A Benchmark for Multicore Machines

Frédéric Boussinot

A Benchmark for Multicore Machines (Note)

Frédéric Boussinot*
EMP-CMA/INRIA - Sophia Antipolis
B.P. 93, 06902 Sophia-Antipolis Cedex, France
Frederic.Boussinot@sophia.inria.fr

October 2007

Abstract

Simulation of collision of particles is proposed as a benchmark program for multicore machines. The focus is put on synchronisation and communication of threads. Some results obtained on a dual-core machine are presented.

1 Introduction

It is not so easy to imagine benchmarks showing the benefit of multicore machines. Of course, there is no difficulty to simultaneously use the processors by launching together several applications (or several times the same application). This is not what we are looking for, which can be formulated as: how can *a single application* benefit from a multicore architecture? The standard answer to this question is *multithreading*, which means that the application is decomposed in several threads that can be executed in real parallelism by the processors.

An example of multithreaded applications are Web servers (e.g. Apache servers) in which requests are implemented as threads. However, in servers, threads basically do not communicate and are quite autonomous and independent computing entities. Actually, servers do not really exploit the shared memory which is at the basis of multicore architectures. Servers are, thus, only partial benchmarks for multicore machines.

In the general context of multithreading, the issue of concurrent accesses to the memory shared by threads is immediately raised. Concurrent accesses to the same memory location possibly produce so-called *data-races* that can lead to data corruption and unpredictable behaviors. Thus, the question should be reformulated as follows: how can a single application, coded by a set of

*with support from ACI ALIDECS

communicating and synchronising threads, *safely* (without data-races) benefit of a multicore architecture?

We propose to consider the graphical simulation of a set of colliding particles as a benchmark for multicore machines. Collision processing is basically an algorithm whose complexity is square in the number of particles (actually $n^2/2$, where n is the number of particles). Thus, the amount needed of computing ressource can grow very rapidly as the number of particles increases. Each particle can be involved in several concurrent steps of collision processing: there is thus a need for protecting the data associated to particles. Moreover, the parallel threads should periodically synchronise, in order to get a realistic simulation (otherwise, a subset of particles could stay idle, while another subset is animated several times).

2 Standard Solutions

Let us suppose for simplicity that there are only two cores. In a standard approach, one should have two threads processing particles, each thread being possibly mapped to a core. One should distinguish two sets of particles: the particles to be processed, and the ones already processed. When all particles are processed, the two sets are exchanged and a new cycle begins.

Particles should be protected from concurrent accesses. This can be done by associating a lock to each particle.

Note that the issue can be rather complex when a first particle is being processed by a thread, while the other thread is processing a second particle and tries to determine if a collision can occur with the first particle. Then, a deadlock could occur if the particle lock is taken during all the time the particle is processed.

Another approach would be to divide particles in two sets, each one being processed by a specific thread. The problem then would be to let a thread access information concerning a particle managed by the other thread. Two variants exist for this approach: in the first one, division in two sets is made once for all; in the second variant, particles dynamically move from one set to the other. In the second variant, belonging to a set can be implemented to reflect *geometric proximity*. In this case, collisions can be processed totally independently by the two threads, except at the border of the two sets. This approach gives a way to break the complexity of collision processing. However, the issue of particles at the border of the two sets remains to be handled. This means that both variants have to consider the issue for a thread of accessing information processed by the other thread.

Note that a static (as opposed to dynamic) distribution of particles gives a simple solution to the problem of load balancing for the two executing threads, even if particles are not equally geometrically distributed in the simulation.

