Designing a Multicore and Multiprocessor Individual-Based Simulation Engine

This article describes the design of an individual-based simulation engine that can harness the full potential of modern general-purpose multicore and multiprocessor computers. This design aims to enable interactive simulations of highly dynamic multiagent systems in which entities can move, change, appear, disappear, and interact with one another and the user at any time.

Fabrice Harrouet
National Engineering School of Brest

In the Computer Sciences for Complex Systems Laboratory at the National Engineering School of Brest, we consider virtual reality, interactive simulation, and multiagent systems as tools to help in the design and tuning of models for complex systems. Individual-based simulation is appropriate when addressing problems in which limit conditions change according to unpredictable influences such as the user's actions. For example, it can facilitate the interactive adjustment of an ethological model producing patterns,[1] enable the injection of a substance if a peculiar situation arises during a medical experiment simulation,[2] or document a measurement process' impact on a molecular dynamic simulation.[3] Moreover, the interactive aspect is decisive when virtual reality is used for learning or training. When simulating an entire system with individual-based models, the higher the number of elements, the more accurate the results are. Thus, these simulations involve a huge amount of individual entities interacting with one another in a common environment, and so require substantial computing power to remain interactive.

Distribution on a computing grid is a common way to provide massive computing power. However, this approach mainly concerns batch processing and suits only problems that are static enough to be presubdivided into many correctly balanced subproblems. Such problems should require minimal internode communication compared to the amount of computation done at each node. Although Chan et al. present an asynchronous approach to minimize the cost of synchronization between nodes,[4] their approach applies only to convection-diffusion problems (homogeneous cells, statically subdivided, and with limited intercell relations). Even if some specific conflict resolution mechanisms allowing inaccuracies could lower the internode communication frequency, it would still be very difficult to generalize, and would hardly give satisfying speedups.[5]

Published by the IEEE Computer Society

On the other hand, even though processor manufacturers can no longer raise the frequency of general-purpose processors, they promise ever more CPU cores and larger cache memories for future processors. Moreover, when giant dies (the silicon piece that a processor is made of) can't be produced, an alternative is to wire several of them in the same package (the first Intel quad-core processors actually consisted of two dual-core dies) and use several such packages in a shared-memory architecture. Today, mass-production workstations consisting of 12 CPU cores enable the simultaneous execution of twice as many threads, thanks to simultaneous multithreading (SMT) technology. Such computers are still expensive for individuals but are affordable for professional use; expensive servers consisting of 40 cores, executing 80 threads, are also already available. For this reason, we drive our work toward a computing approach that's well-suited to the shared-memory multicore and multiprocessor computers that many researchers and engineers will likely have on their desk in the next few years.

This work aims at finding strategies to make an individual-based simulation engine gain as much benefit as possible from multicore and multiprocessor computers. In addition to performance speedup, which is our immediate concern, we consider scalability as decisive to the ability to use future many-core computers to their fullest potential.

Preliminary study

Because individual-based simulations rely on accessing a huge set of data many times, we've explored some common pitfalls with such accesses. Well-known parallelization tools such as OpenMP and Threading Building Blocks[6] encourage developers to design their applications in a data-parallel way. However, we don't use such high-level tools because, although they typically require the developer to subdivide every single iterative processing task into many parallel iterations, they don't offer sufficient control when addressing nonclassic numeric applications, as Massaioli et al. explain.[7] Moreover, when it comes to real applications, even the developers of Threading Building Blocks report rather low speedups in their own book: the best reported speedups are 1.29 or 1.5 when switching from one to four cores.[6]

The simulation of a huge amount of individual entities is a fine-grained problem. Therefore, it can be naturally spread across native threads. Operating systems expose hardware threads as distinct CPUs, whether they're provided by multiprocessors, multicores, or SMT technology. In this article, we accordingly refer to all of these hardware threads as "CPUs," and we use "threads" for software threads. The latter generally provide accurate control (number to launch, attachment to a specific CPU, and so forth), letting us experiment with memory accesses from multiple CPUs.

For this purpose, we use a computer containing two quad-core (Bi Xeon E5405) processors. Each processor has two dual-core dies in a single package, and each dual-core die has a 6-Mbyte Level 2 (L2) common cache and two distinct Level 1 (L1) data caches. The total memory of the L2 cache is thus 24 Mbytes, but each core can use no more than its own 6-Mbyte cache. This architecture enables experiments using either distinct or common caches and packages by placing computations on specific CPUs. To study the impact of such placements, a multithreaded program accesses an array to read or write its data many times so that we can observe how performance (the total number of read or write accesses per second) is affected by several criteria, such as:

- the memory access pattern,
- whether or not the amount of data accessed fits in the cache, and
- how threads are placed according to shared caches and packages.

This program uses GCC (the GNU Compiler Collection) on a Linux-32 operating system and is run many times to produce the results reported in Figure 1.

