Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study

Chao Mei, **Gengbin Zheng, Fillipo Gioachin, Laxmikant V. Kale**

08/03/2010, TeraGrid'10

Motivation

- **Almost all clusters consist of multicore nodes**
  - Node size continues to grow
- **The whole software stack needs to be adapted to the multicore architecture**
  - Application-level
  - Parallel languages (including its runtime system)
  - System-level
- **Potential benefits**
  - Latency is much reduced for intra-node messages
  - Shared-memory data structure

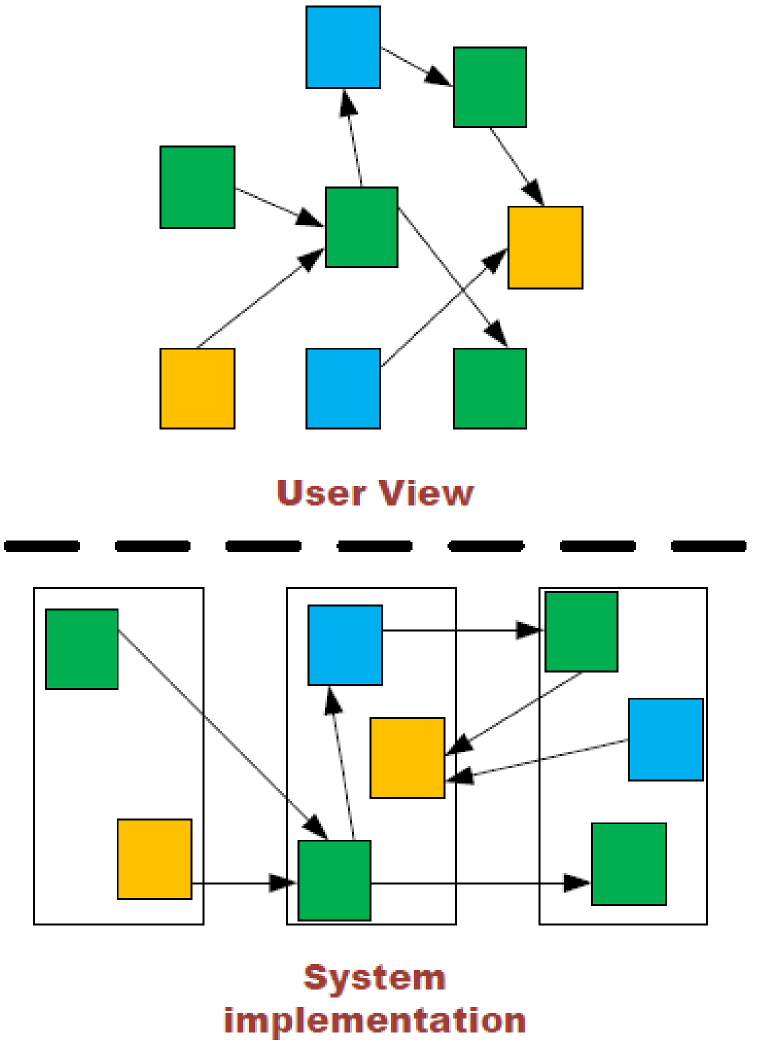
Initial porting of a runtime system doesn't necessarily lead to benefits!

Outline

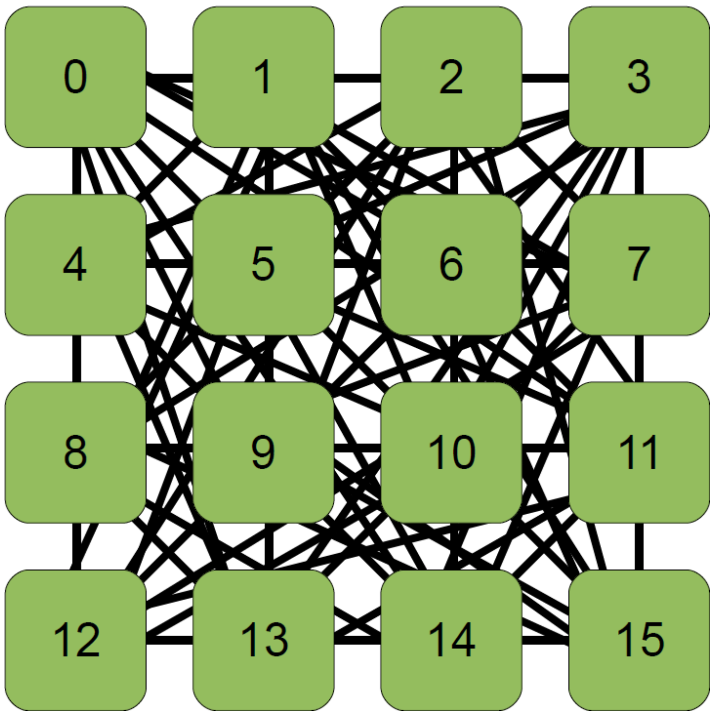
- **Introduction to the runtime system**
  - Charm++
- **Experiment Setup**
  - Benchmark
  - 5 multicore machines
- **Issues and Optimization Techniques**
  - Synchronization overhead
  - Affinity settings
  - …
- **Performance for real applications**

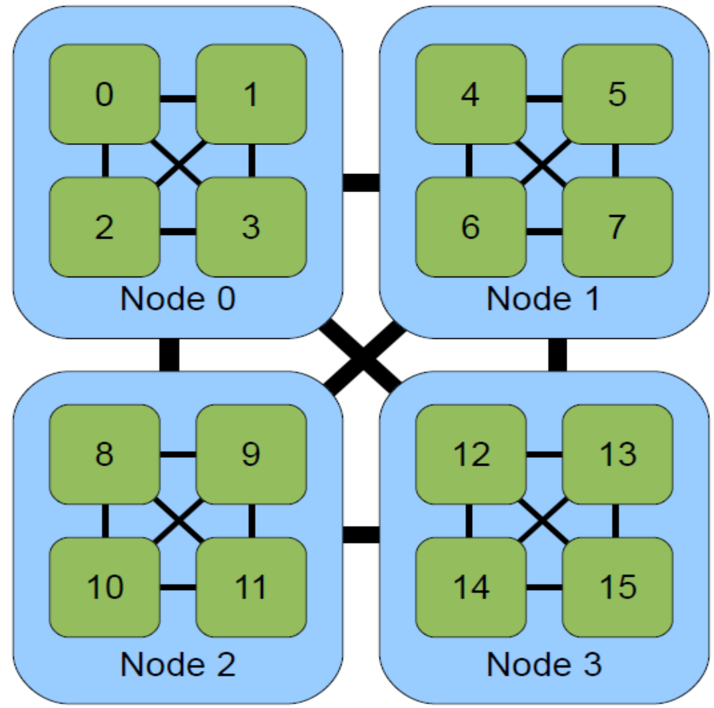
The Runtime System Case: Charm++

- **Objected oriented C++ based**
- **Message driven execution**
  - Asynchronous non-blocking remote method invocation



Architectures of Runtime System





- **non-SMP, process**
  - Network stack
  - POSIX shared memory
- **SMP, process + system thread**
  - Shared memory address space

