Fredrik Degerlund

Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B

Turku Centre *for* Computer Science

TUCS Technical Report
No 1051, August 2012

Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B

Fredrik Degerlund
Åbo Akademi University, Department of Information Technologies
Joukahainengatan 3-5, FIN-20520 Åbo/Turku, Finland
fredrik.degerlund@abo.fi

TUCS Technical Report
No 1051, August 2012

Abstract

Event-B is a tool-supported specification language that can be used e.g. for the 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. While constituting a straightforward method, it involves considerable communication overhead, a problem aggravated by the fine-grained nature of events in Event-B. 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.

Keywords: Parallel computing, Event-B, Scheduling, Message passing, Efficiency

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Event-B [2] 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 the modelling of concurrent systems. Tool support for Event-B has been achieved through the open-source Rodin platform [3, 4, 27], 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 straightforward approach that preserves the event nature has been proposed in [12, 18], 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 using a separate scheduler, which in turn deploys the MPI (Message Passing Interface) [24] 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. An additional benefit is that it also does not require the developer to take a stand on specific schedules and prove that they are compatible with the original model.

However, this approach has a potentially serious drawback in 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 the involvement of the scheduler. The repetitive approach is implemented as part of the scheduling platform, and we let a factorisation model serve as a testbed for benchmarking. This technical report constitutes an extended version of a previously published conference paper [11]. We here provide additional background information as well as a more detailed description of our research than in the original article.

The rest of the paper is structured as follows. We first discuss related work in Section 2. In Section 3, we present background information on the Event-B formalism to the extent needed for understanding this paper. We also discuss how the models can be translated into a programming language (C++). Section 4 is dedicated to concurrent scheduling of models. We also deal with communication overhead and propose a repeating approach to improve efficiency. In Section 5, we present the factorisation model that serves as the testbed for our study, whereas we in Section 6 discuss how the actual benchmarking takes place. We give a number of test configurations that we have used for the test runs, after which we present the resulting execution times as well as an interpretation thereof. Finally, we sum up the paper and draw conclusions in Section 7.

2 Related Work

Unlike the classical B method [1], which focuses on a *correct-by-construction* approach, Event-B [2] was designed with system-level modelling in mind, but it can also be used for pure software development. The formalism has its roots in B Action Systems [30], based on the Action Systems formalism [6], which has been used e.g. for the derivation of parallel algorithms [28]. As a result, Event-B is also suitable for modelling of parallel software. The use of Event-B is facilitated by the Rodin platform [3, 4, 27], which provides tool support for the formalism. Rodin is based on the Eclipse framework [14], and custom plug-ins can also be used in the platform to provide additional functionality.

The Event-B scheduling approach we evaluate in the paper is based on [12, 18], which in turn has its roots in [13]. It is superficially related to the concept of animation as in the ProB [22], AnimB [25] and Brama [29] plug-ins for the Rodin platform. However, animation can be seen as a supplementary methodology during the modelling and development stage, while we (as in [12, 13, 18]) use automated scheduling as a means of executing the final code generated from the model. Furthermore, parallelism is typically not supported in animation, since the primary goal of animation is to analyse models instead of achieving efficient execution.

Another approach to scheduling of Event-B models has been taken in papers [21], [19] and [8]. The basic idea is to provide the models with explicit (sequential) control flow information expressed in dedicated scheduling languages. The developer then has to prove that the desired control flow is correct with respect to the typical Event-B behavioural semantics discussed in Section 3. This kind of scheduling can also be extended to handle parallelism [9]. However, an important difference as compared to the method we study is that scheduling decisions are taken and proven correct by the developer at the modelling stage. The approach we explore can instead be regarded as *on-the-fly* scheduling, where the scheduler takes scheduling decisions during *run-time* based on the current state. This eliminates the need for explicit schedule design and associated proofs, but may, on the other hand, induce a performance penalty.

A means of scheduling is also proposed in [23] for use with code obtained by the Event-B to C/C++/C#/Java code generator EB2ALL, which the authors present in the paper. However, to our knowledge, it supports only sequential execution, and therefore operates in a setting different from the one we consider here. An approach that does support parallelism is given in [15], where Event-B models are translated into Java for concurrent execution. The schedules are expressed by the model developer in a dedicated language called OCB (Object-oriented Concurrent B). In that sense, it bears similarities to the developer-scheduled approaches discussed above, in contrast to an on-the-fly approach. The method has also more recently been adapted [16] for use with the Ada language.

