Performance of the Hybrid LS-DYNA on Crash Simulation with the Multicore Architecture

Yih-Yih Lin[1], Jason Wang[2]

[1] Hewlett-Packard Company, Marlborough, MA, USA
[2] Livermore Software Technology Corporation, Livermore, CA, USA

Summary:

Using crash simulation models, we investigate the multicore performance of the newly developed hybrid LS-DYNA, a method whose speedup arises from both shared-memory and message-passing parallelisms. Theoretically, the hybrid method gains performance advantages over the traditional, message-passing-parallel (MPP) LS-DYNA for two reasons. First, the addition of shared-memory parallelism to the message-passing parallelism reduces the number of messages and their sizes dramatically, which in turn reduces latency and bandwidth requirements on interconnect. Second, the same addition enhances spatial and temporal localities for both code and data accesses, which in turn allows the size-limited cache to work more efficiently. Armed with this theory, we characterize performance of the hybrid method with respect to problem size, core count, core placement, and interconnect speed; thus provide users guidance on when and how to use the hybrid method efficiently. We also attempt to verify the theory by examining message patterns and the effect of core placement.

Keywords:

Hybrid LS-DYNA, Performance, Multicore Architecture

1 Introduction

1.1 A brief history of LS-DYNA parallelism

When DYNA3D, the LS-DYNA precursor, first developed in 1976 [1], all commercial computers were built with uniprocessor architecture; consequently, it did not implement any process-level parallelism and ran serially. By 1989, the year the first version of LS-DYNA appeared, several commercial computers with multiprocessor architecture had already been available, and the shared memory parallelism (SMP) was implemented in LS-DYNA, whose posterior releases have been known as the version of SMP LS-DYNA since. However, the scalability of the SMP LS-DYNA was observed to stop at 8 processors. To solve the limitation of the share memory parallelism, LSTC began to implement distributed memory parallelism in LS-DYNA, using the software MPI (message passing interface) library; distributed memory parallelism implemented with message passing paradigm is customarily called as the message passing parallelism. The first version of LS-DYNA with message passing parallelism, called MPP (Massively Parallel) LS-DYNA, was released in 1993. Since then, MPP LS-DYNA has been observed with scalability up to hundreds and even thousands of processors. However, it has been observed that interconnect speed is a significant limiting factor in the scalability of MPP LS-DYNA. Interconnect speed is determined by its latency and bandwidth: the lower the latency and the higher the bandwidth is, the faster MPP LS-DYNA performs. To reduce MPP LS-DYNA's requirement on low latency and high bandwidth on interconnect, LSTC about a year ago began to implement the hybrid method that combines both the shared memory (as in SMP LS-DYNA) and the message passing (as in MPP LS-DYNA) parallelisms. As to be shown later, the hybrid method indeed has achieved significant performance improvement over the traditional MPP LS-DYNA.

1.2 The shared memory and the message passing parallelisms and their hybrid

In computer terminology, a core is the central processing unit, which has a minimal memory infrastructure that includes caches. A process is a program operating in an address space that is not shared by other processes. Multiple processes can be spawned by a single core. A thread is an executing program that is created from a process and operates in the process' address space; several threads, all sharing the same address space, can be contained in a single process. An application is said to be shared memory parallel if each of its cooperating subprograms, has access to all of a single, shared address space for memory operations. The majority of modern shared memory parallel programs are implemented with threads, and so is SMP LS-DYNA.

An application is said to be distributed memory parallel if it comprises a set of cooperating processes, each of which operates on unshared (local) memory, but each of which is able to communicate its local information with other processes. The best known method for communication among processes is by sending and receiving messages, for which the Message Passing Interface (MPI) Library is the standard software tool. As mentioned earlier, distributed memory parallelism implemented with message passing paradigm is customarily called message passing parallelism. Since its inception, MPP LS-DYNA has been implemented with the MPI Library, and therefore it is also called the pure MPI method in this paper.

In addition to the finite element method, MPP LS-DYNA is based on the domain decomposition method. In the method, the problem is split into small problems on subdomains. Each smaller problem on a subdomain is solved independently, but serially by a distinct process (called *rank* in MPI), and its solution is periodically coordinated with solutions on other subdomains via MPI. In contrast, if each smaller problem is solved with multiple threads, created by its associated MPI rank, then the method is called the hybrid method. Those multiple threads are called SMP threads. Both the traditional MPP LS-DYNA and the hybrid method are called *collectively* as the MPI method in this paper.

1.3 Multiprocessor, multicore architecture

The term *core* is a recent one. Historically, a processor had one and only one central processing unit, and there was no need to call it with another name. However, processors with multiple processing units were invented recently and have become prevailing in the industry, and so the term *core* has been reserved for calling a central processing unit; that way, a processor with two central processing units can be simply called a dualcore processor, with four central processing units a quadcore processor, and so on.

