Multicore programming support in Ada

José F. Ruiz
Senior Software Engineer
AdaCore

FOSDEM 2012, Brussels
Outline

• Motivation

• Ada history and advantages

• New exciting things

• How to use them
Why addressing multiprocessors?

• The answer to increasing processing demands
  – We cannot increase the clock frequency forever
  – We cannot increase the instruction-level parallelism forever
  – Provides higher performance for less consumed energy
Periodic activity in Ada and C

---

**C**

<table>
<thead>
<tr>
<th>Periodic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Freq = 10 Hz</strong></td>
</tr>
<tr>
<td><strong>CPU affinity = 1</strong></td>
</tr>
</tbody>
</table>

---

**Ada**

```
task Periodic with CPU => 1;

task body Periodic is
    Activation : Time := Clock;
begin
    loop
        delay until Activation;
        -- Do something
        Something;
        -- Compute next activation time
        Activation := Activation + Milliseconds (100);
    end loop;
end Periodic;
```

**C**

```
static void *periodic (void *arg) {
    struct timespec activation;

clock_gettime (CLOCK_REALTIME, &activation);

    while (1) {
        clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME, 
                         &activation, NULL);

        /* Do something */
        something ();

        /* Compute next activation time */
        if (1000000000 - activation.tv_nsec < 100000000) {
            activation.tv_nsec  += 100000000 - 1000000000;
            activation.tv_nsec  += 100000000;
        } else {
            activation.tv_nsec += 100000000;
        }
    }
}

int main(void) {
    cpu_set_t cpuset;
    pthread_t thread;
    pthread_attr_t attr;

    CPU_ZERO (&cpuset);
    CPU_SET (0, &cpuset);

    pthread_attr_init (&attr);
    pthread_attr_setaffinity_np
        (&attr, sizeof (cpu_set_t), &cpuset);

    pthread_create (&thread, &attr, &periodic, NULL);
}
```
Some advantages of Ada

• Better readability
  – No doubts, right?

• Semantics
  – The Ada run time enforces required dispatching policy, signal mask, master-dependent tasks, …

• Portability

<table>
<thead>
<tr>
<th>Platform</th>
<th>Ada</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Solaris</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Windows</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>VxWorks</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Bare board</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Ada history (in short)

- Concurrency a first-class citizen in Ada
  - Easy to use and analyze
  - Since the beginning
    - Well-developed tasking in Ada 83
  - Ada 95, Ada 2005, Ada 2012 improved and extended tasking

Decades of experience in using Ada on multiprocessors
• **Tasks**
  – Unit of concurrent/parallel execution
  – The notion of thread

• **Task entries**
  – Synchronization and communication

• **Shared variables**
  – Force synchronization point for read and update

```ada
task body DB_Server is
begin
    Initialize_DB;
    Initialized := True;
    loop
        select
            accept Load (I : out Item) do
                Get_Value (I);
                end;
            or
            accept Store (I : in Item) do
                Set_Value (I);
                end;
            or
                terminate;
        end select;
    end loop;
end DB_Server;

task body Customer is
begin
    Tmp : Item;
    while not Initialized loop
        null;
    end loop;
    DB_Server.Load (Tmp);
    Process (Tmp);
    DB_Server.Store (Tmp);
    ...
end Customer;
```

```
Copyright © 2012 AdaCore
Slide: 7
```
• **Addition of protected objects and suspension objects**
  – Synchronization and communication

• **Addition of requeue**
  – Rendezvous and protected objects
    – No communication until the conditions are met
    – Caller is blocked
  – Requeue
    – Complimentary mechanism
    – Allows an accepted request to be moved to another entry
Example of protected objects and requeues

protected body \textit{Dispatcher} is
\begin{verbatim}
entry Send (I : Item)
  when Capacities (LineA) > 0
  or else Capacities (LineB) > 0
  or else Capacities (LineC) > 0
  -- In Ada 2012
  -- when (for some \textit{C} of Capacities \Rightarrow C > 0)
  is
  begin
    if Capacities (LineA) >= Capacities (LineB)
      and then Capacities (LineA) >= Capacities (LineC)
      then
        requeue LineA_Server.Send;
      elsif Capacities (LineB) >= Capacities (LineC) then
        requeue LineB_Server.Send;
      else
        requeue LineC_Server.Send;
      end if;
  end Send;

procedure Set_Capacity (L : Line; C : Natural)
begin
  Capacities (L) := C;
end Set_Capacity;
\end{verbatim}

end \textit{Dispatcher};

end \textit{Line_Server};

end \textit{LineA_Server};

end \textit{LineB_Server};

end \textit{LineC_Server};
- **Ravenscar profile (Ada 2005)**
  - Safe, efficient, resource-constrained concurrency