Effect of memory footprint

The top part of Figure 1 shows that, once caches are overflowed, we can't expect any significant gain from parallelization, because the limiting factor obviously lies in the global-memory transfer rate. The most noticeable result is that the block pattern lets

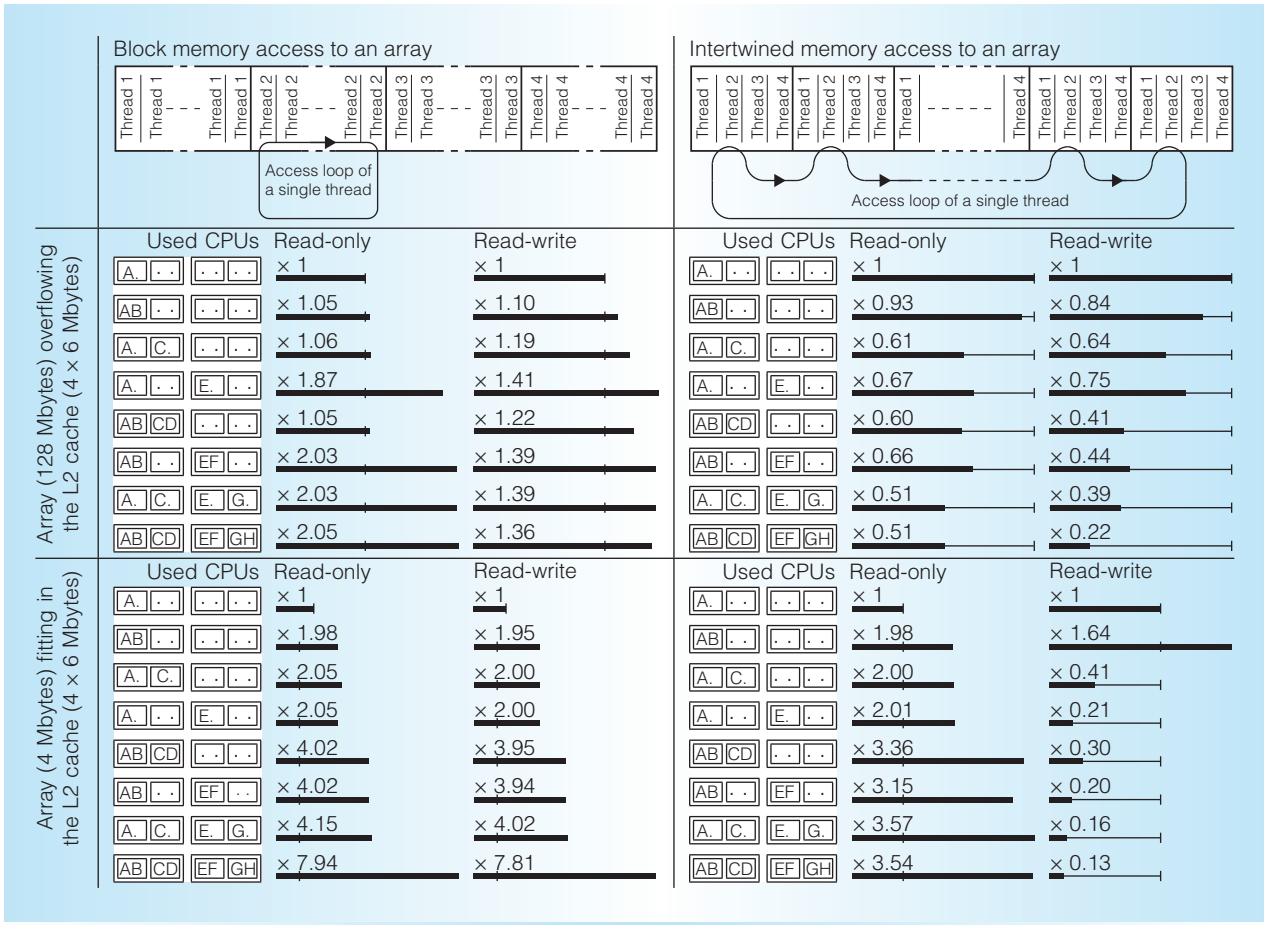

Figure 1. Speedups of various CPU uses and memory access patterns. Eight experiments show the performance of each multithreaded CPU combination compared to a single-threaded CPU combination (the higher the speed ratio, the better). In each CPU combination, letters indicate used CPUs, and dots represent unused CPUs. We group these CPUs by common cache and package, merging many equivalent combinations (AB ≡ CD, EF, or GH; ABEF ≡ CDEF, ABGH, or CDGH; and so on). Each CPU of a specific combination runs a thread that performs many loops for reading or changing an array's content (8-byte elements) according to one of the memory access patterns shown in the top part of the figure. (L2: Level 2.)

two processor packages work reasonably well simultaneously, especially when reading, whereas performance collapses dramatically with the intertwined pattern, even when reading. Conversely, the bottom part of Figure 1 shows that, as soon as the accessed data fit entirely in the available caches, the results rank from an ideal speedup (N times faster with N CPUs) to a catastrophic slowdown.

