Efficient Barrier Implementation on the POWER8 Processor

C.D. Sudheer
IBM Research
New Delhi, India
sudheer.chunduri@in.ibm.com

Ashok Srinivasan
Dept. of Computer Science
Florida State University
Tallahassee, USA
asriniva@cs.fsu.edu

***Abstract*—POWER8 is a new generation of POWER processor capable of 8-way simultaneous multi-threading per core. High-performance computing capabilities, such as high amount of instruction-level and thread level parallelism, are integrated with a deep memory hierarchy. Fine-grained parallel applications running on such architectures often rely on an efficient barrier implementation for synchronization. We present a variety of barrier implementations for a 4-chip POWER8 node. These implementations are optimized based on a careful study of the POWER8 memory sub-system. Our best implementation yields one to two orders of magnitude lower time than the current MPI and POSIX threads based barrier implementations on POWER8. Apart from providing efficient barrier implementations, an additional significance of this work lies in demonstrating how certain features of the memory subsystem, such as NUMA access to remote L3 cache and the impact of prefetching, can be used to design efficient primitives on the POWER8.**

***Keywords*-Barrier; POWER8; MPI; pthreads;**

I. Introduction

The POWER8 is a new high-end processor from IBM, and IBM's first high-end processor to be available under OpenPOWER Foundation. It provides up to 8-way SMT per core, thus permitting a high degree of parallelism in node. Further details on its architecture are presented in Section II. Applications may implement parallelism using multiple threads, multiple processes, or a combination. In many applications involving fine-grained parallelization, the performance of barrier calls play a crucial role [1]. In this paper, we present our efficient MPI and POSIX pthreads barrier implementations on a 4-chip POWER8 server.

An efficient barrier implementation relies on characterization of the memory sub-system. We present early related work on the POWER8 memory sub-system in section III, and also related work on similar architectures, especially in the context of collective communication implementations. We present our results on additional characterization of the memory subsystem in section IV. We then describe a variety of popular barrier algorithms in section V.

Section VI presents our comprehensive evaluation of these algorithms, along with results of optimization for the POWER8 architecture. In particular, the most effective optimizations account for NUMA effects in remote L3 cache access and the impact of pre-fetching to cache. Our best implementation achieves latency of around 1 μs - 4 μs on 1 and 8-way SMT respectively. This improves on the existing implementations by one to two orders of magnitude. Our implementation uses characteristics of the memory subsystem, which can also help optimize other applications, as mentioned in section VII on future work.

The primary contributions of this paper are as follows.

- Developing efficient MPI and POSIX threads based barrier implementations that improve on existing ones by one to two orders of magnitude.
- Demonstrating the potential for optimization through control of hardware prefetching by the POWER8 core. This will be useful for optimizing other applications too on POWER8.
- Demonstrating optimization of data movement on multi-chip POWER8 servers by accounting for NUMA effects in access to L3 cache.

II. POWER8 ARCHITECTURE

POWER8 is the eighth-generation POWER processor [2], designed for both high thread-level performance and system throughput on a wide variety of workloads. POWER8 processor is implemented in IBM's 22nm SOI technology [3]. Each POWER8 chip has up to 12 processor cores, a 50% increase over the previous generation POWER processor. The 649mm2 die, shown in Figure 1, includes 12 enhanced 8-way multithreaded cores with private L2 caches, and a 96MB high-bandwidth eDRAM L3 cache.

The POWER8 memory hierarchy includes a per-core L1 cache, with 32KB for instructions and 64KB for data, a per-core SRAM-based 512KB L2 cache, and a 96MB eDRAM-based shared L3 cache. The cache line size on POWER8 is 128 bytes. In addition, a 16MB off-chip eDRAM L4 cache per memory buffer chip is supported. There are 2 memory controllers on the chip supporting a sustained bandwidth of up to 230GB per second. The POWER8 chip has an interconnection system that connects all components within the chip. The interconnect also extends through module and board technology to other POWER8 processors in addition to DDR3 memory and various I/O devices.

The POWER8 based Scale Out servers [4] such as the one we used in our work are based on IBM‘s Murano

dual-chip module (DCM) [5], which puts two half-cored POWER8 chips into a single POWER8 socket and links them by a crossbar [6]. Each DCM contains two POWER8 chips mounted onto a single module that is plugged into a processor socket. For example, a 12-core DCM actually has two 6-core POWER8 processors mounted on the module. The particular server we used in this work is 8247-42L [7], and it has 2 sockets, each with one DCM containing two 6-core POWER8 chips. In total, it has 24 cores with 192-way parallelism.

Each POWER8 core can issue and execute up to ten instructions and 4 load operations can be performed in a given cycle [2]. This higher load/store bandwidth capability compared to its predecessor makes POWER8 more suitable for high bandwidth requirements of many commercial, big data, and HPC workloads.

