Scheduling of Compute-Intensive Code Generated from Event-B Models: An Empirical Efficiency Study

Fredrik Degerlund

Scheduling of Compute-Intensive Code Generated from Event-B Models: An Empirical Efficiency Study*

Fredrik Degerlund

Åbo Akademi University &
TUCS - Turku Centre for Computer Science
Joukahainengatan 3-5, FIN-20520 Åbo, Finland
fredrik.degerlund@abo.fi

Abstract. Event-B is a tool-supported specification language that can be used e.g. for modelling of concurrent programs. This calls for code generation and a means of executing the resulting code. One approach is to preserve the original event-based nature of the model and use a run-time scheduler and message passing to execute the translated events on different computational nodes. In this paper, we consider the efficiency of such a solution when applied to a compute-intensive model. In order to mitigate overhead, we also use a method allowing computational nodes to repeat event execution without the involvement of the scheduler. To find out under what circumstances the approach performs most efficiently, we perform an empirical study with different parameters.

1 Introduction

Event-B [1] is a formal modelling language based on set transformers and the stepwise refinement approach. While designed for full-system modelling, it can also be used for correct-by-construction software development. Event-B also has a parallel interpretation, which allows for modelling of concurrent systems. Tool support for Event-B has been achieved through the open-source Rodin platform [2], to which further functionality can be added in the form of plug-ins. Code generation from Event-B can be achieved in a number of different ways. A straight forward approach that preserves the event nature has been proposed in [3,5], for which a preliminary plug-in has been developed. In this approach, the model is translated into a C++ class, where events are directly translated into methods. The methods are invoked by using a run-time scheduler, which in turn deploys the MPI (Message Passing Interface) [7] library to achieve parallel execution on a multi-core/multi-processor system, or even on a cluster. This solution has the advantage that code execution very closely reflects the operating mechanisms of the Event-B model. It also does not require the developer to take a stand on specific schedules and prove that they are compatible with the original model.

* This research was supported by the EU funded FP7 project DEPLOY (214158). http://www.deploy-project.eu

However, a potentially serious drawback is the amount of overhead introduced by the scheduler and the MPI communication. Due to the practical nature of communication overhead, we recognise that it is difficult to evaluate the impact from a strictly mathematical-logical perspective. The purpose of this paper is, instead, to evaluate the viability of the scheduling approach by performing an empirical study. Since preliminary tests indicate that the overhead is unacceptably large, we propose a means of repeating execution of events without involving the scheduler. The repetitive approach is implemented as part of the scheduling platform, and we use a factorisation model as a testbed for benchmarking.

2 Event-B and Code Generation

Event-B models consist of *static* and *dynamic* parts, denoted *contexts* and *machines*, respectively. Contexts may contain e.g. *constants*, *carrier sets* and *axioms*, and can be used by one or several machines. Machines, in turn, contain elements such as *variables*, *events* and *invariants*. The variables v form the state space of the model, whereas events model atomic state updates. There is also a special *initialisation* event that gives initial values to the variables.

Each event, except for the initialisation, contains a *guard* $G(v)$ and an *action* $v :| A(v, v')$. The guard contains a condition that must hold in order for the event to be allowed to take place, whereby the event is said to be *enabled*. The action describes how the state space is to be updated once the event is enabled and triggered. An event can be expressed in the following general form [6]:

$$E \triangleq \textbf{when } G(v) \textbf{ then } v :| A(v, v') \textbf{ end}$$

Here, v and v' represent the variables before and after the event has taken place, respectively. The operator $:|$ represents non-deterministic assignment, whereby $v :| A(v, v')$ intuitively means that the variables v are updated in such a way that the *before-after predicate* $A(v, v')$ holds. A special case of the non-deterministic assignment operator is the deterministic assignment, $:=$, which closely resembles the assignment operator in standard programming languages. Note that the initialisation is an exception in that it contains no guard and does not depend on a previous state. After initialisation, enabled events are non-deterministically chosen for execution until all events are disabled.