Effect of memory access pattern

When a single CPU is the only one to access some cache lines, it works independently of the others (see the block pattern in the bottom part of Figure 1). The speedup is then ideal, whether the memory is accessed for reading or writing. The opposite situation occurs when many CPUs access data so close to one another that they access the same cache line (see the intertwined pattern in the bottom part of Figure 1). This is known as *false sharing* (details are available elsewhere[6]), and it produces the same effect as sharing exactly the same data: repeated write accesses produce recurrent cache line invalidations and updates.

Performance gets worse as the number of CPUs involved in this situation increases. The situation is slightly less negative when only two CPUs from the same cache are involved. In this peculiar case, the L2 cache is common, and no cache line invalidation

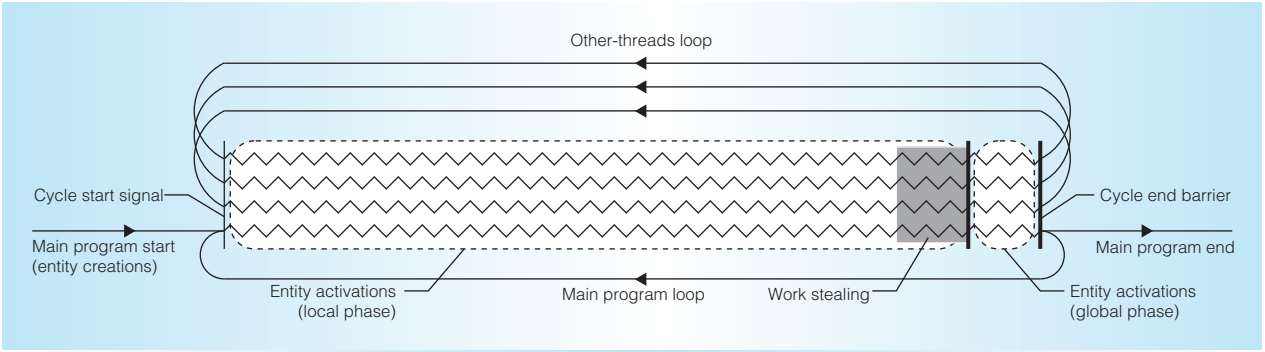

Figure 2. Outline of the simulation cycle in our simulation engine. The main thread is responsible for the loop in charge of launching every new simulation cycle. Time runs from left to right (except when looping back to a new cycle). Threads are synchronized on the two thick vertical lines.

occurs at this level; however, cache line invalidation still occurs between L1 caches and prevents the computer from obtaining an ideal speedup (1.64 instead of 2). Although the results for read-only intertwined accesses are not dramatically bad, they are still a bit disappointing. Indeed, when a cache line is accessed for reading, it is locked just in case another CPU would write into the same cache line. Even if we know that our program won't attempt any write access to this shared cache line, the cache coherency system isn't aware of that and must always prevent any potential inconsistency.

Superlinear speedup

Although some results give an ideal speedup, certain programs could benefit far more from parallelization if their memory footprint is slightly larger than the cache capacities. For example, reading and writing a 16-Mbyte array by contiguous blocks gives an amazing superlinear speedup (more than N times faster with N CPUs) of 30 when switching from one to eight CPUs. This is because a single CPU can't keep the entire 16-Mbyte array in its 6-Mbyte L2 cache; its data permanently goes back and forth between cache and main memory. When we involve the eight available CPUs in the same problem, each one needs to access only a 2-Mbyte subarray. Therefore, each 6-Mbyte cache can easily contain the necessary 4-Mbyte data accessed by its two CPUs; there is no longer any transfer with the main memory.

Thus, multicore and multiprocessor computers offer far more than computing units. Their cache architecture can lead to dramatic slowdowns when badly used, but they can bring significant speedups when correctly employed. Even if modern processors provide a Level 3 (L3) cache that is common to many cores on a single die, the lower-level caches remain distinct and still imply coherency penalties. Moreover, the same attention is necessary when using many such common-cache processors together. Consequently, thread assignment to CPUs and data placement have a central place in our simulation engine design.

Design choices

Here, we describe the simulation engine from a user's viewpoint, and we give technical details based on the experiments just discussed.

Main scheme

We've designed our simulation engine, a software library written in C99, to activate a set of autonomous entities. As the general shape of Figure 2 shows, the main application instantiates the initial entities (some user-defined data structures with their own activation function) and runs a loop to make the simulation engine activate them. Each entity's activation function detects its environment (other entities or global data structures) and, according to its own rules, modifies its data structure and eventually its environment.

The simulation engine runs exactly one thread per CPU, including the main program, and prevents that thread from moving

to another CPU. This strict binding inhibits context switches and gives fine control of data assignment to CPUs while avoiding cache trashing. The entire set of entities is split across these threads, so that each thread needs to activate only a subset of entities. Synchronization barriers occur at each cycle, so every entity is activated exactly once per cycle, ensuring a consistent clock in the simulation.