- **synchronized, protected and task interface (Ada 2005)**
  - Integrating tasking and OOP

- **Scheduling policies**
  - Non-preemptive
  - Round Robin
  - Earliest Deadline First (EDF)
  - Mixed policies within a partition

- **Execution time support**
  - CPU clocks and timers

- **Timing events**
Timing events and execution-time control

---

**Budget monitoring**

```ada
protected body Overrun is
    procedure Timer (TM : in out Timer) is
    begin
        Alarm;
    end Timer;
end Overrun;
```

```ada
task body Enforced is
    WCET : Time_Span := Milliseconds (10);
    WCET_Manager : Timer (Current_Task);
    Cancelled : Boolean;
begin
    loop
        Set_Handler
            (WCET_Manager,
             WCET,
             Overrun.Timer’Access);
        Something;
        Cancel_Handler (WCET_Manager, Cancelled);
    end loop;
end Enforced;
```

Budget enforcing

```ada
protected Watchdog is
    pragma Interrupt_Priority (Interrupt_Priority’Last);
    procedure Timer (Event : in out Timing_Event);
end Watchdog;
```

```ada
protected body Watchdog is
    procedure Timer (Event : in out Timing_Event) is
    begin
        -- We have a problem
        Alarm;
    end Timer;
end Watchdog;
```

Event : Timing_Event;

Set_Handler
    (Event, Clock + Milliseconds (10), Watchdog.Timer’Access);

-- Do something in less than 10 milliseconds
Something;