POWER8 also has enhanced prefetching features such as Instruction speculation awareness and Data prefetch depth awareness. Designed into the load-store unit, the data prefetch engine can recognize streams of sequentially increasing or decreasing accesses to adjacent cache lines and then request anticipated lines from more distant levels of the memory hierarchy. After a steady state of data stream prefetching is achieved by the pipeline, each stream confirmation causes the engine to bring one additional line into the L1 cache, one additional line into the L2 cache, and one additional line into the L3 cache [2]. Prefetching is automatically done by the POWER8 hardware and is configurable through the Data Streams Control Register (DSCR). The POWER ISA supports instructions and corresponding programming language intrinsics to supply a hint to data prefetch engines to override the automatic stream detection capability of the data prefetcher [3]. Our shared memory based barrier implementation is optimized as described later by configuring this DSCR register.

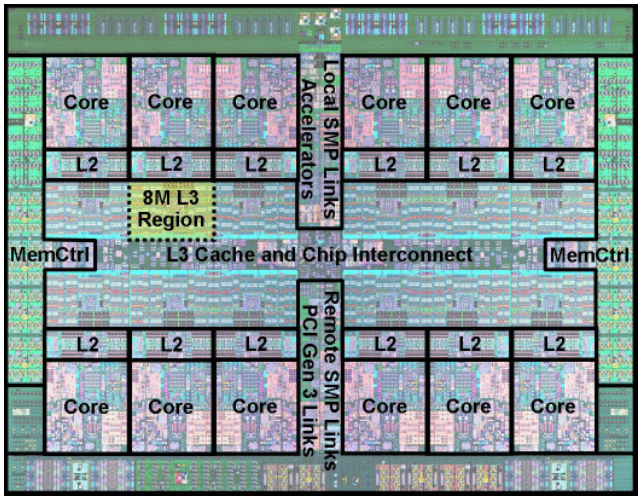

Figure 1. POWER8 processor chip

III. RELATED WORK

In [8] various aspects of a POWER8 system were studied using microbenchmarks, and the performance of OpenMP directives is evaluated. The performance of few scientific applications using OpenMP is evaluated and it highlights the overhead of OpenMP primitives especially when 8-way SMT is used in each core.

Communication between the cores in a cache coherent system is modeled in [9] using Xeon Phi as a case study. Using their model, they designed algorithms for collective communication.

Our barrier implementation is not built on top of point-to-point primitives. Rather, all processes directly use a common region of shared memory. Such use of shared memory has been shown to be more efficient than implementing collectives on top of point-to-point primitives [10] on the Opteron. We have also implemented collectives using shared memory for Cell BE and Xeon Phi. In our earlier work, we had used double buffering on the programmer controlled caches (local stores) of the Cell processor to optimize MPI [11]. We also showed that double buffering can be effective even on the Xeon Phi [12]. Since the caches on Xeon Phi are not programmer controlled, we induce the same benefits by careful design of the algorithms. Though our implementations on POWER8 use shared memory, the implementation is tuned considering the cache hierarchy and prefetching features of POWER8 rather than through the above optimization techniques.

IV. EVALUATION OF MEMORY SUBSYSTEM

Understanding the memory bandwidth, memory access latency or the cache-to-cache bandwidth on the POWER8 is crucial to optimize parallel codes. Here we focus on the chip organization used in the POWER8 scale-out servers, which uses a 2 socket configuration. The L3 cache is shared among the the cores and a core can access a remote L3 cache block and the latency for the access depends on the placement of the remote core. The remote core could be present on the same chip, or same socket or a remote socket, and access latency varies accordingly. To achieve better parallel performance, core binding and NUMA placement are crucial.

To better understand the NUMA effects of the POWER8 memory subsystem, we used two benchmarks to measure the core to core access latency. First, we use the standard ping-pong benchmark [13] to measure the point-to-point communication latency. Message passing latency between two nodes or between two cores within a node is commonly determined by measuring ping pong latency. MPI latency is usually defined to be half of the time of a ping pong operation with a message of size zero.

The table I shows MPI latency between two POWER8 cores with the two MPI processes placed onto different

chips. For convenience, we use the following naming convention. The chips on the 2-socket POWER8 server is referred as socket 0 having two chips A and B, and socket 1 having two chips C and D. One MPI process is always bound onto a core in chip A, and latency is measured by placing the other task onto different chips. As can be seen from the latency numbers, when the two cores are on the same chip (A), the latency is less compared to when the cores across the chips communicate. Also, notice the latency with cores between A and B, which are on the same socket, is less compared to the same between A and C. MPI tasks are bound to cores using MPI runtime process binding options. For this particular experiment OpenMPI rankfile option was used to bind the processes.