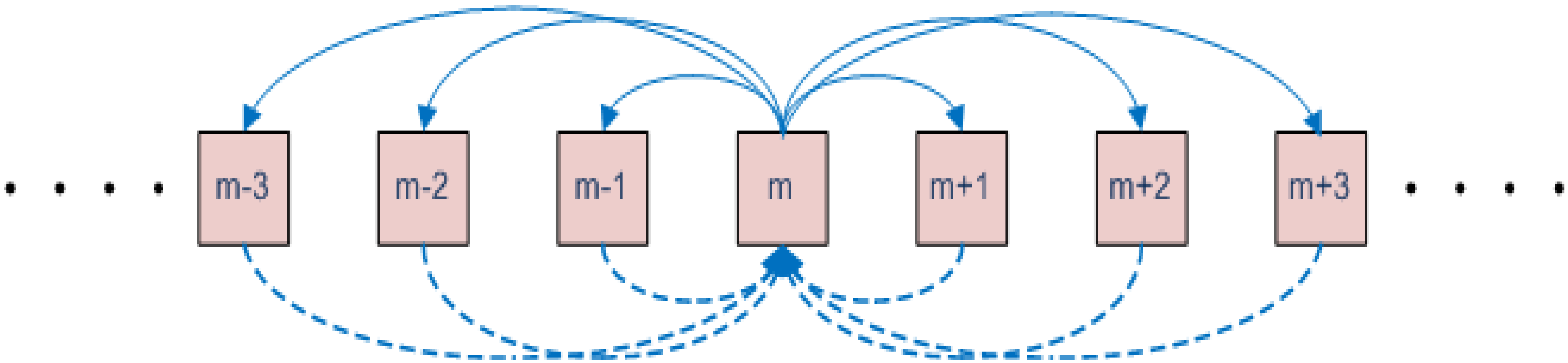
Initial Experiments Result

- Applications do not have any performance improvement
  - NAMD: ~10% degradation ☹
  - ChaNGa: ~2% degradation

- Attack the problem in two steps
  - Issues on a single node
  - Issues on multiple nodes

Experiment Setup: Benchmark

- **kNeighbor (k=3 in our study)**

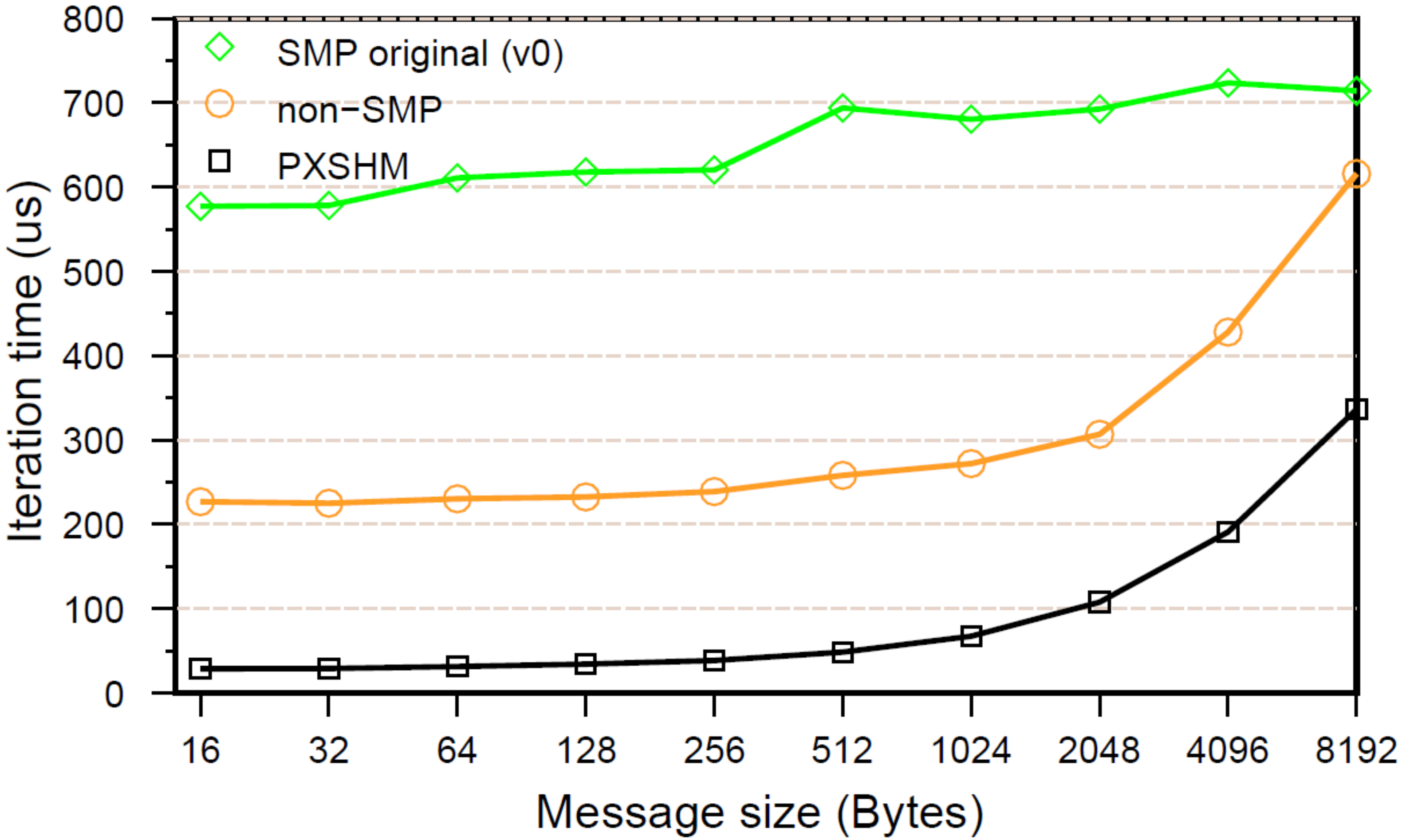


  - Benchmark one iteration time
  - Touch every byte of the message when received
  - Emphasize the performance of message latency in the presence of contention

Experiment Setup: Multicore Machines

- **Five multicore machines**
  - A: AIX 6.1/IBM Power 5, a 16-core (SMT=2) node
  - B: Ubuntu 8.04/Intel Nehelem Xeon E5520, a 8-core (SMT=2) node
  - C: Ubuntu 8.04/Intel Harpertown Xeon E5405, a 8-core node
  - D: Ubuntun 8.04/AMD Barcelona Opteron 2356, a 8-core node
  - E: CentOS 5.4/Intel Dunnington Xeon E7450, a 24-core node