Event-B does not specify how to generate executable code from models, and the Rodin tool in its basic form cannot translate models into a programming language without the use of extensions. However, a number of different approaches have been proposed. In [9], a code generator plug-in was developed. It was mainly intended for use as part of a virtual machine project, and supported translation of the most important Event-B constructs. This approach was taken a step further towards a more general-purpose tool, albeit an experimental one, in [3,5]. The model first has to be refined according to the Event-B refinement rules (e.g. using the Rodin tool) until the events only contain concrete constructs that have direct equivalents in C++. The guard of the event is translated into a method

returning a boolean value reflecting enabledness, whereas the action results in a separate method containing the C++ equivalent of its assignments. The idea was that the resulting methods could be invoked by an accompanying scheduler. The testbed model (see Sect. 4) we benchmark in this paper (Sect. 5) is based upon a model originally used in [3,5], and the translated code thereof. The model has, however, been amended in ways that could not be handled by the translation plug-in, and the code used for in this paper has, to a certain degree, been translated manually.

3 Scheduling

When an Event-B model has been translated into C++ code, a means of scheduling the resulting code is required. For the evaluations performed in this paper, we use a scheduler from [3,5] that allows for execution on a multi-core/multi-processor computer. However, we have improved it further in a number of ways, e.g. to handle 64-bit integers and to support repetition of events as presented later in this section. The scheduler code, which is written in C++, technically runs as part of both the scheduling process and a number of slave processes. The processes are mapped to physical processors or cores by the MPI framework, which the scheduling software uses for all inter-process communication. Communication takes place according to a star topology with the scheduling process in the centre, delegating event execution to the slaves. The scheduling process keeps track of the state space of the model, and when delegating an event for execution, it submits the current values of the variables involved to the slave. When the slave has executed the event, it returns the updated values of the variables to the scheduling process. To avoid conflicts, events that have variables in common must not be scheduled in parallel. It is also the responsibility of the scheduler to verify that events are enabled prior to delegating them. A more detailed description of the scheduling algorithm can be found in [3,5].

To reduce overhead, we have amended the above scheduling approach so that the slave processes may execute an event several times on their own. Before the scheduling process first delegates an event, it verifies the enabledness and passes on the values of the variables to the slave process as previously described. However, after execution, the slave checks whether the event is still enabled. If that is the case, it may run it again without any involvement of the scheduling process. This procedure may take place several times, until the event has been executed at most *REPEAT* times (including the initial execution delegated by the scheduling process), after which the updated variable values are reported to the central scheduler. The constant *REPEAT* can be seen as a parameter of the scheduling platform, and it applies to all slaves processes and, in principle, to all events. However, since events may disable themselves even after only one or a few consecutive executions, *REPEAT* is to be seen as an upper limit. Also note that an event does not automatically become disabled after being executed *REPEAT* times, but to continue running it, it must once again be chosen for execution by the scheduling process.

4 Testbed model

We use a trial division based integer factorisation model as our testbed. While trial division is not particularly efficient, we are not primarily interested of evaluating the performance of the algorithm *per se*, but rather that of the scheduling method as compared to a sequential program. The model has its roots in [3,5], but we have revised it e.g. so that it can make use of the automatic repetition of events. The goal of the model is to find a factor of a given integer n, such that it is ≥ 2 and $< n$. However, if n is a prime number, the result reported is n itself.

At the core of the model are the factorisation events *process1*, *process2*, etc., up till the number of computational, or slave, processes. This typically corresponds to the number of hardware computational nodes (processors or cores) to be used for slave computations. The Event-B notation of the factorisation events, in a model designed for two computational processes, is given in Fig. 1. Note that we use separate events instead of parametrisation, since we want the factorisation events to be separate from each other.

process1 $\triangleq$	process2 $\triangleq$
when	**when**
$continue\_1 > 0$	$continue\_2 > 0$
$result\_1 \neq 0$	$result\_2 \neq 0$
$i\_1 < n/2$	$i\_2 < n/2$
then	**then**
$result\_1 := n \bmod i\_1$	$result\_2 := n \bmod i\_2$
$i\_1 := i\_1 + STEP$	$i\_2 := i\_2 + STEP$
$continue\_1 := continue\_1 - 1$	$continue\_2 := continue\_2 - 1$
end	**end**

Fig. 1. Factorisation events for two computational processes.

