



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Critical Path-Based Thread Placement for NUMA Systems

C. Y. Su, D. Li, D. S. Nikolopoulos, M. Grove, K.
Cameron, B. R. de Supinski

November 1, 2011

2nd International Workshop on Performance Modeling,
Benchmarking and Simulation of High Performance Computing
Systems (PMBS11)
Seattle, WA, United States
November 13, 2011 through November 13, 2011

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Critical Path-Based Thread Placement for NUMA Systems

ChunYi Su
Virginia Tech
Blacksburg, VA, USA
sonicat@vt.edu

Dong Li
Oak Ridge National Lab
Oak Ridge, TN, USA
lid1@ornl.gov

Dimitrios S.
Nikolopoulos^{*}
FORTH-ICS
Heraklion, Crete, GREECE
dsn@ics.forth.gr

Matthew Grove
Virginia Tech
Blacksburg, VA, USA
mat@vt.edu

Kirk Cameron
Virginia Tech
Blacksburg, VA, USA
cameron@vt.edu

Bronis R. de Supinski
LLNL
Livermore, CA, USA
bronis@llnl.gov

ABSTRACT

Multicore multiprocessors use a Non Uniform Memory Architecture (NUMA) to improve their scalability. However, NUMA introduces performance penalties due to remote memory accesses. Without efficiently managing data layout and thread mapping to cores, scientific applications, even if they are optimized for NUMA, may suffer performance loss. In this paper, we present algorithms and a runtime system that optimize the execution of OpenMP applications on NUMA architectures. By collecting information from hardware counters, the runtime system directs thread placement and reduces performance penalties by minimizing the *critical path* of OpenMP parallel regions. The runtime system uses a scalable algorithm that derives placement decisions with negligible overhead. We evaluate our algorithms and runtime system with four NPB applications implemented in OpenMP. On average the algorithms achieve between 8.13% and 25.68% performance improvement compared to the default Linux thread placement scheme. The algorithms miss the optimal thread placement in only 8.9% of the cases.

Categories and Subject Descriptors

D.4 [Operating Systems]: Thread Management Scheduling

Keywords

Multicore processors, NUMA, Thread Placement, OpenMP, Critical Path, Shared Resource Contention

1. INTRODUCTION

Many shared-memory multicore multiprocessors use a Non Uniform Memory Architecture (NUMA) to dedicate different memory lanes to different processors and to distribute

^{*}Also with the Department of Computer Science, University of Crete, Heraklion, Crete, Greece.

system DRAM between processors. Compute nodes of high-end systems such as the Cray XMT [9] and XE series, as well as compute nodes based on leading multicore processors, such as the IBM Power 7 [13] and the Intel Single-chip Cloud Computer (SCC) [7], use a NUMA organization for off-chip DRAM. NUMA systems provide more memory bandwidth per core compared to UMA systems. Therefore, their scalability is superior to that of UMA systems.

Performance optimization for NUMA systems typically relies on data localization, so that each thread accesses local off-chip memory upon cache misses. Such localization may be achieved either with NUMA-aware data placement or with NUMA-aware thread placement. Typically, NUMA systems use first-touch or round-robin page placement on DRAM to achieve a reasonably balanced initial distribution of memory accesses between nodes. However, maximizing local data accesses can create contention on the node-level cache hierarchy and memory controllers by placing too many threads on the same memory node. Moving some data to a remote memory node may alleviate contention on shared resources and outperform a thread or data placement scheme that enforces strict localization. NUMA performance limitations due to either contention or remote memory accesses may limit application scalability [12].

NUMA may also break performance and power optimizations, such as Dynamic Concurrency Throttling (DCT) [3, 5, 11]. DCT dynamically adjusts the number of threads between parallel regions, according to a performance prediction that indicates the optimal concurrency configuration (number and layout of cores to be used) of each parallel region. Theoretically, appropriately selecting the number and placement of threads for each parallel region can achieve optimal performance if the implicit overhead of DCT is ignored. Unfortunately, applying DCT on a NUMA system is challenging, because adjusting the number and placement of threads between parallel regions can break any data localization and the balancing of memory accesses that may have been performed initially in the application.

Conventional OS schedulers do not adequately address the NUMA issue, because they emphasize other optimization criteria, such as fairness, throughput and responsiveness. OS schedulers often perform thread migrations that ignore

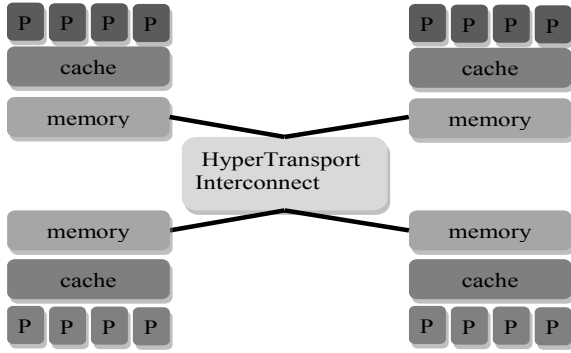


Figure 1: NUMA test platform.

the implications of data locality. Existing work studies how to co-locate data and threads on NUMA architectures [10, 12] and how to build NUMA-aware task schedulers. While OS-level NUMA-aware task schedulers are transparent, they ignore important application-specific information, such as the existence of application execution phases with different memory access intensity and memory access patterns, or the criticality of threads in an application, i.e., the threads that execute the critical path. By contrast, such information is available in the runtime system of languages used to parallelize applications. In this work, we leverage critical path analysis to optimize the thread placement of OpenMP applications on NUMA systems.