Remote core on	Latency (us)
chip A	0.67
chip B	1.27
chip C	1.38
chip D	1.47

Table I
0-BYTE PING PONG LATENCY

These results convey that careful placing of tasks onto cores is essential for obtaining better MPI latency. Another benchmark we use is a 2-process barrier. The table II shows the barrier latency when the two MPI processes are mapped onto cores on different chips. One process is always bound to chip A and the second process moved across the chips and measure the barrier latency. These results also demonstrate the significance of NUMA effects on the MPI latency and potential importance of binding for scaling parallel codes on POWER8.

2nd process on	Latency (us)
chip A	0.33
chip B	0.39
chip C	0.43
chip D	0.46

Table II
2-TASK BARRIER LATENCY

Yet another important feature we need to understand is effect of mapping within a chip. For this, we consider a 6-process barrier with all the 6 processes mapped onto the 6 cores in the same chip. The barrier code was run with all the possible 6! permutations of the process-core mapping, and results showed that affinity does not have a significant effect on the latency as shown in the table III. The understanding from these benchmarks is used in optimizing our barrier implementation as described in section VI.

Min	0.295
Max	0.328
Average	0.303

Table III
BARRIER LATENCY IN A CHIP WITH ALL POSSIBLE MAPPINGS.

V. BARRIER ALGORITHMS

We now present various algorithms we have used for barrier [11][12]. We used POSIX shared memory as a communication channel between the processes using POSIX functions *shm_open* and *mmap*. The shared memory region is contiguous and is memory mapped at the time of the creation of the communicator for MPI and during the pthread_init for pthreads implementation. We design our algorithms and implementations based on our observations on the performance of the memory subsystem. We explain our rationale further in section VI.

Let the number of processes be P. All the algorithms except dissemination, follow the *gather/broadcast* style. In this approach, a designated process, called the root, waits for all the processes to enter the barrier. After that, the root broadcasts this information to all processes. When a process receives the broadcast, it exits the barrier.

Dissemination [14]: In the *k*th step, process *i* sets a flag on process $i + 2^k$ (*mod* P) and polls on its flag until it is set by process $P + i - 2^k$ (*mod* P). This algorithm takes $\lceil log_2 P \rceil$ steps.

K-ary tree [15]: The processes are logically mapped to the nodes of a tree of degree k. Every process calculates its children using the following formula: $(k \times rank) + i$, where $i = \{i \in N \wedge (1 \leq i \leq k) \wedge (1 \leq i \leq P)\}$. We have modified the algorithm slightly from [15] to avoid false sharing; in the *gather* phase, each node polls on the flags of its children till they are set before setting its own flag. In the *broadcast* phase, every node resets the flags of its children before exiting the barrier. The algorithm takes $2\lceil log_k(P) \rceil$ steps.

Centralized: In Centralized algorithm, a shared memory segment consists of P flags is used. In the *gather* phase, a process sets its flag and polls on it until it is unset. The root process resets the flags after all the processes have entered the barrier. A process exits the barrier only when its flag has been reset.

Tournament [14]: In each round of the *gather* phase of the algorithm, processes are paired and the winner goes to the next round. A process with lower rank is considered to be the winner and waits for its flag to be set by its losing partner. The overall winner initiates the broadcast phase. The broadcast phase is similar to the one used in the *tree* algorithm. The *gather* phase takes $\lceil log_2(P) \rceil$ steps and the *broadcast* phase takes $\lceil log_k(P) \rceil$ steps.

Binomial Spanning Tree (BST) [16]: The processes

are logically mapped to a binomial spanning tree. The working principle differs from the *tree* algorithm only in the fashion in which it constructs the tree. Each process calculates its children by adding 2^i to its rank, where $i = \{i \in N \wedge (log_2(rank) < i < \lceil log(P) \rceil) \wedge (rank + 2^i < P)\}$.

VI. EMPIRICAL EVALUATION

We first describe the computational infrastructure and then present results of the empirical evaluation.

A. Computational Platform

We ran all the benchmarks on one node of an IBM cluster with 8 POWER8 processor nodes clocking at 3.32 GHz. Each node has two socket dual-chip module (DCM), with 6 cores per chip and a total of 24 cores per node. We use the IBM Parallel Environment (PE) Runtime Edition for Linux Version 1.3 [17] for MPI codes. We use the IBM XL C/C++ 13.1.2 compiler for the thread based codes. For testing the OpenMPI barrier, we use the recent OpenMPI version 1.8.4. We use a modified version of the OSU MPI benchmark suite [13], that benchmarks collectives on MPI COMM WORLD, to measure the performance of MPI collective operations on the POWER8. To obtain statistically sound results, we repeated the measurements 2 million times and we give the results as the average values.