There are variables i_1, i_2, etc., associated with the respective factorisation events. Variable i_1 is initialised to the value 2 (i.e. 1+1), i_2 to the value 3 (i.e. 2+1), etc., and each time a factorisation event m is executed, it checks whether the constant n is divisible by the current value of its associated variable i_m. If that is the case, a factor has been found. To distribute the work evenly among the processes, i_m is after each trial division incremented by a constant *STEP* containing the number of factorisation events. Each factorisation event m is also associated with a counter $continue\_m$. Initially set according to a constant *CONTINUES*, it is decreased by 1 after every trial division. By checking that $continue\_m > 0$ as part of the guard, the number of consecutive executions of each factorisation event is limited to *CONTINUES*.

Since the factorisation events must not have any variables in common, they cannot directly check whether another event has found a factor. This is where a synchronisation event *newround* comes into play. After the factorisation events have been executed for a maximum of *CONTINUES* times, they disable them-

selves, and can only be re-enabled by *newround*, given that none of them has already found a factor. The listing for *newround* is given to the left in Fig. 2. Note that *newround* is disabled if the value of all variables i_m is greater than $n/2$. Each of the m factorisation events also disables itself if the corresponding i_m exceeds $n/2$. This is because a factor (less than n itself) cannot exist beyond this threshold. It would actually be enough to check numbers up till $\sqrt{n}$, but since Event-B does not support square root, we use $n/2$ as the limit.

newround $\triangleq$
when
$result\_1 \neq 0 \wedge result\_2 \neq 0$
$\neg(i\_1 > n/2 \wedge i\_2 > n/2)$
$continue\_1 < CONTINUES$
$\vee continue\_2 < CONTINUES$
then
$continue\_1 := CONTINUES$
$continue\_2 := CONTINUES$
end

found0 $\triangleq$
when
$result\_1 \neq 0$
$result\_2 \neq 0$
$result = -1$
$i\_1 > n/2$
$i\_2 > n/2$
then
$result := n$
end

Fig. 2. Events for re-enabling the factorisation events (left) and for finalising when it becomes clear that the number is prime (right).

In the case that no factorisation event finds a factor, and all i_m exceed $n/2$, event *found0* becomes enabled. This event is shown to the right in Fig. 2, and simply sets a variable *result*, storing the final result, to n. There are also events *found1*, *found2*, etc., related to the factorisation events *process1*, *process2*, etc., respectively. These events, as shown in Fig. 3, set the *result* variable to the value found by their associated factorisation events.

found1 $\triangleq$
when
$result\_1 = 0 \wedge result = -1$
then
$result := i\_1 - STEP$
end

found2 $\triangleq$
when
$result\_2 = 0 \wedge result = -1$
then
$result := i\_2 - STEP$
end

Fig. 3. Events for finalising when process 1 (left) or process 2 (right) has found a factor.

When benchmarking the model, in the following section, we also compare execution times to that of a sequential C++ program. Though designed to be as closely as possible a sequential version of the algorithm above, there are a few differences. For example, it obviously contains no synchronisation mechanisms, and it always finds the lowest factor that is ≥ 2, whereas the Event-B model may find a greater factor depending on the relative progress of the processes.