Initial Comparison for *k*Neighbor

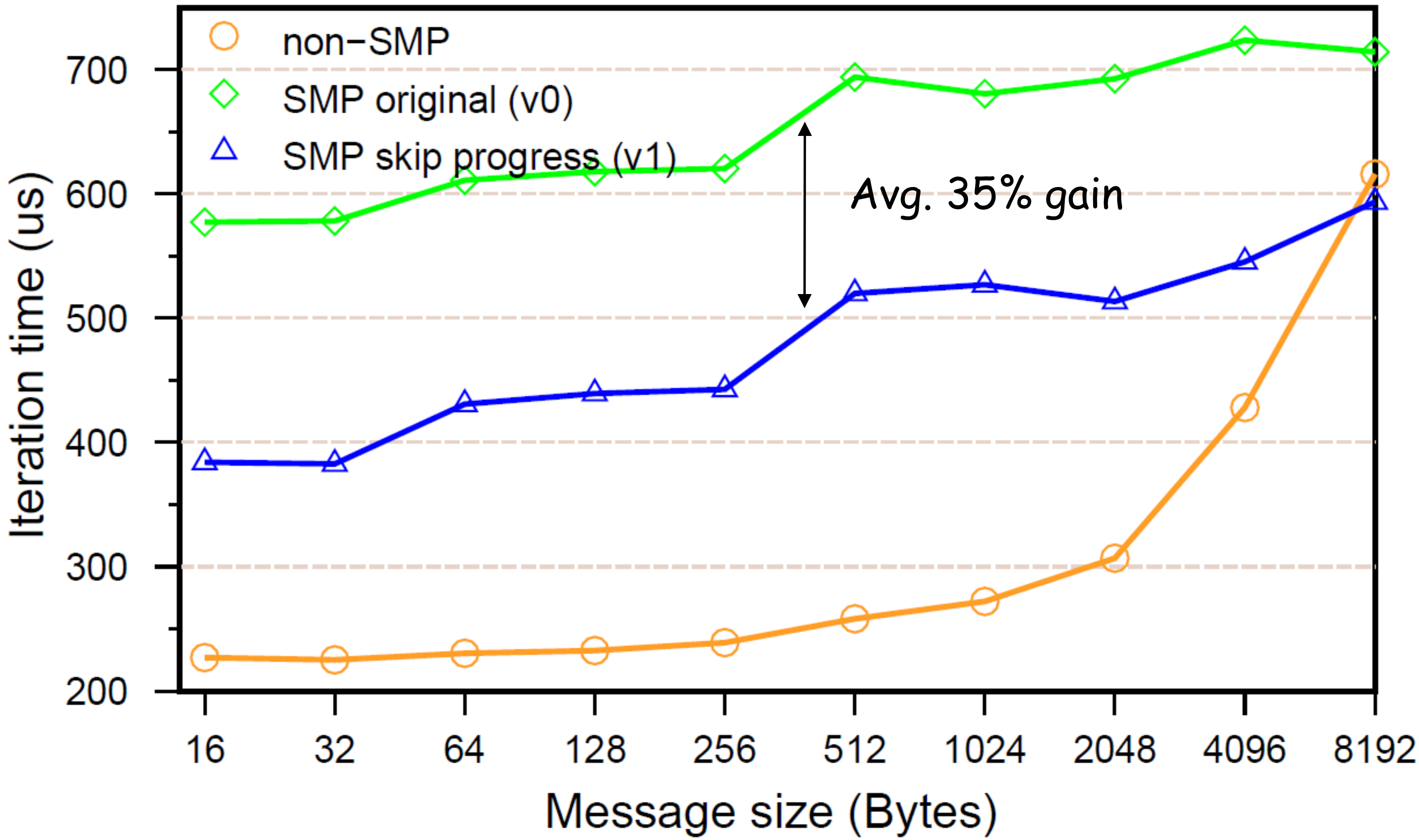


Network Progress Engine Issue

- **Network progress engine**
  - Process incoming messages and send outgoing message immediately
  - Expensive

- **Initial Usage**
  - Invoked every time a message is sent
    - contention on the engine

- **Current Usage**
  - Not necessary for intra-node message
  - Only invoke network progress engine if it is an inter-node message



Not simply change processes to threads and make it thread safe, but re-think the overall design of the architecture

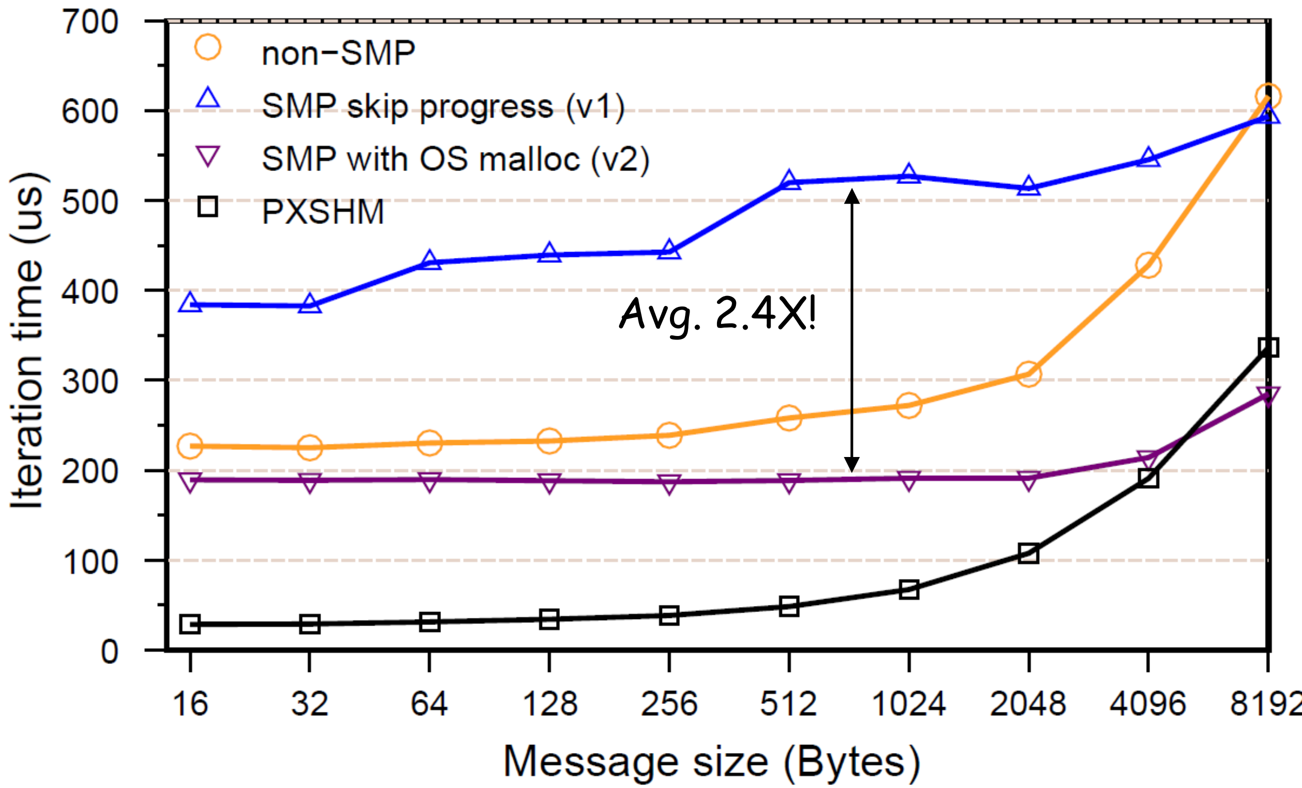
Multi-threaded Performance Issues

- **Efficient locking and synchronization among threads**
  - **key factor for fast fine-grained intra-node communication**

- **Three issues**
  - **Memory management**
  - **Granularity of critical sections**
  - **Message queues**

Memory Management

- **Charm++ uses its own memory allocator**
  - Based on a GNU memory allocator developed seven years ago
  - Every malloc/free is protected with a lock ☹
- **Switched to OS provided memory module**



Performance of OS-provided Memory Module

- **Synthetic benchmark: every thread simultaneously allocates memory of the same size for 100,000 times, then free**

#thds	A(us)	[redacted]	B(us)	C(us)	D(us)	E(us)
1	1.06		0.78	0.80	1.13	0.68
2	2.23		1.30	1.44	1.53	2.03
4	6.06		3.95	2.14	2.36	2.73
8	15.35		8.71	3.69	4.72	7.06
16	36.89		22.63	n/a	n/a	14.58
24	n/a		n/a	n/a	n/a	21.31
32	210.96		n/a	n/a	n/a	n/a