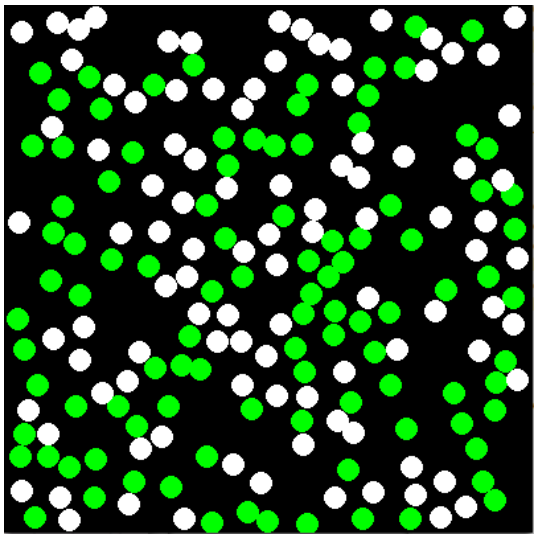

Figure 1: Simulation Snapshot

3 Benchmark Proposal

One considers a simulation made of particles having an inertial movement, and bouncing on the borders of the window. Moreover, collision of particles is processed by using a simple brute force algorithm. The particles are statically partitioned in two sets of equal size. Each set is animated by a thread and the challenge is to execute these two threads in the most parallel way, on the two cores of a dual-core machine. The implementation should be checked against data-races (ideally, absence of data-races should be proved). The number of simulated particles should typically be of several thousands.

The benchmark has been coded in FunLoft[1], a recently proposed concurrent language (the code is given in annex), with results described below.

The machine characteristics are: a Mac running Mac OS X 10.4.10, processor Intel Core 2 Duo, 2.33 GHz, 2GB of memory. Graphics is based on SDL[4].

The simulation with 200 particles is shown in Figure 1 (100 particles in one color are animated by one thread and 100 particles in another color are animated by the other thread).

The CPU usage (for 500 particles) is shown in the right part of Figure 2 (the graphical window is masked during the measure). This is to compare with the usage obtained with the single threaded version of the simulation, shown in the left part.

The time for simulating 100 instants (one instant corresponds to the execution of all the particles) is shown on Figure 3 (representing the output of the unix command `time`), with a memory footprint of about 70MB. Note that the total number of performed interactions is about $100 * 500^2 = 25^6$. The case of

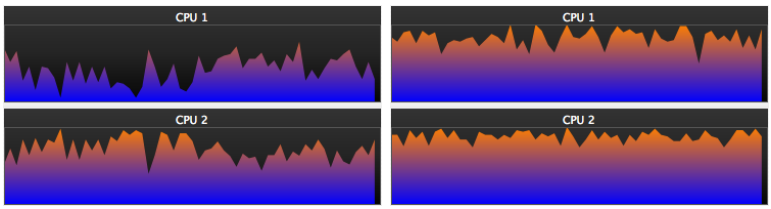

Figure 2: CPU usage. Left: 1 scheduler. Right: 2 schedulers

500 particles	1 sched	2 scheds
real	0m21.832s	0m14.189s
user	0m21.102s	0m21.369s
sys	0m0.220s	0m0.379s

1K particles	1 sched	2 scheds
real	1m20.564s	0m53.724s
user	1m19.587s	1m23.508s
sys	0m0.491s	0m1.078s

Figure 3: Time for 500 and 1000 particles during 100 instants

1000 particles is also shown on the bottom part of Figure 3.

4 Related Work

The Threading Building Block[2] (TBB) of Intel is a C++ library which proposes templates to facilitate data-parallel programming. It comes with several examples and benchmarks. No benchmark however concentrates on applications in which threads are strongly synchronised and communicate often, like in the proposal presented here.

Game Of Life, a particular cellular automaton, has been programmed using TBB in [3]. In the multithreaded version, two treads are running, one for computing the next state of cells, and the other to get information from neigbours. Note that the two threads do not communicate at all and just synchronise at each global step of the automaton evolution. Moreover, this architecture does not fit well with machines having more than 2 cores.

Cellular automata have also been considered in [6] in the context of multithreading. By contrast with the computing effort demanded for each particle in the benchmark proposed in this paper, the computing effort needed for each cell of a cellular automaton is usually very light. Cellular automata have been implemented in FunLoft. The array of cells is decomposed in slices of equal length, run by specific native threads. Except for the special case of self-replicating loops (see [6] for details), the benefit of using several threads does not clearly

appear in these examples. This, however, is to be confirmed by further experimentations.

