Beyond Ada 2005: Allocating Tasks to Processors in SMP Systems

A.J. Wellings and A. Burns
Department of Computer Science
University of York, UK
{andy,burns}@cs.york.ac.uk

Abstract

Ada 2005 has added no new facilities to support applications that want to run on multiprocessor systems. Following the example set by Ada 95, the language facilitates multiprocessor implementations but provides no direct support for an application-controlled mapping of tasks to processors. Such partitioning is often required to obtain feasible real-time systems. This paper argues that multiprocessors systems are becoming so prevalent that the current position is no longer tenable. A proposal for minimal support is presented.

1 Introduction

Multiprocessors system are becoming more prevalent. In particular SMP systems are often the default platform for large real-time systems rather than a single processor system. The scheduling of processes on these systems can be

1. global – all processors can execute all processes
2. fully partitioned – each process is executed only by a single processor; the set of processes is partitioned between the set of processors
3. mixed – each process can be executed by a subset of the processors; hence the tasks set may be partitioned into groups and each group can be executed on a subset of the processors.

The Ada Reference Manual allows a program's implementation to be on a multiprocessor system. However, it provides no direct support that allows programmers to partition their tasks onto the processor in the given system. The following ARM quotes illustrate the approach.

> "NOTES 1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way." ARM Section 9 par 11.

This simply allows multiprocessor execution and also allows parallel execution of a single task if it can be achieved, in effect, "as if executed sequentially".

> "In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains." D.2.1 par 15.

This allows the full range of partitioning identified above. However, currently the only way that an implementation can provide the mechanisms to allow the programmers to partition their tasks amongst the available processors is via implementation-defined pragmas, or non standard library packages.

This paper argues that multiprocessor systems are becoming so prevalent that it is now time for the language to provide more direct support.

Unfortunately, as of yet, there has been no standardisation of support for multiprocessor systems in the operating system community. Hence, if Ada is being implemented on top of a real-time operating system, it is difficult to know what facilities it can rely on. Consequently, the challenge is to provide a set of mechanisms that can be both expressive enough to support a wide range of application requirements and yet can be implemented (possibly with degraded services) on a wide range of operating systems.

This paper proposes the introduction of a new package `System.Processor_Elements` to capture the interface between the programmer and the underlying system's multiprocessor architecture. A new pragma, `Affinity` is also introduced. The focus is on support for SMP (Symmetric MultiProcessor) Systems.

The paper is structured as follows. In Section 2 we present the main motivations for wanting to provide more explicit support for multiprocessor systems. In Section 3 we briefly review the support that has been discussed or provided by current operating systems. Drawing on this work, we then present (in Section 4) some *initial* thoughts on how to integrate multiprocessor support into Ada. Finally we present our conclusions.

2 Motivation

Whilst many applications do not need more control over the mapping of tasks to processors in an SMP environment, there are occasions when such control is important. They include:

- To allow more flexible approaches to scheduling. – Although the state of the art in schedulability analysis for multiprocessor systems continues to advance [2], the current state is such that partitioned systems offer more guaranteed schedulability than global systems. Quoting from [3]:

  "The choice between global and partitioned approaches to multiprocessor scheduling is a conundrum. Setting aside pragmatic questions about queue contention overhead and differences in cache behavior, the theoretical results are equivocal.

  In favor of global scheduling, it has long been known from queueing theory that single-queue (global) FIFO multiprocessor scheduling is superior to queue-per-processor (partitioned) FIFO scheduling, with respect to average response time.

  Apparently in favor of partitioned scheduling, the application of well-known single processor scheduling algorithms appears superior to the global application of those same algorithms for some task sets with hard-deadlines. For example, it is known that all periodic implicit-deadline task sets with utilization below $m(2^{1/2} - 1)$ can be scheduled on m processors using a first-fit-decreasing-utilization (FFDU) partitioning algorithm and and local rate monotonic scheduling, but Dhall's example shows that there are hard-deadline periodic task sets with total utilization arbitrarily close to 1.0 that cannot meet all deadlines if scheduled on m processors using global rate monotonic scheduling. Dhall's example also applies to global EDF scheduling, yet FFDU partitioned EDF scheduling is guaranteed up to utilization $(m + 1)/2$.

