodurov, "User-level scheduling on NUMA multicore systems under Linux.", in Linux symposium (Vol. 2011)., 2011, June. .

[8] A. N. I. .. Kleen, "A numa api for linux.", 2005.

[9] D. H. e. a. Bailey, "The NAS parallel benchmarks summary and preliminary results.", in Supercomputing, 1991. Supercomputing'91.Proceedings of the 1991 ACM/IEEE Conference on. IEEE,, 1991..

[10] J. G. H. a. G. W. Treibig, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments.", in Parallel Processing Workshops (ICPPW), 2010 39th International Conference on. IEEE, ., 2010.

[11] D. R. A. a. M. S. Tam, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors.", in ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, , 2007..

[12] D. a. Y. L. ,. Chase, "Dynamic circular work-stealing deque.", in Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2005..