

Designing a Multicore and Multiprocessor Individual-based Simulation Engine

Fabrice Harrouet (harrouet@enib.fr)

LISyC-ENIB-UEB

*Université Européenne de Bretagne, École Nationale d'Ingénieurs de Brest,
Laboratoire d'Informatique des Systèmes Complexes, Centre Européen de Réalité Virtuelle
Technopôle Brest-Iroise, CS 73862, 29238 Brest Cedex 3, France*

Abstract

This article describes the design of an individual-based simulation engine which aims to use modern general purpose multicore and multiprocessor computers to their fullest potential. It is dedicated to interactive simulations of highly dynamic multiagent systems where entities can move, change, appear, disappear and interact with each other and the user at any time. After studying some of the common memory access issues associated with this kind of computer, we list the main choices considered for the design of our engine. Experiments with various computers, operating systems and compilers yield very satisfying results in terms of performance and scalability relating to the number of CPUs used.

Keywords

Parallel simulation, multiagent systems, multicore, multiprocessor.

1 Introduction

In the *LISyC*, we consider virtual reality, interactive simulation and multiagent systems as tools helping in the design and the tuning of models involved in complex systems. We find individual-based simulation appropriate when addressing problems in which limit conditions change according to unpredictable influences such as the user's actions. As for example, it facilitates the interactive adjustment of an ethological model producing patterns [1], it makes the injection of a substance possible if a peculiar situation is observed during a medical experiment simulation [2] or it enables the study of

the measurement process impact in a molecular dynamic simulation [3]. Moreover, the interactive aspect is decisive when using virtual reality for learning or training. When simulating a whole system with individual-based models, the higher the number of elements is, the more accurate the results are. Thus, these simulations involve a huge amount of individual entities interacting with each other in a common environment and require a substantial computing power to remain interactive.

Distribution on computing grids is a common way to provide a massive computing power. However, this approach mainly concerns batch processing and suits only to problems static enough to be pre-subdivided in many subproblems correctly balanced; they should require a minimal inter-node communication compared to the amount of computation done on each node. Although an asynchronous approach to minimise the cost of synchronisation between nodes is presented in [4], this only applies to convection-diffusion problems (homogeneous cells, statically subdivided, well-defined and limited inter-cell relations). Even if some specific conflict resolution mechanisms allowing inaccuracies could lower the inter-node communication frequency, this is quite difficult to generalise and it hardly gives satisfying speedups [5].

On the other hand, despite 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 silicium piece a processor is made of) cannot be produced, an alternative consists in wiring several of them in the same package—the first Intel quad-core processors were actually made of two dual-core dies—and using several of such packages in a shared memory architecture.

Today, mass production workstations dotted with twelve CPU cores enable the simultaneous execution of twice as many threads thanks to the simultaneous multithreading technology. Such a computer is still expensive for an individual but is affordable for professional usage—very expensive servers dotted with forty cores executing eighty threads are also already available. For this reason, we drive our work towards a computing approach that suits well to the shared memory multicore and multiprocessor computers many researchers and engineers could reasonably expect to have on their desk in the next few years.

This work aims at finding strategies to make an individual-based simulation engine take 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 to their fullest potential the future computers dotted with many more cores. The preliminary study described in section 2 highlights the guidelines for the technical choices exposed in section 3. Section 4 reports the results obtained with our simulation engine in terms of performance and scalability.

2 Preliminary study

Because individual-based simulations rely on accessing many times a huge set of data, this section explores common pitfalls with such accesses. Although well-known parallelisation tools such as *OpenMP* or the *Threading Building Blocks* [6] encourage the developers to design their applications in a data-parallel way, we won't use them. This kind of high-level tools tends to subdivide every single iterative treatment in many parallel iterations but, as related in [7], they do not offer sufficient control when addressing non-“*classic*” numerical applications. Moreover, even the developers of the *Threading Building Blocks* report in their own book [6] quite low speedups when it comes to real applications: the best reported speedups are 1.29 or 1.5 when switching from one to four cores. 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, were they provided by multiprocessors, multicores, or simultaneous multithreading. In this paper, we accordingly name them *CPU* and we use *thread* for software threads. These latter generally provide a very accurate control (number to launch, attachment to a specific CPU, etc.) enabling us to experiment memory accesses from multiple CPUs.