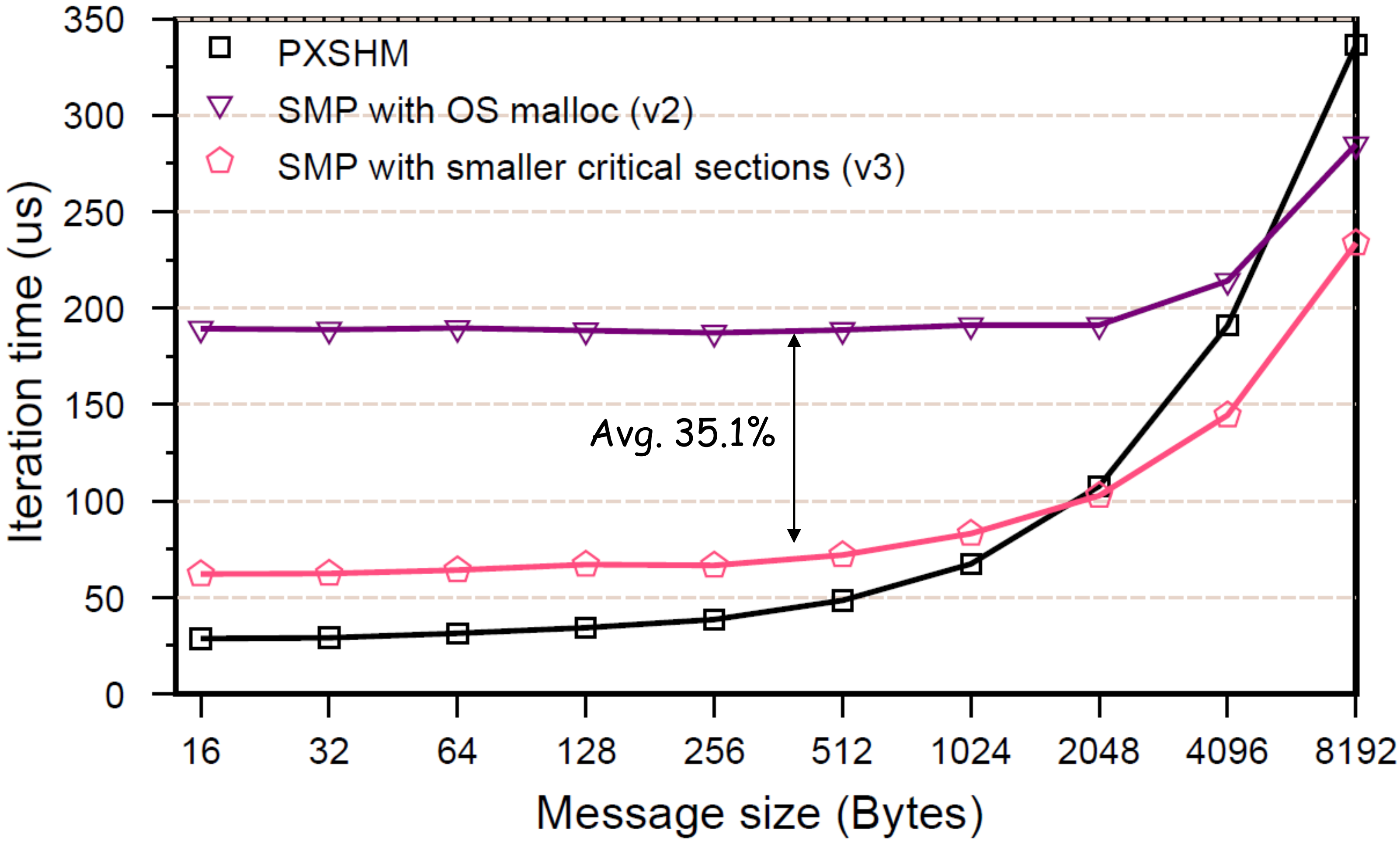
Performance of OS-provided Memory Module

- **Synthetic benchmark: every thread simultaneously allocates memory of the same size for 100,000 times, then free**

#thds	A(us)	A/M(us)	B(us)	C(us)	D(us)	E(us)
1	1.06	1.03	0.78	0.80	1.13	0.68
2	2.23	1.02	1.30	1.44	1.53	2.03
4	6.06	1.05	3.95	2.14	2.36	2.73
8	15.35	1.03	8.71	3.69	4.72	7.06
16	36.89	1.06	22.63	n/a	n/a	14.58
24	n/a	n/a	n/a	n/a	n/a	21.31
32	210.96	1.02	n/a	n/a	n/a	n/a

Granularity of Critical Sections

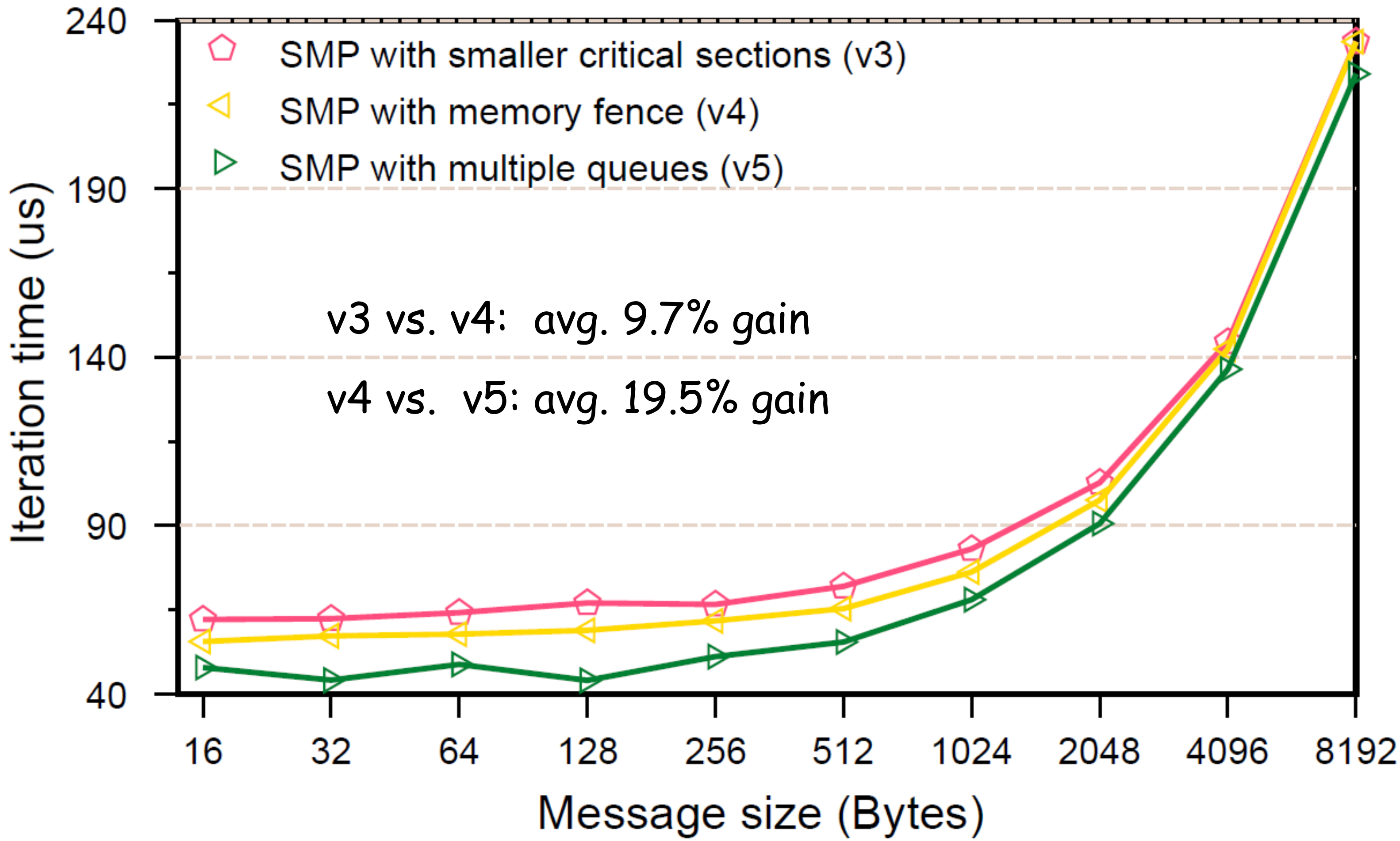
- **Trade-off between productivity and performance**