Multiple multicore processors have been organized to make a node, in which all its processors, and hence all their cores, share a global memory. Multiple nodes are further interconnected to make a cluster, in which all nodes, and hence all their cores, can communicate. Consequently, the SMP LS-DYNA can be run only on a single node. In contrast, both the pure MPI method and the hybrid method can be run on a single node or on a cluster.

1.4 Communication and computation costs

Performances of the MPI method are determined by computation and communication costs. Communication cost is the messaging passing cost incurred for iteratively coordinating solutions on subdomains. Each solution on a subdomain, solved either serially or by multiple threads, incurs a cost; their maximum is computation cost. Communication cost is determined by latency and bandwidth of interconnect, which is the conduit for message passing. On the other hand, with a given computer architecture, computation cost is primarily affected by memory access speed, which in turn is affected by CPU affinity and core placement, from users' perspective. CPU affinity is the tendency of a process or a thread to run on a core as long as possible without being moved out to some other core, and core placement means simply which core is run on. For serial or SMP LS-DYNA, CPU affinity with appropriate core placement is almost always beneficial to its performance, but inappropriate core placement can degrade it, sometimes greatly.

2 Performances of the pure MPI method and the hybrid method

2.1 Models, LS-DYNA versions and cluster systems

The car2car model, which is based on NCAC minivan model and is available on the TopCrunch site http://www.topcrunch.org, is the main problem used in this investigation. The model comprises 2.4 million elements and has a termination time of 120 ms. Additionally, the 0.8-million-element 3-vehicle-collision model, also available on the TopCrunch site, is used, but only in Section 3.4 to demonstrate that the hybrid method gains no performance advantage over the pure MPI method when the problem size is small.

Tests are performed with MPP LS-DYNA Version R4.2, using HP-MPI, and its corresponding version of the hybrid method.

Benchmarks are run on two clusters: One comprises nodes containing two of the newly released Nehalem processors, and the other comprises the HP BL2x220c nodes containing two Xeon Harpertown processors. A summary of the hardware configuration for the two clusters is shown below.

Node type	Intel Nehalem white box	HP BL2x220c
Architecture	Xeon X5560 (quadcore) (2.8 GHz)	Xeon E5450 (Harpertown) (quadcore) (3.0 GHz)
Processors/Cores per node	2 processors/8cores per node	2 processors/8cores per node
Highest level cache size per processor	8 MB	12 MB
Cache Configuration	One 8 MB L3 cache shared among 4 cores, containing memory controller	Two 6 MB L2 caches, each shared between 2 cores, requiring separate memory controller
Memory per node	18 GB	16 GB
Interconnect	InfiniBand QDR	InfiniBand DDR
O/S	RH 5.3	SLES 10.2

The left diagram in Figure 1 depicts the cache structure of the Nehalem processor, while the right one depicts that of the Harpertown processor. Three differences in the two cache structures are noticed:

- The Nehalem has 2 MB cache per core, while the Harpertown has 3 MB per core.
- All 4 cores share one same cache in the Nehalem, while one pair of cores shares one cache and the other pair shares another in the Harpertown.
- The Nehalem contains the memory controller, while the Harpertown requires a separate memory controller.

How these differences in cache structures affect the performance of the hybrid method will be discussed later.

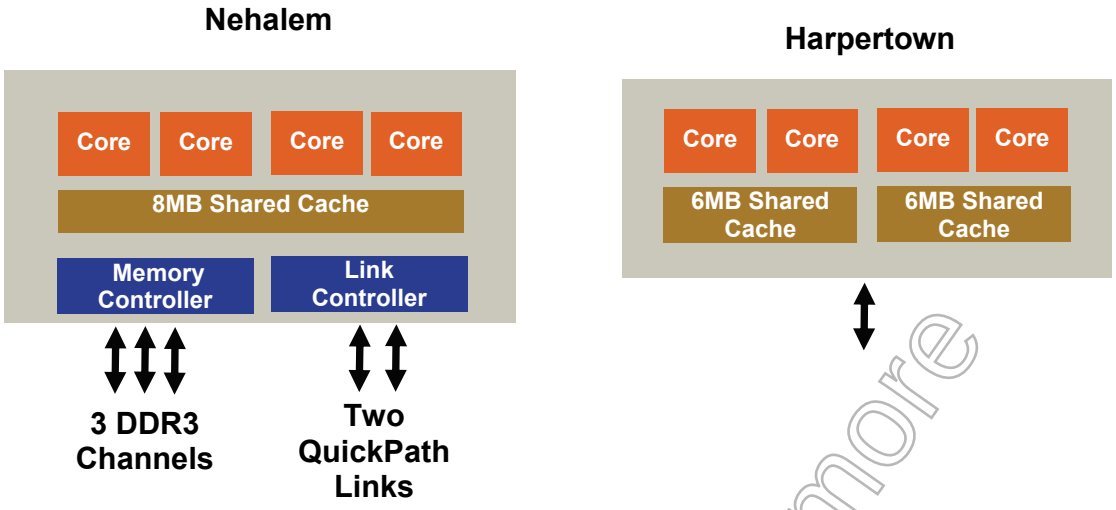

Figure 1: The cache structures of the Nehalem processor and the Harpertown processor