In this paper we propose several algorithms to solve the thread placement problem on NUMA systems and apply these algorithms to OpenMP applications. Our algorithms differentiate from prior work by using critical path analysis and the properties of threads executing this path to guide thread placement. When scheduling threads, our algorithms consider data locality and avoid local resource contention. The paper makes the following contributions:

- We propose a stable algorithm to address the critical path problem. This algorithm determines the best thread mapping in linear time with low overhead. This scalable algorithm is suitable for many-core systems.
- We develop a runtime system to predict optimal thread mappings for applications written in OpenMP.
- We implement and evaluate a runtime system for optimizing thread placement in OpenMP programs. The runtime system provides placement error-resilience for threads that are poorly mapped initially.

On a set of NPB OpenMP applications, our runtime system improves performance on average by 8.13% and up to 25.68% compared to the default Linux scheduler. The runtime system mispredicts the optimal thread placement for only 8.9% of the parallel execution phases.

2. BACKGROUND AND MOTIVATION

NUMA performance issues arise in two cases. First, modern multicore processors often have multiple memory controllers

(MCs) distributed across the same chip. Access by a core to the memory attached to the closest memory controller on the chip has longer latency than access to the memory attached to another memory controller. For example, the 48-core Intel Single-chip Cloud Computer processor has four DDR3 memory controllers [12]. The four MCs are placed at the four corners of the SCC 2D on-die mesh. This implies non-uniform memory access latencies for cores within the socket. Second, NUMA may impose remote accesses across sockets. Latency is lower for accesses to memory attached to the memory controller on the same socket as a core than accesses to memory attached to another socket. Our test platform, which Figure 1 illustrates, exhibits this difference. Our platform has four quad-core AMD Opteron 8350 processors (16 cores in total), with an MC for each socket with a memory nodes attached to each MC. Besides remote memory access latency, NUMA may reduce performance due to congestion on the interconnect and bandwidth saturation when accessing memory.

NUMA system performance is sensitive to the page placement policy of the operating system (OS). Typically, the OS uses a "first touch" policy that places physical pages on the node on which the thread that first touches the page executes. Other page allocation policies, such as round robin and interleaving are also used, as they produce balanced distribution of data between nodes. In this work, we assume the data is always allocated with the first touch policy, which is also the default setting in Linux.

The performance of an OpenMP parallel region is sensitive to how OpenMP threads are mapped to processor cores. Figure 2 shows the performance of the SP benchmark from the NAS parallel benchmark suite (OpenMP version) with 85 possible mappings and with the system default scheduling on our test platform. The first bar of each group shows the performance difference with the best and the worst mapping cases. The second bar of each group shows the performance difference with the default OS scheduling and with the best mapping. Performance with the default OS scheduling can be as much as 16.85% (region 9) worse than the best case. Therefore, relying on the system default thread mapping without information on data location is far from enough to achieve best performance. Motivated by this example, we explore new algorithms for NUMA-aware thread placement.

We base our algorithms on the following assumptions:

- We assume that applications are iterative. The outermost iterations are executed sequentially and typically correspond to simulation time steps. Within the outermost application loop, OpenMP directives parallelize code regions. This assumption is valid for many scientific applications [3, 5];
- We assume that application data is already touched and thus memory locations are fixed across iterations of the outermost application loop.
- We assume that static OpenMP loop scheduling.
- We assume that the OS NUMA-aware page placement policy is first touch, which is the default setting in Linux kernel.

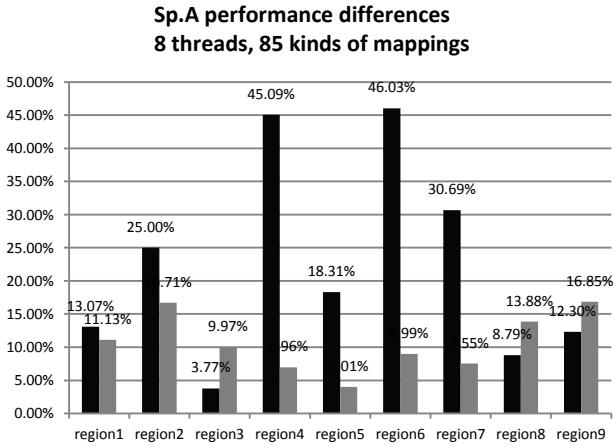


Figure 2: The performance difference of 85 thread mappings of OpenMP parallel regions in NAS SP.