Message Queues

- **Producer-Consumer Queues (PCQueue)**
  - Commonly used data structure for implementing scheduler queues
- **Scenario in Charm++**
  - Single consumer, multiple producers
- **Use memory fence instead of locks**
  - A general API across multiple platforms for read/write fence
  - Two steps of optimizations
    - Remove locks for consumer
    - Remove locks for producers by having a queue pair between the single consumer and each producer
      - Polling overhead increased

Perf. of Optimizing Message Queues

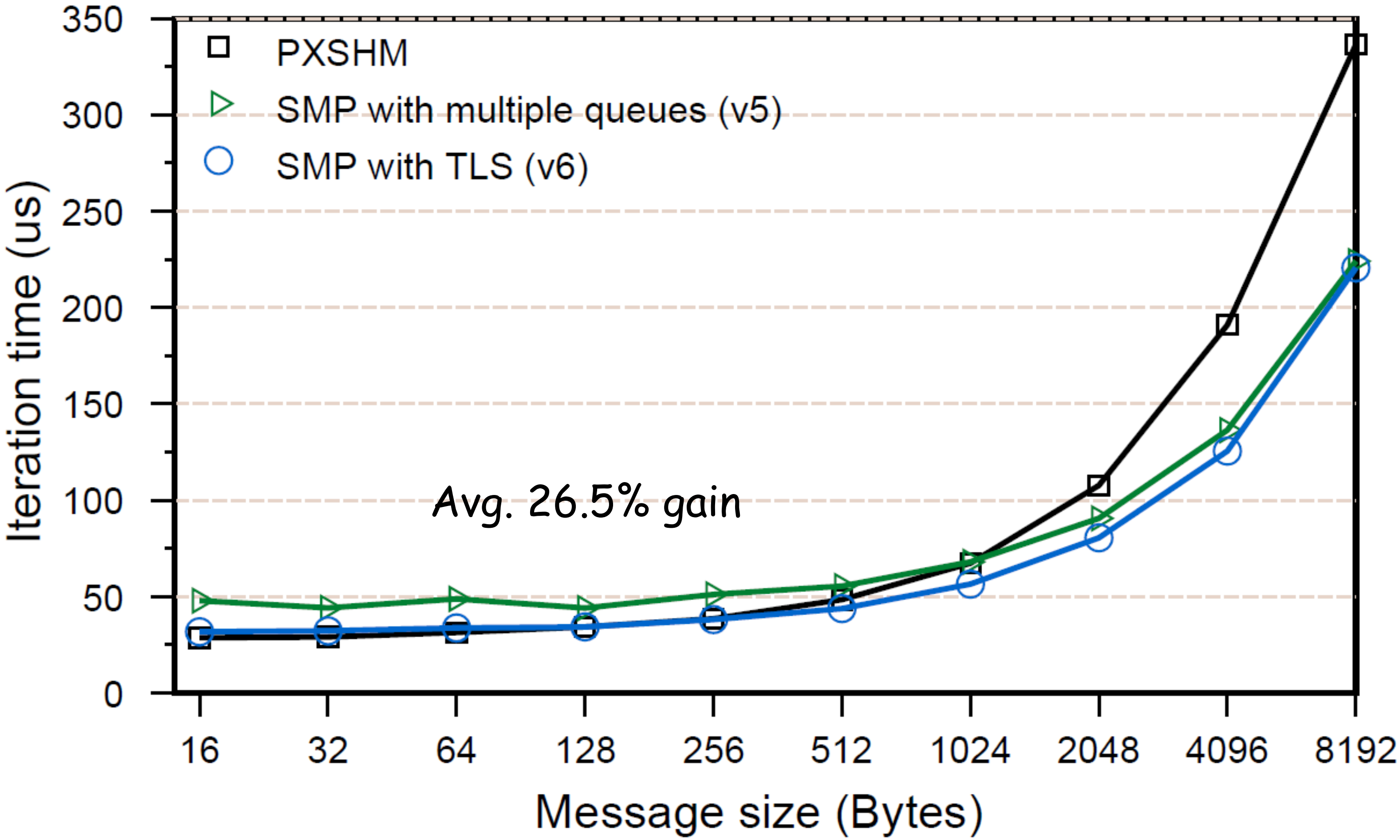


Handling Processor Private Variables

- **Similar to the thread private variables in OpenMP**
  - "Cpv" macros providing transparent usage in non-SMP/SMP mode, e.g. CpvAccess(var)
- **Initial implementation is array-based:**
  - CpvAccess(var) ➔ var[myrank]
  - false sharing ☹
- **Solution: Thread Local Storage (TLS): explict or implicit**
  - pthread_setspecific/pthread_getspecific on Unix-like
  - TlsSetValue/TlsGetValue on Windows
  - "__thread" if supported by compiler and assembler

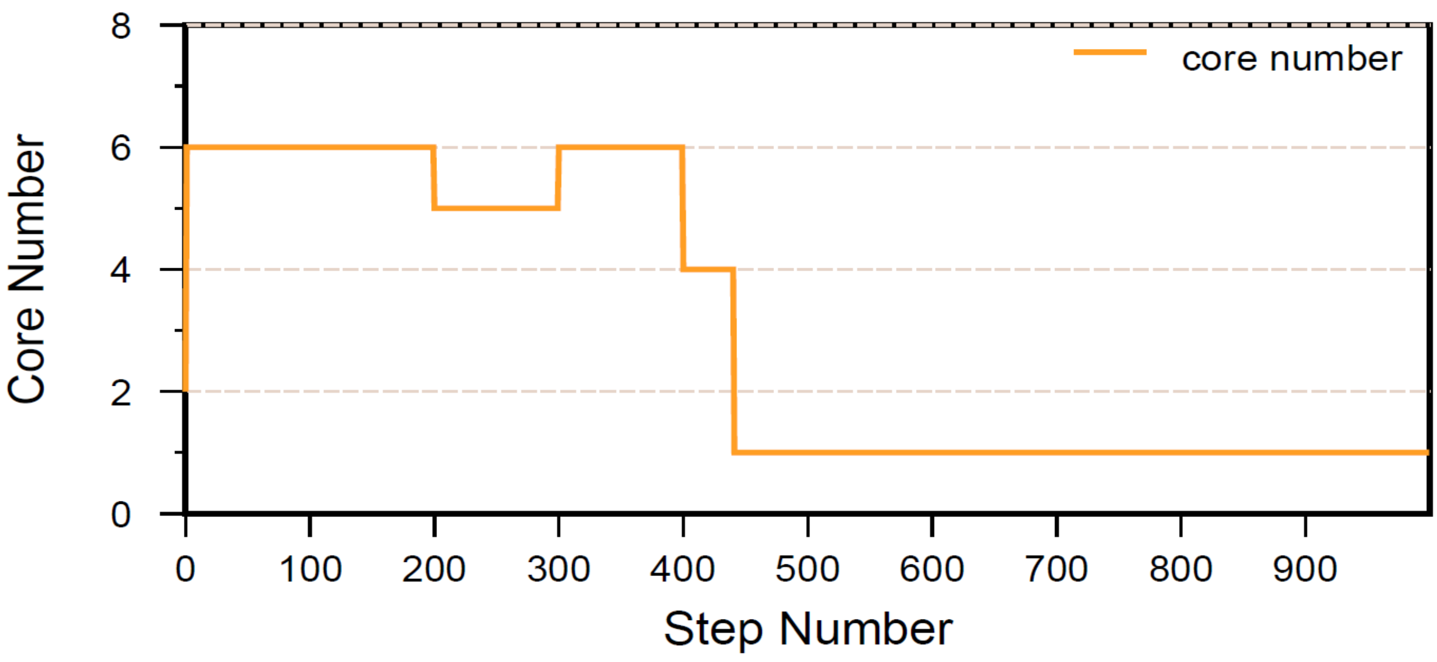
Platform	A (ns)	B (ns)	C (ns)	D (ns)	E (ns)
TLS	0.40	1.27	1.5	1.75	1.26
Array-based	51.58	17.52	10.03	9.61	8.50