5 Benchmarking

Performance of the scheduling approach discussed in previous sections has been evaluated by scheduling the testbed model on a multi-core/multi-processor system using different parameters. The scheduler was compiled together with the C++ translation of the model using the GNU Compiler Collection (GCC) [4] with the maximum (O3) level of optimisation. Since some parameters were part of the model and could not be changed afterwards, we technically compiled different models with minor changes from each other. To facilitate scripting for benchmarking purposes, we also slightly modified the scheduler as well as the model code to support additional parametrisation. We do not expect these changes to have disrupted test results by having any relevant impact on performance. The system used for the test runs consists of two Xeon E5430 (2.66 GHz) processors, each of which has four computational cores, running a GNU/Linux operating system and the MPICH2 [8] implementation of MPI.

From the perspective of the scheduling platform, there are especially two parameters of interest: the number of slave processes and the value of *REPEAT* used in the scheduler. Important parameters related to the model are n, i.e. the number to factorise, and the value of the constant *CONTINUES*. Even though we will not mention it explicitly, the number of slave processes also has implications on the model in that the number of factorisation events has to match, and the value of *STEP* must be set accordingly. To keep the repeat cycles of the scheduler and the model synchronised, we decided to let the values of *REPEAT* and *CONTINUES* be bound to each other, and we will from now on refer to the value of $REPEAT = CONTINUES$ as c. For each set of parameters, we performed eight timed test runs. The initial one was disregarded, since it may not be comparable should subsequent executions have any caching benefits. The timings of the subsequent seven runs (numbered 1-7) were recorded, and the mean value was computed. The time unit used was seconds and fractions thereof.

Our first set of runs was performed with the parameter $n = 2,147,483,647$ with three slave processes. An additional process was used for the scheduler, so technically, the execution involved four processes. Note that we chose n to be a prime number in order to achieve benchmarking times long enough to draw conclusions. We ran several subsequent test sets, with the values of $c = REPEAT = CONTINUES$ being 10^2, 10^3, ..., 10^9, respectively. With the c value set to 100 (i.e. 10^2), the execution times are several times higher than that of the sequential program with a mean value of 13.36 seconds for the sequential version versus 89.08 seconds for the parallel one. However, if c is set to 1000, timings approach those of the sequential model, and with a c value of 10000, the parallel model is faster at 10.09 seconds on average. Values of c beyond 10^5 do not seem to provide further gains, and execution times level out at about 9 to 9.5 seconds, which constitutes approximately 70% of the running time of the sequential version. However, we also realise that execution times of only a few seconds may not necessarily be representative of performance in general. For example, the time taken to initialise the scheduling platform may have an unduly large impact. Therefore, we performed a new set of test runs with the

same parameters, except for setting the value of n to 68,720,001,023, which is also a prime number. We present the results in Table 1.

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
Sequential	498.63	427.11	549.03	448.51	555.63	567.38	516.91	509.03
Par. $c = 10^2$	2844.53	2883.50	2850.31	2802.62	2846.89	2840.90	2864.31	2847.58
Par. $c = 10^3$	544.07	555.74	542.31	560.58	557.56	552.23	550.53	551.86
Par. $c = 10^4$	320.24	320.19	319.80	320.67	320.69	317.67	318.21	319.64
Par. $c = 10^5$	292.76	293.95	293.61	293.13	294.09	292.09	292.34	293.14
Par. $c = 10^6$	288.50	290.00	290.34	288.24	290.16	290.23	288.50	289.42
Par. $c = 10^7$	288.32	286.88	288.05	288.52	289.86	296.57	288.13	289.48
Par. $c = 10^8$	289.03	287.51	289.40	288.18	287.32	286.79	287.81	288.01
Par. $c = 10^9$	288.06	288.29	287.24	288.24	287.97	288.34	287.68	287.97

Table 1. Test runs with 3+1 processes, $n = 68{,}720{,}001{,}023$.