We use performance counters to collect memory access information during the first few iterations of the application and to direct thread mapping to cores. In particular, we monitor the event CPU_TO_DRAM_REQUESTS_TO_TARGET_NODE_X, where X indicates the target memory node. This event counts all DRAM read and write requests generated by cores on the local node to the targeted node in the coherent fabric. This event can be used to observe processor data affinity [1]. By monitoring this event, we can set up a thread-node table (TNT) that records the number of memory references to each memory node from each thread. TNT provides data distribution information that we use to design our thread mapping algorithms. Table 1 gives an example of a TNT. The data is collected from the NPB MG benchmark using 8 OpenMP threads. We use this table as an example to describe our algorithm in the following sections.

3. DESIGN

In this section, we present three algorithms that optimize performance of OpenMP regions by appropriately assigning threads to cores. The algorithms use the TNT to keep snapshots of the distribution of memory references for each OpenMP thread. Each element of the TNT table is represented by $e(T_i, D_j)$. Table 2 shows our notation. The algorithms attempt to maximize the total local memory accesses (LMA) across all threads for an OpenMP region. With this policy, the algorithms attempt to reduce remote memory references, thus optimizing data access locality.

3.1 Algorithm 1

Algorithm 1, based on the memory reference information collected in the TNT, enumerates all possible thread mappings and calculates the total number of LMA for each mapping. The algorithm then selects the mapping with the highest LMA. This algorithm does not scale. Since it must enumerate all possible mappings, the runtime overhead increases quickly. For example, an OpenMP region executed on 4 quad-core processors must enumerate $(16!) / (4!4!4!) = 63,063,000$ mappings. As the number of cores increases, the overhead can easily offset any performance benefit.

Thread #	Memory Node 1	Memory Node 2	Memory Node 3	Memory Node 4
1	1770108	1765296	1766348	1765584
2	1631249	1530389	1529758	1532284
3	1554151	1554991	1552323	1552409
4	331659	330097	330903	329727
5	984706	985755	987233	987138
6	985833	986215	985754	988217
7	988661	986670	989070	988749
8	984706	985755	987233	987138

Table 1: A TNT with 8 threads

Symbol	Definition
N	Number of threads
Nd	Number of memory nodes, or sockets
NUMA factor	The ratio of remote access latency to local access latency
$e(T_i, D_j)$	An element at i_{th} row and j_{th} column on TNT table
$e.T_i$	Thread ID of element $e(T_i, D_j)$. The i_{th} row of TNT
$e.D_j$	Memory node ID of element $e(T_i, D_j)$. The j_{th} column of TNT
V_{ij}	NMemory requests of element $e(T_i, D_j)$.
LMA	Local memory accesses
RMA	Remote memory accesses
$IF(e(T_i, D_j))$	The performance Impact Factor of the thread T_i on memory node D_j toward critical path

Table 2: Notations used in the paper.

3.2 Algorithm 2

Algorithm 2 also tries to find the optimal thread mapping in terms of LMA. However, it uses a sorting algorithm that significantly reduces runtime overhead. We discuss the time complexity of the algorithms in Section 3.4. In this section, we describe Algorithm 2 in detail. To find the maximal LMA, Algorithm 2 first sorts all elements V_{ij} in the TNT in descending order and generates a linked list. The algorithm then iteratively selects the element with the “max” value from the list until all threads are selected. Each selection iteration chooses an element $e(T_i, D_j)$ to pin thread i to a processor attached to memory node j . Algorithm 2 has two additional properties. First, the assigned number of threads per processor should not be higher than the available number of cores per processor. Otherwise, the processor will be oversubscribed. Second, the algorithm considers contention on the memory node when placing multiple threads on processors attached to the same node. To avoid contention, the algorithm does not always select the element with the maximum value at a specific iteration. Instead, it may choose an element with a lower value that alleviates contention in other memory nodes. The element with the maximum value in the specific iteration is deferred to a later iteration.

We use the example in Table 1 to illustrate how Algorithm 2 avoids contention. In the example, the algorithm finds the element $e(1,1)$, has the maximum value (1770108) after sorting, so it places thread 1 on processor 1, which is attached to

the memory node 1. Then the elements $e(1,2)$, $e(1,3)$, $e(1,4)$ are removed from the list, because the position of thread 1 has been decided. The algorithm then adds the memory reference count of $e(1,1)$, 1770108, to the total number of local memory references of node 1: $LC[1] += 1770108$. Next, the algorithm finds that element $e(2,1)$ has the maximum value, so it attempts to place thread 2 on the processor attached to memory node 1 to maximize the local memory references of thread 2. However, memory node 1 is already assigned thread 1. Placing thread 2 close to memory node 1 will introduce contention and load imbalance. In this situation, the algorithm chooses pinning thread 2 close to another memory node by considering the elements $e(2,2)$, $e(2,3)$ and $e(2,4)$. In this situation, the algorithm sacrifices some locality to reduce contention.