Perf. Improvement After Using TLS



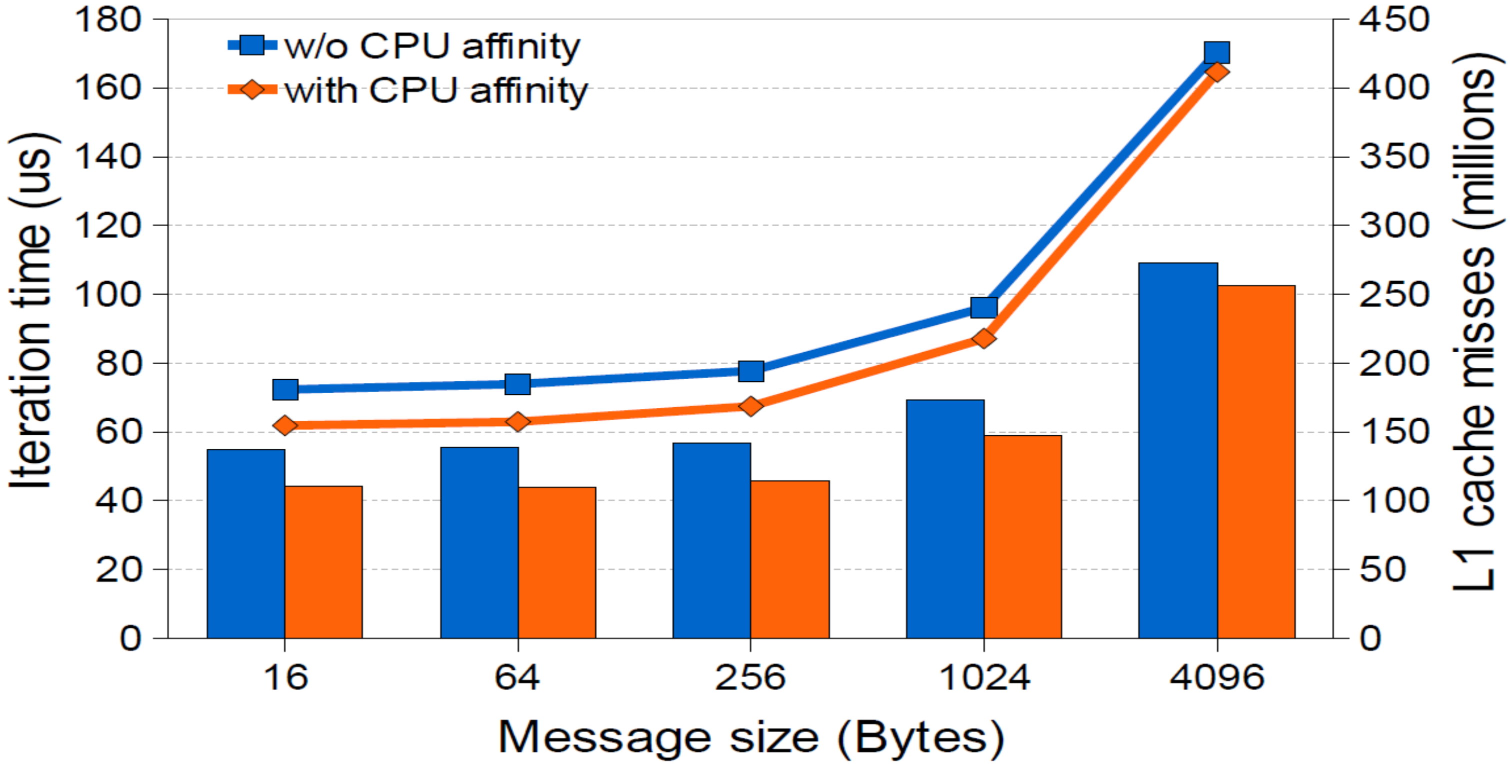
CPU Affinity (1)

- ***OS adopts natural affinity***
  - ***Keep process/thread on the same CPU as long as possible***



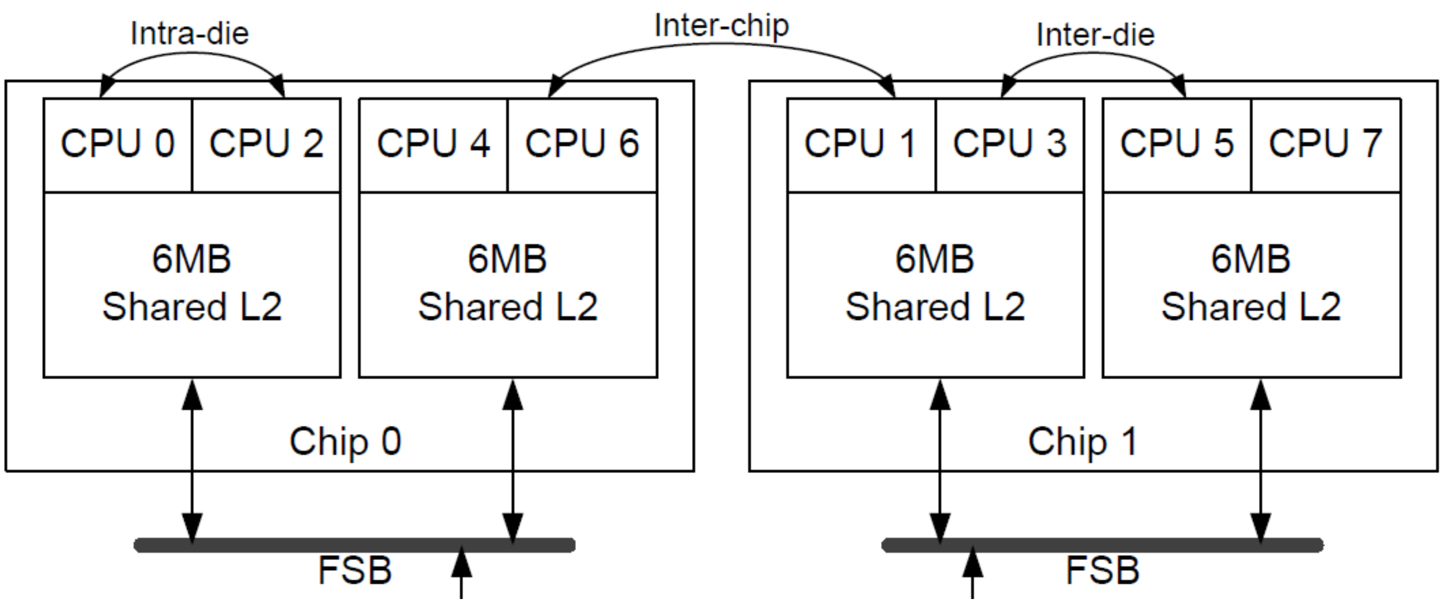
CPU Affinity (2)