Synchronization and sharing

Activating entities on multiple threads leads to a drawback: some of them modify their own state while consulted by other entities. Furthermore, global data structures, such as a spatial index, could be used to store these entities and help find them. Using synchronization primitives or lock-free algorithms to prevent concurrent access inconsistencies would lead to a dramatic slowdown.[8,9] Indeed, such primitives and algorithms rely on many write accesses to a specific variable protecting critical operations,[10] and our experiments clearly show that cache line coherency penalties are damaging.

To bypass this situation, we split the simulation cycle into two distinct phases, separated by a synchronization barrier (see Figure 2):

- The *local phase* lets each entity read all the data it needs in the environment without any synchronization.
- The *global phase* enables entities to appear, disappear, and modify their environment under the strict control of synchronization primitives.

The local phase contains neither synchronization nor write access overhead, so it represents an ideal situation for parallelism. An entity should compute most of its behavior in this phase. However, the result of this computation can't just remain in local variables and is likely to induce changes in the entity itself, as well as in the environment. Because an entity can't change its current state during the local phase, its own data structure comprises two distinct states, *current* and *next*, which are swapped after each simulation cycle. This corresponds exactly to the classical synchronous simulation approach: each entity adjusts its next state according to the immutable current state of its environment—including itself. To avoid false sharing, we must fit these two states in different cache lines by using appropriate padding[11] or cache-line-aligned dynamic allocation. Then, each entity can modify its own next state without perturbing the read-only accesses performed by other entities during the same local phase (this situation is similar to the left part of Figure 1).

Conversely, the global phase lets entities make changes to anything beyond their own next states. This phase mainly concerns global modifications, such as updating indexation inside a global data structure, creating a new entity, or an entity destroying itself. Strict synchronization is necessary because many threads can make these changes at any time during this phase. This strict synchronization is likely to sacrifice parallelization efficiency, but we have no choice; global changes must be made sooner or later. We must reduce global-phase use and ideally avoid it if some entities don't require any global changes. Thus, every entity's local phase tells the simulation engine whether or not it needs a global phase.

Beyond the activation strategy itself, synchronization and concurrent writing to shared data are implied in several services commonly used in individual-based simulations, such as memory allocation,[12] pseudo-random number generation, and data preparation for rendering. These services perform poorly when solicited by many threads, even if system libraries make some of them reentrant. Duplicating these services in every thread and making them accessible through thread-local storage dramatically improves performance because each thread can reuse its own data structures without interacting with others.

Dynamic load balancing

Even if write-access conflicts between the simulation engine's threads have been reduced, an important issue remains: every CPU must be fully used to harness the full potential of the computer so as to speed up our simulations. Unfortunately, threads reaching a synchronization barrier must stay

idle until the last one arrives. Ideally, every thread should be previously assigned an equivalent workload so that all threads reach the synchronization barrier at the same time. Some problems execute the same computation on every element for a predetermined neighborhood. These problems are adapted to a distributed approach,[4] and thus can consider precomputed load balancing. However, the highly dynamic individual-based simulations that we're interested in don't enable this prediction of neighborhood and require dynamic load balancing.

For this purpose, we consider the *work-stealing* method. Every thread has a *to-do* and a *done* entity subset, respectively populated and emptied at the beginning of each cycle, and swapped at the end. To activate an entity, a thread peeks from its own to-do subset and stores this entity in its own done subset afterward. When one thread's to-do subset becomes empty, it "steals" entities to the most loaded thread's to-do subset to help it end its work sooner. This tends to keep all the threads busy until no more entities need to be activated. Even if synchronization becomes mandatory on every to-do subset, it doesn't significantly alter cache behavior. Indeed, most of the time, a to-do subset is accessed only by its owner thread; it only needs to be present in one CPU's cache. Cache invalidation occurs only when a thread approaches the synchronization barrier (see the gray rectangle in Figure 2); only then do some threads begin accessing others' to-do subsets. Furthermore, these invalidations occur less often when a significant amount of entities are stolen at once rather than one by one.

Affinity-based assignment

Another benefit arises from giving each thread its own to-do and done subsets. The main part of the entities are likely to be activated by the same thread from one cycle to another, except those that are stolen. Thus, their data structures keep staying in the same caches. To go further with this idea, we consider applicative relationships between entities to assign them to the threads.

When an entity detects some other entities and interacts with them, it loads their data structures in the current CPU's cache—whatever the detection means and the details of these relations are. If the same thread activates these other entities, their respective data will already be present and up to date in the same cache. Moreover, when these entities are activated, they will probably use a similar neighborhood that will already be present in this cache ("most of my neighbors' neighbors are probably my neighbors"). Every entity knows the thread it is assigned to, so it can poll its neighbors to find which thread is represented the most within them. The entity is then stored in this latter thread's done subset to be activated by this thread during the next cycle. This simply leads to automatically and dynamically placing entities' data structures in caches according to their applicative affinity.