Selecting cases in which reduced locality is beneficial merits further discussion. We use an example to explain this point further. Assume the element $e(T_i, D_j)$ has the maximum value of local memory references in an iteration, but the algorithm attempts to place thread i to remote memory node k instead of the local node j to avoid contention. From the TNT table, the algorithm finds that the number of memory references to memory node k for thread i is T by checking element $e(T_i, D_k)$. Pinning thread i to memory node k instead of memory node j is beneficial only if the remote memory access time to memory node k is no less than the local memory access time to memory node j :

$$T \cdot \text{RMA latency} \geq \text{MAX VALUE} \cdot \text{LMA latency} \quad (1)$$

Equation 1 implies that:

$$T \geq \frac{\text{MAX VALUE}}{(\text{RMA latency})/(\text{LMA latency})} = \frac{\text{MAX VALUE}}{\text{NUMA Factor}} \quad (2)$$

In Equation 2, the *NUMA Factor* is the ratio of the remote memory access latency to the local memory access latency. It usually varies between 1.5 and 2, depending on the interconnect design [8]. In our test platform we set *NUMA Factor*=1.5. Threshold T is used to decide whether sacrificing locality is beneficial. The memory references to the remote memory node k should be above T for the algorithm to decide to sacrifice locality. In other words, the algorithm reduces local memory accesses by no more than:

$$\begin{aligned} & \text{MAX VALUE} - \frac{\text{MAX VALUE}}{\text{NUMAFactor}} \\ & = \text{MAX VALUE} \cdot \left(1 - \frac{1}{\text{NUMAFactor}}\right) \end{aligned} \quad (3)$$

In our example, since $e(2,1)$ has the largest value in the second iteration, T is calculated as $1631249/1.5=1087500$ for the element $e(2,1)$. The elements $e(2,2)$, $e(2,3)$, $e(2,4)$ are above T . Further, the algorithm finds that $e(3,2)$, $e(3,3)$, and $e(3,4)$ are more eligible than $e(2,2)$, $e(2,3)$, $e(2,4)$ because they have more local memory references. The algorithm eventually chooses $e(3,2)$ because it has most local memory references. The elements $e(3,1)$, $e(3,3)$, and $e(3,4)$ are removed. The algorithm maps thread 3 to the processor attached to memory node 2, instead of mapping thread 2 to the processor attached to memory node 1. Eventually, the algorithm adds the value of $e(3,2)$, to the total local memory references of node 2: $LC[2] += 1554991$, and finishes the

Algorithm 2 Maximize total LMA in all threads.

Input: A TNT Table T

Output: Map $M_{MaxLocal}$ //A mapping with maximum local memory references

```

1: LC[Nd]= 0; //Local memory reference counts
2: ElementList SortedList = SortTable(T);
3: while sizeof ( $M_{MaxLocal}$ )  $\neq$   $N$  do
4:   element  $e_{max}(T_i, D_j)$ =getNextMax(SortedList);
5:    $M_{MaxLocal}$ .Insert( $e_{max}(T_i, D_j)$ );
6:   removeElementsWithThread(SortedList,  $e.T_i$ );
7:   Return  $M_{MaxLocal}$ ;
8: end while

```

9: **getNextMax**(List l)

Input: a sorted element List l ;

Output: an element e_{decide} with smallest local contention;

```

10: element  $e_{max}$  = getFistMaxElement( $l$ )
11: List  $lc$ = findAllPossibleCandidateElements( $e_{max}$ );
12:  $e_{decide}$ = findLowestLocalContentionElement( $lc$ );
13: RemoveThreadFromList( $e_{decide}.T_i$ );
14: AppendLocalContention( $e_{decide}.D_j, e_{decide}.V_i j$ );
15: Return  $e_{decide}$ ;

```

second iteration. The algorithm keeps $e(2,1)$ in the sorted list and waits for next selection iteration. The pseudo-code of Algorithm 2 is listed above.

In essence, Algorithm 2 attempts to improve the performance of each thread by maximizing LMA. However, it is not aware of the critical path and thus cannot ensure the critical thread has higher priority for performance optimization. We solve this problem in Algorithm 3.

3.3 Algorithm 3

Ideally, computation and data are evenly assigned to each thread and thus all threads have the same execution time. In practice, however, the execution time of threads may vary. The computation time of a thread may be longer than that of the other threads. This thread is then on the critical path. Changes in the memory access latency of threads may also change the critical path.

The critical path problem, while easy to understand, can be hard to analyze, because the memory reference time is influenced by many factors, such as last level cache (LLC) misses, resource contention on memory controllers and memory links from other memory operations, prefetching, cache coherence protocol and even page faults. Previous work [2, 14] uses LLC misses as a simple metric to compare the performance of threads. However, LLC misses is insufficient to estimate memory performance. Instead, we use the event CPU_TO_DRAM_REQUESTS_TO_TARGET_NODE_X to estimate the memory performance of threads. This event not only measures accesses to the LLC but also all DRAM access requests, including resent requests due to resource contention on shared resources and data prefetch requests.

Algorithm 3 is a NUMA-aware and critical path aware algorithm. Estimating the real execution time of the thread in the critical path is generally difficult. The algorithm avoids directly estimating the real time of the critical path; as an

Algorithm 3 Find a map with minimal critical path.