For this purpose, we use a computer dotted with two quad-core processors (Bi-Xeon E5405). Each of them is made of two dual-core dies in a single package. Each dual-core has a six megabyte level-2 common cache and two distinct level-1 data caches. The total amount of level-2 cache is then twenty-four megabytes but each core or dual-core can only use up to its own six megabyte cache. This architecture enables experiments on the impact of using distinct or common caches and packages, by placing computations on specific CPUs. To do so, a multithreaded program accesses an array to read or write its data many times in order to observe how performances (the total number of read or write accesses per second) are affected, depending on several criteria such as:

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

This program uses the *Gnu gcc* compiler on a *Linux-32* operating system and is run many times to produce the average results reported in figure 1.

2.1 Effect of memory footprint

The top part of figure 1 shows that once caches are overflowed we cannot expect any significant gain from parallelisation since the limiting factor obviously lies in the global memory transfer rate. The most noticeable result is that the *block* pattern enables two processor packages to reasonably work simultaneously—especially when reading—whereas

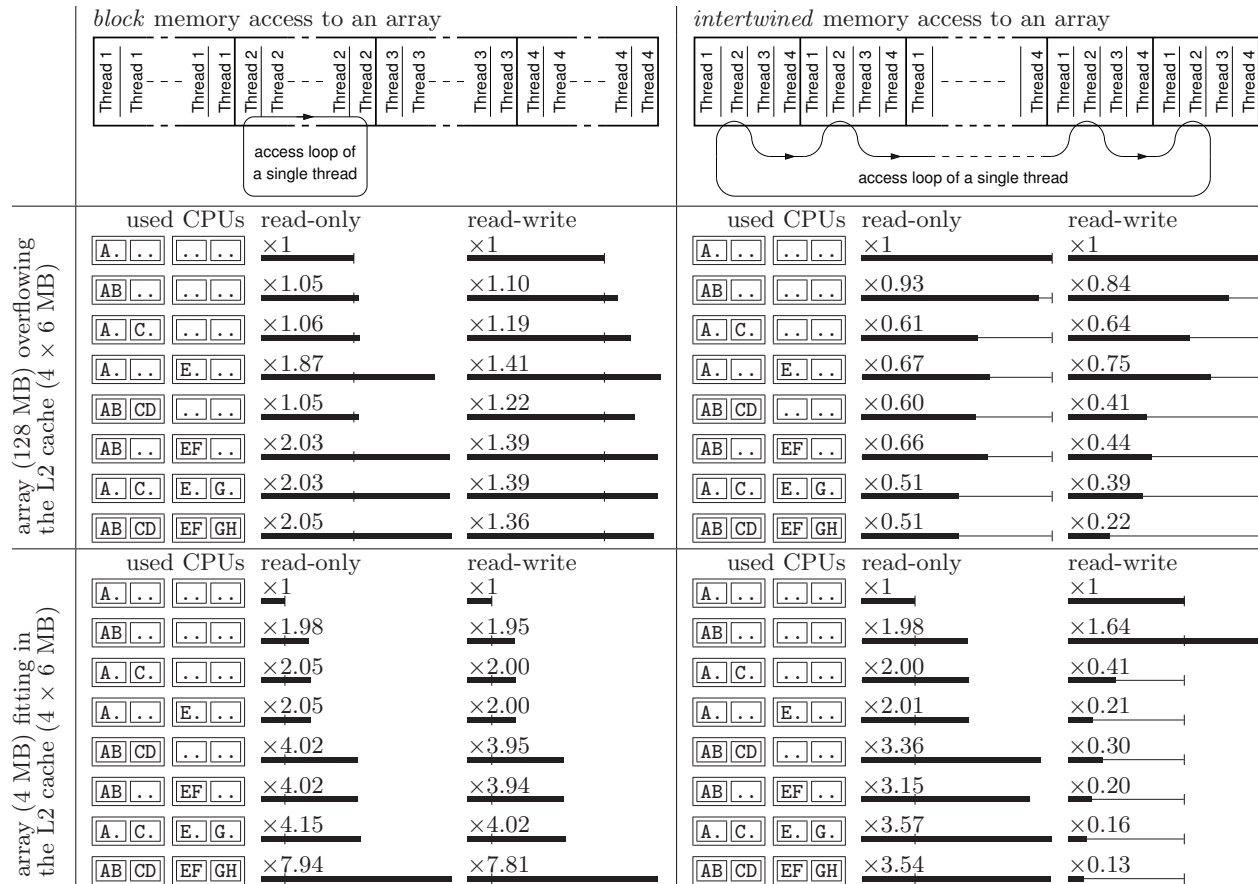


Figure 1: Speedups of various CPU usages and memory access patterns — Eight experiments showing the performance of each multithreaded CPU combination compared to a single threaded one (time spent ratio, higher is better). In each CPU combination, letters point out the used CPUs while dots relate unused ones. These CPUs are grouped by common cache and package; many equivalent combinations are merged ($AB \equiv CD$, EF or GH ; $ABEF \equiv CDEF$, $ABGH$ or $CDGH$ etc.). Each CPU of a specific combination runs a thread which performs many loops reading or changing the content of an array (8-byte elements) according to one of the above patterns.

performances collapse 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.