While running MPI based benchmarks for OpenMPI barrier and IBM PE MPI barrier, the code was compiled with appropriate compiler flags such that the shared memory (sm) is used for MPI communication. For both the MPIs, appropriate core binding runtime options are used to map the processes onto the cores. With the IBM PE MPI, environment variables such as MP_TASK_AFFINITY, MP_BIND_MODE, MP_BIND_LIST are used for process binding. With OpenMPI, rankfiles and -map-by options are used. For pthreads based codes, we use pthread_attr_setaffinity_np library call to map a thread onto a specific core. With the XL based OpenMP, OMP_PLACES is used for explicitly binding the threads to the cores.

We use the OpenMP micro-benchmark suite (version 3.X) from EPCC to quantify these overheads [18] and with this benchmark we used a sample size of 10000.

Our analysis of various implementations for barrier is presented next.

B. Flag representation

All the algorithms use a shared memory segment of size P flags, one important aspect is to evaluate the different options for representing the flag. Since a flag is accessible by all the cores, the size for the flag need to be chosen carefully considering the data prefetching and cache coherency effects. We evaluate different options for representing the flag.

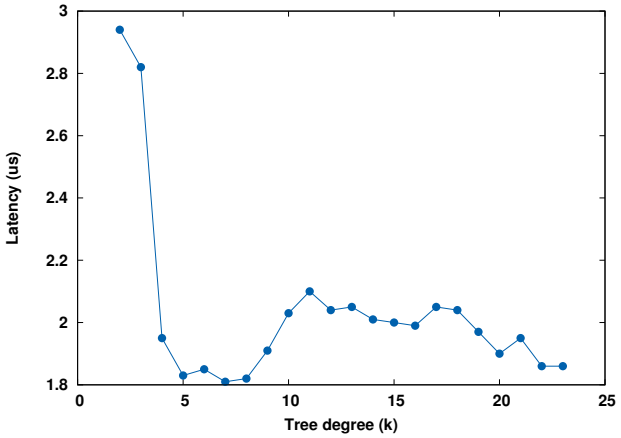

Figure 2. *k-ary tree* barrier latency

One possibility is to use one integer for representing the flag. We use the *k-ary tree* algorithm to evaluate this. POWER8 cache line size is 128B, and hence for a 24 process MPI, with this representation all the flags fit in one cache line. So, many processes access the same cache line, and this could potentially lead to higher false sharing cache misses. In effect, a store access to the cache line by a core results in invalidation of the cache line at the other cores. Using this flag representation, a latency of 21 us was observed for the barrier and the hardware counter data showed high amount of cache misses. We then experimented using the other possibility of using one cache line for representing the flag, and this incurs in much lesser latency of 1.8 us. Hence, using one integer for flag is not a good option, and hence we represent flag with one cache line. This observation is different from what we observed with our implementation for Xeon Phi, where, despite false sharing, the first option was better. As discussed in [12], with the first option, better vectorization was observed and effect of false sharing was not seen. However, on POWER8 the first option results in more cache misses.

C. Optimal degree for k-ary tree

Given that *k-ary tree* has many different implementations based on the degree, we need to identify the optimal degree. We discuss about this next.

Figure. 2 compares the influence of the tree degree for the *k-ary tree* algorithm. It gives the latency for a 24 process barrier, where a process is mapped onto one core. All the implementations represent the flag with one cache line. The number of steps in the algorithm reduces as the degree of the tree increases. This is reflected in the decrease in the latency of the algorithm as the tree degree increases. The best latency of roughly around 1.81 us is observed with tree degrees 5, 7 or 8. Each POWER8 chip has 6 cores, and possibly with these degrees, the number of inter-chip data transfers are less thus resulting in better latency. Next,

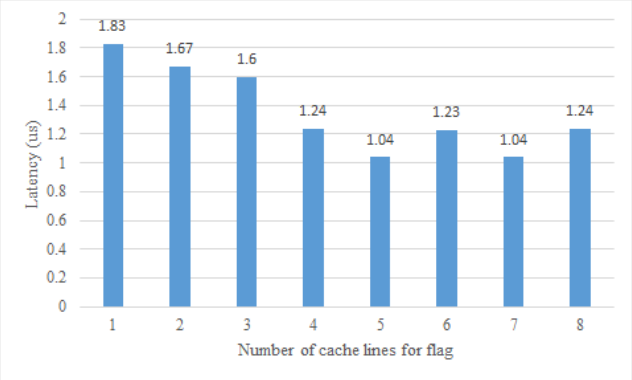

Figure 3. *k-ary tree* degree (5) with different cache lines per flag

we look at the *k-ary tree* barrier operation flow in terms of memory accesses to evaluate it better.

