Virtual Processing in ParMod

Franz Abstreiter
Institut für Informatik
Technische Universität München
D-8000 München 2, F.R.G.

Abstract
The programming language ParMod supports distributed processing as well as parallel processing. The latter one allows dynamical unbounded parallelism. These parallel activities have to be mapped by well-suited scheduling mechanisms and policies (system scheduling) to the given physical processors. On a higher level, implementation of scheduling mechanisms and policies is supported by several ParMod language constructs (user scheduling). A semantical correct implementation of user scheduling demands for certain requirements on system scheduling which reduce the set of possible mechanisms and policies for system scheduling.

Introduction
Dynamical unbounded parallelism is recognized to be the best way for learning parallel thinking because parallel activities should be generated whenever they are necessary. In a parallel program the degree of parallelism should be determined by the problem itself, not by a given hardware configuration. The mapping of unbounded parallelism to a hardware configuration should not be done by the user or by the programmer but by the runtime environment of the parallel language. The term virtual processing, as used in this paper, describes all techniques that allow us to execute a parallel program whose degree of parallelism may be greater than the number of physical available processors.

The number of (physical) processors is bounded in every multiprocessor system. Thus, there has to be a mapping of the virtual processors of the parallel program to the physical processors of a given multiprocessor system. Scheduling policies and scheduling mechanisms for parallel programs differ from those for multi-user systems in the way that there is merely one parallel program whose parallel activities (i.e. virtual processors) are to be scheduled, and, furthermore, that a parallel activity may cooperate with other parallel activities.

Since parallel activities may depend on each other, scheduling can no longer be considered solely as a matter of concern to the virtual processing system. Hence there should be some language constructs which influence the scheduling policies. ParMod offers several language constructs for this purpose.

The Parallel Language ParMod — A Survey
Based on the procedural language Pascal [10] ParMod supports two major concepts: distribution and parallelization. A ParMod program consists of several (at least one) modules which are executed in parallel not sharing any memory. Similar to the concept of distributed processes [3] or guardians [8] communication between modules is is only performed by calling global parallel procedures. ParMod, however, provides non-blocking calls [6]. This means that calling a parallel procedure does not block the caller (in contrast to remote procedure calls [3,6] or rendezvous [1]). Results of a global parallel procedure call are sent back asynchronously. Parallelism between modules is called global parallelism.

Parallelism within a module (local parallelism) is achieved by executing all calls of global parallel procedures in parallel and, in addition, by calling local parallel procedures [6] which will also be executed in parallel.

The degree of global parallelism is fixed statically whereas the degree of local parallelism changes dynamically and may be unbounded.

A global parallel procedure may have a sequence part and a pre part. By the sequence part an execution order (corresponding to the calling order) is specified for all calls of global parallel procedures with the same sequence part from the same parallel activity. The pre part specifies a statement which is executed with high priority. After executing the pre
part the body part starts to execute with low priority.

Within a module parallel activities may operate on common variables. Synchronization is provided by a region statement which is similar to the concept of Hoare's conditional critical regions [7].

The ParMod Virtual Processing Model

In ParMod every parallel activity (calls of parallel procedures, answers, init part and main part) is executed on its own virtual processor. A parallel activity may be in one of two states: active or delayed. Initially the init part is active and all calls of global parallel procedures from outside are delayed until the init part terminates. Then the main part gets active.

A parallel activity is delayed
- with respect to the sequence part, if there is a (yet not terminated) parallel activity having the same sequence part and called by the same parallel activity earlier, or
- with respect to a region statement if the evaluation of the region condition (i.e., accessibility to the region elements and evaluation of the Boolean expression) yields the value false.

A parallel activity is activated
- with respect to the sequence part, when the direct predecessor in the calling order terminates, or
- with respect to a region statement, when the region condition yields the value true.

System Scheduling

The programming language ParMod allows dynamic unbounded parallelism. Therefore, for each given multiprocessor system there obviously exists a ParMod program whose degree of parallelism is greater than the number of available physical processors. In all those cases, where there are available too few processors, scheduling mechanisms and policies are demanded for (system scheduling), which allocate the physical processors to the virtual processors in such a way that every parallel activity will be executed.

On the other hand, if we consider a with a given input terminating ParMod program, then there will exist a multiprocessor system whose number of processors equals the number of necessary virtual processors. In this case processor utilization is minimal and the degree of parallelism is maximal. Since we want to obtain a good ratio of processor utilization to the degree of parallelism and the costs of a physical processors, we will put up with a lower degree of physical parallelism in favour of a higher utilization of the physical processors. Hence again system scheduling is required.