The general pattern turned out to be the same as for the lower value of n. For a c value of 100, execution times are poor in this case, as well, but from $c = 10000$ and beyond, we see performance gains. While they also level out for higher values of c, executions times are around 50%-60% as compared to the corresponding sequential program. This it better than in the previous case. However, we were also interested in testing how the framework scales when the number of processes increases. Therefore, we did yet another set of test runs. We kept the value of n at 68,720,001,023, but increased the number of slave processes to six, in addition to the scheduling process, which is always present. The results are given in Table 2. Note that the sequential test runs used for comparison were not redone, since the value of n remained unchanged.

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
Sequential	498.63	427.11	549.03	448.51	555.63	567.38	516.91	509.03
Par. $c = 10^2$	2574.73	2074.18	2609.96	2647.20	2494.59	2577.47	2632.28	2515.77
Par. $c = 10^3$	338.11	319.87	347.55	335.40	324.07	348.34	346.58	337.13
Par. $c = 10^4$	159.96	137.46	165.59	141.58	160.85	158.15	153.70	153.90
Par. $c = 10^5$	147.62	146.77	147.72	147.02	121.49	148.72	146.45	143.68
Par. $c = 10^6$	113.44	144.23	136.24	145.56	145.35	145.51	145.68	139.43
Par. $c = 10^7$	145.03	145.50	134.56	146.20	129.59	145.29	146.63	141.83
Par. $c = 10^8$	119.44	146.29	144.89	134.04	145.75	139.15	130.20	137.11
Par. $c = 10^9$	140.61	120.94	139.71	138.91	140.97	142.09	141.35	137.80

Table 2. Test runs with 6+1 processes, $n = 68{,}720{,}001{,}023$.

While we see the same pattern as before, execution times are considerably shorter. The scenario where $c = 100$ is still highly inefficient, but it is nonetheless slightly faster than with three slave processes. We also note that for a value

of $c = 1000$, performance is now better than for the sequential comparison, whereas it was a bit slower than sequential in the 3+1 set-up. At $c = 10000$, and especially from $c = 10^5$, where the levelling out seems to start, performance is greatly increased as compared to using three slave processes. For such values of c, execution times in the 6+1 process set-up are around half of those in the 3+1 setting, indicating a good scalability of the scheduling approach.

6 Conclusions

In this paper, we have performed an empirical study on the efficiency of MPI-based parallel scheduling of compute-intensive code translated from an Event-B model. The purpose was to evaluate whether an on-the-fly scheduling approach taken is feasible from a practical perspective. We used an integer factorisation model as a testbed for the study. To mitigate excessive overhead due to the fine-grained nature of events in Event-B, we introduced an optimisation in the form of repeated event execution without the involvement of the scheduler. To benefit from this strategy, the model should be designed so that computational events are enabled a large number of times in a row.

We performed a number of test runs on a multi-core/multi-processor system to evaluate the performance of the testbed factorisation model when using the optimisation. The tests involved different numbers of processor cores in use, and different limits on how many times events can be executed consecutively without involving the scheduling process. The runs showed that given a large enough number of repetitions, the performance increased to a degree where the program clearly benefits from parallel execution, as compared to a corresponding sequential program. We also found that when increasing the cores in use from 3 slave processes + 1 scheduler, to a 6+1 configuration, performance increased considerably, indicating good scalability of the approach.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. J.-R. Abrial *et al.* An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.
3. F. Degerlund, R. Grönblom, and K. Sere. Code generation and scheduling of Event-B models. Technical report, Turku Centre for Computer Science (TUCS), 2011.
4. GNU Compiler Collection (GCC) web site. http://gcc.gnu.org/.
5. R. Grönblom. A framework for code generation and parallel execution of Event-B models. Master's thesis, Åbo Akademi University, 2009.
6. S. Hallerstede. On the purpose of Event-B proof obligations. *Formal Aspects of Computing*, 23:133–150, January 2011.
7. Message Passing Interface forum. http://www.mpi-forum.org/.
8. MPICH2 web site. http://www.mcs.anl.gov/research/projects/mpich2/.
9. S. Wright. Using EventB to create a virtual machine instruction set architecture. *Abstract State Machines, B and Z*, pages 265–279, 2008.