Cancel_Handler (Event, Cancelled);
```

---

Timing Events
• Ada 2012 added explicit support for controlling processor allocation
  – Dispatching_Domain

• Ravenscar profile for multiprocessors

• Parallel task synchronization

• Memory barriers
Ada has always taken into account parallel architectures

- Allow concurrent/parallel execution
  - Multicomputers, multiprocessors, interleaved execution

- Even allow parallel execution of a single task
  - ... if its effect is as executed sequentially

- A task can be on the ready queues of more than one processor

- Many partitioning schemes allowed
  - Via implementation-defined pragmas or non standard library packages
Synchronization and communication of parallel activities

- **Task synchronization**
  - Protected objects
    - No language-defined ordering or queuing presumed for tasks competing to start a protected action
      - Tasks are intended to spin lock on multi-processors
  - Shared variables
    - Cache coherence
  - Rendezvous
    - The call to the task entry is blocking
Symmetric Multi-Processor (SMP)

• Several similar processors
  – All processors can perform the same functions

• Centralized memory with uniformed access time
  – Problem of cache coherence
• Tasks assigned to a given processor
• How to schedule a group of tasks on a processor is known
  – Rate Monotonic Scheduling (static priorities)
  – Earliest Deadline First (dynamic priorities)
• But, dividing the tasks into groups is NP-hard

• Task migration is permitted
  – Overhead of task migration increases with the number of CPUs
  – Reduced cache performance

None is better than the other in terms of guaranteed CPU utilization
Typical OS support for multiprocessors

- **Set CPU affinity**
  - Allocate tasks to one CPU (or to a group of CPUs)

- **Get CPU affinity**

- **Task migration**
  - From one CPU to another
  - Either user-requested or performed by the OS

- **Spin locks**
  - Tasks wait in a loop until lock is free (busy waiting)
  - Multiprocessor synchronization
Support for multiprocessors in Ada 83, 95, 05

- Ada has always allowed a program’s implementation to be on a multiprocessor system
  - Real parallelism
  - Inter-processor synchronization

- No direct support for affinities
  - The OS can decide the best allocation
  - The developer
    - Implementation-defined pragmas or non standard library packages

- Allows the full range of partitioning
  - But no user control defined in the standard
Explicit support for multiprocessors in Ada 2012

- Notion of dispatching domain
- Safe multiprocessor tasking
- Parallel task synchronization
- Memory barriers
Ada 2012 dispatching domains

- **Focus on SMPs**

- **Handle mapping of tasks to processors**
  - Support all schemes
    - Partitioned
      - Tasks allocated to a subset of CPUs
    - Global
      - Implicit task migration supported
      - Explicit task migration allowed

- **Notion of processor dispatching domain**
  - Group of processors across which global scheduling occurs
    - Non-overlapping dispatching domains
  - Tasks are assigned to an unique dispatching domain
    - A task may be allocated to a given processor within the dispatching domain
    - Or free to be in any of the domain
task type \texttt{Allocated}\_\texttt{Task} (Affinity : CPU)
    with CPU => Affinity;

T1 : \texttt{Allocated}\_\texttt{Task} (1);
T2 : \texttt{Allocated}\_\texttt{Task} (2);
T3 : \texttt{Allocated}\_\texttt{Task} (3);
T4 : \texttt{Allocated}\_\texttt{Task} (4);

GroupA : \texttt{aliased Dispatching}\_\texttt{Domain} := \texttt{Create} (1, 2);
GroupB : \texttt{aliased Dispatching}\_\texttt{Domain} := \texttt{Create} (3, 4);
GroupC : \texttt{aliased Dispatching}\_\texttt{Domain} := \texttt{Create} (5, 6);
GroupD : \texttt{aliased Dispatching}\_\texttt{Domain} := \texttt{Create} (7, 8);

\textbf{task type Grouped}\_\texttt{Task} (Group : \texttt{access Dispatching}\_\texttt{Domain})
    with Dispatching\_\texttt{Domain} => Group.all;

T1, T2, T3, T4 : \texttt{Grouped}\_\texttt{Task} (GroupA'\texttt{Access});
T5, T6 : \texttt{Grouped}\_\texttt{Task} (GroupB'\texttt{Access});
T7, T8, T9 : \texttt{Grouped}\_\texttt{Task} (GroupC'\texttt{Access});
T10, T11 : \texttt{Grouped}\_\texttt{Task} (GroupD'\texttt{Access});
Dynamic affinity handling

GroupA : Dispatching_Domain := Create (1, 2);
GroupB : Dispatching_Domain := Create (3, 4);
GroupC : Dispatching_Domain := Create (5, 6);
GroupD : Dispatching_Domain := Create (7, 8);

task T_In_A with Dispatching_Domain => GroupA;
task T_Non_Allocated;

task body Driver is
begin
  -- Allocate T_Non_Allocated to GroupB
  Assign_Task (GroupB, 3, T_Non_Allocated'Identity);
  Do_Something;

  -- Move it to a different processor
  if Proc_3_Overloaded then
    Set_CPU (4, T_Non_Allocated'Identity);
  end if;
  Do_Something;

end Driver;

start body T_In_A is
  Current_CPU : CPU;
begin
  -- In processor 1 or 2
  Do_Something;

  -- In processor 1 only
  Set_CPU (1);
  Do_Something;

  -- In processor 2 only
  Set_CPU (Get_Last_CPU (GroupA));
  Do_Something;

  -- Now again in processor 1 or 2
  Set_CPU (Not_A_Specific_CPU);
  Do_Something;

  -- Now I am lost. Where am I?
  Current_CPU := Get_CPU;
  pragma Assert (Current_CPU = Not_A_Specific_CPU);
end T_In_A;
Handle affinity and dispatching policy

• What I want
  – Create a group of processors
  – Define an specific scheduling policy for the group
  – Execute a set of tasks within the group

• What I have to do
  – Create a dispatching domain
  – Define a non-overlapping priority band
  – Allocate tasks to the dispatching domain
  – Use priorities in the priority band

```
pragma Priority_SpecificDispatching
  (FIFO_Within_Priorities, 20, 25);

Group : Dispatching_Domain := Create (1, 2);

task T1
  with Dispatching_Domain => Group,
    Priority => 22;

task T2
  with Dispatching_Domain => Group,
    Priority => 23,
    CPU => 1;

task T3
  with Dispatching_Domain => Group,
    Priority => 24;
```
Synchronization on multiprocessors

• **Protected objects**
  – There is a lock-free optimization for monoprocessors (using priorities)
    – No longer viable on multiprocessors
  – Currently Ada advise that tasks should busy-wait (spin) at their active priority for the lock

• **Task entries**
  – Requires internal synchronization primitives aware of multiprocessor
    – Spin locks
• **We need to address:**
  - Reliability
  - Predictability
  - Analyzability

• **The Ravenscar profile for monoprocessors is**
  - Deterministic
  - Time analyzable
  - Simple to use and implement

• **Extend the Ravenscar profile model from monoprocessor to multiprocessor**
  - Fully partitioned model
  - Fixed-priority scheduling
  - Static model

• **Allow for**
  - Simple implementation
  - Verifiable
  - Schedulability analysis
Static model

• Concurrent entities fixed and static
  – Tasks and shared memory defined before execution

• Static fixed priority scheduling algorithm
  – Preemptive fixed priority scheduling in each CPU
    – Analyzable as in Ravenscar for monoprocessors
  – Dynamic-priority scheduling algorithms could increase CPU utilization but:
    – Higher complexity
    – Higher run-time overhead
    – Lower predictability, lower robustness in case of overload

• Partitioned
  – Each task allocated to an user-defined processor forever
    – CPU utilization of partitioned scheduling is neither better nor worse than global
  – It relies on very well known monoprocessor techniques for priority allocation and timing analysis
  – It is much simpler to implement
    – No task migration
Task scheduling

- **Tasks statically allocated to processors**
  - No task migration

- **Preemptive fixed-priority scheduling**

- **Single shared run time**
  - Per-CPU ready queues
  - Spin-locks to protect shared data
    - Disabling interrupts is not enough

- **Operations on a different processors**
  - Triggering a special interrupt in the target processor