If one thread is represented the most in an entity's neighborhood, it is beneficial to choose this one. Conversely, if the poll is neutral, this choice is not so important. Because the entity subsets are actually queues, they're mainly traversed from one end to the other. Entities at the beginning of a to-do queue are sure to be activated by the owner thread, whereas entities at the other end are more likely to be stolen by other threads. Thus, when the poll indicates a strong tendency one way or another, the simulation engine places the entity at the beginning of the chosen thread's done subset; otherwise, the engine places it at the other end.

Our simulation engine supports many other functionalities, which we don't describe here because they go beyond the scope of the engine's internals (see http://www.enib.fr/~harrouet/transprog.html).

Testing the engine

Here, we give some experimental results concerning our simulation engine. Although performance is important, scalability is even more crucial because it determines whether future computers consisting of more CPUs will be able to run ever-larger simulations. All the experiments reported here involved measuring the cycle frequency of some simulations. We waited for this cycle frequency to be stable and then computed its temporal mean over several minutes. We report an arithmetic mean of such results for many equivalent simulations.

Neutral benchmark simulation

Potentially, any simulation scenario can have peculiarities that could drive entities to a state in which only a few specific aspects of the simulation engine are relevant. For example, inter-entity relationships create a performance bottleneck for gregarious entities, whereas they're irrelevant for entities ignoring one another. Therefore, our first benchmark aims to implement an average entity behavior that is representative of various simulations.

In this simple scenario, 50,000 entities take place in a limited 3D toroidal world from which none of them can escape. These entities all detect their respective neighbors within a given range (using a global spatial index, which isn't described in this article). They neither avoid nor pursue one another, but rather just go on their random walks as if no detection occurred. We adjust the detected neighborhood's average size by simply choosing the world's limits when launching the simulation. Although trivial, this example uses many features commonly involved in more relevant simulations: inter-entity detection, dynamic memory management, pseudorandom number generation, and indexing in a global data structure.

We spent hundreds of hours running this benchmark on eight-core computers similar to the one described earlier, under various operating systems (including both 32-bit and 64-bit versions of Linux, FreeBSD, NetBSD, Mac OS X, and Windows Vista) and with various compilers (including Gnu GCC, Microsoft cl.exe, and Intel C++ Compilers). Running all the benchmark's available CPU combinations on these platforms gave simulation cycle frequencies varying from 1 Hz to 135 Hz. Although raw performance differs from one operating system or compiler to another, the results show that our solution scales from one to eight CPUs in a similar way: the speedup curves have the same general shape regardless of the platform. Thus, it makes sense to mix these results and produce the mean curves reported in Figure 3. This property implies that our design choices aren't specific either to an operating system or a compiler; they're likely to always be relevant for this kind of general-purpose multicore or multiprocessor computer.

Figures 3a through 3c show that not using inter-entity affinity tends to make the speedups become asymptotic, although more CPUs would be necessary to confirm this observation. On the other hand, Figures 3d through 3f show that when using affinity, we can expect considerable scalability with the number of available CPUs. This difference obviously makes more sense when the average neighborhood is large. But, even when only a few neighbors are detected, this caution isn't damaging; it just provides an ideal speedup as if these computations were using totally disjointed data.

To explain the superlinear speedups obtained here, we focus on the 50-neighbor case running on eight threads, and we use simple reasoning regarding average values. In this scenario, the memory footprint of an entity's data structure is approximately 400 bytes, so the total footprint for the entire set of entities is 19 Mbytes. Each thread must activate approximately 6,250 (50,000/8) entities. When entities are arbitrarily assigned to threads, each detecting 50 neighbors, one thread must deal with the total footprint of 19 Mbytes ($6{,}250 \times 50 = 312{,}500$, which greatly exceeds 50,000). Conversely, when considering the affinity between entities to assign them to threads, we could ideally reach a situation in which entities only detect the ones that are activated by the same thread. In this case, each thread only needs to embrace the 2.4-Mbyte footprint of 6,250 entity data structures. Each shared 6-Mbyte L2 cache can deal with the necessary 4.8-Mbyte footprint, whereas the total 19-Mbyte footprint would imply cache exhaustion and memory transfers.