2.2 Results

Figure 2 shows 256-core performances of the pure MPI method and the hybrid method (with 4 SMP threads) and their comparisons on the Nehalem cluster. Figure 3 shows 256-core performances with varying number of threads on the HP BL2x220c cluster. Figure 4 shows performances of the two methods, and their comparisons, with varying core counts from 32 to 512 on the HP BL2x220c cluster. These three figures give out the following three main results for the investigation:

- The hybrid method obtains a performance gain of 1.20X over the pure MPI method with 256 cores.
- The best performance is achieved when the number of threads is equal 4, which is the number of cores per processor.
- The hybrid method begins to gain performance over the pure MPI method when the core count exceeds 128.

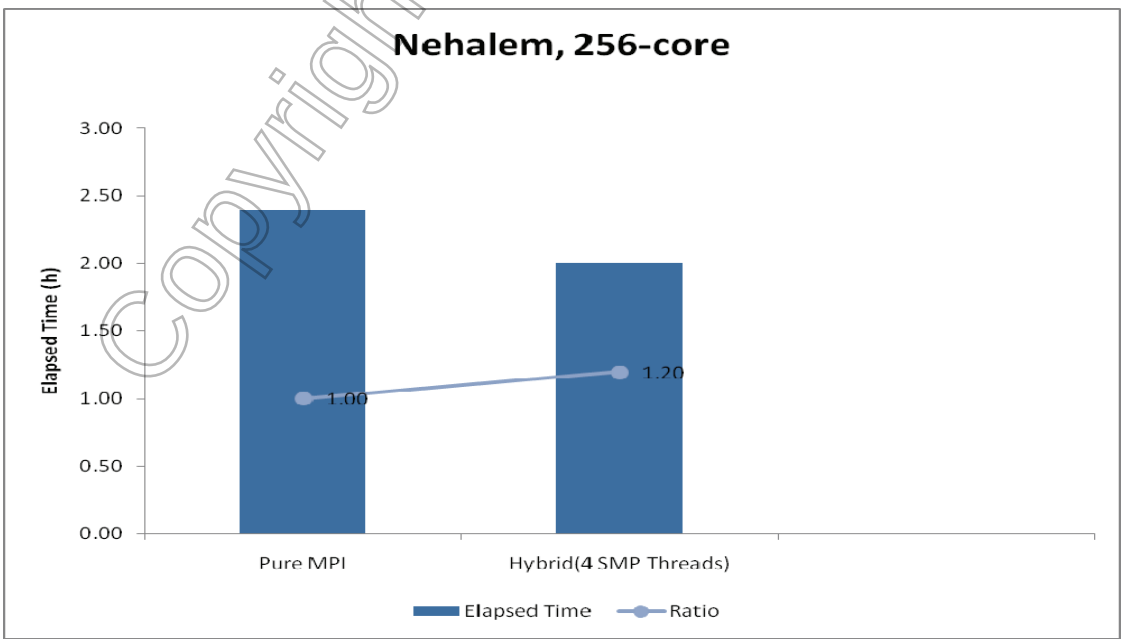

Figure 2: 256-core elapsed times of the pure MPI method and the hybrid method, and their comparison, on the Nehalem cluster