As described in section V, the *k-ary tree* barrier involves two phases, gather and broadcast. In the gather phase, all the processes having children polls on their flags, and this involves remote cache load accesses. In the broadcast phase, every process resets the flags of its children, and this involves remote cache store accesses. So, in essence, the cache line (flag) corresponding to a process 'i' is moved between the caches of process 'i' and its parent. This results in cache conflict misses which effects the overall latency. The effect of conflict misses increases as we increase the degree k. Another issue that should be effecting the latency is the number of steps involved in the algorithms with different degrees. With lower degrees, the number of steps in the gather and broadcast phases are high, resulting higher inter-chip data transfers. With degrees 5, 7 or 8, these transfers are less, and we could use one of these as our as our optimal *k-ary tree* degree.

The additional complexity in the analysis results due to the interplay of the aggressive prefetching done by the data prefetch engine. We now discuss the potential impact of prefetching on the performance of the barrier.

D. Prefetch effects

In our implementation, all the process flags are stored contiguously in memory. The access to a flag by a core results in placing the corresponding cache line in its L1 cache, however, due to the aggressive prefetching used by the core, a stream of cache lines are also brought into its L1 cache. This stream of lines contains the cache lines corresponding the flags of the other processes. So, when the other cores attempt for a prefetch issue for their respective flags, those prefetch requests are rejected as the requested lines are already in transit. The others cores will then get access to their lines only through a demand load request rather through a prefetch request. The demand request for the line by the these cores will be serviced from the remote cache which already have these. This request is referred to as an L3 lateral cast-out or an intervention request, and this access to remote caches incurs in more latency.

With an aim to gain more insight into the processing of this implementation of *k-ary tree*, we monitor hardware counters related to the memory. They indicate a high ratio of L3 store misses and a high amount of L3 prefetches. This validates our above reasoning about the detrimental effects of prefetching for this representation of one cache line per flag.

To reduce the miss rate due to the prefetching, one possible approach is to store the flags corresponding to the process far apart such that prefetch engine will not load the valid lines of other processes. One possibility is to use multiple cache lines to represent the flag instead of just one cache line. Figure 3 gives the latency for the *k-ary tree* barrier with degree 5 for varying number of cache lines per flag. The best latency of 1.04 us is observed when 5 or 7 cache lines are used per flag. With 5 cache lines, the hardware counter data showed less L3 misses. POWER8 uses a default DSCR value of 0 and with this a ramp depth of 4 is used by the data prefetch engine [2], so possibly 4 cache lines are brought in per one prefetch data stream request. Hence, when 5 cache lines are used per flag, each core get its flag into its cache as a prefetch request rather than as a demand load request. So, when the memory access stream stride is beyond 4 cache lines, the prefetch will not be loading the valid lines corresponding to the remote cores.

However, using 5 cache lines for a flag is not a scalable solution as each MPI communicator in an application uses a separate shared memory segment, leading to possible high memory requirement. Hence, we need other ways to address prefetching effects, and we could achieve it by configuring the prefetch engine at the run time. As described in section II, the POWER8 core has the support to configure the data prefetch engine. We used the intrinisic __prefetch_set_dscr_register() [3] to set the required value in the DSCR register. Intrinsic __prefetch_get_dscr_register() can be used to know the default DSCR value. We used these intrinsics with the 1 cache line per process implementation, and set the DSCR value such that hardware prefetching is disabled. We set the DSCR to 40. The results for this implementation is compared with the other two prior ones next.

Figure. 4 compares the three implementations, i.e., i. one cache line for flag with the default hardware prefetch enabled, ii. 5 cache lines for flag, and iii. one cache line for flag with hardware prefetch disabled. These implementations are tested with different tree degrees, and ii. and iii. perform consistently better than i. Considering the high memory requirement for the approach of using 5 cache line for flag, we select iii. as our best barrier implementation. And clearly, degree 5 gives the best latency, and hence from here any reference to *k-ary tree* based implementation corresponds to the implementation of iii. with degree 5. Next, we do further

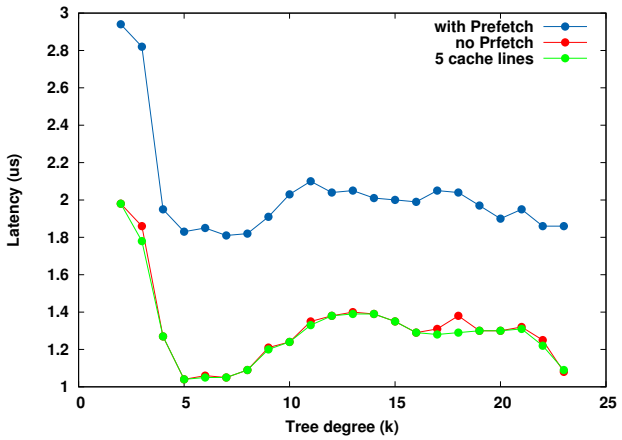

Figure 4. *k-ary tree* barrier latency for tree order

analysis of *k-ary tree* based implementation to understand its operation better.