3 Event-B and Code Generation

3.1 The Event-B Formalism

Models in Event-B 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. The invariant $I(v)$ is used to assign types to the variables, as well as to restrict the valid state space. Consequently, the initialisation event must *establish* the invariant, whereas the rest of the events must *preserve* it.

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 [20]:

$E \triangleq$ **when** $G(v)$ **then** $v :| A(v, v')$ **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 event is an exception, containing only an action but no guard. It also does not depend on a previous state.

Refinement [5, 7, 31] is a key concept in Event-B, enabling models to be developed in a stepwise manner. The idea is to achieve a chain of models, beginning from an abstract one and gradually turning it into more concrete ones. For each step, it must be shown that the new model is correct with respect to the previous one. We omit a detailed description of refinement in this paper, since we only focus on the last refinement step, which is the one to be converted to program code.

Event-B does not mandate any specific behavioural semantics. Instead, it defines a number of proof obligations, and any semantics compatible with them can be used. Typically, the same behavioural semantics as in the Action Systems formalism is deployed, and that one has also been used in this paper. First, the initialisation event is executed, after which the rest of the execution can be thought of as the events of the machine residing inside a loop. In each iteration, any enabled event is non-deterministically chosen for execution, and the loop only terminates

when no event is enabled any longer. This can be interpreted as a deadlock situation in control systems, but for the input-output focused models we are interested in, it corresponds to termination.

3.2 Code Generation

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 [32], 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 [12, 18]. 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 Section 5) we benchmark in this paper (Section 6) is based upon a model originally used in [12, 18], 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. Even though we here rely on manually generated code for evaluation of the scheduling approach, the process is time-consuming and error-prone. Due to the latter, in particular, manual translation may negate the correctness benefits of formal methods and does not constitute a realistic option for use in industrial projects. A possible path forward would be further development of the code generator of [12, 18]. An alternative approach would be to use the translation tool EB2ALL, even though adaptations would have to be made for the resulting code to be in a form compatible with the desired scheduling as discussed in the next section.

4 Scheduling

4.1 Scheduling Platform

When an Event-B model has been translated into C++ code, a means of scheduling the resulting code is required. Since we in this paper are interested in evaluating the viability of run-time scheduling, we need a scheduling platform that can invoke the methods that have been translated from the events. A prototype version of such a scheduler, called ELSA, was developed in [13] for running code

generated from the Atelier B tool [10] when used for developing B Action Systems. The goal was to be able to execute the code of compute-intensive models in parallel on a multi-processor computer or a cluster using the MPI framework. In [12, 18], ELSA was adapted for use with code translated from Event-B models using a plug-in developed as part of the same research. We use this Event-B compatible version of ELSA for the evaluations performed in this paper, but 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 in Section 4.2.

The scheduler code, which is written in C++, technically runs as part of both the scheduling process and a number of slave processes. The code takes a separate execution path on the scheduling process than on the event-executing slave processes, reflecting the different roles they play. 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 scheduling process is 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. Enabledness is easy to check, since the guards are translated as boolean functions separate from the event actions. Though much simplified, the workings of the scheduling process can be explained as checking events for enabledness and delegating them for execution to slaves that are currently not processing any other events. This takes place until no events are enabled, whereby the scheduler terminates execution. A more detailed description of the scheduling algorithm can be found in [12, 18].

4.2 Repeated Execution of Events

The scheduler in its basic form, as described above, has a practical problem that needs to be tackled. After initial testing, it became evident that the overhead involved outweighs the benefits that parallelism can provide, resulting in poor execution times. The heart of the problem is not only the overhead in itself, but it becomes particularly problematic when combined with the fine-grained nature of Event-B events (or the corresponding C++ code). Events cannot contain structures such as sequential composition or loops, and complex behaviour instead has to be modelled in an alternative way, such as by repeated execution of events.

The scheduling approach above would imply that if an event is executed several times in a row, the scheduling processes would be involved in every invocation, resulting in excessive overhead. For this reason, we have amended the scheduling platform 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 slave 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. In fact, the repetition mechanism has no impact on the enabling/disabling of events, and it operates within the limits of the behavioural semantics as described in Section 3.1.

5 Testbed Model

A suitable testbed model for our study should be compute-intensive, easily parallelisable, convenient to express, and, for generality, as representative as possible of how other high performance computation models would be expressed in Event-B. The generality of the model is particularly important, since our goal is to draw as universal conclusions as possible on the viability of the scheduling approach. We find that an integer factorisation example given in [12, 18] for the most part fulfils these requirements. However, since we have made improvements to the scheduling approach as compared to [12, 18], especially by introducing repetition of events, we have also revised the model accordingly.