2.2 Effect of CPU/data placement

When a single CPU is the only one to access some cachelines, it works without worrying about 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 purpose. The opposite situation occurs when many CPUs access data so close to each other that they concern the same cacheline (see the *intertwined* pattern in the bottom part of figure 1). This is known as *false-sharing* (well explained in [6]) and produces the same effect as sharing exactly the same data: repeated write accesses produce recurrent cacheline invalidations and updates. Performances get worst 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 level-2 cache is common and cacheline invalidation occurs at this level; however it still happens between level-1 caches and prevents from obtaining an ideal speedup (1.64 instead of 2). The results are not so bad but are still a bit disappointing when performing read-only intertwined accesses. Indeed, when a cacheline is accessed for reading, it is locked just in case another CPU would write into the same cacheline. Even if we know that our program will not attempt any write access to this shared cacheline, the cache coherency system is not aware of that and must always prevent from any potential inconsistency.

2.3 Superlinear speedup

Although some results give an ideal speedup, one peculiar program could take much more benefits from parallelisation if its memory footprint is slightly larger than the cache capacities. As for example, reading and writing a sixteen megabyte array by contiguous blocks gives an amazing superlinear speedup (more than N times faster with N CPUs) of thirty when switching from one to eight CPUs. This is due to the fact that one single CPU cannot keep the whole sixteen megabyte array in its six megabyte level-2 cache; its data permanently go back and forth between cache and main memory. When involving the eight available CPUs on the same problem, each one needs to access only a two megabyte subarray. Therefore, each six megabyte cache can easily contain the necessary four megabyte data accessed by its two CPUs: there is no longer any transfer with the main memory.

This section shows that multicore and multiprocessor computers offer much more than computing units. Their cache architecture can lead to dramatic slowdowns when badly used but can also bring very significant speedups when correctly understood. Even if modern processors provide a level-3 cache which 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 still required when using together many of such common cache processors. Consequently, thread assignment to CPUs and data placement will

take a central place in the design of our simulation engine.

3 Design choices

This section firstly describes the simulation engine from the user point of view, and then gives technical details inspired from the experiments of section 2.

3.1 Main scheme

Our simulation engine, a software library written in *C99*, aims at activating a set of autonomous entities. As shown on the general shape of figure 2, 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 is responsible for detecting its environment (other entities or global data structures) and, according to its own rules, modifying its own data structure and eventually its environment (explained in section 3.2).

The simulation engine does not make any difference between a CPU and a thread: it runs exactly one thread per CPU—including the main program—and prevents it from moving to another CPU. This strict binding inhibits context switches and gives a fine control on data assignment to CPUs without worrying about cache trashing. The whole set of entities is split across these threads so that each one is in charge of activating only a subset (this splitting is discussed in sections 3.3 and 3.4). Synchronisation barriers occur at each cycle, thus every entity is activated exactly once per cycle; this ensures a consistent clock in the simulation.

3.2 Synchronisation and sharing

Activating the entities on multiple threads leads to a first drawback: some of them modify their own state while consulted by others. Furthermore, global data structures—such as a spatial index—could be used to store these entities and help finding them. Using synchronisation primitives or *lock-free* algo-

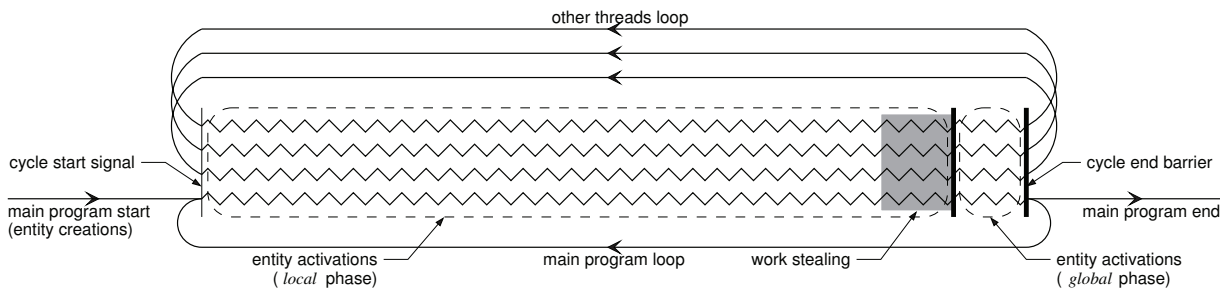


Figure 2: Outline of the simulation cycle — The main thread is responsible for the loop in charge of launching every new simulation cycle. Time runs from left to right (except when looping back to a new cycle). Threads are synchronised on the two thick vertical lines.