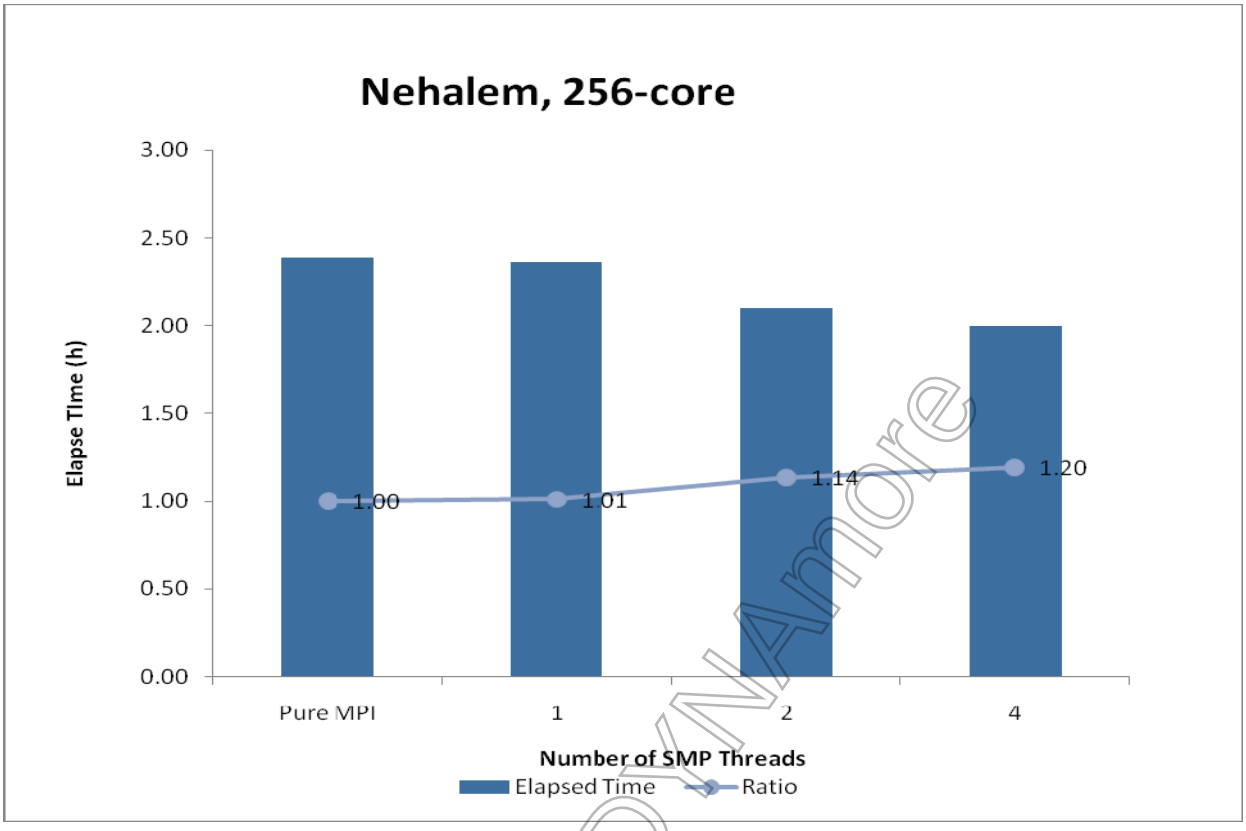

Figure 3: 256-core elapsed times of the pure MPI method and the hybrid method, and their comparisons, with varying numbers of SMP threads on the Nehalem cluster

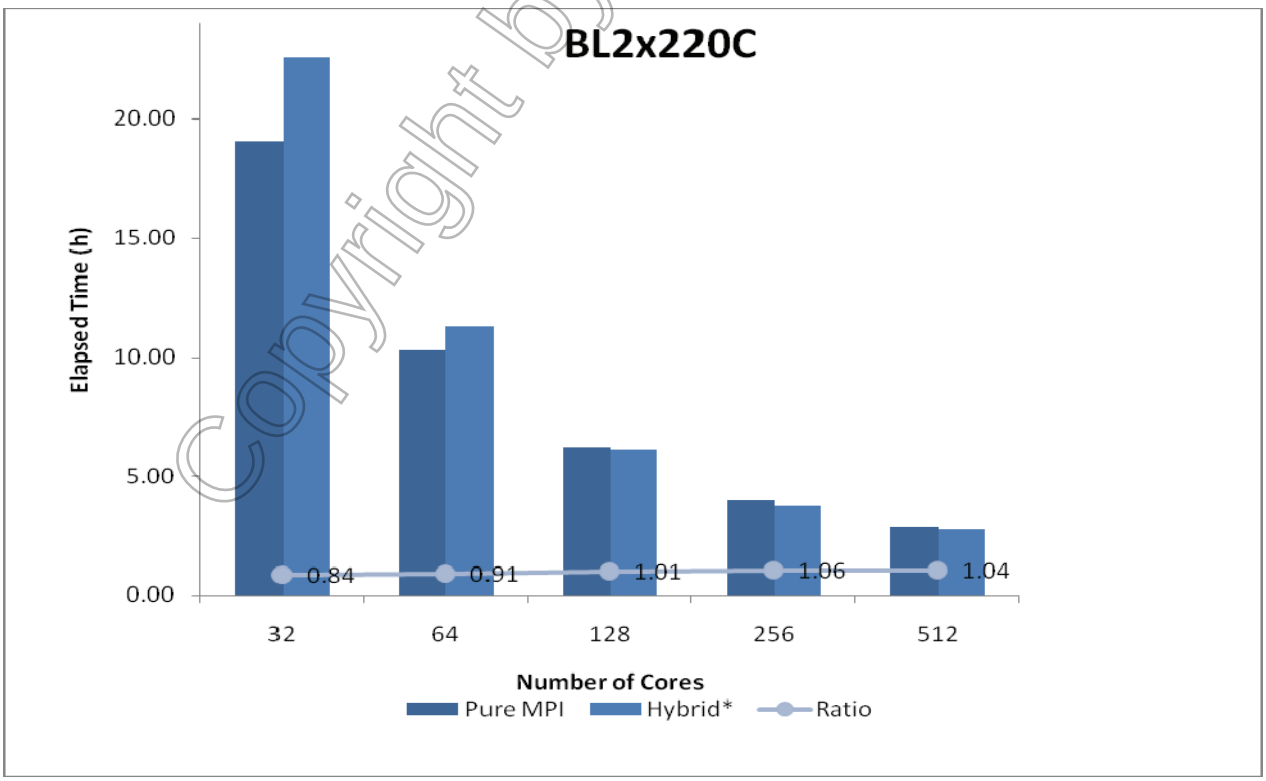

*Figure 4: Elapsed times of the pure MPI method and the hybrid method, and their comparison, with varying core counts from 32 to 512 on the HP BL2x220c cluster. * With 4 SMP threads*

3 Discussion

3.1 Communication and computing costs

As mentioned before, performance of an MPI method is determined by communication and computation costs; therefore, we will discuss factors affecting performance of the hybrid method along this division.