  However, the supposed advantage of partitioned scheduling above disappears if one considers hybrid global priority schemes. The Dhall example can easily be handled by the $EDF - US(1/2)$ or $EDF(k_min)$ schemes, in which top priority is given to a few "heavy" tasks, as can any implicit deadline sporadic task system with utilization up to $(m+1)/2$. This is exactly the same bound as for FFDU partitioned scheduling!

  The experiments we performed on large numbers of pseudo-randomly generated task sets were intended to provide some additional evidence on which to base a choice between these two approaches. From those experiments, statistically, the chance of being able to satisfy all the deadlines of a randomly chosen periodic or sporadic task set appears to be highest with partitioned scheduling. In particular, the partitioned EDF scheduling appeared to be the overall best performer in this statistical sense. At the same time, there are certainly specific task sets where global scheduling is more effective.

  While the schedulability tests used in the experiments probably could be improved, it remains unclear whether they can be improved enough to erase the statistical margin of partitioned scheduling with the available schedulability tests."

- To support temporal isolation. – Where an application consists of tasks of mixed criticality level, some form of protection between the different levels is required. The strict typing model of Ada provides a strong degree of protection in the spatial domain. The CPU budgeting facility provides a limited form of temporal protection but at the expense of flexibility. More flexible temporal protection is obtainable by allowing tasks in each criticality level to be executed on partitions of the processor set.

- To obtain performance benefits. – For example, dedicating one CPU to a particular process will ensure maximum execution speed for that process. Restricting a process to run on a single CPU also prevents the performance cost caused by the cache invalidation that occurs when a process ceases to execute on one CPU and then recommences execution on a different CPU [4].

- To be able to respond to dynamic changes to the processor set. – In a parallel computing environment the set of processors allocated to an application may vary depending on the global state of the system. An application may be able to optimize its algorithms if it is informed when these changes in the processor set occur.

One of the application areas where use of Ada remains strong is in high-integrity systems. It is important to anticipate how the requirements on these systems will change over the coming years so that we can ensure that Ada remains competitive.

Currently there is limited use of general multiprocessor shared memory systems in Safety Critical Systems. Traditionally, where multiprocessors are required they are used in a distributed processing mode: with boards or boxes interconnected by communications busses, and bandwidth allocation, and the timing of message transfers etc carefully managed. This "hard" partitioning simplified certification and testing since one application cannot affect another except through well-defined interfaces. More recently, there has been a move towards more integrated distributed systems where functions are more distributed across a single computing infrastructure (e.g. Integrated Modular Avionics). The goal of this approach is to save space and weight, reduce wiring, provide cheaper fault toleranc and reduce overall costs. Partitioning here is "softer" and is supported by a combination of hardware and software techniques (e.g. memory management support to protect address spaces, some form of CPU budgeting to enforce temporal firewalls, and TDMA on the network).

There has been some use of shared memory modules between processors but access to these memory modules are very restricted and typically only used to coordinate computational activity. Where it has been necessary to use an SMP, only one processor has been enabled[1].

However, there is evidence that future systems will use SMP. For example, the LynxSecure Separation Kernel has recently been announced. The following is taken verbatim from their web site[1]:

- Optimal security and safety – the only operating system to support CC EAL-7 and DO-178B level A
- Real time – time-space partitioned real-time operating system for superior determinism and performance
- Virtualization technology – supports multiple heterogeneous operating system environments on the same physical hardware
- Highly scalable – supports Symmetric MultiProcessing (SMP) and 64-bit addressing for high-end scalability
- Support for open standards – supports 100% binary compatibility for Linux or POSIX-based software application to migrate to a highly robust, secure environment
- Faster time to market – enables developers to begin early development for secure applications

This work has been undertaken by Intel and LynuxWorks to demonstrate the MILS (Multiple Independent Levels of Security/Safety) architecture[2].