rithms [8, 9] to prevent concurrent access inconsistencies would lead to a dramatic slowdown. Indeed, they rely on many write accesses [10] to a specific variable protecting the sensible operation and section 2 clearly shows how cacheline coherency penalties are damaging.

In order to bypass this situation, the simulation cycle is split in two distinct phases separated by a synchronisation barrier (see figure 2):

- a *local* phase permitting each entity to consult everything it needs in the environment without any synchronisation,
- a *global* phase allowing entities to appear, disappear and modify their environment under the strict control of synchronisation primitives.

The *local* phase contains neither synchronisation nor write access overhead, so it represents an ideal situation for parallelism; an entity should compute the main part of its behaviour in this phase. However, the result of this computation cannot just stay in local variables and is likely to induce changes on the entity itself as well as on the environment. Since an entity cannot change its current state during the *local* phase, its own data structure is made of two distinct states, *current* and *next*, swapped after each simulation cycle. It 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*, these two

states must fit in different cachelines by using appropriate padding [11] or cacheline-aligned dynamic allocation. Then, each entity modifies its own *next* state without perturbing the read-only accesses performed by other entities during the same *local* phase (this situation is quite similar to the left part of figure 1).

Conversely, the *global* phase lets entities make changes to anything but their own state. It mainly concerns global modifications such as updating one's indexation inside a global data structure, creating a new entity or destroying itself. Strict synchronisation is required since these changes are made by many threads at any time within this phase. This is likely to misbehave regarding parallelisation efficiency but we have no choice; global changes have to be made soon or later. We ought to reduce the *global* phase usage and ideally avoid it if no global changes are required by some entities. Thus, every entity's *local* phase tells the simulation engine whether it needs a *global* phase or not.

Beyond the activation strategy itself, synchronisation 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 or data preparation for rendering. These services behave very badly 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 performances since

each thread can reuse its own data structures without interacting with others.

3.3 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. Unfortunately, threads reaching a synchronisation barrier have to stay idle until the last one arrives. Ideally, the best situation occurs when every thread is previously assigned an equivalent workload so that they reach the synchronisation barrier at the same time. Some problems execute the same computation on every element the neighbourhood of which is predetermined. They are adapted to a distributed approach [4] and thus can consider such a precomputed load-balancing. But the highly dynamic individual-based simulations we are interested in hardly enables this prediction and need a dynamic load-balancing.

We consider for this purpose the *work-stealing* method. Every thread has a *todo* and a *done* entity subset, respectively populated and empty at the beginning of each cycle, and swapped at the end. To activate an entity, a thread peeks from its own *todo* subset and stores in its own *done* subset afterwards. When one thread's *todo* subset gets empty, it "*steals*" entities to the most loaded thread's *todo* 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 synchronisation becomes mandatory on every *todo* subset, it does not alter so much cache behaviours. Indeed, most of the time a *todo* subset is accessed only by its owner thread; it just has to be present in one CPU's cache. Cache invalidation only happens when approaching the synchronisation barrier (see the grey rectangle in figure 2); only then, some threads begin to access others' *todo* subset. Furthermore, these invalidations occur less often when stealing a significant amount of entities at once instead of one by one.

3.4 Affinity-based assignment

Another benefit arises from giving each thread its own *todo* and *done* subsets: the main part of the entities are very likely to be activated by the same

thread from cycles to cycles—except those which are stolen. Thus, their data structures keep staying in the same caches. In order to go further within this idea, we take into account applicative relationships between entities to assign them to the threads.

When an entity detects some other ones 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 these other entities are activated by the same thread, their respective data will already be present and up to date in the same cache. Moreover, when these entities are activated, they probably use a similar neighbourhood which shall already be present in this cache ("*most of my neighbours' neighbours are probably my neighbours*"). Every entity knows the thread it is assigned to, so it can poll its neighbours to find which thread is the most represented within them. It is then stored in this latter's *done* subset in order to be activated by this thread at 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 mostly represented among an entity's neighbourhood, then it is really gainful to choose this one. Conversely, if the poll is quite neutral this choice is not so important. Since the entity subsets are actually queues, they are mainly traversed from one end to the other. Entities at the beginning of a *todo* queue are sure to be activated by the owner thread, while entities at the other end are more likely to be stolen by other threads. Thus, when the poll gives a strong tendency, the entity is placed at the beginning of the chosen thread's *done* subset, otherwise it is placed at the other end.