User Scheduling in ParMod

Scheduling the virtual processors of a parallel program to the physical processors of a multiprocessor system can no longer be considered solely as a matter of concern to the virtual processing system because parallel activities may depend on each other. Hence there should be offered control mechanisms by the parallel programming language.

An example for this is an unbounded buffer which is accessed simultaneously by several producers and consumers. If there exist more producers than consumers, consumers will get higher priority than producers, so that a consumer should be preferred. Here, the property "priority for consumers" becomes part of the problem specification. Therefore an implementation language should make it possible to realize such properties.

In the following the term "User Scheduling" describes the mechanisms for controlling the execution of the virtual processors by the parallel program itself. ParMod supports User Scheduling by several language features:
- suspension
- critical regions
- serialization
- priorities

The detach statement

\[\text{detach}\]

is used to release a physical processor from the virtual processor. The virtual processor will be suspended, this means it will be ready to execute but there is no physical processor idle.

Critical regions can be expressed by region statements:

\[\text{await } \langle\text{Boolean expression}\rangle\]
\[\text{region } \langle\text{region elements}\rangle\]
\[\text{do } \langle\text{statement}\rangle\]

The \(\langle\text{statement}\rangle\) will be executed if the evaluation of the region condition (i.e., accessibility to the region elements and evaluation of the Boolean expression) yields the value true. Exclusion is achieved by the fact that each region element can be accessed simultaneously by at last one parallel activity.

Serialization respectively priorities are provided within global parallel procedures:

\[\text{global procedure } \langle\text{proc ident}\rangle \{ \langle\text{parameters}\rangle \}; \]
\[\{ \langle\text{local declaration part}\rangle \}
\[\langle\text{sequ}\rangle \langle\text{sequ ident}\rangle \}
\[\{ \langle\text{pre}\rangle \langle\text{statement}\rangle \}
\[\text{begin } \langle\text{statement}\rangle \{ ; \langle\text{statement}\rangle \} \text{ end}\]

If a sequence part is specified all incarnations of this procedure called from the same parallel activity are serialized, this means they are executed sequentially one after the other according to the calling order.

The pre statement is executed with high priority whereupon the body part (begin ... end) is executed with low priority.
By two examples we will show how user scheduling can be realized in ParMod.

Example 1: Fair Readers/Writers Problem with Writer Serialization

Specification:
- Readers and writers may not be active at the same time
- At most one writer may write the resource at the same time
- Arbitrary many readers may read simultaneously
- Neither a reader nor a writer has to wait
- Writers of the same master task get access to the resource in the order they are called.

module ReadersWritersSerial;
var
  aw: integer; { number of waiting and active writers }
  ar: integer; { number of active readers }

global procedure Read (...);
begin
  await aw = 0 region registration do
  ar := ar + 1;
  read the resource;
  ar := ar - 1;
end;

global procedure Write (...);
sequ writing;
begin
  await aw = 0 region registration do
  aw := aw + 1;
  await ar = 0;
  write the resource;
  aw := aw - 1;
end;

init
begin
  aw := 0;
  ar := 0;
end
endmodule

Program 1. Fair Readers/Writers Problem with Writer Serialization

A ParMod solution of this problem is given in program 1. The global procedures Read and Write realize the reading and writing access to the resource. Mutual exclusion of readers and writers is achieved by the module variables aw and ar. If a writer is active (aw > 0), then every reader must wait until the writer has finished (await aw = 0). If readers are active (ar > 0), then a writer must wait until no reader is active (await ar = 0).

The registration of a waiting writer (aw := aw + 1) is exclusive (region registration do), and every writer must wait until no writer is waiting or active (await aw = 0). Hence there exists at last one registered writer at the same time.

Arbitrary many readers may read simultaneously the resource because every reader is waiting only for the condition that no writer is registered (await aw = 0).

Readers as well as writers are waiting for the same condition (await aw = 0 region registration do) whenever they are created. Since this condition holds infinitely often, neither a reader nor a writer will be deferred infinitely often. Thereby fairness among readers and writers is guaranteed.

The serialization part (sequ writing) takes care that the execution order of the writers of the same master corresponds to the calling order.