The issue of game programming in a multicore context is considered by several papers (for example, in [5]).

Several benchmarks for Haskell are presented in [7]. These benchmarks basically use transactions. It would be interesting to code the collision example using transactions in Haskell, to observe the behavior on multicore architectures.

5 Conclusion

We have proposed a benchmark application for multicore machines. The benchmark illustrates the use of heavily communicating and synchronising threads. A priori, such threads are not good candidates for exploiting the full parallelism offered by multicore machines. However, the benchmark shows that a benefit can still be drawn, because the computing effort that has to be done by each thread is more important than the one needed for synchronisation.

The benchmark has been implemented in FunLoft and the obtained results are given in the paper. Note that, due to the specific characteristics of FunLoft, the code is automatically free of data-races.

References

[1] FunLoft. `http://www.inria.fr/mimosa/rp/FunLoft`.

[2] Intel Threading Building Blocks. `http://osstbb.intel.com/documentation.php`.

[3] Multi Threading Sample - The Game of Life. `http://aeshen.typepad.com/-aeshen/2007/02/the_game_of_life.html`.

[4] Simple Directmedia Layer. `http://www.libsdl.org`.

[5] Game Programming in a Multi Core World, august 2006. Report by Mohamed Eldawy, available at `http://adlcommunity.net`.

[6] Frédéric Boussinot. *Reactive Programming of Cellular Automata*. Inria research report, RR-5183, 2004.

[7] Perfumo C., Sonmez N., Cristal A., Unsal O.S., Valero M., and Harris T. Dissecting Transactional Executions in Haskell. In *Proc. Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

ANNEX: Code of Collision Simulation

This section contains the FunLoft[1] code for the simulation. This code is the one that has been used for producing the results previously given.

It should be noticed that threads in FunLoft (produced as instances of modules using the `thread` construct) are basically user-threads, defined at a logical level. On the opposite, FunLoft schedulers are mapped on physical (native) threads (pthreads in the current implementation). Thus, schedulers are the units executed in real parallelism on multicore machines.

External Functions

```
/* External functions returning the applet dimensions, and the size of particles. */
let get_maxx : unit -> int
let get_maxy : unit -> int
let get_size : unit -> int
let get_instant_max : unit -> int
let get_update_factor : unit -> int
let get_zoom : unit -> int

// Interface with the graphical level
let start_graphics : unit -> bool // true = ok
let end_graphics : unit -> unit
let update_display : unit -> unit

// Color type
type color_t =
   BLACK | BLUE | GREEN | CYAN | RED | MAGENTA | YELLOW | WHITE |
   GRAY25 | GRAY50 | GRAY75 | DARK_BLUE | DARK_GREEN | DARK_CYAN |
   DARK_MAGENTA | OCRE

// External function to draw a rectangle
let draw_rectangle :
    int // x
  * int // y
  * int // size in x
  * int // size in y
  * color_t -> unit

// External function to draw a ball
let draw_ball :
    int // x
  * int // y
  * int // radius
  * color_t -> unit

let square_root_f : float -> float

let zoom = get_zoom ()
```

Tracing of Instants

```
/* Tracing of instants and termination of simulation. Tracing should
   be launched in a scheduler in which objects are placed (not in the
   implicit scheduler). */
let module trace_instants (n) =
 let x = ref 0 in
  begin
    repeat n do
    begin
      print_int (!x); print_string (" "); flush ();
      x++;
```

```
      cooperate;
    end;
    print_string ("\nend of simulation\n");
    end_graphics ();
    quit (0)
  end
```