Many other functionalities are provided in our simulation engine but are not described in this article since they go beyond the scope of the engine's internals (soon to be made accessible from <http://www.enib.fr/~harrouet/transprog.html>).

4 Testing the engine

This section gives some experimental results concerning our simulation engine. Despite performances are an important factor, we consider scalability

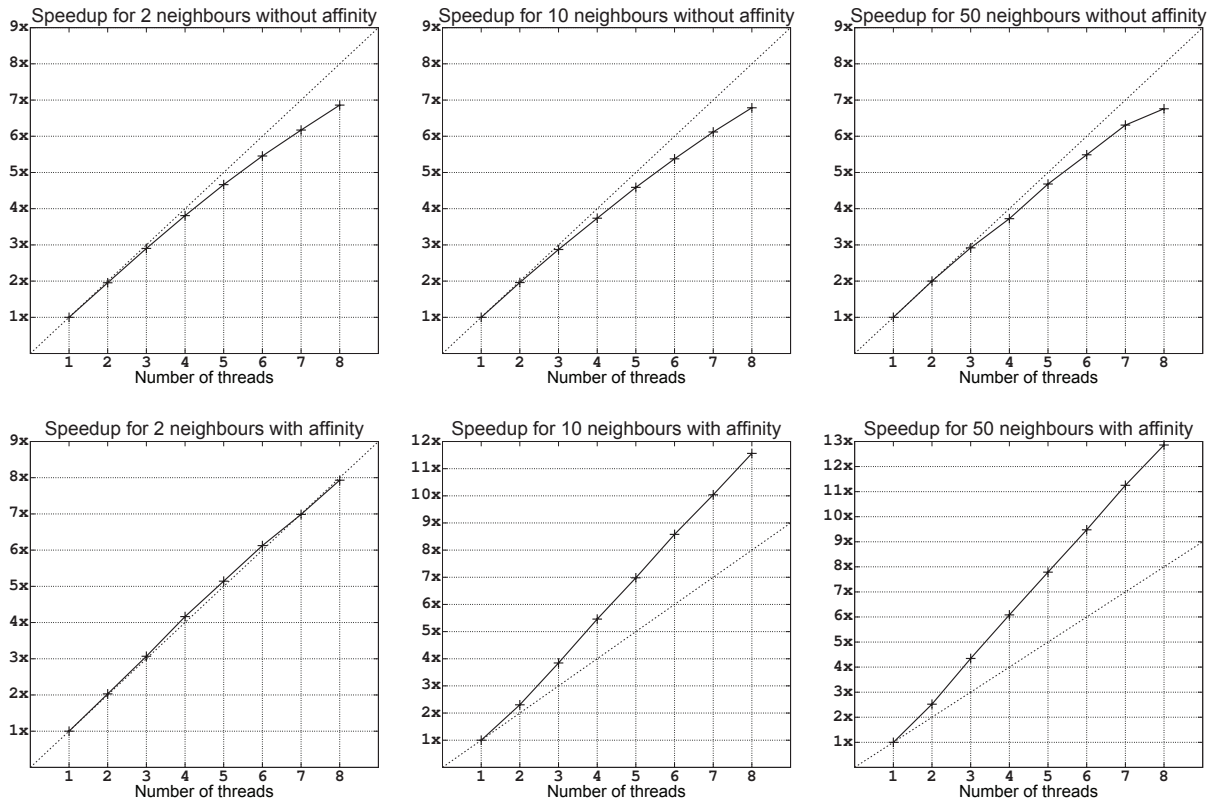


Figure 3: Neutral benchmark simulation — 50 000 entities detecting their neighbours while performing a random walk on an octo-core computer. The density of entities is statically adjusted to obtain different average neighbourhood sizes. These results are means of simulation cycle frequency speedups obtained with various platforms. The bottom row shows that entity assignment to threads according to their applicative affinities dramatically improves the scalability.

to be even more crucial; it determines whether future computers dotted with more CPUs would be able to run ever bigger simulations. All the experiments reported here consist in measuring the cycle frequency of some simulations. We wait for it to be stable and then compute its temporal mean during several minutes. We finally report an arithmetic mean of such results for many equivalent simulations.