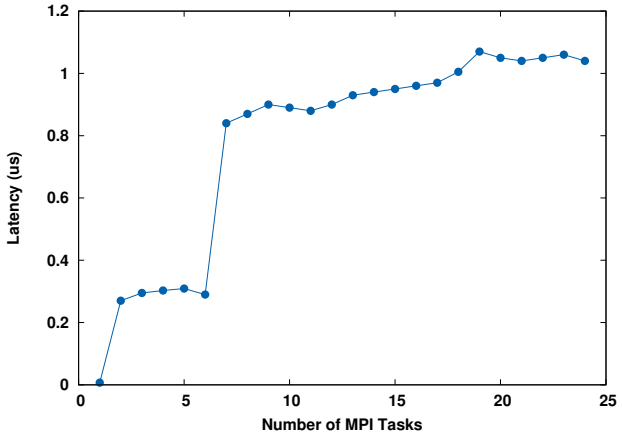

Figure 5. Barrier latency with increasing number of MPI tasks (one task mapped onto one core)

E. Further Analysis of k-ary tree

The scalability of barrier for the *k-ary tree* based implementation as the number of processes participating in the barrier are increased is shown in Figure. 5. When the number of processes involved in a barrier are less than 6, all the memory accesses involved in the barrier are contained within a single chip and hence results in a good latency. However, as can be seen with sudden jump in the figure, when 7 or more processes are involved in the barrier, the accesses go across the chip resulting in higher latency. Table IV shows the latency for the 7 process barrier, when the 7th process is placed across the different sockets. Higher latency is incurred as the process moves farther from the chip, this confirms with our earlier benchmarks discussed in section IV.

The 7 process barrier test reveals that the inter-chip accesses are the major contributor to the barrier latency and the algorithm that reduces the number of transfers performs the best. The small difference in latency between the 7 process and 24 process barrier also shows that the effect of inter-chip accesses with the 24 process barrier is less.

Having found the best combination of tree degree and flag representation for the *k-ary tree* barrier, we now compare it with the other barrier implementations.

7th process on	Latency (us)
chip B	0.81
chip C	0.90
chip D	0.93

Table IV
7 PROCESS MPI BARRIER

F. Comparison with other implementations

We compare the barrier latency for the *Disemmination*, *Binomial*, *Tournament*, and *k-ary tree* algorithms. The flag is represented with one cache line even in these algorithms. Figure. 6 shows the barrier latency for these algorithms. The *k-ary tree* algorithm gives performs among all the algorithms.

In the *Dissemination* algorithm, the flag of a process is updated by a different process in each round. Each time the cache line is updated, it becomes a candidate to be invalidated in the other caches. And also, in every round of the algorithm there are P processes communicating across the cores, whereas in the *k-ary tree* algorithm there are much fewer processes communicating across the cores in a typical step. Apart from the higher number of steps, the data access pattern of dissemination involves a lot of data transfer from remote processes. That is likely the reason for its worse performance.

The broadcast phase in both the *Tournament* and *k-ary tree* algorithm is similar. However the degree used with tournament is lower, and also in the gather phase, *Tournament* algorithm involves more stages thus it involves more data transfers.

G. With SMT

Until here, we consider only the cases which only use the 1-way SMT of each POWER8 core. Now, we look at the latency of the barrier where there more processes are mapped onto a core using upto 8-way SMT.

Figure 7 gives the timings for the *k-ary tree*, *Binomial* and *Centralized* algorithms. (note: The *Centralized* algorithm is essentially *k-ary tree* with degree 23 for 1-way SMT and with degree 191 for 8-way SMT). Our best barrier implementation for 1-way SMT was the *k-ary tree* with degree 5, and it also performs consistently well as we increase the

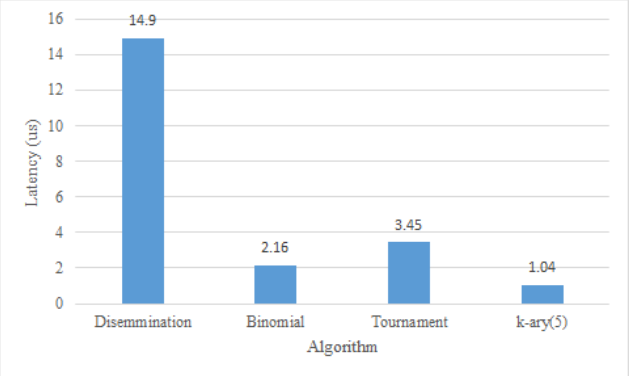

Figure 6. 24 process MPI barrier latency for different algorithms

number of processes occupied per core. The performance of *Centralized* implementation decreases compared to other algorithms from the 3-way SMT, potentially due to higher cache conflict misses. In the *Centralized* algorithm, the gather and broadcast phase involve just one step. In this algorithm every process is polling on its flag to be reset by the root process. So, while every other process wants to read, the root intends to perform a write. In the worst case, there will be a cache miss for every process whose flag the root is updating. Though cache miss effects on latency are not observed when 1 process per core was used, with higher number of processes per core, the performance of the algorithm diminishes due to higher cache misses.