The goal of the model is to find a factor of a given integer n, such that it is greater than or equal to 2 and less than n. However, if n is a prime number, the result reported will be n itself. The approach we take is based on trial division. While there are much more sophisticated factorisation algorithms available, they are not as straightforward, resulting in models much more difficult to follow and evaluate. We are also not primarily interested in evaluating the efficiency of the algorithm *per se*, but rather that of the scheduling method.

At the core of the model are the factorisation events *process1*, *process2*, etc., up till the number of computational 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 Figure 1. Note that we use separate events instead of parametrisation, since we want the factorisation events to be separate from each other. It was also of utmost importance that the model be

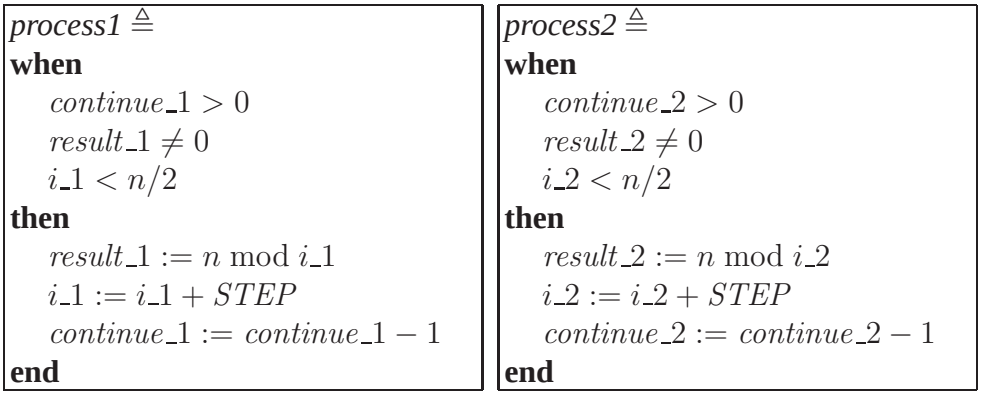

Figure 1: Factorisation events for two computational processes.

expressed in such a way that the factorisation events have no variables in common, since the scheduler would otherwise be unable to run them in parallel. They may, nevertheless, refer to the same constants.

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 in the model. In addition to the variable i_m, 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 themselves, and can only be re-enabled by *newround*, provided that none of them has already found a factor. The listing for *newround* is given to the left in Figure 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.

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 Figure 2, and it 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 Figure 3, set the *result* variable to the

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

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

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

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

value found by their associated factorisation events. Note that even though the final result has been found once *found0* or any of the *found1*, *found2*, etc. events has been executed, there is a possibility that one or several of the factorisation events may still be executed several times afterwards. This undesired behaviour is a side effect of the independence of events, and it is aggravated by setting the *CONTINUES* constant to a large value. The choice of value for *CONTINUES* is, however, a trade-off, since setting it to a value that is too small results in excessive synchronisation by the *newround* event.

In Figure 4, we give a sequential C++ function designed to perform factorisation similarly to the model presented above. A program based on the function is used as comparison in Section 6 when evaluating the efficiency of the parallel model. Though designed to resemble as closely as possible a sequential version of the algorithm above, there are a number of differences. For example, since the program is sequential, it obviously contains no synchronisation or other process-related mechanisms, resulting in much simpler code. The sequential version also always finds the lowest factor greater than or equal to 2, whereas the Event-B model may find a greater factor depending on the relative progress of the processes.

```
long long factor(long long n) {
  long long i = 1;
  long long res = -1;
  while(i < n/2 && res != 0) {
    i++;
    res = n % i;
  }
  if(res == 0) return i; else return n;
}
```

Figure 4: The C++ function for sequential factorisation used as comparison.

6 Benchmarking

6.1 Approach

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) [17] 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 [26] implementation of MPI. While the *numerical* results will be dependent upon factors such as the clock frequency of the processors, instruction set architecture, performance of the system memory, etc., we believe that the *interpretation* of the results is representative of modern computer systems with similar topology (e.g. the same number of processor cores). This is because we are mainly interested in the overall feasibility of the scheduling framework and the impact of different parameter values. Since all our test runs, including the comparison with a sequential program, have been done on the same system, the results are mutually comparable to each other.