3 Review

Although POSIX currently does not provide specific support for SMP systems, the issue has been raised [5]. POSIX.1 defines the "Scheduling Allocation Domain" as the set of processors on which an individual thread can be scheduled at any given time. POSIX states that [6]:

- "For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used;"
- "For application threads with scheduling allocation domains of size greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner."
- "The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time."

With this approach, it is only possible to write strictly conforming applications with real-time scheduling requirements for single-processor systems. If an SMP platform is used, there is no portable way to specify a partitioning between threads and processors.

Additional APIs have been proposed but currently these have not been standardized. The approach has been to set the initial allocation domain of a thread as part of its thread-creation attributes. The proposal is only draft and so no decision has been taken on whether to support dynamically changing the allocation domain.

Since Kernel version 2.5.8, Linux has provided support for SMP systems [4] via the notion of CPU affinity. Each process in the system can have its CPU affinity set according to a CPU affinity mask. A process's CPU affinity mask determines the set of CPUs on which it is eligible to run.

[1]http://www.lynuxworks.com/rtos/secure-rtos-kernel.php.

[2]See http://www.intel.com/technology/itj/2006/v10i3/5-communications/6-safety-critical.htm.

```
#include <sched.h>

int sched_setaffinity(pid_t pid,
    unsigned int cpusetsize,
    cpu_set_t *mask);

int sched_getaffinity(pid_t pid,
    unsigned int cpusetsize,
    cpu_set_t *mask);

void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);

void CPU_SET(int cpu, cpu_set_t *set);

void CPU_ZERO(cpu_set_t *set);
```

A CPU affinity mask is represented by the `cpu_set_t` structure, a "CPU set", pointed to by the mask. Four macros are provided to manipulate CPU sets. `CPU_ZERO` clears a set. `CPU_SET` and `CPU_CLR` respectively add and remove a given CPU from a set. `CPU_ISSET` tests to see if a CPU is part of the set. The first available CPU on the system corresponds to a cpu value of 0, the next CPU corresponds to a cpu value of 1, and so on. A constant `CPU_SETSIZE (1024)` specifies a value one greater than the maximum CPU number that can be stored in a CPU set.

`sched_setaffinity` sets the CPU affinity mask of the process whose ID is pid to the value specified by mask.

If the process specified by pid is not currently running on one of the CPUs specified in mask, then that process is migrated to one of the CPUs specified in mask.

`sched_getaffinity` allows the current mask to be obtained.

An error is returned if the affinity bitmask mask contains no processors that are physically on the system, or `cpusetsize` is smaller than the size of the affinity mask used by the kernel.

The affinity mask is actually a per-thread attribute that can be adjusted independently for each of the threads in a thread group. The value returned from a call to gettid (get thread id) can be passed in the argument pid.

Other operating systems provide slightly different facilities. For example IBM's AIX allows a kernel thread to be bound to a particular processor [3]. Further more, the set of processors (and the amount of memory) allocated to a partition in AIX can change dynamically. In AIX a partition appears to be a subset of resources allocated to a particular subsystem.

The Expert Group responsible of development of the Real-Time Specification for Java (JSR 282) is also considering the appropriate level to support SMP systems. The proposal given here is compatible with their current view.

[3] see http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/-com.ibm.aix.basetechref/doc/basetrf1/bindprocessor.htm

4 Proposal

In the general case, the following may be supported by the underlying platform (operating system and hardware).

1. An application program may be allocated (by the operating system) the full set of the processors in the system or only a subset of them. An initial allocation is performed at the start of program execution time.

2. The operating system may only support global scheduling of threads or it may allow threads to be constrained to one or more processors in the set allocated to the program.

3. The operating systems may dynamically change the allocation of processors allocated to a program during the program's execution. If it does this, it is done in a safe manner.

4. Mechanisms may be provided by the operating system to inform the application (if the operating system supports task to processor allocation) or they may not (if it only supports global scheduling).

From an Ada perspective, there are two possible approaches to supporting task to processor allocation:

1. associate Ada partitions with processor sets

2. associate individual tasks with processor sets

We use the latter approach, as partitions in Ada are more a unit of distribution (or at least implies that each partition executes in a separate address space) and are not first class entities. Here, we are concerned with entities that share memory. Hence the mechanisms we define here are on a per partition basis and we allow tasks to set their processor affinity.