A 192-way MPI process barrier just takes 4.5 us with our implementation. And the implementation scales well as we use more processes in a core.

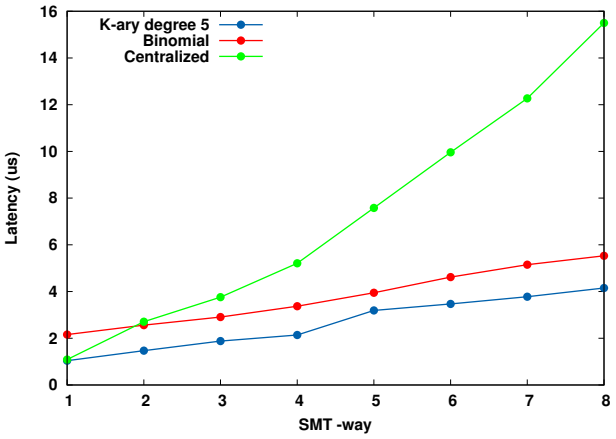

Figure 7. Latency comparison *k-ary tree* with *Binomial Tree* based and *Centralized* barrier implementations for 1-way upto 8-way SMT

H. Comparison with MPI libraries

Now that we have the *k-ary tree* as the best among all our implementations, we compare it with the barrier implementations available in the standard MPI libraries. Note that these benchmarks were run using the good compile time and run time options specified for the optimal Intra-node collectives with the respective libraries. Figure. 8 gives the latency for the *k-ary tree* barrier, IBM PE MPI barrier and the OpenMPI barrier. Our barrier implementation perform consistently better than others, and interestingly even the latency for a 192 process (8-way SMT) MPI barrier with our implementation is faster than even a 1-way MPI barrier latency with the others.

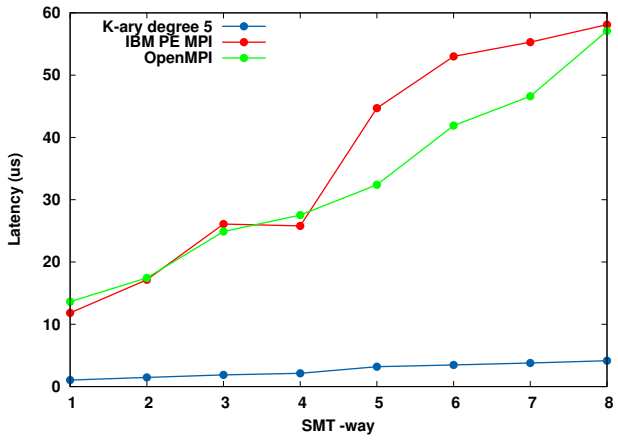

Figure 8. Latency comparison *k-ary tree* with IBM PE MPI and OpenMPI barriers for 1-way upto 8-way SMT

Having established that we have an efficient barrier implementation for Intra-node MPI, we now look at implementing our barrier algorithm using a shared memory programming model.

I. Shared memory based barrier

We use pthread programming model to implement the barrier. We compare our barrier implementation with the pthread barrier available in the POSIX library. Table V shows the latency for the POSIX library pthread barrier call and for our pthreads based barrier from 24-way up to 192-way thread level parallelism. Our barrier implementation is few orders of magnitude better than the library implementation.

SMT	*k-ary tree* degree(5) pthreads	POSIX pthreads barrier
1	1.05	127.6
2	1.41	397.8
3	1.85	624
4	2.1	1229
5	2.55	1546
6	3.06	1929
7	3.48	2252
8	4.07	2611

Table V
PTHREADS BARRIER LATENCY (US)

We also compare its performance with the barrier available in other shared memory programming models such as OpenMP. Figure 9 shows the latency for our pthreads based barrier and the OpenMP BARRIER[1] primitive from 24-way up to 192-way thread level parallelism. Our implementation performs around about 2-3 times faster than the OpenMP barrier. This speedup would have a significant effect on the parallel applications.