4.1 Neutral benchmark simulation

Potentially, any simulation scenario has peculiarities that could drive entities in a situation just few specific aspects of the simulation engine are relevant.

For example, inter-entity relationships are a performance bottleneck for gregarious entities, whereas it is irrelevant for entities ignoring each other. Then, our first benchmark aims to implement an average entity behaviour representative of various simulations.

In this quite simple scenario, fifty thousand entities take place in a limited three-dimensional toroidal world none of them can escape from. They all detect their respective neighbours within a given range—using a global spatial index not described in this article. They neither avoid nor pursue each other but just go on their random walk as if no detection occurred. We adjust the average size of the detected neighbourhood by simply choosing

the limits of the world when launching the simulation. Although trivial, this example uses many features commonly involved in more relevant simulations: inter-entity detection, dynamic memory management, pseudo-random number generation and indexing in a global data structure.

We spent hundreds of hours to run this benchmark on octo-core computers similar to the one described in section 2, under various operating systems (both 32bit and 64bit versions of *Linux*, *FreeBSD*, *NetBSD*, *MacOsX* and *Windows Vista*) and with various compilers (*Gnu gcc*, *Microsoft cl.exe*, *Intel icc* and *Intel icl.exe*). Running all the available CPU combinations of our benchmark on these platforms gives some simulation cycle frequencies varying from 1Hz to 135Hz. Although raw performances differ from an operating system or a 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 whatever the platform is. Thus, it makes sense to mix them and produce the mean curves reported in figure 3. This property implies that our design choices are not specific neither to an operating system nor to a compiler; they are likely to always be relevant for this kind of general purpose multicore and multiprocessor computer.

The top row of this figure shows that not using inter-entity affinity tends to make the speedups become asymptotic—more CPUs would be necessary to confirm this latter feeling. On the other hand, the bottom row shows that when using it (as described in section 3.4), we can expect a great scalability with the number of available CPUs. This difference is obviously more sensible when the average neighbourhood is large. But, even when only few neighbours are detected, this caution is not 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 fifty neighbour case running on eight threads, and perform a simple reasoning on average values. In this scenario, the memory footprint of an entity's data structure is approximately four hundred bytes; the total footprint for the whole set of entities is then nineteen megabytes. Each thread has to activate approximately 6 250 (50 000/8) en-

tities. When entities are arbitrarily assigned to threads, and each of them detects fifty neighbours, one thread has to deal with the total footprint of nineteen megabytes ($6\,250 \times 50 = 312\,500$ largely exceeds 50 000). Conversely, when taking into account the affinity between entities to assign them to the threads, we could ideally reach a situation where entities only detect the ones which are activated by the same thread. In this case, each thread has only to embrace the 2.4 megabyte footprint of 6 250 entity data structures. Each shared six megabyte level-2 cache can deal with the necessary 4.8 megabyte footprint while the total nineteen megabyte footprint would imply cache exhaustion and memory transfers as explained in section 2.3.

4.2 Realistic simulation

This second scenario involves more complicated behaviours based on Reynolds' flocking rules [13]. Fish form schools while avoiding each other as well as obstacles. In order to make the simulation less predictable, they accelerate to escape from sharks pursuing them. Thus, the size of the neighbourhood 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 consequence of the gregarious behaviour of fish. Running this simulation on two different computers, using the *Gnu gcc* compiler on a *Linux-64* operating system, gives the results reported in figure 4.

When dealing with only five thousand entities, there are so few of them in each thread that the cycle frequencies are very high. The synchronisation barriers occur so often that we cannot expect a great performance increase with many more CPUs.

With an approximate individual memory footprint of seven hundred bytes, the same reasoning as at the end of section 4.1 explains that thirty-six thousand entities would give an optimal superlinear speedup. This is due to a threshold in cache usage: the twenty-four megabyte total footprint fits in the four (top computer) and two (bottom computer) higher level caches. The measurements confirm this expectation and show that the effect is more no-