- **Just fixing the affinity shows performance improvement**
  - **Fewer L1 cache misses**
  - **Performance better and more stable**



CPU Affinity (3)

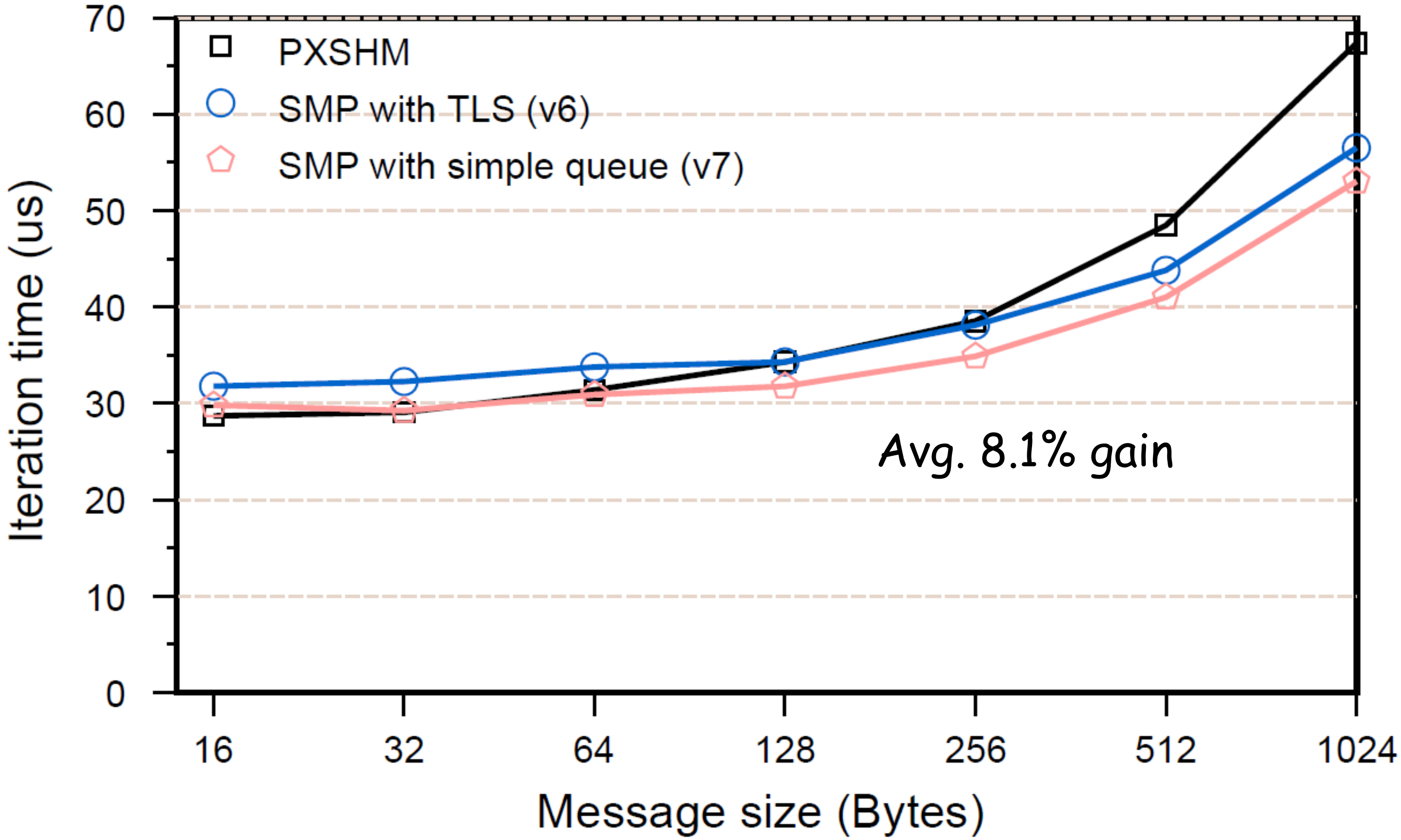
- How to set the CPU affinity generally?
  - A cross-platform function API in Charm++
  - Some TeraGrid sites also provide such functionality when lunching the job
- What's the optimal affinity setting?
  - Depends on the communication pattern of the program
- Example
  - *k*Neighbor in the case of k=1 with 7 elements
  - Message size: 256 bytes
  - Immediate neighbor communication



- Elem(0,1,2,3,4,5,6) → CPU(0,2,4,6,1,3,5): 11.66 us
- Elem(0,1,2,3,4,5,6) → CPU(0,1,2,3,4,5,6): 13.37 us
- Why?
  - Inter-chip: 8 vs. 24
  - Inter-die: 8 vs. 4
  - Intra-die: 12 vs. 0

Other Issues

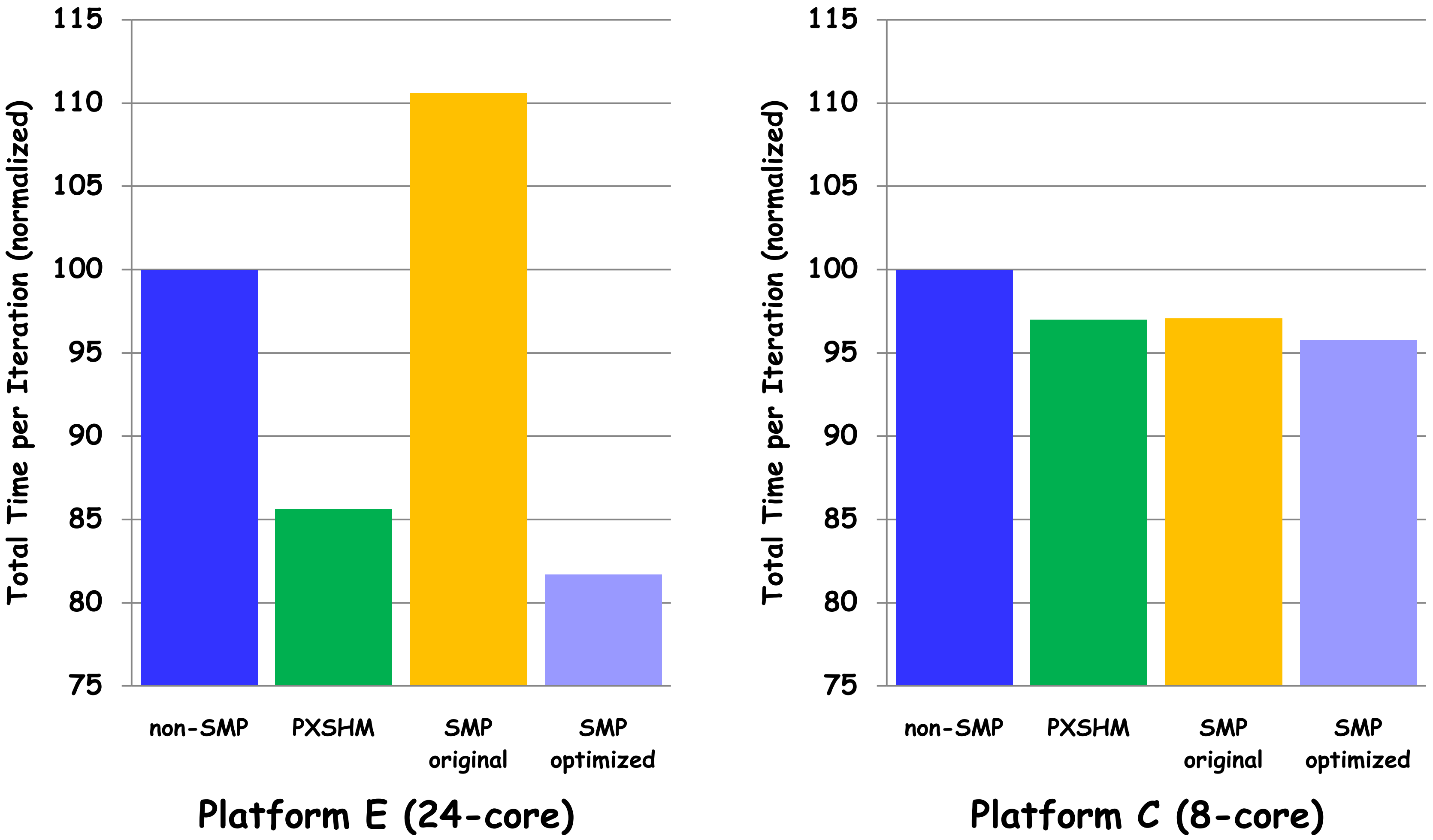
- **Reducing memory accesses in operations of message queues**
  - Very fine-grained performance tuning