3.1.1 Communication cost

The reason that the hybrid method using 256 cores attains its best performance with 4 threads can be understood via communication cost. (The reason why the case with 8 SMP threads is excluded here will be discussed later in Section 3.1.2 next.) The communication cost is the cost of iteratively coordinating solutions on subdomains, which involves coupling contacts and coordinating nodal values on shared boundaries, among subdomains, via message passing. The amount of data for coupling contacts is independent of decomposition, but the amount of data for coordinating nodal values on the boundaries is of course dependent on the size of shared boundaries among subdomains, which is proportional to the number of subdomains. (Dividing a square, of side length *a*, into two equal areas, results in a shared boundary line of length *a*; in contrast, dividing the same square into four equal areas results in two shared boundary lines, horizontal and vertical, with a total length of 2*a*.) Therefore, barring contact coupling from consideration, the number of messages and their sizes increase proportionally to the number of subdomains.

For the hybrid method, we have the following relationship among core count, number of MPI ranks, and number of SMP threads:

$$\text{Core count} = \text{Number of MPI ranks} \times \text{Number of SMP threads}$$

It follows that given a core count, say 256, the larger the number of SMP threads, the smaller the number of messages and their sizes are, and hence the less the communication cost is, since the number of MPI ranks is equal to the number of subdomains. That is so can be seen evidently from Figure 5, in which the 256-core message histograms for the hybrid method with 1, 2, and 4 SMP threads are shown. As a corollary, the reasoning also explains why the hybrid method with higher SMP numbers outperform the pure MPI method, since the message pattern of latter is the same as that of the hybrid method with 1 SMP thread.

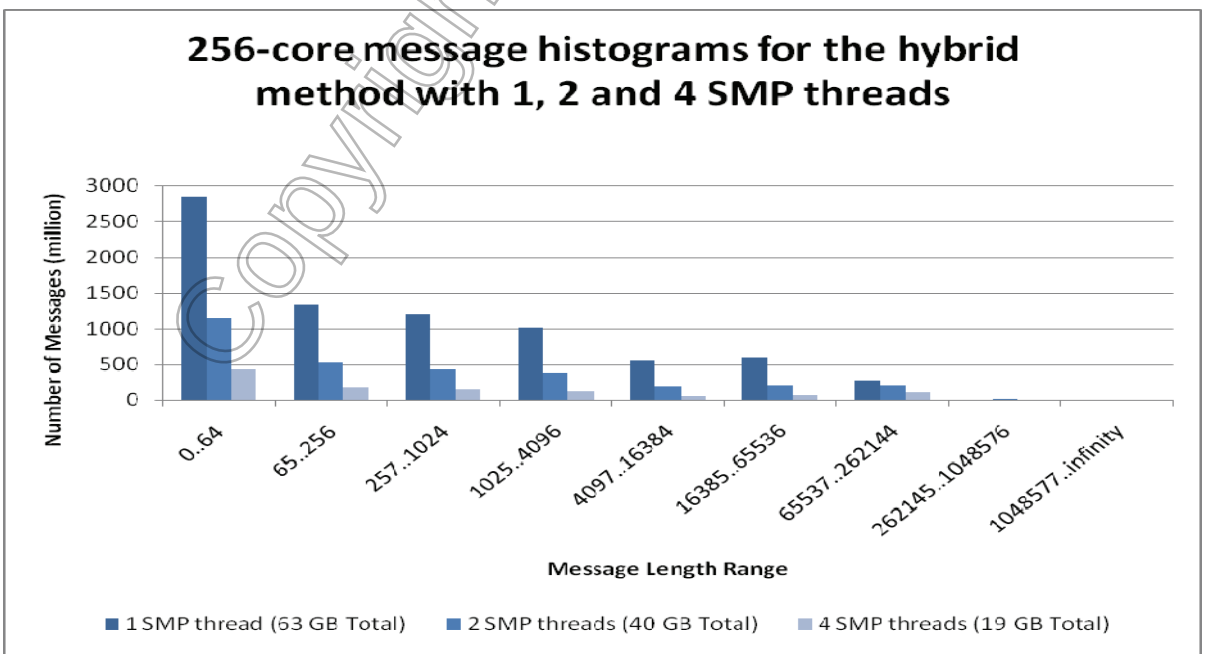

Figure 5: 256-core message histograms for the hybrid method with 1, 2 and 4 SMP threads

3.1.2 Computation cost—effects of CPU affinity and core placement

CPU affinity improves memory access speed for an SMP program by preserving more spatial locality and temporal locality in code and data access, which in turn allows the size-limited cache to work more efficiently. This benefit from CPU affinity can be seen from Figure 6, which shows 256-core elapsed times with and without CPU affinity, and their comparisons, for the pure hybrid method and the hybrid method with 1, 2, and 4 threads on the HP BL2x220c cluster. It shows that CPU affinity improves performance for the hybrid method with more than one SMP thread, though it has little effect on the pure MPI method and no effect on the hybrid method with 1 SMP thread.

