International. Conference on Particle-Based Methods
Particles 2009
E. Oñate and D. R. J. Owen (Eds.)

PARTICLE METHODS ON MULTI-CORE AND GPGPU MACHINES

JOHN R. WILLIAMS[*], DAVID HOLMES[*] AND PETER TILKE[†]

[*] Massachusetts Institute of Technology, 77 Massachusetts Av., Cambridge, MA 02139, USA
e-mail: jrw@mit.edu, dholmes@mit.edu,web page: http://autoid.mit.edu

[†] Schlumberger-Doll Research Center, 1 Hampshire St, Cambridge, MA 02139-1578, USA
Email: tilke@slb.com - Web page: http://www.slb.com

Key words: SPH, Particle Method, Multi-Phase Fluids, Oil Reservoir Simulation, Multi-Core

Summary. This presents a strategy for programming particle simulation methods on multi-core shared memory machines.

1 INTRODUCTION

Multi-core machines can increase the speed at which applications execute. In particular, on board data access is more than 10,000 times faster than across machine access.[1] However, new parallel programming challenges are introduced because each core can address all of the main memory, leading to potential memory access conflicts, such as race conditions and deadlock. Some address by using a process on each core to ensure memory isolation. Here we show that a large class of computational physics problems, including "particle" simulations, can be decomposed into orthogonal compute tasks that can be executed safely in parallel on multi-core machines. A new task management algorithm called H-Dispatch [1,2] is developed that allows optimal use of memory by matching the task size to the available L3 cache, while optimizing the CPU usage by employing a "hungry" task pull strategy rather than the more common push strategy.

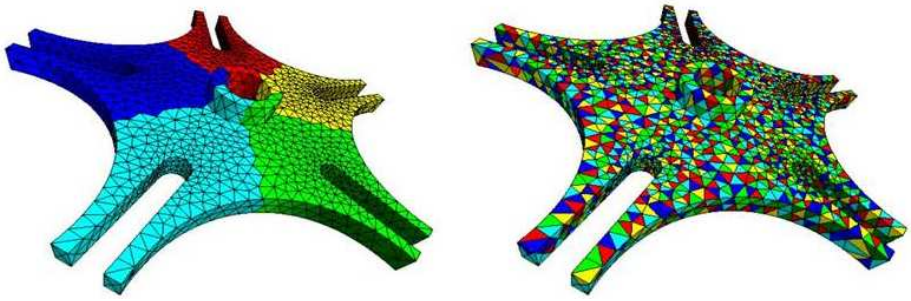

Figure 1 Optimized Domain Decomposition for (a) Cross Machine (b) for Multi-Core

The technique is demonstrated on SPH problems and it is shown that an optimal task size exists. If the task size is too small adding more cores can actually slow down execution

[1] RAM access is on the order of 100 cycles while cross-machine message latency is on the order of 1 millesecond or 1 million cycles, assuming a 1 GHz CPU with 1 cycle=1 nanosecond.

because the problem becomes dominated by messaging latency. However, when the task size is increased an optimal speedup is attained. It is shown that a near linear speedup is attained on a 24-core machine. It is noted that the algorithm is quite general and can be a applied to a wide class of computational tasks on heterogeneous architectures involving multi-core and GPGPU hardware. One solution that ensures memory isolation is to run a separate MPI [3] process on each core. The operating system then ensures an isolated virtual memory space for each core. Data is then shared across cores by sending MPI messages between the processes, typically requiring object marshalling and de-marshalling. The problem of memory conflicts is avoided but the cross-core communication overhead is significant, on the order of 10,000 machine cycles or about 100 times slower than direct memory access. We note that not all data needs to be communicated across cores, and in computational mechanics problems only "ghost region data" is shared. In essence, the MPI strategy turns each core into an information island, with information transfer being limited by the speed at which MPI messages can be delivered across the cores. While this is around 100 times faster than cross-machine MPI messages, this is still relatively slow compared to sharing main memory between the cores.

An alternative strategy, which allows memory sharing across cores, is to share a single operating system process across all cores, but use separate threads of execution on each core. In order to avoid memory contention "thread safety" must now be managed explicitly by the programmer. "Thread safe" programming can be complex even for the best programmers and the non-deterministic nature of running multiple threads makes detection of race conditions difficult. However, there are specific classes of problem where thread safety can be guaranteed. Indeed, this is the basis of OpenMP [4] and Cilk++ [5] that break "for loops" into parallel execution.