Input: A TNT Table T

Output: Map M_{MinCp}

```

1: Map  $M_{MinCp} = \Phi$ 
2:  $CPImpact[Nd] = 0$ ; //Impacting Extent on each domain
3: ElementList SortedList = SortElementInTable( $T$ );
4: while  $sizeof(M_{MinCp}) \neq N$  do
5:    $e(T_i, D_j) = \text{getMinCriticalPathElement}(\text{SortedList})$ ;
6:    $M_{MinCp}.Add(e(T_i, D_j))$ ;
7: end while
8: Return  $M_{MinCp}$ 

```

9: $\text{getMinCriticalPathElement}(\text{Listl})$

Input: a sorted element List l;

Output: an element e with smallest impacting to critical path;

```

10: element  $e_{max} = \text{getFistMaxElement}(l)$ ;
11: List lc = findAllPossibleCandidateElements( $e_{max}$ );
12:  $e_{decide} = \text{findLowestCPElement}(CPImpact, lc)$ ;
13: RemoveThreadFromList( $e_{decide}, T_i$ );
14: AppendCriticalPathImpact( $CPImpact, IF(e_{decide})$ );
15: Return  $e_{decide}$ 

```

16: $\text{findLowestCPElement}(CPImpact[], \text{Listl})$

Input: a sorted element List l;

Output: an element e with smallest impact on critical path;

```

17:  $minVal = \text{UINT64\_MAX}$ ; element  $e_{decide} = \Phi$ 
18: for any element  $e$  in l do
19:   if  $(IF(e) + CPImpact[e.Dj]) < minVal$  then
20:      $minVal = IF(e) + CPImpact[e.Dj]$ ;  $e_{decide} = e$ ;
21:   end if
22: end for

```

alternative, the algorithm uses *Impact Factor* (IF) to represent memory reference effects on performance:

$$IF(T_i, D_j) = \text{number of local requests} + \text{NUMA Factor} \cdot \Sigma(\text{number of remote requests}) \quad (4)$$

IF represents the effects of memory references on the memory system, including both local and remote memory references. A thread with a large IF value has high tendency to be on the critical path. Table 3 provides an example using the data from Table 1. Algorithm 3 sorts the TNT then picks the element with the maximum value and the rest of the candidates in other nodes, similarly to Algorithm 2. Algorithm 3 uses an array, $CPImpact$ of size Nd to record the IF on each domain.

Table 3 illustrates the main idea of Algorithm 3. Assume the algorithm has already selected $e(1,1)$ and pinned thread 1 to memory node 1. It then assigns $CPImpact[1] += IF(1,1)$ to record the IF from placing thread 1 on memory node 1. In the next iteration, the algorithm chooses the next element with maximum local references, $e(2,1)$ and three other candidates $e(3,2)$, $e(3,3)$ and $e(3,4)$ on the other three memory nodes. Then the algorithm selects one with the lowest value by computing $IF(e.T_i, e.D_j) + CPImpact[e.Dj]$.

According to Table 3, the algorithm will select $e(3,2)$, and add $IF(3,2)$ to $CPImpact[2]$, (i.e., $CPImpact[2] += IF(3,2)$).

element	$IF(e.T_i, e.D_j)$	$CPImpact[e.Dj]$	$IF(e.T_i, e.D_j) + CPImpact[e.Dj]$
$e(2,1)$	4592431	9715950	14308381
$e(3,2)$	4659723	0	4658883
$e(3,3)$	4661551	0	4661551
$e(3,4)$	4661465	0	4661465

Table 3: $IF(e.T_i, e.D_j) + CPImpact[e.Dj]$ comparison among elements

Algorithm 3 improves upon Algorithm 2 by considering additional remote memory contention while estimating the IF , while Algorithm 2 only uses local memory contention. The pseudo code of Algorithm 3 is given to the left.

3.4 Time Complexity Analysis

Algorithm 1 uses a brute-force method to find all possible thread mappings. Therefore it is cumbersome, slow and impractical. The time complexity of Algorithm 1 is $O(N!)$. Since there are many redundant thread combinations, we can select some “good” mappings by avoiding the check of symmetric cases. “Good” here means load balanced and symmetric [4]. The time complexity of Algorithm 2 and 3 is determined by the sorting algorithm and the iterative selection process. The process of iterative selection can be done in linear time: $O(k \cdot N \cdot Nd)$, where k is a constant, $N \cdot Nd$ is the total number of elements in the TNT.

Our implementation uses parallel radix sort as our sorting method. Theoretically, we can achieve a constant time complexity for parallel radix sort. The time complexity of the parallel radix sort is $O(\frac{1}{p} \cdot k \cdot N \cdot Nd)$, where k is a constant, p is the level of parallelism, $N \cdot Nd$ is the total number of elements in the TNT. When we use $p = N$, the time complexity becomes a constant, $O(k \cdot Nd)$. Since Algorithms 2 and 3 use parallel radix sort, their time complexity is dominated by the linear iterative selection process. This scales only with N . Theoretically, if $N = 16$, Algorithm 2 and 3 can be 12300 times faster than Algorithm 1. Algorithm 2 and 3 are thus more scalable and suitable for many-core systems.

4. PERFORMANCE

4.1 Experimental Environment

We used a system with four quad-core AMD Opteron 8350 HE processors (16 cores in total), each with private L1 and L2 cache per core and a shared 2MB L3 cache. Each processor has one memory controller. The machine has 64GB of RAM. The inter-processor communication is enabled by a HyperTransport interconnect. We tested OpenMP implementations of benchmarks from the NAS Parallel Benchmarks Suite, version 3.1 using Intel C/C++ compilers and Fortran compilers with “-O” optimization flag. The OS was Linux version 2.6.32.