Linux kernel does provide a mechanism to enforce CPU affinity, but it requires the software developer to activate it properly. HP-MPI currently provides CPU-affinity and core-placement capabilities to bind an MPI rank to a core in the processor from which the MPI rank is issued. Children threads, including SMP threads, can also be bound to a core in the same processor, but not to a different processor; additionally, core placement for SMP threads is by system default and cannot be explicitly controlled by users.

The inability to bind an SMP thread to a core in a different processor than the one the parent MPI rank resides will, when the number of SMP threads is greater than the number of cores per processor, force two different SMP threads to bind to one of the 4 cores in the processor; such binding of 2 executing programs to a single core, also known as oversubscribing, always greatly degrades an application's performance. The abnormally long elapsed time (24 plus hours) for the 256-core hybrid method with 8 SMP threads shown in Figure 7 is caused by oversubscribing.

Also, the inability for users to explicitly control core placement with SMP threads explains why the hybrid method with 2 SMP threads on Harpertown processors does not perform well as that on Nehalem processors (relative performance 1.04 in Figure 7 versus 1.14 in Figure 3), even though the former has a larger cache (3 MB versus 2 MB pre processor). Examining the core placement in the hybrid method with 2 SMP threads, we have found that the two threads are bound to two cores that do not share a cache in the Harpertown processor; in contrast, any two cores in a Nehalem processor always share a cache (see Figure 1). Sharing a cache among SMP threads provides spatial and temporal localities for both code and data accesses and helps performance, and therefore we have the observation.

3.1.3 Future CPU affinity project

It is reasonable to expect that if the ability to bind an SMP thread to a core in a different processor than the one the parent MPI rank resides is available, the hybrid method with 8 SMP may outperform that with 4 SMP threads, as the number of messages and their sizes will be further reduced with the former. It is also likely that for the hybrid method with 2 SMP threads may outperform that with 4 on a Harpertown cluster for smaller problems, provided that core placement is done correctly. Therefore, we are planning a project on CPU affinity to allow these two capacities.

3.2 Cache structures

The cache structure of the Nehalem processor with memory controller, as depicted in Figure 1, is with NUMA (Non-Uniform Memory Access) architecture [2]. It has been known that CPU affinity helps NUMA architecture more than non-NUMA in most applications. This can partially explain why the Nehalem processor outperforms the Harpertown processor for an MPI method.

3.3 Interconnect speed

As the table Section 2.1 shows, the Nehalem cluster has a faster interconnect than the BL2x220c cluster. Faster interconnect always helps performance of an MPI application such as the pure MPI method or the hybrid method. But with limited study, it is difficult for us to assess the effect of interconnect speed on the hybrid method vis-à-vis the pure MPI method.

3.4 Problem sizes

Figure 8 shows that the 256-core performance of the hybrid method is actually a little bit slower than the pure MPI method with the 0.8-million 3-vehicle-collision model. The reason is because the number of messages and their sizes are much smaller for the 3-vehicle-collision model than those for the car2car model, so that reduction in the number of messages and their sizes will not help performance as much.