Example 2: Nonpreemptive Scheduling Policies

The second example presents a general method, how nonpreemptive access to a resource can be controlled (cf. [5]). Policies as for instance first-come-first-served, last-come-first-served, shortest-job-next, priority-queue, path expressions, can be implemented using the programming scheme presented in program 2.

module Resource;
declaration of data structure A
function NewRequest (...): p;
structure A and
  insert a request with the given parameters into data structure A
  gives as result the identification of this request in A
function Schedule (r: p): boolean;
  • uses data structure A and
  • gives the result true if the request identified by r is the request to be processed next with respect to the processing strategy

init
begin
  r := NewRequest(...);
end:
endmodule

Program 2. Nonpreemptive Scheduling Policies

Each access to the resource is registrated and processed by
the global parallel procedure Request. The registration is done within the pre part because this guarantees that the data structure always is kept up-to-date even if there is a lack in processing capacity. Each request has to wait for being scheduled, but then it is processed completely and nonpreemptive with regard to the resource.

Two examples were presented how user scheduling is implementable in ParMod. But there exist several scheduling mechanisms and policies which cannot be implemented by ParMod. For instance, preemptive scheduling policies cannot be described in ParMod because the decision, whether a request is processed, is made only one time, namely at the beginning of the request. We can approximate preemptive scheduling policies by partitioning the processing of a request into several parts, whereby each part is preceded by a scheduling decision. This kind of scheduling is also known as self scheduling [10].

Requirements to System Scheduling

A correct implementation of user scheduling demands for requirements which are not fulfilled by each scheduling mechanism or policy. For instance, if parallel activities with high priority are scheduled noninteruptable, then nontermination may be possible in appropriate programs, whenever too few processors are available.

In the following requirements of user scheduling to system scheduling are elaborated by means of distinguished examples.

(a) Local Parallelism

ParMod semantics (cf. [4]):

No virtual processor is suspended for an infinite amount of time.

Program 3. Local parallelism

(b) Priority Mechanism

ParMod Semantics (cf. [4]):

If within a module a parallel activity with low priority is executing then there is no parallel activity with high priority which is suspended.

Program 4. Priority mechanism

Corresponding to the previous example the main part of program 4 calls the global parallel procedure terminate and enters a loop which it leaves when the global parallel procedure assigns the value true to the module variable terminating.

The global parallel procedure terminate has a pre part which is executed immediately after the call because the main part is just active with low priority. If there is available merely one processor, then the main part must be interrupted, so that the procedure terminate can be executed.

Requirements:

- Two priority classes (high and low)
o Preemption of low priority parallel activities by high priority parallel activities

o Justice (and preemption) within each priority class

(c) Activating Mechanism

ParMod Semantics (cf. [4]):
If a region condition is established infinitely often, then every parallel activity waiting for this region condition will sometimes be activated.

module Termination;
var terminating: boolean;
x: integer;
global procedure WaitAndTerminate
  (var ready: boolean);
  pre begin
    answer ready;
    await x > 0 region x do;
  end;
  begin
    terminating := true
  end;
global procedure Wait
  (var ready: boolean);
  pre begin
    answer ready;
    await x > 0 region x do;
  end;
  begin
    end;
main
begin
  terminating := false;
x := 0;
  repeat
    region x do
    begin
      Wait(ready);
      await result(ready);
      x := x + 1
    end
    until terminating do
  end
endmodule

Program 5. Activating mechanism

In program 5, first of all the global parallel procedure WaitAndTerminate is called and then within a loop the global parallel procedure Wait is called iteratively whereby variable x counts the number of calls. After each call of a global parallel procedure the main part waits (by means of the result variable ready) until the pre part of the new parallel activity will be executed. Each incarnation of the global parallel procedures is delayed at the region statement because the main part occupies the region element x.

The program will terminate, if the global parallel procedure WaitAndTerminate terminates. This will be the so, if it is not delayed at the region statement (await x > 0 region x do) for an infinite amount of time. Therefore it is necessary that a region condition will be evaluated sometimes again as often as there exist parallel activities waiting for it.

On the other hand there should be justice among parallel activities waiting for the same region condition. Otherwise the procedure WaitAndTerminate would never be executed, perhaps.

Requirements:

o A region condition tasks are waiting for will be evaluated sometimes.

o Justice among delayed parallel activities waiting for the same region condition

(d) Sequence Mechanism

module Termination;
var terminating: boolean;
x: integer;
global procedure WaitAndTerminate;
  sequ Waiting;
  pre
    x := x + 1;
  begin
    await x > 1;
    terminating := true
  end;
main
begin
  x := 0;
  terminating := false;
  WaitAndTerminate;
  until terminating;
end
endmodule

Program 6. Sequence mechanism

ParMod Semantics (cf. [4]):
All calls of global parallel procedures with the same sequence
part, which are made by the same parallel activity, will be executed sequentially according to their calling order (with respect to high and low priority parts).