4.2 Results

Due to limited space we only discuss the performance results of Algorithms 2 and 3. First, we demonstrate the ability of Algorithm 2 to adapt to good thread mappings regardless of the initial thread mapping. Figure 3 shows a histogram generated from 100 runs of the SP and BT benchmarks. The experiment is conducted as follows: First, we randomly

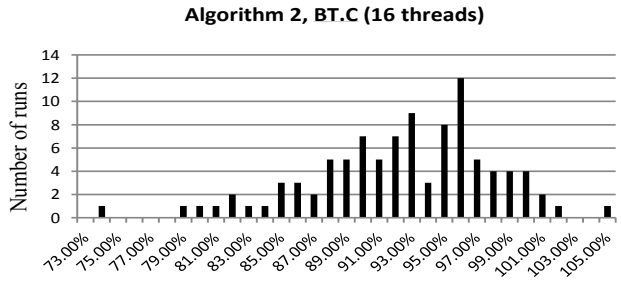
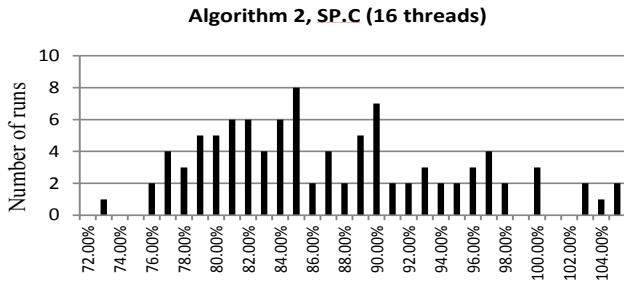


Figure 3: Performance comparison between Algorithm 2 and random mapping. X-Axis: Execution time under Algorithm 2 divided by execution time under random mapping

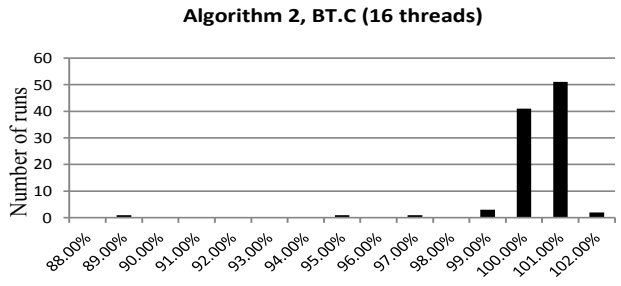
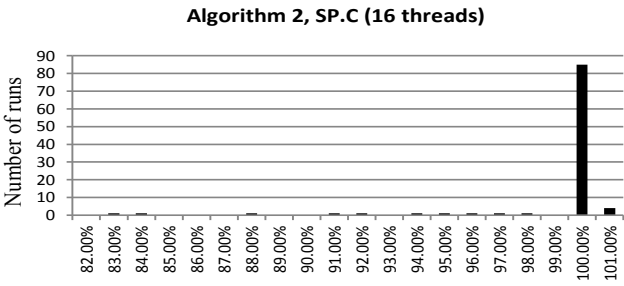


Figure 4: Performance comparison between Algorithm 2 and the system default mapping. X-Axis: Execution time under Algorithm 2 divided by execution time under the system default mapping