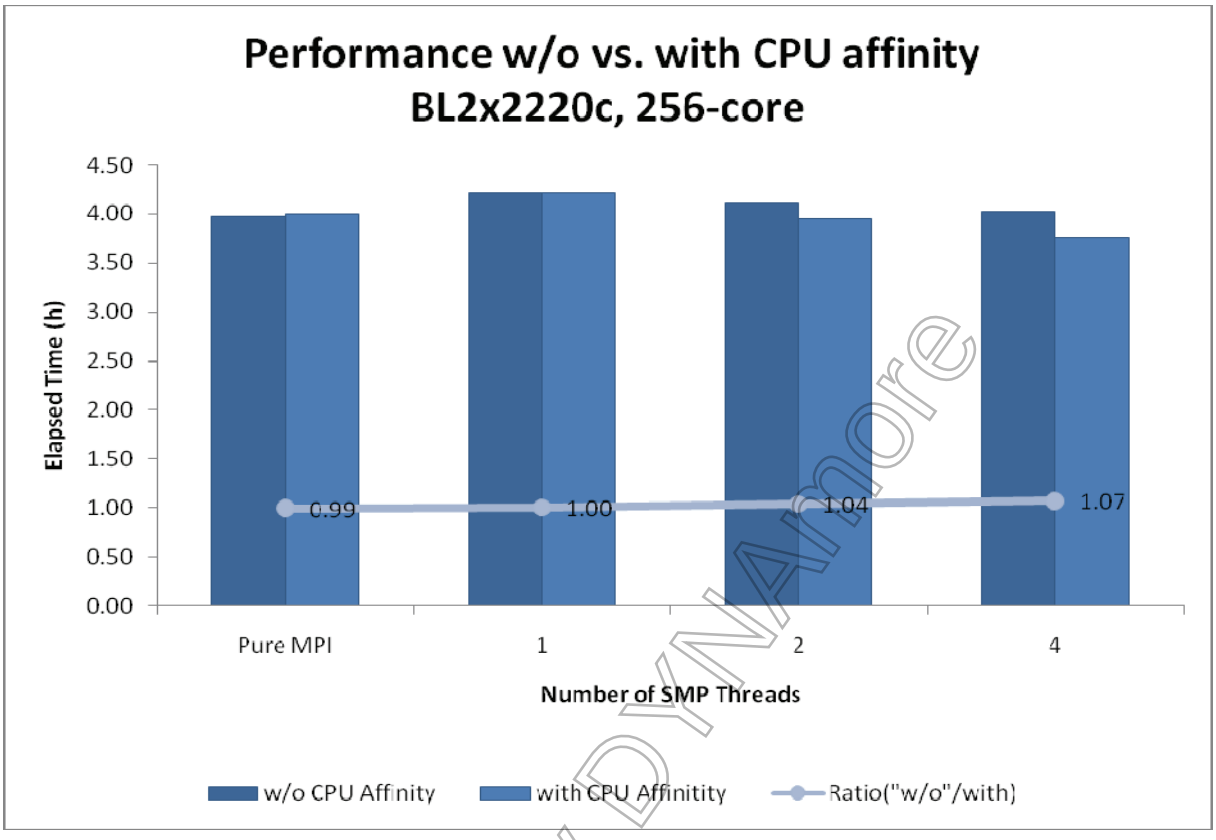

Figure 6: 256-core elapsed times with and without CPU affinity, and their comparison, for the pure MPI method and the hybrid method with 1, 2 and 4 threads, on the HP BL2x220c cluster

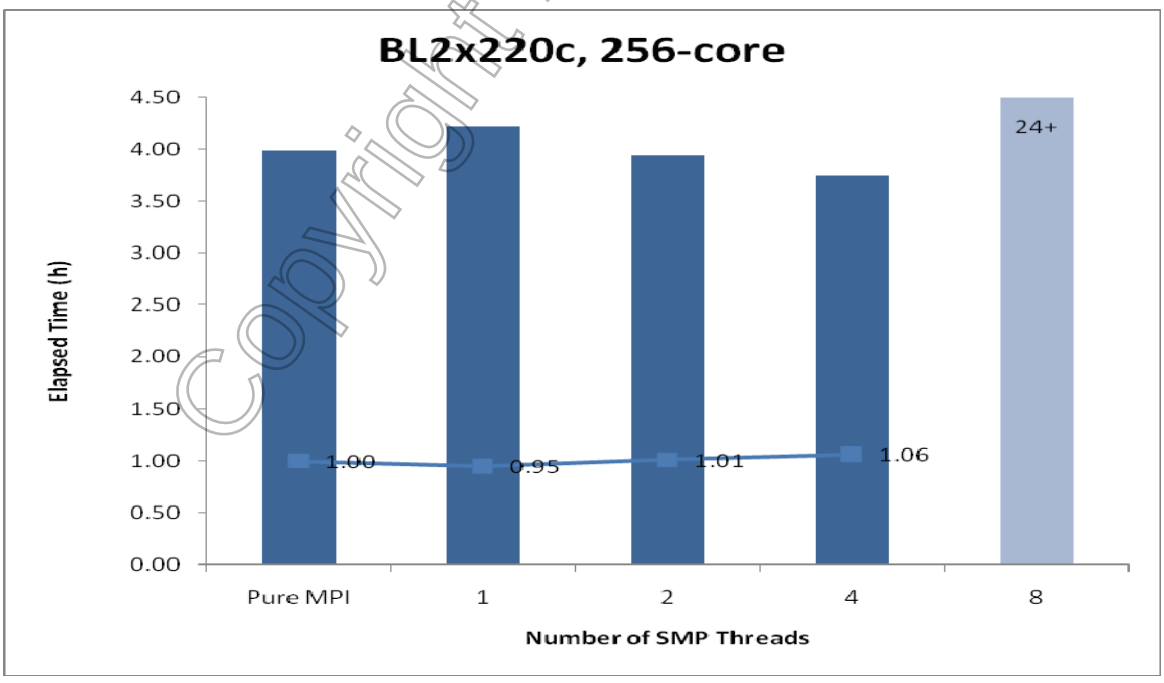

Figure 7: 256-core elapsed times for the pure MPI method and the hybrid methods with 1, 2, and 4 SMP threads, and the abnormally long elapsed time (24 plus hours) for the hybrid method with 8 SMP threads