Realistic simulation

This second scenario involves more complicated behavior based on Reynolds' flocking rules.[13] Fish form schools, while avoiding one another as well as obstacles. To make the simulation less predictable, we make the fish accelerate to escape from sharks pursuing them. Thus, the size of the neighborhood detected by each entity varies dramatically throughout the simulation. The only autonomous entities are fish and sharks; obstacles are passive objects of the environment, and schools are just the visible

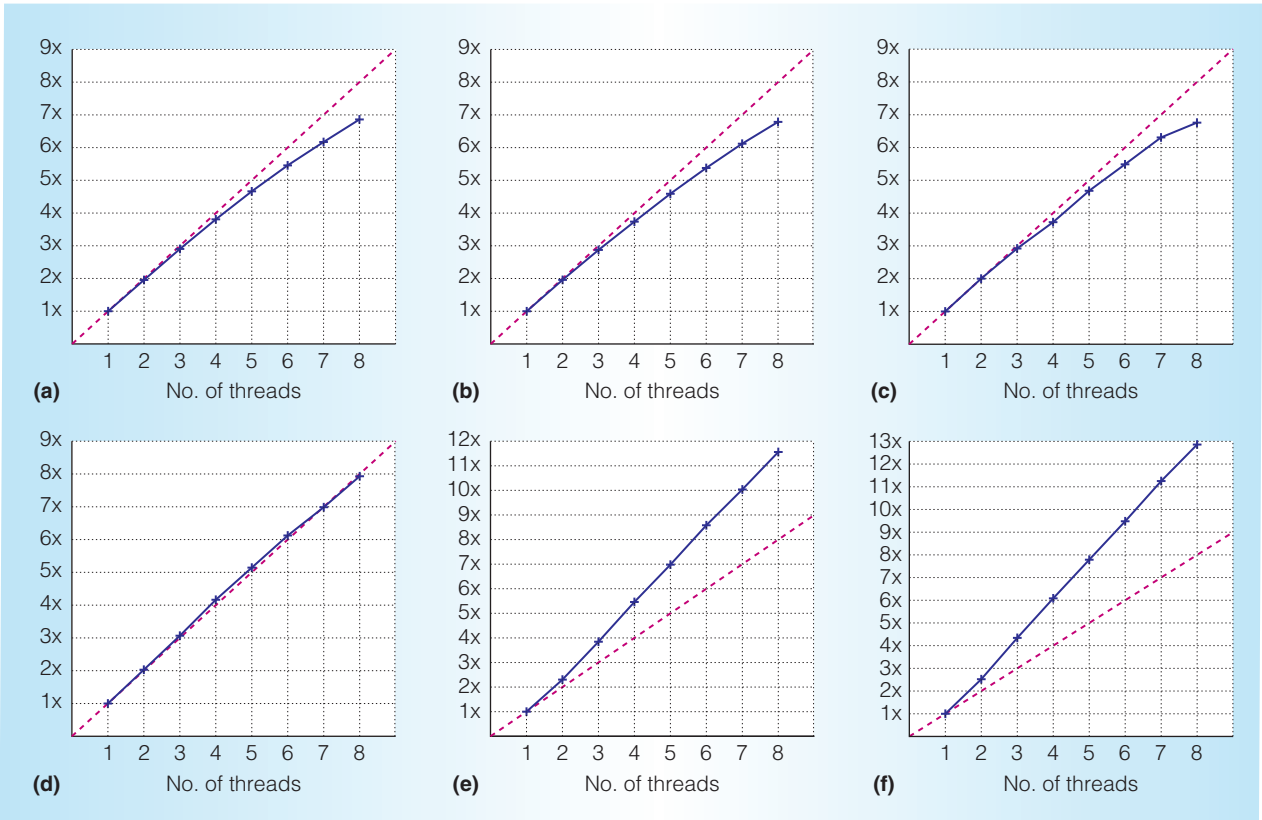

Figure 3. Neutral benchmark simulation for 50,000 entities detecting their neighbors while performing random walks on an 8-core computer: mean simulation cycle frequency speedup for two neighbors without affinity (a), 10 neighbors without affinity (b), 50 neighbors without affinity (c), two neighbors with affinity (d), 10 neighbors with affinity (e), and 50 neighbors with affinity (f). The entities' densities are statically adjusted to obtain different average neighborhood sizes. It's clear from the curves in (d) through (f) that entity assignment to threads according to their applicative affinities dramatically improves scalability.

consequence of the gregarious behavior of fish. Running this simulation on two different computers, using the Gnu GCC compiler on a Linux-64 operating system, gave the results reported in Figure 4.

When dealing with only 5,000 entities, so few entities end up in each thread that the cycle frequencies are very high. The synchronization barriers occur so often that we can't expect a great performance increase with many more CPUs.

With an approximate individual memory footprint of 700 bytes, we can use the same reasoning as at the end of the previous section to explain why 36,000 entities would give an optimal superlinear speedup. It is because of a threshold in cache usage: the 24-Mbyte total footprint fits in the four (Figure 4a) and two (Figure 4b) higher-level caches, respectively. The measurements confirm this expectation and show that the effect is more noticeable when quadrupling the cache capacity than when doubling it.

One hundred thousand entities make the frequency reach some values, below which the simulation is hardly interactive. Nevertheless, the previous cache threshold still maintains a superlinear speedup (Figure 4a) or at least a nearly ideal one (Figure 4b).