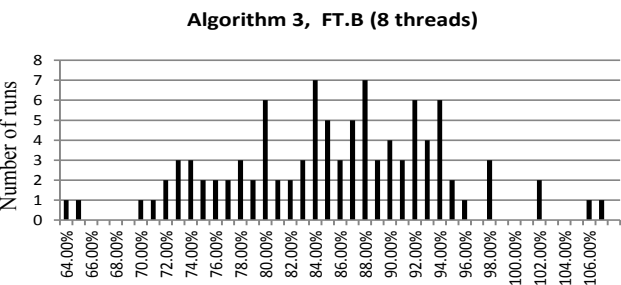
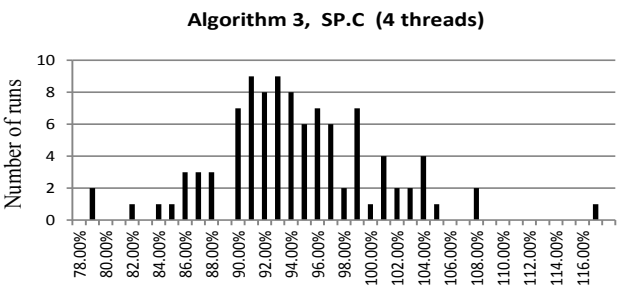
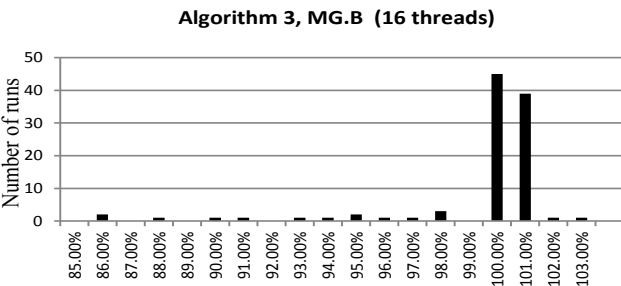
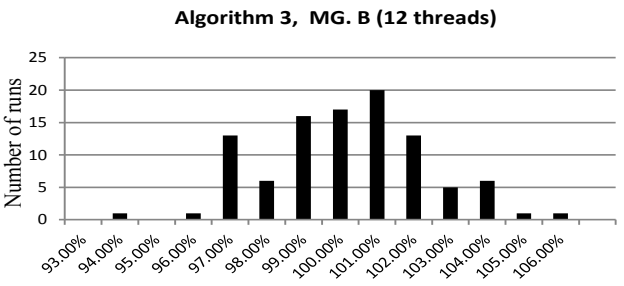
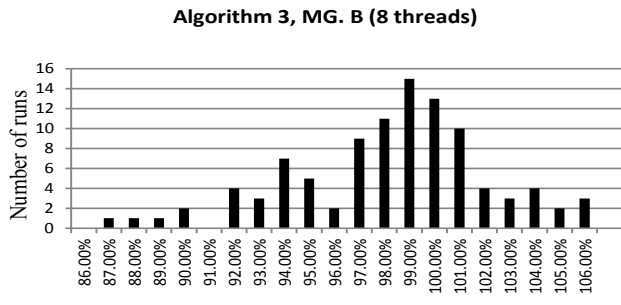
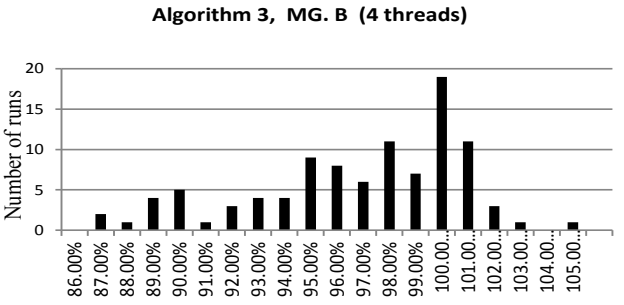


Figure 5: Performance comparison between Algorithm 3 and the system default mapping X-Axis: Execution time under Algorithm 3 divided by the execution time under the system default mapping

Statistical results of NPB benchmarks		MG		FT		BT		SP	
		Test counts	average	Test counts	average	Test counts	average	Test counts	average
thread num=4	correct	65	94.22%	29	94.37%	86	96.24%	83	92.00%
	wrong	2	103.25%	42	104.10%	1	103.21%	8	106.35%
	same as system default	33	100.00%	29	100.00%	13	100.00%	9	100.00%
thread num=8	correct	61	95.43%	81	91.95%	96	90.74%	95	84.21%
	wrong	12	103.88%	4	103.08%	0	100.00%	2	105.79%
	same as system default	27	100.00%	15	100.00%	4	100.00%	3	100.00%

Table 4: Statistical results of NPB benchmarks

map 16 threads on cores using a balanced mapping (one thread per core) and measure total execution time of iterations 1 through 20 (we do not measure iteration 0 to avoid warm-up effects). Then, we use 4 more iterations to collect snapshots of memory behavior in the TNT, apply Algorithm 2, make a prediction of the new mapping and measure the execution time of the next 20 iterations, from 25 to 44. The results in Figure 3 show the ratio of execution time (iterations 25–44) after prediction to the execution time before prediction (iterations 1–20) with random balanced mapping (less than 100% means better). In most cases, Algorithm 2 improves performance. The algorithm outperforms the random mapping by up to 28%, therefore it is robust and adapts effectively regardless of the initial mapping.

Figure 4 shows the histogram of execution time compared to the system default (Linux with first-touch policy) after applying Algorithm 2. Due to ignorance of the critical path, Algorithm 2 cannot outperform the system default. In most cases, the performance of the predicted mapping is the same as the system default.

We test Algorithm 3 under the following scenario: We first assign a random number of threads to run from iteration 1 to 20, then we change the number of threads to a specific number (i.e., 4, 8, 12 or 16) to run the next 20 iterations (21–40), using the system default scheduler. We use iterations 41–44 to collect snapshots of memory behavior in the TNT, then we apply Algorithm 3 and measure the execution time of the next 20 iterations (45–64). Figure 5 shows this histogram of MG, SP, and FT using Algorithm 3 with 4, 8, 12, and 16 threads in iterations 20 to 64. We find that Algorithm 3 performs well with 4, 8, or 12 threads. With 16 threads, in most cases, the performance of Algorithm 3 is the same as the system default. In executions with fewer threads Algorithm 3 is better because the system default scheduler tends to select an imbalanced thread mapping after the number of threads changes. These mappings lengthen the critical path.

To validate the accuracy of the derived mappings (in terms of whether the algorithms find the optimal mapping), we exhaustively ran each benchmark 100 times with 4 and 8 threads, for a total of 800 runs. We categorized the thread mappings as “correct predictions”, “wrong predictions” and “same as system default” according to the differences between predictions and the system default. We classify a performance rate under 99% of the system default as “correct”,

larger than 101% compared to the system default as “wrong” and between 99% and 101% of the system default as “same as system default”. Table 4 shows the statistical results and average ratio of execution time compared to the system default of four NPB benchmarks. We found that only 8.9% of the predictions are wrong. These predictions incur a 4.27% weighted performance loss. 74.50% of the predictions are correct and achieve 8.13% weighted performance gain. 16.63% of the predictions are the same as system default. When the system default has a suboptimal mapping, Algorithm 3 can improve performance by up to 27.29%, 35.09%, 17.08% and 23.26% for MG, FT, SP and BT respectively.