The mechanisms supported by the proposed package (see Figure 1) have been designed with the constraints that should degrade if the program is executing

1. on a single processor system

2. under an operating system which imposes a global partitioning approach.

3. under an operating system that does not change the processor set allocated to a program.

The minimum functionality is for the operating system to allow an Ada program to determine how many processors are available to it.

```
with Ada.Task_Identification; use Ada.Task_Identification;
package System.Processor_Elements is
  Affinity_Error : exception;
  Unsupported_Operation : exception;

  type Processors is range 0 .. <<implementation-defined>>;
  -- The number of processors available on this system.
  -- Each processor has a logical Id in the range.
  -- On a single processor system, the range is 0..0

  type Processor_Set is array(Processors) of Boolean;
  -- A set of processors. A boolean set to True, indicates
  -- that the logical processor is included in the set

  function Available_Processors return Processor_Set;
  -- Indicates which of the processors in the system are
  -- current available to the program. In some
  -- systems this will never change, others it may.


  Dynamic_Set_Changes_Supported : constant Boolean := <<implementation-defined>>;
  -- Indicates if the current system might dynamically change the
  -- Available_Processor set

  Processor_Affinity_Supported : constant Boolean := <<implementation-defined>>;
  -- Indicates whether the system allows a task's affinity to be
  -- set by the programmer

  function Set_Default_Affinity(Processors: Processor_Set) return Processor_Set;
  -- Raises Unsupported_Operation if Processor_Affinity_Supported = False
  -- Raise Affinity_Error if Processors is incompatible with Available_Processors

  function Get_Default_Affinity return Processor_Set;
  -- The default affinity is the set of processors that can
  -- execute a newly created task. The initial system default is
  -- the set returned from Available_Processors, i.e. global
  -- scheduling on any of the processors available to the system.
  -- If Processor_Affinity_Supported = False, then this always
  -- returns Available_Processors

  function Set_Affinity(Processors : Processor_Set; TID :Task_Id := Current_Task)
           return Processor_Set;
  -- Sets the affinity for a particular task.
  -- Raises Unsupported_Operation if Processor_Affinity_Supported = False
  -- Raises Affinity_Error if Processors is in conflict with Available_Processors
  -- The new affinity is set immediately if the task is not executable.
  -- If it is current executable,
  -- the new affinity is set when the task next becomes non-executable
  -- Returns the old set allocated???

  function Get_Affinity(TID :Task_Id := Current_Task) return Processor_Set;
  -- Returns the current affinity of the task

  type Change_Handler is access protected procedure(Processor : Processor_Set);

  procedure Set_Available_Processor_Changed_Handler(
        New_Handler : in Change_Handler; Old_handler : out Change_Handler);
  -- Raises Unsupported_Operation if Dynamic_Set_Changes_Supported = False
  -- If the system allows processors to be added to or subtracted
  -- from the Available_Processors, then the program can request
  -- notification of these changes via a call to a protected
  -- procedure. Here a new call of Set_Available_Processor_Changed_Handler
  -- overwrites any previous call. Whenever a change occurs, the
  -- system calls the last set handler.
end System_Processor_Elements;
```

Figure 1. Proposed API

The API allows for systems that support the dynamic addition and removal of processors from the set allocated to the program. If an operating system does not support this facility then the set will not dynamically change. An operating system is also allowed to maintain a set of logical processors allocated to the program and to transparently change its logical to physical mapping. Again, from the Ada programs perspective the set has not changed. However, it should be noted that this may have an impact on the application if a) it is handling interrupts directly on the processor or b) if the change undermines any feasibility analysis assumptions. For many Ada applications this may not be a problem. In all of the above circumstances `Dynamic_Set_Changes_Supported` is set to `False` in the following package.

If the operating system does support dynamic changes to the processor set, the assumption is that it will inform the Ada program of the changes (e.g. via a signal). The Ada run-time system will pass this information to the application via the calling of a protected procedure. In this circumstances, `Dynamic_Set_Changes_Supported` is set to `True`.