Bi-Xeon X5472 (2×2 dual-cores, 6MB level-2 cache each) at 3.0GHz with 800MHz RAM				
used CPUs	5 000 entities	36 000 entities	100 000 entities	1 000 000 entities
<u>A.</u> <u>..</u> <u>..</u> <u>..</u>	×1 (40.40 Hz)	×1 (3.55 Hz)	×1 (1.02 Hz)	×1 (0.073 Hz)
<u>AB</u> <u>..</u> <u>..</u> <u>..</u>	×1.98 (79.83 Hz)	×2.00 (7.09 Hz)	×1.92 (1.96 Hz)	×1.93 (0.141 Hz)
<u>A.</u> <u>C.</u> <u>..</u> <u>..</u>	×1.90 (76.73 Hz)	×2.41 (8.57 Hz)	×2.25 (2.29 Hz)	×2.03 (0.148 Hz)
<u>A.</u> <u>..</u> <u>D.</u> <u>..</u>	×1.93 (77.83 Hz)	×2.42 (8.59 Hz)	×2.24 (2.28 Hz)	×2.00 (0.146 Hz)
<u>AB</u> <u>CD</u> <u>..</u> <u>..</u>	×3.90 (157.50 Hz)	×4.74 (16.83 Hz)	×4.25 (4.34 Hz)	×3.47 (0.253 Hz)
<u>AB</u> <u>..</u> <u>EF</u> <u>..</u>	×3.81 (154.04 Hz)	×4.82 (17.10 Hz)	×4.40 (4.49 Hz)	×3.63 (0.265 Hz)
<u>A.</u> <u>C.</u> <u>E.</u> <u>G.</u>	×3.71 (149.84 Hz)	×5.54 (19.65 Hz)	×5.36 (5.47 Hz)	×3.79 (0.277 Hz)
<u>AB</u> <u>CD</u> <u>EF</u> <u>GH</u>	×6.87 (277.59 Hz)	×10.39 (36.87 Hz)	×10.12 (10.32 Hz)	×6.41 (0.468 Hz)

Bi-Xeon X5680 (2 hexa-cores with 2-way SMT, 12MB level-3 cache each) at 3.33GHz with 1333MHz RAM				
used CPUs	5 000 entities	36 000 entities	100 000 entities	1 000 000 entities
<u>A.</u> <u>....</u> <u>....</u> <u>....</u>	×1 (50.82 Hz)	×1 (5.98 Hz)	×1 (1.69 Hz)	×1 (0.113 Hz)
<u>A.</u> <u>....</u> <u>G.</u> <u>....</u>	×1.85 (94.24 Hz)	×2.08 (12.46 Hz)	×2.07 (3.50 Hz)	×1.75 (0.198 Hz)
<u>ABCDEF</u> <u>....</u> <u>....</u> <u>....</u>	×5.75 (292.26 Hz)	×5.77 (34.50 Hz)	×5.61 (9.48 Hz)	×4.81 (0.543 Hz)
<u>ABC</u> <u>...</u> <u>GHI</u> <u>...</u>	×5.43 (276.09 Hz)	×6.14 (36.74 Hz)	×6.03 (10.19 Hz)	×4.54 (0.513 Hz)
<u>ABCDEF</u> <u>GHIJKL</u> <u>....</u> <u>....</u>	×10.37 (527.02 Hz)	×12.24 (73.19 Hz)	×11.88 (20.07 Hz)	×8.27 (0.935 Hz)
<u>ABCDEF</u> <u>GHIJKL</u> <u>....</u> <u>....</u> using 2-way SMT	×12.94 (657.94 Hz)	×15.60 (93.27 Hz)	×15.96 (26.98 Hz)	×11.12 (1.257 Hz)

Figure 4: Realistic simulation — Fish forming schools while avoiding obstacles and sharks pursuing them. These simulation cycle frequency speedups are obtained for different numbers of entities and various CPU combinations on two different computers. In each CPU combination, letters point out the used CPUs while dots relate unused ones. These CPUs are grouped by common cache and package; many equivalent combinations are merged.

ticeable 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 (top computer) or at least a nearly ideal one (bottom computer).

When largely exceeding the cache capacities with one million entities, the simulation is very slow but still responsive. As seen in the top-left part of figure 1, the memory transfer rate does not scale with the number of CPUs; the speedup is consequently limited.

Although the *2-way SMT* technology provided by the bottom computer is not as efficient as adding cores (along with their cache capacities), it brings

some substantial improvements and it is worth using it for the simulations.

This section shows that, as long as the cache capacities increase with the number of cores, we can expect to run ever bigger individual-based interactive simulations. Because the cache usage is central to the design of our simulation engine, its performance does not rely on the availability of a shared cache; the superlinear speedups observed here show that it can deal with multidiel packages and multiprocessor computers as easily as with multicore dies.