We show below that a large class of computational physics problems, including "particle" simulations, can decompose the problem into orthogonal compute task that share memory but execute "safely" in parallel on multi-core machines.

2 COMPUTATIONAL PHYSICS USING PARTICLE METHODS

In computational mechanics problems involving cross machine computing, we divide the unknowns into non-overlapping domains. However, there is spatial coupling of unknowns across domains so each domain must keep a copy of the "ghost region" belonging to its neighboring domains. Updating unknowns from time step N to step N+1 within a domain then proceeds in parallel. Only unknowns "belonging" to the domain are updated at this stage. Once the time step is complete the "ghost regions" are then updated by sending MPI messages from one machine to the other. We note that there needs to be a synchronization point that ensures all machines have finished the update on their own domains before messages updating the "ghost regions" are sent. Such synchronizations generally mean that computations on every machine must halt. Synchronization points are critical in coordinating parallel computing, as we shall see below.

If an MPI process is launched on each core then the computational process is almost identical to that described above for cross machine computation. The only difference is that the MPI messages can be optimized for in-machine communication. Typically, marshalling

and un-marshalling objects across process boundaries allows approximately 100,000 messages per second, so that each message takes roughly 10,000 machine cycles.

In the case of shared memory there are no "ghost regions" since any unknowns from neighboring domains may be read directly from memory. However, we must now devise a strategy for ensuring that writing does not corrupt data being read. One method is to provide two memory slots for each variable, one for v_n and one for v_{n+1}. Using this strategy only one synchronization point is necessary at the end of the time step.

Typically, the domain boundaries are minimized so that the "ghost regions" are as small as possible and message passing is minimized. However, using our shared memory strategy there is no penalty involved in dividing the problem up into smaller domains (Figure 1 (b)). Indeed, there is a benefit in doing this because we can now optimize the task size to match the underlying hardware, particularly the L3 cache size. On a 24 core machine we have shown that we get better efficiency in breaking the problem into hundreds of smaller domains. (see Figure 2 below). Figure 2 shows the performance of the H-Dispatch strategy compared to MPI and traditional scatter-gather.

When using modern languages such as Java and C#, we need also to minimize garbage collection because, during such collection phases, all cores must be stopped while the heap is re-mapped. We achieve this by essentially managing memory on each core explicitly. We allocate a block of memory for each core at the start of the computation and hold it until the end of the computation. This is achieved by allocating a master thread on each core. The thread "pulls" tasks from a single dispatcher queue as fast as it can. The task size is such that it can be mapped into the memory allocated. Furthermore, load balancing across cores is ensured no matter if one core runs slower than another. Indeed, even if a core "fails" by not responding within some given period of time, we can resubmit the task to the queue and it will execute on another core. The end of the time-step occurs when there are no more tasks in the queue.

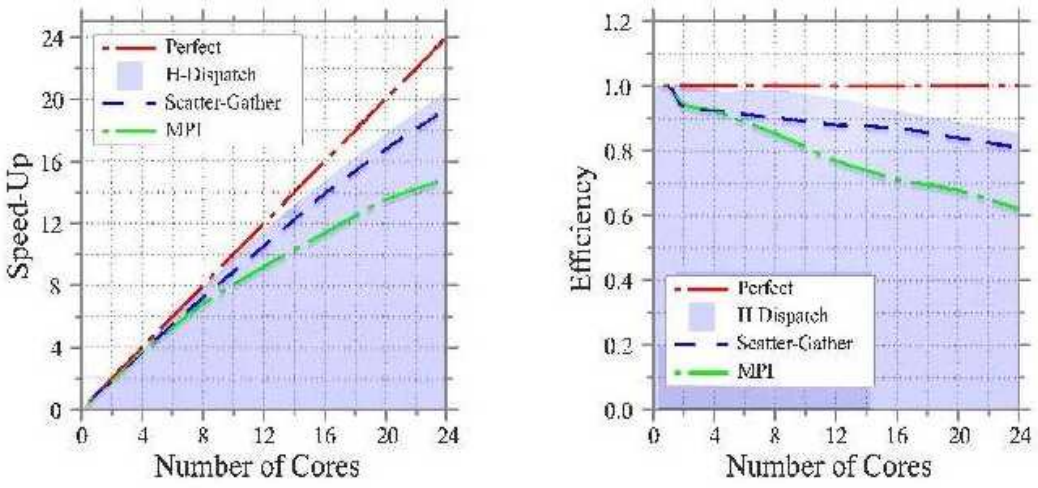

Figure 2 (a) Speed-up versus number of cores, (b) Efficiency versus number of cores

Figures 3 and 4 show typical results of multi-phase flow applied to pore scale analysis of an oil reservoir.