The assumption is that the application will maintain its own list of which tasks are mapped to which processors (logical or physical). It will then undertake whatever reconfiguration it deems appropriate.

If a processor fails and the platform cannot transparently recover, the Ada program abnormally ends (with assumed fail stop semantics). Any recovery must be performed outside of the Ada program. This is because a processor failure can leave the application in an inconsistent state (e.g. with a corrupted heap) from which it is unlikely to be able to recover.

The API supports the setting of the affinity tasks by the programmer. If the operating system doesn't support this facility then all of the associated operations, raise the `Unsupported_Operation` exception, and `Processor_Affinity_Supported` is set to `False`.

The full API is shown in the AFigure 1, annotated with the semantics of the subprograms. For convenience, the affinity mask is shown as a boolean array. In practice, a more efficient representation of the affinity mask would be needed.

Open Issues

- Defaults – The current proposal has default affinity arrays. In Ada, the default priority of a task is the same as its parent, and a pragma is defined to allow the priority to be set at task creation time. Hence, a pragma such as `pragma Set_Affinity(Mask'Access)` could be provided.
- Dispatching Policies – Where a task can be executed on more than one processor it may be appropriate to define a new dispatching policy to obtain efficient use of caching. For example, the current policies could be extended and a new one added as follows:
  - FIFO_Within_Priorities. With this policy, a task can be migrated between its allocated processors whenever it is preempted.
  - Non_Preemptive_FIFO_Within_Priorities. With this policy, a task once dispatched to a processor will not be migrated from that processor whilst it is still executable. Furthermore, it cannot be preempted.
  - FIFO_Within_Priorities_Without_Migration. A new policy, a preempted task cannot be migrated from the processor from which it was preempted whilst it is still runnable.
  - EDF_Across_Priorities. It is not clear what this policy means if the tasks assigned to the priority range can be executed on a possible disjoint set of processors.
- Interrupt handling – Some SMPs allow the affinity of an IRQ to be set. Hence, certain interrupt handlers can only run on that processor set (e.g. on Redhat linux /proc/irq/IRQ#/smp_affinity specifies which target CPUs are permitted for a given IRQ (Interrupt ReQuest line) source). An alternative version of the `Attach_Handler` pragma could be provided to allow the mask to be set. Also a new subprogram in `Ada.Interrupts` could allow the mask to be set in the dynamic case.
- Asynchronous task control – The current definition of `Ada.Asynchronous_Task_Control` seems to work adequately for the multiprocessor case. However, setting the affinity of a task to be "no processors" also needs to be considered in this context. In particularly when it is waiting at an accept/select statement.
- Current Processor – A mechanism may be needed for a task to determine the actual processor upon which it is currently executing. Such a facility could be provided in the above package.

5 Conclusions

Historically, Ada has always taken a neutral position on multiprocessor implementations. On the one hand, it tries to define its semantics so that they are valid on a multiprocessor. On the other hand, it provides no direct support for allowing a task set to be partitioned. This paper has argued that multiprocessors are becoming more ubiquitous, and that there are advantages to be gained by allowing the program more control over which task executes where. Unfortunately the POSIX standards do not currently address this issue, and consequently it is difficult to know what mechanisms Ada can rely on existing in the underlying execution platform. Consequently, the paper has proposed an API which can gracefully degrade according to the facilities provided.

Acknowledgements

The authors gratefully acknowledge the contributions of the JSR 282 and JSR 302 Expert Groups where many of the ideas presented in this paper have been discussed (albeit from within a Java context).

References

[1] B.S. Anderson. Saftey critical systems and SMPs, private communication, 2006.

[2] T.P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 32(1-2):41–71, 2006.

[3] T.P. Baker. Global versus partitioned scheduling in multiprocessor systems, private communication, 2006.

[4] Linux Manual Page. sched_setaffinity(), 2006.

[5] Michael Gonzalez Harbour. Supporting SMPs in POSIX, private communication, 2006.

[6] Open Group/IEEE. The open group base specifications issue 6, ieee std 1003.1, 2004 edition. IEEE/1003.1 2004 Edition, The Open Group, 2004.