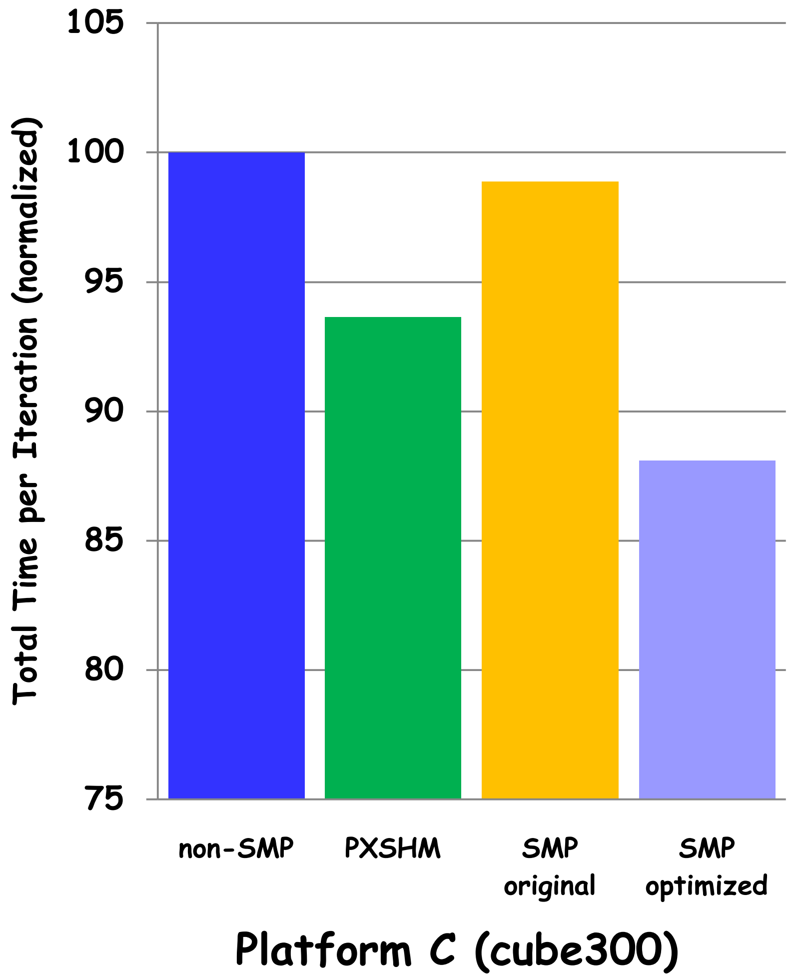
Overall Improvement for *k*Neighbor

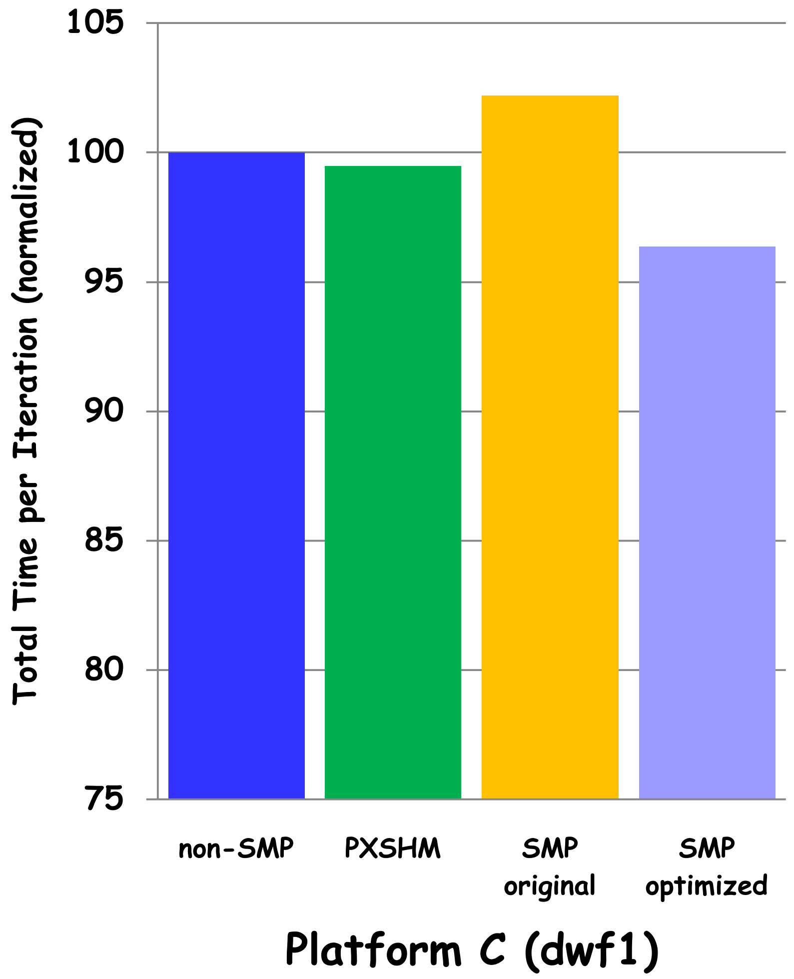
- **14.4X over initial SMP**
- **4.87X over non-SMP**
- **1.21X over non-SMP in PXSHM**

Application Performance: NAMD



Application Performance: ChaNGa





Conclusion

- **Studied the parallelization of a parallel language runtime system for mutlicore platforms via Charm++**
  - Described various issues for the initial implementation
  - Applied optimization techniques correspondingly
    - Lock and synchronization overhead
    - CPU affinity
    - False sharing

- **Should be general enough and useful to other runtime system**

Thank you !

☺

http://charm.cs.uiuc.edu