Graphics

```
/* Initialisation of graphical environment. */
let initialise_graphics (maxx,maxy) =
    if start_graphics () then
        begin
           print_string ("\ndisplay ");
           print_int (maxx);
           print_string ("x");
           print_int (maxy);
           print_string ("=");
           print_int (maxy*maxy);
           print_string (" ok\n");
        end
    else
       begin
         print_string ("can't initialize display\n");
         quit (1)
       end

/* At each instant, display is updated and a black background is issued. */
let module graphics (maxx,maxy,color) =
let zoom = get_zoom () in
  begin
     initialise_graphics (maxx*zoom,maxy*zoom);
     loop
        begin
          update_display ();
          draw_rectangle (0,0,maxx*zoom,maxy*zoom,color);
          cooperate;
        end
  end
```

Drawing Processor

```
/* An auxiliary type is defined to hold images of particles. */
type image_t = Image of
    int // x
  * int // y
  * int // size
  * color_t

// Drawing function: image is drawn as a ball (default)
let draw_ball_image (i) =
   match i with Image (x,y,r,c) -> draw_ball (x*zoom,y*zoom,r*zoom,c) end

/* The module that process drawing orders: at each instant, values of
   draw_event are collected and the function draw_image is called for
   each of them */
let module draw_processor (draw_event) =
   loop for_all_values draw_event with i -> draw_ball_image (i)

// Drawing function: image is drawn as a square
let draw_square_image (i) =
     match i with Image (x,y,r,c) ->
         draw_rectangle (x*zoom,y*zoom,r*zoom,r*zoom,c) end

let module draw_square_processor (draw_event) =
   loop for_all_values draw_event with i -> draw_square_image (i)
```

Particles

```
/* A particle is a structure holding four references: x,y
coordinates, x,y speeds, and two constants: radius and color. */

type particle_t = Particle of
      float ref      * // x coord
      float ref      * // y coord
      float ref      * // x speed
      float ref      * // y speed
      int            * // radius
      color_t          // color

// Maximum speed of particles
let max_speed = 5

// Random speed (positive or negative)
let random_speed (m) =
  let x = random_int (2) in
  let v = random_int (m) + 1 in
     if x = 0 then v else -v

/* Creation of a new particle. The particle is randomly placed, and
   has a random speed */

let new_particle (maxx,maxy,size,color) =
   let x = int2float (random_int (maxx)) in
   let y = int2float (random_int (maxy)) in
   let sx = int2float (random_speed (max_speed)) in
   let sy = int2float (random_speed (max_speed)) in
      Particle (ref x,ref y,ref sx,ref sy,size,color)

// Invert the speed of a particle
let invert_x_speed (s) =
   match s with Particle (_,_,sx,_,_,_) -> sx:=-.!sx end

let invert_y_speed (s) =
   match s with Particle (_,_,_,sy,_,_) -> sy:=-.!sy end

// Moves a particle in the four directions
let go_right (s,dist) =
   match s with Particle (x,_,_,_,_,_) -> x:=!x+.dist end

let go_down (s,dist) =
   match s with Particle (_,y,_,_,_,_) -> y:=!y+.dist end

let go_left (s,dist) = go_right (s,-.dist)

let go_up (s,dist) = go_down (s,-.dist)
```

Inertia and Bouncing Behaviors

```
/* Give inertia to the particle at each instant. Inertia simply
   increment coordinates by speed. */

let inertia (me) =
  match me with Particle (x,y,sx,sy,_,_) ->
    begin x:=!x+.!sx; y:=!y+.!sy end
  end

// Let the particle bounce on the applet borders at each instant.
let module bounce_behavior (me,maxx,maxy) =
  match me with Particle (x,y,sx,sy,radius,_) ->
    let r = int2float (radius) in
    let d = int2float (2*radius) in
    let mx = int2float (maxx)-.r in
    let my = int2float (maxy)-.r in
```

```
    let mx2 = 2.*.mx in
    let my2 = 2.*.my in
    loop begin
      if !x <. r then
        begin invert_x_speed (me); x:=d-.!x end
      else if !x >. mx then
        begin invert_x_speed (me); x:=mx2-.!x end
      else ();
      if !y <. r then
        begin invert_y_speed (me); y:=d-.!y end
      else if !y >. my then
        begin invert_y_speed (me); y:=my2-.!y end
      else ();
      cooperate
    end
  end
```