When we use 1 million entities, largely exceeding the cache capacities, the simulation is very slow but still responsive. As the top left of Figure 1 shows, the memory transfer rate doesn't scale with the number of CPUs, and the speedup is consequently limited.

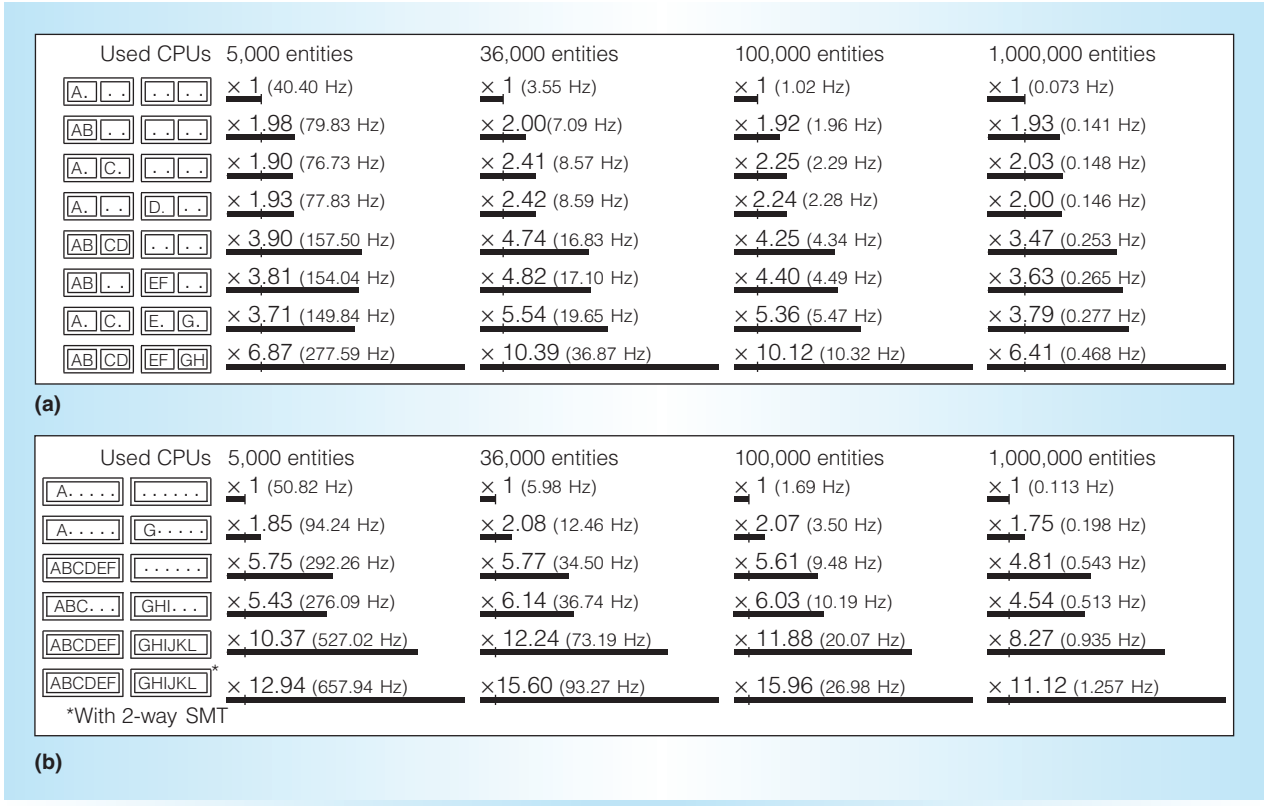

Figure 4. Cycle frequency speedups for different numbers of entities and various CPU combinations on two different computers for realistic simulation of fish forming schools while avoiding obstacles and sharks pursuing them: Bi Xeon X5472 (2 × 2 dual-core dies, each with a 6-Mbyte L2 cache) at 3.0 GHz with 800-MHz RAM (a) and Bi Xeon X5680 (two 6-core dies, each with a 12-Mbyte Level 3 cache) at 3.3 GHz with 1,333-MHz RAM (b). In each CPU combination, letters indicate used CPUs, and dots represent unused CPUs. These CPUs are grouped by common cache and package, and many equivalent combinations are merged. (SMT: simultaneous multithreading.)

Although the two-way SMT technology provided by the computer in Figure 4b isn't as efficient as adding cores (along with their cache capacities), it brings some substantial improvements and is worth using for the simulations.

Thus, as long as the cache capacities increase with the number of cores, we can expect to run ever larger individual-based interactive simulations. Because cache use is central to the design of our simulation engine, its performance doesn't rely on a shared cache's availability. The superlinear speedups observed here show that our engine can deal with multidie packages and multiprocessor computers as easily as with multicore dies.

While pursuing the design of an individual-based simulation engine, we studied common pitfalls regarding data access by multicore and multiprocessor computers. Because parallelization doesn't provide any physical performance gain with memory footprints largely exceeding cache capacity, we focus on smaller footprints. Indeed, below and around the caches' capacity, memory-access patterns dramatically impact performance, as well as scalability with the number of CPUs used.