5. RELATED WORK

Terboven et al. [12] proposed a data placement policy, “next touch”, to migrate pages with heavy remote accesses dynamically. Ribeiro et al. [10] used different data access patterns to guide the memory placement policy on NUMA systems. Both attempted to improve performance by changing data placement. However, they must guarantee that the benefit surpasses the penalty of migrating data.

Majo et al. [6] proposed a NUMA-aware task scheduler by measuring LLC pressure and NUMA penalty. Their algorithm requires application parameters that must be obtained offline, which prevents dynamic adjustments to improve performance. Zhuravlev et al. [2, 14] argue that LLC misses are not the only factor that causes performance degradation and that the memory controller and prefetch mechanism are also important. They propose an online task scheduler but they still use LLC miss rate as a metric to measure the extent of local contention. McCurdy et al. [8] argue that NUMA problems can be identified by the help of hardware counters that track remote memory references. These crossbar events can now be counted in modern AMD and Intel architectures. We find that LLC misses are not the only factor of performance degradation and use the memory request event mentioned by Blagodurov et al. [2] as the metric to capture NUMA performance degradation.

Curtis-Maury et al. [3] proposed the concept of DCT to adjust the number of threads in different OpenMP regions dynamically to improve performance. Li et al. [5], extended the concept of DCT and built a power-aware prediction model to save energy with Hybrid MPI/OpenMP programs. We extend their work with algorithms that optimize thread placement on NUMA systems.

6. CONCLUSIONS

NUMA architectures raise significant performance issues due to mismatches between data and thread placement. We presented NUMA-aware, thread placement algorithms that consider the critical path to address NUMA issues in OpenMP programs. To the best of our knowledge, these algorithms are the first to use prediction and critical path analysis to derive nearly optimal thread mappings. In the future, we plan to validate the performance of our tool on non-NUMA optimized benchmarks, such as Parsec and Sequoia benchmarks. We also plan to release a beta-version of the tool.

7. ACKNOWLEDGMENTS

This work is partially supported by a Marie Curie International Reintegration Fellowship, through the I-Cores project (Grant ID FP7-MCF-IRG-224759). Partly performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

8. REFERENCES

- [1] AMD. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. AMD, 2010.
- [2] BLAGODUROV, S., ZHURAVLEV, S., FEDOROVA, A., AND KAMALI, A. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 557–558.
- [3] CURTIS-MAURY, M., SHAH, A., BLAGOJEVIC, F., NIKOLOPOULOS, D. S., DE SUPINSKI, B. R., AND SCHULZ, M. Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 250–259.
- [4] KLUG, T., OTT, M., WEIDENDORFER, J., TRINITIS, C., AND MÜNCHEN, T. U. autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems.
- [5] LI, D., DE SUPINSKI, B., SCHULZ, M., CAMERON, K., AND NIKOLOPOULOS, D. Hybrid MPI/OpenMP Power-Aware Computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (April 2010), pp. 1–12.
- [6] MAJO, Z., AND GROSS, T. R. Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead. In *Proceedings of the International Symposium on Memory Management* (New York, NY, USA, 2011), ISMM '11, ACM, pp. 11–20.
- [7] MATTSON, T. G., RIEPEN, M., LEHNIG, T., BRETT, P., HAAS, W., KENNEDY, P., HOWARD, J., VANGAL, S., BORKAR, N., RUHL, G., AND DIGHE, S. The 48-Core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.
- [8] MCCURDY, C., AND VETTER, J. Memphis: Finding and Fixing NUMA-Related Performance Problems on Multi-core Platforms. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2010).
- [9] MIZELL, D., AND MASCHHOFF, K. Early Experiences with Large-Scale Cray XMT Systems. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (may 2009), pp. 1–9.
- [10] RIBEIRO, C., MEHAUT, J.-F., CARISSIMI, A., CASTRO, M., AND FERNANDES, L. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on* (Oct. 2009), pp. 59–66.
- [11] SINGH, K., CURTIS-MAURY, M., MCKEE, S. A., BLAGOJEVIĆ, F., NIKOLOPOULOS, D. S., DE SUPINSKI, B. R., AND SCHULZ, M. Comparing Scalability Prediction Strategies on an SMP of CMPs. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*.
- [12] TERBOVEN, C., AN MEY, D., SCHMIDL, D., JIN, H., AND REICHSTEIN, T. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?* (New York, NY, USA, 2008), MAW '08, ACM, pp. 377–384.
- [13] WARE, M., RAJAMANI, K., FLOYD, M., BROCK, B., RUBIO, J., RAWSON, F., AND CARTER, J. Architecting for Power Management: The IBM POWER7 Approach. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (Jan. 2010), pp. 1–11.
- [14] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 129–142.