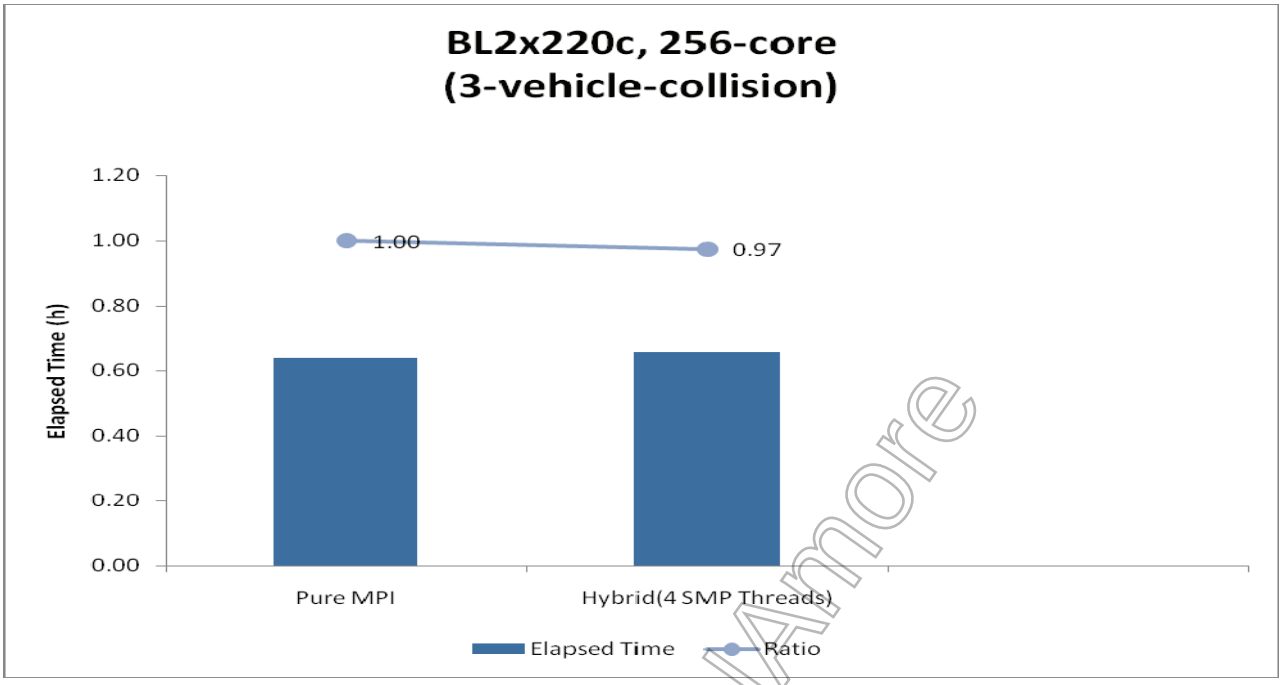

Figure 8: 256-core performances of the pure MPI method and the hybrid method, and their comparison, with the smaller 3-vehicle-collision model, on the HP BL2x220c cluster

3.5 Additional SMP parallelism

The major computation cost of SMP LS-DYNA is from element processing loops and contact algorithm, etc. Currently, SMP parallelism is only applied to element processing loops, and it already, as shown in this investigation, shows an encouraging speedup. We expect the breakeven point for the pure MPI and the hybrid MPI will be even lower when the contact algorithm is applied with more SMP parallelism.

4 Conclusion

As described in the *Introduction* section, there is a natural correspondence between the organization of a multicore cluster and the hybrid method: A multicore cluster comprises a number of processors, each of which in turn comprises a number of cores; in contrast, the hybrid method comprises a number of MPI ranks, each of which in turn issues a number of SMP threads.

First, in this investigation we have established that such a correspondence between hardware and software organizations can be exploited to gain performance advantage, provided that the problem size and the core count are big enough. For the 2.4-million-element car2car model, it is shown that the hybrid method gains 20 percent over the traditional, pure-MPI LS-DYNA method at 256 cores. Second, we have shown that CPU affinity with an appropriate core placement enhances the performance of the hybrid method in a multicore architecture. The best performance is achieved when the number of SMP threads is equal to the number of cores per processor. Finally, we have depicted that reduction in the number of messages and their sizes is the main reason for the performance gain in the hybrid method.

5 Acknowledgment

The result on the Nehalem cluster was obtained with the help from Nick Meng, of Intel Corporation.

6 Literature

[1] Hallquist, J.: "A Procedure for the Solution of Finite Deformation Contact-Impact Problems by the Finite Element Method," University of California, Lawrence Livermore National Laboratory, Rept. UCRL-5066, 1976.

[2] Patterson, D. and Hennessy, J.: "Computer Architecture: A Quantitative Approach", Second Ed., 1996, p.640.