In program 6 the main part calls with a loop iteratively the global parallel procedure \textit{WaitAndTerminate} as long as the first of these calls terminates. The first call will terminate merely, if the serializing distinguishes high and low priority parts. Otherwise, if there exist only one waiting queue for each sequence part and caller, then the first incarnation of the procedure \textit{WaitAndTerminate} would not terminate because it would be delayed at the region condition \((\text{await } x > 1)\) forever, and therefore the waiting queue would be blocked forever.

Requirements:

- Two first-come-first-served waiting queues (high and low priority) for each pair of sequence part and caller.

\section*{(e) Global Parallelism}

\begin{verbatim}
module Master;
external Slave procedure Start;
external Slave procedure Stop;
main
  begin
    Start;
    Stop
  end
endmodule

module Slave;
var terminating: boolean;
global procedure Start;
  pre
    while not terminating do;
    begin
      end;
global procedure Stop;
  pre
    terminating:= true;
  begin
    end;
init
begin
  terminating:= false
end
endmodule
\end{verbatim}

Program 7. Global parallelism.

\textbf{ParMod Semantics} (cf. [4]):

No virtual multiprocessor is delayed for an infinite amount of time.

In module Master of program 7 two global parallel procedures (Start and Stop) of the module Slave are called. In its pre part procedure Start enters a loop, which it leaves when procedure Stop has finished its pre part.

If there exists only one processor, then the pre part of the procedure Start takes care that the main part is suspended without having called procedure Stop. Guaranteeing the termination of this program, we must require that the procedure Stop will actually be called sometimes.

Requirements:

- Justice (and preemption) between the sets of parallel activities of different modules.

\textbf{Mechanisms and Policies for System Scheduling}

Summed up the previous section there are the following requirements to mechanisms and policies of system scheduling:

(R1) Justice (and preemption) between the sets of tasks of different modules

(R2) Two priority classes (high and low) within a module

(R3) Justice (and preemption) within each priority class

(R4) Preemption of low priority parallel activities by high priority activities with a module

(R5) Two first-come-first-served waiting activities for high priority modules

(R6) Each region condition parallel activities are waiting for will be evaluated sometimes

(R7) Justice among delayed parallel activities waiting for the same region condition.

Requirements (R1)-(R7) will be fulfilled by a variety of system scheduling mechanisms and policies which result from combining well-known mechanisms and policies fulfilling particular requirements.

(R1) is fulfilled by round robin, for instance, whereby the time slice may depend on the number of parallel activities of a module. If priority scheduling is used then there will have to be some technique (e.g. aging) which will guarantee justice between modules. (R3) is fulfilled by the same mechanisms and policies. Additionally, nondeterministic group scheduling (cf. [2] is possible. Preemptive priority scheduling fulfills (R2) and (R4).

(R5) is fulfilled by a multi-level queue which is scheduled (non-preemptive) first-come-first-served. Scheduling takes place whenever a parallel activity enters an empty waiting queue or whenever a parallel activity leaves the waiting queue.

(R6) and (R7) are fulfilled by evaluating region conditions according to a first-delayed-first-checked mechanism. The evaluation of region conditions must be activated by a timer, for instance.
Implementing requirements (R1)–(R5) will be cheap, whereas fulfilling requirements (R6) and (R7) will be more expensive: the decision, when and how often a region condition is to be evaluated, may considerably influence the execution of time critical parallel programs. A frequent evaluation will approximate the virtual execution behaviour better, but will lead to a higher scheduling overhead.

Sequence parts can be implemented by ParMod itself. So, they can be removed from the language without reducing the expressing power of ParMod. In this case the requirement can be dropped that the communication network has to preserve the order of messages.

Stropping off pre parts we have to take into bargain execution time displacements of parallel activities. Infinite execution time displacements will be obtained if the requirements (R1) and (R3) are dropped. The gain of stropping off preemption will be that interrupting parallel activities is no longer needed and that the number of context switches is reduced drastically.

Conclusion

This paper shows that scheduling is necessary for implementing programming languages which allow unbounded parallelism. On the other hand programming of scheduling mechanisms and policies for parallel activities is desired in the parallel programming language itself. ParMod supports user scheduling by means of suspension, region statements, sequence parts and pre parts. User Scheduling restricts system scheduling. Nevertheless, there exist several system scheduling mechanisms and policies which guarantee a semantical correct implementation of user scheduling.

References