Our simulation engine optimizes cache usage to employ modern general-purpose multicore and multiprocessor computers to their fullest potential. A two-phase adaptation of the classical synchronous simulation method significantly minimizes the need for synchronization and limits competition for memory during write accesses. A careful work-stealing ensures a correct load balance between the multiple CPUs to minimize their idle time without requiring too many

concurrent write accesses. Finally, we let the applicative affinities between the simulation's entities dynamically determine their assignment to specific CPUs, thus ensuring good data locality in the caches.

When testing the simulation engine, the results validated our design choices because the engine behaved as expected and performed well for interactive simulations with many autonomous entities. Moreover, it scaled well with the number of CPUs used—even with multiple distinct caches—and provided superlinear speedups when the memory footprint was appropriate. Therefore, we predict that running increasingly larger simulations will be possible when using computers comprising many more cores and processors.

Of course, we are aware of the limits of our results. Our last assumption concerning scalability is correct only if the cache capacities increase along with the number of CPUs. These results are also very specific to the cyclic and fine-grained character of this kind of application. Problems that are stable enough to be easily parallelized and distributed will not benefit greatly from our approach.

In the future, we'd like to experiment with a memory allocation strategy that considers modern computers' nonuniform memory access (NUMA) architecture to improve data locality even when caches are overflowed. Because computing power also dramatically increases in GPUs, committing straightforward computations to this kind of device while keeping less predictable computations on CPUs could be challenging. MICRO

References

1. L. Gaubert et al., ''A First Mathematical Model of Brood Sorting by Ants: Functional Self-Organization without Swarm-Intelligence,'' *Ecological Complexity,* vol. 4, no. 4, 2007, pp. 234-241.
2. G. Desmeulles et al., ''*In Virtuo* Experiments Based on the Multi-interaction System Framework: The *RéISCOP* Meta-Model,'' *Computer Modeling in Eng. & Sciences,* vol. 47, no. 3, 2009, pp. 299-329.
3. M. Combes et al., ''Multiscale Multiagent Architecture Validation by Virtual Instruments in Molecular Dynamics Experiments,'' *Proc. Int'l Conf. Computational Science* (ICCS 10), Elsevier, 2010, pp. 761-770.
4. M. Chau et al., ''MPI Implementation of Parallel Subdomain Methods for Linear and Nonlinear Convection–Diffusion Problems,'' *J. Parallel and Distributed Computing,* vol. 67, no. 5, 2007, pp. 581-591.
5. J. Liu et al., ''Distributed Individual-Based Simulation,'' *Proc. 15th Int'l Euro-Par Conf. Parallel Processing* (Euro-Par 09), LNCS 5704, Springer, 2009, pp. 590-601.
6. J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism,* O'Reilly Media, 2007.
7. F. Massaioli, F. Castiglione, and M. Bernaschi, ''OpenMP Parallelization of Agent-Based Models,'' *Parallel Computing,* vol. 31, nos. 10-12, 2005, pp. 1066-1081.
8. H. Gao and W.H. Hesselink, ''A General Lock-Free Algorithm Using Compare-and-Swap,'' *J. Information and Computation,* vol. 205, no. 2, 2007, pp. 225-241.
9. G. Cong and D.A. Bader, ''Designing Irregular Parallel Algorithms with Mutual Exclusion and Lock-Free Protocols,'' *J. Parallel and Distributed Computing,* vol. 66, no. 6, 2006, pp. 854-866.
10. M. Chynoweth and M.R. Lee, ''Implementing Scalable Atomic Locks for Multi-core Intel EM64T and IA32 Architectures,'' 8 Nov. 2009; http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures.
11. E. Raman, R. Hundt, and S. Mannarswamy, ''Structure Layout Optimization for Multithreaded Programs,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO 07), IEEE CS Press, 2007, pp. 271-282.
12. D. Tiwari et al., ''MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management,'' *Proc. IEEE Int'l Parallel & Distributed Processing Symp.* (IPDPS 10), IEEE CS Press, 2010, doi:10.1109/IPDPS.2010.5470428.
13. C.W. Reynolds, ''Flocks, Herds and Schools: A Distributed Behavioral Model,'' *ACM SIGGRAPH Computer Graphics,* vol. 21, no. 4, 1987, pp. 25-34.

Fabrice Harrouet is a lecturer at the National Engineering School of Brest, France, where he conducts research in the Computer Sciences for Complex Systems Laboratory and the European Center for Virtual Reality. His research focuses on interactive multiagent simulations, particularly parallel computing and 3D rendering. Harrouet has a PhD in computer science from the University of Western Brittany.

Direct questions and comments about this article to Fabrice Harrouet, Centre Européen de Réalité Virtuelle, Technopôle Brest-Iroise, CS 73862, 29238 Brest Cedex 3, France; harrouet@enib.fr.