Collision Behavior

```
// Auxiliary type to hold particle information
type coord_t = Coord of float * float * float * float * int
let particle2coord (p) =
   match p with Particle (x,y,sx,sy,r,_) -> Coord (!x,!y,!sx,!sy,r) end

let dot_product (d1,d2,d3,d4) = (d1*.d3) +. (d2*.d4)

/* The collision function processes possible collision beetwen a
   particle and another one given by its coordinates.  The
   distance between the two particles is computed, then the collision
   is processed. */

let collide (me,other) =
  match me with Particle (rx1,ry1,rsx1,rsy1,rad1,_) ->
    let x1 = !rx1 in
    let y1 = !ry1 in
    let sx1 = !rsx1 in
    let sy1 = !rsy1 in
      match other with Coord (x2,y2,sx2,sy2,rad2) ->
        let max_dist = int2float (rad1+rad2) in
        let dx = x2 -. x1 in
        let dy = y2 -. y1 in
        let dist = square_root_f ((dx*.dx) +. (dy*.dy)) in
          if (dist <. (max_dist /. 2.)) || (dist >. max_dist) then return
          else
            let d3 = dot_product (sx1,sy1,dx,dy) in
            let d5 = d3 /. dist in
            let d6 = dot_product (sx2,sy2,-.dx,-.dy) in
            let d7 = d6 /. dist in
            let d8 = d5 +. d7 in
               if (d8 <. 0.0) || (d8 = 0.0) then return
               else
                 let dsx = d8 *. (dx /. dist) in
                 let dsy = d8 *. (dy /. dist) in
                   begin
                     rx1 := x1 -. dsx;
                     ry1 := y1 -. dsy;
                     rsx1 := sx1 -. dsx;
                     rsy1 := sy1 -. dsy;
                   end
      end
  end

let process_all_collisions (me,list) =
  match list with
      Nil_list -> ()
    | Cons_list (other,tail) ->
```

```
        begin
          collide (me,other);
          process_all_collisions (me,tail);
        end
  end

let module collide_behavior (me,collide_event) =
  let r = ref Nil_list in
    loop
      begin
        generate collide_event with particle2coord (me);
        get_all_values collide_event in r;
        process_all_collisions (me,!r);
        inertia (me);
      end
```

Draw Behavior

```
// Generate a drawing order for the particle at each instant.
let module draw_behavior (me,draw_event) =
 loop
   begin
      match me with Particle (x,y,_,_,r,c) ->
         generate draw_event with Image (float2int (!x),float2int (!y),r,c)
      end;
      cooperate
   end
```

Particle Behavior

```
/* Each particle is animated by three threads: one for bouncing on
   the applet borders, one for collision processing, and one for
   graphics. All these threads share the particle. */

let module particle_behavior (collide_event,draw_event,color) =
  let s = new_particle (mx,my,size,color) in
    begin
      thread bounce_behavior (s,mx,my);
      thread collide_behavior (s,collide_event);
      thread draw_behavior (s,draw_event);
    end
```

Global System

```
// number of instants in the simulation
let instants = 1000

// number of particles in the simulation
let particle_number = 200

// Dimension of the applet and size of particles.
let mx = get_maxx ()
let my = get_maxy ()
let size = get_size ()

let s1 = scheduler
and s2 = scheduler
//let s2 = scheduler

let module main () =
  let draw_event = event in
  let collide_event = event in
    begin
```

```
  link s1 do
    begin
      thread graphics (mx,my,BLACK);
      thread draw_processor (draw_event);
      thread trace_instants (instants);
      repeat particle_number / 2 do
         thread particle_behavior (collide_event,draw_event,WHITE);
     end;
  link s2 do
    begin
     repeat particle_number / 2 do
       thread particle_behavior (collide_event,draw_event,GREEN);
    end
end
```