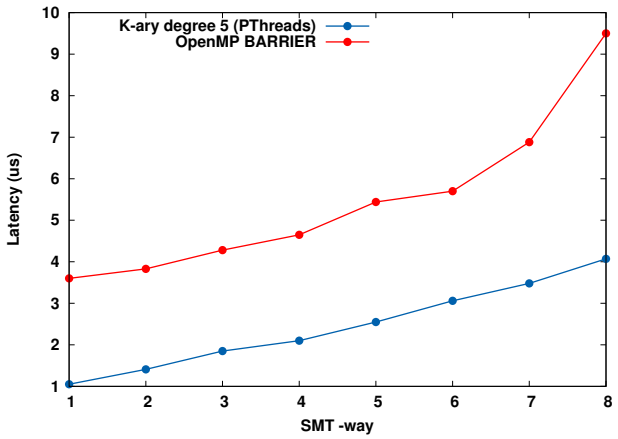

Figure 9. Latency comparison *k-ary tree* based pthread barrier with the OpenMP BARRIER for 1-way upto 8-way SMT

VII. CONCLUSIONS AND FUTURE WORK

We have identified certain important performance characteristics of the memory sub-system on the IBM POWER8 Processor. In particular, we have evaluated the impact of NUMA on L3 cache access and also analyzed the effect of prefetching on the performance of algorithms that access data in more complex ways than a stride-based pattern. We have used this characterization to develop an efficient barrier implementation that is one to two orders of magnitude faster than current MPI and pthreads implementations. In addition, we note that the optimal implementation differs from that on the Intel MIC architecture due to differing impacts of cache access overhead.

The memory sub-system characteristics that we have identified can help optimize other applications, especially if they have an irregular memory access pattern. In future work, we plan to use our results to optimize other collective operations on a single node. Efficient collective communication implementations for large parallel systems build on top of efficient primitives on single node. We intend combining our implementation with inter-node barrier implementations in order to develop optimal solutions on large parallel systems.

[1]Our benchmark results on OpenMP do not match with what was presented in [8]. As they mention, OpenMP runs were not optimized for POWER8. In our work, the OpenMP barrier is evaluated for the best possible OpenMP runtime for POWER8. The OpenMP based benchmarks are compiled with the IBM XL compiler toolchain with the POWER8 specific compilation flag.

ACKNOWLEDGMENT

This material is partly based upon work supported by the National Science Foundation under Grant No. 1525061. We thank Sameer Kumar, Vaibhav Saxena, Manoj Dusanapudi and Shakti Kapoor for discussions on different topics related to this work.

REFERENCES

[1] C. D. Sudheer, S. Krishnan, A. Srinivasan, and P. R. C. Kent. Dynamic load balancing for petascale quantum Monte Carlo applications: The alias method. *Computer Physics Communications*, 184(2):284–292, 2013.

[2] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, 2015.

[3] Performance Optimization and Tuning Techniques for IBM Processors, including IBM POWER8, An IBM Redbooks Publication. 2014.

[4] Power scale-out servers: http://www-03.ibm.com/systems/in/power/hardware/scale-out.html, 2015.

[5] IBM Power Systems Facts and Features: Enterprise and Scale-out Systems with POWER8 Processor Technology. May 2015.

[6] Power8 iron to take on four-socket xeons: http://www.theplatform.net/2015/05/11/power8-iron-to-take-on-four-socket-xeons/, May 2015.

[7] IBM Power System S824L Technical Overview and Introduction, An IBM Redbooks Publication.

[8] Andrew V. Adinetz, Paul F. Baumeister, Hans Bottiger, Thorsten Hater, Thilo Maurer, Dirk Pleiter, Wolfram Schenck, and Sebastiano Fabio Schifano. Performance Evaluation of Scientific Applications on POWER8. In *5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS14) held as part of SC14*, 2014.

[9] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 97–108. ACM, 2013.

[10] Richard L Graham and Galen Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140. Springer, 2008.

[11] M.K. Velamati, A. Kumar, N. Jayam, G. Senthilkumar, P.K. Baruah, S. Kapoor, R. Sharma, and A. Srinivasan. Optimization of collective communication in intra-Cell MPI. In *Proceedings of the 14th IEEE International Conference on High Performance Computing (HiPC)*, pages 488–499, 2007.

[12] P. Panigrahi, S. Kanchiraju, A. Srinivasan, P.K. Baruah, and C.D. Sudheer. Optimizing MPI collectives on Intel MIC through effective use of cache. In *in Proceedings of the 2014 International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pages 88–93, 2014.

[13] OSU MPI Benchmarks: http://mvapich.cse.ohio-state.edu /benchmarks/.

[14] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[15] Michael L Scott and John M Mellor-Crummey. Fast, Contention-Free Combining Tree Barriers for Shared-Memory Multiprocessors. *International Journal of Parallel Programming*, 22(4):449–481, 1994.

[16] Nian-Feng Tzeng and Angkul Kongmunvattana. Distributed shared memory systems with improved barrier synchronization and data transfer. In *Proceedings of the 11th international conference on Supercomputing*, pages 148–155. ACM, 1997.

[17] IBM Parallel Environment Runtime Edition for Linux on Power Version 1.3 simplifies parallel application development. 2014.

[18] J. M. Bull, F. Reid, and N. McDonnell. A microbenchmark suite for openmp tasks. In *in Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP '12)*, pages 271–274, 2012.