```
task Cyclic
  with Priority => 100,
  CPU => 3;
end Cyclic;
```
Task synchronization

• **Library-level protected objects**
  – Shared data with mutual exclusion
  – Both for inter- and intra-processor communication

• **Simple and efficient mutual exclusion changing priority for intra-processor communication**
  – As in Ravenscar monoprocessor
  – Could be statically detected
    – Efficiency
    – Simple timing analysis

• **Spin-locking for inter-processor synchronization**

• **Awaking tasks from other processors**
  – Inter-processor interrupt facility to modify the ready queues
Parallel task synchronization

• Goal
  – Effective parallel task synchronization
  – Set of tasks blocked and released at once

• Typical case
  – A group of tasks must wait until all of them reach a synchronization point
  – And then be released together to work in parallel

• Mimic the POSIX barrier mechanism
package Ada.Synchronous_Barrriers is
   pragma Preelaborate (Synchronous_Barrriers);
   subtype Barrier_Limit is Positive range 1 .. <imp-def>;

   type Synchronous_Barrier
      (Release_Threshold : Barrier_Limit) is limited private;

   procedure Wait_For_Release
      (The_Barrier : in out Synchronous_Barrier;
       Notified   : out Boolean);

private
   -- not specified by the language
end Ada.Synchronous_Barrriers;

Number_Of_Tasks : constant := 8;
Barrier : Synchronous_Barrier (Number_Of_Tasks);

task type Worker (Affinity : CPU) with CPU => Affinity;

task body Worker is
   Notified : Boolean;
begin
   loop
      Wait_For_Release (Barrier, Notified);

      -- Do something in parallel at the same time
      Something;

      if Notified then
         -- Only one task does this
         Ask_For_More_Work;
      end if;
   end loop;
end Worker;
Memory barriers

• **Goal**
  - Have control over cache memories

• **Typical case**
  - Non-blocking algorithms to effectively exploit hardware parallelism
    - lock-free and wait-free

• **Problem to solve**
  - How to ensure the correct order of loads and stores with multi-level caches
  - Modern multicore processors do not guarantee this ordering between processors
    - Optimizations that can result in out-of-order execution
    - Unless special instructions are used

• **How to do it with Volatile**
  - Until Ada 2005
    - They can never be in cache or registers
  - The Ada 2012 (more realistic) approach
    - Volatiles can be handled in cache memories, but
    - Guarantee serial ordering
      - All tasks of the program (on all processors) that read or update volatile variables see the same order of updates to the variables
      - May need the use of an appropriate memory barrier to flush the cache
Example of memory barriers

### In Ada 83, 95, 2005
- Shared_Data and Barrier can **never** be in cache

### In Ada 2012
- Shared_Data and Barrier can be in cache

---

```ada
Shared_Data : Integer;
pragma Volatile (Shared_Data);

Barrier : Boolean := False;
pragma Volatile (Barrier);

task Producer with CPU => 1;
task Consumer with CPU => 2;

task body Producer is
begin
  -- Produce (Shared_Data);
  Barrier := True;
end Producer;

task body Consumer is
begin
  -- If we see that Barrier has been updated,
  -- we must see the produced value of
  -- Shared_Data.

  Use(Shared_Data);
end Consumer;
```
Conclusion

• Ada has supported execution on parallel architectures since its inception

• Ada 2012 dispatching domains
  – Good flexibility and analyzability
  – Implementable on top of typical operating systems and kernels

• Ravenscar for multiprocessors
  – Simple extension to Ravenscar on monoprocessors
  – Partitioning into a set of monoprocessor Ravenscar systems
  – Keep desired properties found in monoprocessor Ravenscar

Ada is a great language for programming multicores