5 Conclusion

Through the purpose of designing an individual-based simulation engine, we studied common pitfalls about data access by multicore and multiprocessor computers. Since parallelisation does not provide

any sensible performance gain with memory footprints largely exceeding the caches' capacity, we focus on smaller footprints. Indeed, below and around the caches' capacity, memory access patterns exhibit a decisive impact on performance and scalability regarding the number of CPUs used.

The design of our simulation engine tends to optimise the cache usage in order to use modern general purpose multicore and multiprocessor computers to their fullest potential. A two-phase adaptation of the classical synchronous simulation method significantly minimises the needs for synchronisation and limits competition for memory during write accesses. A careful *work-stealing* ensures a correct load balance between the multiple CPUs, in order to minimise their idle time, without implicating too much concurrent write access. And finally, we let the applicative affinities between the entities of the simulation dynamically determine their assignment to specific CPUs: this ensures a good data locality in the caches.

When it comes to testing the simulation engine, the results validate our design choices since it behaves as expected and gives good performances for interactive simulations with many autonomous entities. Moreover, it scales well regarding the number of CPUs used—even with multiple distinct caches—and provides superlinear speedups when the memory footprint is appropriate. Therefore, we predict that running ever bigger simulations will be possible by using computers doted with 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 which are stable enough to be easily parallelised and distributed will not benefit greatly from our approach.

To go further with this work, we would like to experiment a memory allocation strategy aware of the *NUMA* architecture of the modern computers to improve data locality even when caches are overflowed. As computing power also dramatically increases in *Graphical Processing Units*, it could be challeng-

ing to commit straightforward computations to this kind of device while keeping the less predictable ones on the CPUs.

References

- [1] Gaubert, L., Redou, P., Harrouet, F., Tisseau, J.,
A first mathematical model of brood sorting by ants: Functional self-organization without swarm-intelligence.
Ecological Complexity, (Dec 2007)
- [2] Desmeulles, G., Bonneaud, S., Redou, P., Rodin, V., Tisseau, J.,
In virtuo Experiments Based on the Multi-Interaction System Framework: the RéISCO Meta-Model.
CMES, Computer Modeling in Engineering & Sciences, (Oct 2009)
- [3] Combes, M., Buin, B., Parenthoën, M., Tisseau, J.,
Multiscale multiagent architecture validation by virtual instruments in molecular dynamics experiments.
ICCS 2010, Procedia Computer Science (Jun 2010)
- [4] Chau, M., El Baz, D., Guivarch, R., Spiteri, P.,
MPI implementation of parallel subdomain methods for linear and nonlinear convection–diffusion problems.
J. Parallel Distrib. Comput., 67, 581–591 (2007)
- [5] Liu, J., Dillencourt, M.B., Bic, L.F., Gillen, D., Lander, D.,
Distributed Individual-Based Simulation,
Euro-Par 2009 Parallel Processing, 5704, 590–601 (2009)
- [6] Reinders, J.,
Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.
O'Reilly, (2007)
- [7] Massaioli, F., Castiglione, F., Bernaschi, M.,
OpenMP parallelization of agent-based models.
Parallel Comput., 31, 1066–1081 (2005)

- [8] Gao, H., Hesselink, W.H.,
A general lock-free algorithm using compare-and-swap.
Inf. Comput., 205, 225–241 (2007)
- [9] Cong, G., Bader, D.A.,
Designing irregular parallel algorithms with mutual exclusion and lock-free protocols.
J. Parallel Distrib. Comput., 66, 854–866 (2006)
- [10] Chynoweth, M., Lee, M.R.,
Implementing Scalable Atomic Locks for Multi-Core Intel EM64T and IA32 Architectures.
(Nov 2009)
http://isdlibrary.intel-dispatch.com/isd/85/AtomicLocks_r2.pdf
- [11] Raman, E., Hundt, R., Mannarswamy, S.,
Structure Layout Optimization for Multi-threaded Programs.
CGO '07: International Symposium on Code Generation and Optimization, 271–282 (2007)
- [12] Tiwari, D., Lee, S., Tuck, J., Solihin, Y.,
MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management.
IEEE International Parallel and Distributed Processing Symposium (Apr 2010)
- [13] Reynolds, C.W.,
Flocks, Herds, and Schools: A Distributed Behavioral Model.
Computer Graphics, 25–34 (1987)

Biography

Fabrice Harrouet was born in Nantes, France, went to the ENIB engineering school in Brest, where he studied computer science, and obtained his PhD in 2000. He works as a lecturer in this school and does his research at the Computer Sciences for Complex Systems Laboratory and in the European Center for Virtual Reality in Brest. His research focuses on interactive multiagent simulations; this mainly concerns parallel computing and 3D rendering.