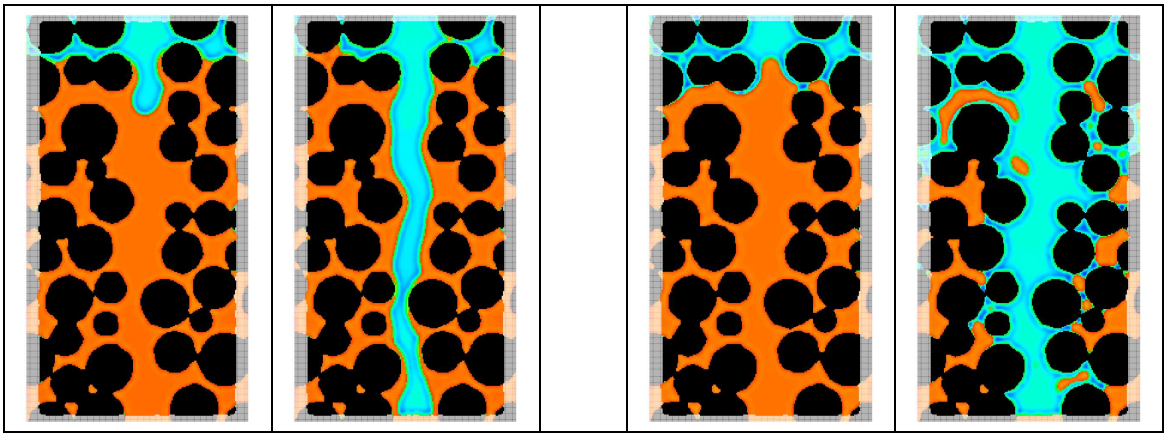

Figure 3 SPH multi-phase fluid simulation (a) water-rock non-wetting (b) water-rock wetting

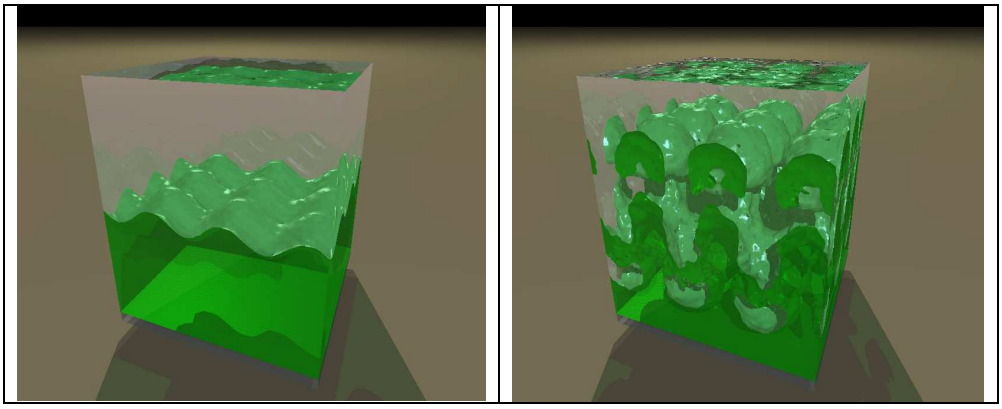

Figure 4 SPH multi-phase fluid simulation of Rayleigh-Taylor instability

REFERENCES

[1]

[3] Gropp W, Lusk E, Skjellum A. Using MPI: Portable Parallel Programming With the Message-Passing Interface. Cambridge: MIT Press, 1999.

[4] Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R. Parallel Programming in OpenMP. San Francisco: Morgan Kaufmann Publishers, 2001.

[5] Leiserson CE, Mirman IB. How to Survive the Multicore Software Revolution (or at Least Survive the Hype). Cambridge: Cilk Arts, 2008. URL www.cilk.com.

[6] Chrisanthakopoulos, G. and Singh, S., An asynchronous messaging library for C#. In Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages, San Diego, OOPSLA 2005, pp 89-97

[7] Monaghan JJ. Smoothed particle hydrodynamics. Annual Review of Astronomy and Astrophysics 1992;30:543–574. http://dx.doi.org/ 10.1146/annurev.aa.30.090192.002551.

[8] Monaghan JJ. Simulating free surface flows with SPH. Journal of Computational Physics 1994;110:399–406. http://dx.doi.org/ 10.1006/jcph.1994.1034.