6.2 Parameters and Results

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

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
Sequential	13.37	13.36	13.36	13.36	13.36	13.36	13.36	13.36
Par. $c = 10^2$	91.42	85.52	90.45	92.00	85.37	91.34	87.43	89.08
Par. $c = 10^3$	16.57	16.59	16.78	16.63	16.94	16.55	16.07	16.59
Par. $c = 10^4$	10.23	10.28	9.92	10.17	9.57	10.18	10.26	10.09
Par. $c = 10^5$	9.29	8.90	9.31	9.55	9.38	8.68	9.60	9.24
Par. $c = 10^6$	9.19	9.46	8.13	9.13	8.82	9.19	9.24	9.02
Par. $c = 10^7$	9.31	8.47	9.44	9.15	9.18	9.32	9.33	9.17
Par. $c = 10^8$	9.26	9.48	9.47	9.46	9.49	9.58	9.38	9.45
Par. $c = 10^9$	9.51	9.44	9.54	8.70	9.46	9.51	9.30	9.35

Table 1: Test runs with 3+1 processes, n = 2,147,483,647.

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 from now on, 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. Furthermore, we decided to keep the values of *REPEAT* and *CONTINUES* bound to each other, even though it would not absolutely have to be that way. We motivate our decision as follows. The value of *REPEAT*, being a property of the scheduler, may have an impact on the performance of execution, but it does not change the logics of the model. In contrast, *CONTINUES* is part of the model, which is nevertheless constructed to produce a correct result for different values of *CONTINUES*. A value of *REPEAT* less than *CONTINUES* would imply that there may be unnecessary involvement of the scheduler even in cases where the slave processes could have been repeatedly executed events on their own. Since the model is not aware of the impact of the repetition mechanism of the scheduler, though interrupted, it would not even have a chance of synchronising by executing the *newround* event. A *REPEAT* value greater than *CONTINUES* is also not motivated, since repeated execution of the factorisation events would be limited by *CONTINUES* anyway.

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 executions (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. The results are shown in Table 1.

As can be seen in Table 1, with the c value set to 100 (i.e. 10^2), the execution

	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 2: Test runs with 3+1 processes, n = 68,720,001,023.

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. It can be explained by overhead that, in this case, is clearly not outweighed by the potential benefits of parallelism. This is apparently the case even though the slave processes may allow the factorisation events to be executed up to 100 times without involving the scheduling process. The overhead may in part be due to MPI communication, but also behaviour specific to the parallel model, such as the *newround* event, may have an impact. 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 2.

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, execution times are around 50%-60% as compared to the corresponding sequential program. This is 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 3. Note that the sequential test runs used for comparison were not redone, since the value of n remained unchanged.

While we see the same pattern as before, execution times are considerably

	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 3: Test runs with 6+1 processes, n = 68,720,001,023.

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 c value of 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.

7 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. The main pitfall we suspected in the basic form of the framework was that the overhead of the scheduler and the MPI library communication would defeat the potential speed gains of parallelism. This is because individual events in Event-B are typically very fine-grained.

In an attempt to mitigate excessive overhead, we introduced an optimisation in the form of repeated event execution without the involvement of the scheduler. A benefit of this solution is that it directly reduces the communication overhead. The repetitive behaviour introduced is compatible with the original behavioural semantics typically used in Event-B, and can therefore be considered correct from a theoretical point of view. 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. This indicates a good scalability of the approach. In conclusion, the empirical study we have performed hints at a potential practical applicability of the run-time scheduling framework in question.

Acknowledgements

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

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[4] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer Berlin / Heidelberg, 2006.

[5] R.J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer Berlin / Heidelberg, 1990.

[6] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *Proc. of the 2nd ACM SIGACTS-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[7] R.J.R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer Berlin / Heidelberg, 1990.

[8] P. Boström. Creating sequential programs from Event-B models. In *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM 2010)*, pages 74–88. Springer-Verlag, 2010.

[9] P. Boström, F. Degerlund, K. Sere, and M Waldén. Concurrent scheduling of Event-B models. In *Proceedings 15th International Refinement Workshop*, pages 166–182, June 2011.

[10] ClearSy. Atelier B web site. `http://www.atelierb.eu/` .

[11] F. Degerlund. Scheduling of compute-intensive code generated from Event-B models: An empirical efficiency study. In K. Göschka and S. Haridi, editors, *Proceedings of Distributed Applications and Interoperable Systems (DAIS) 2012*, volume 7272 of *Lecture Notes in Computer Science*, pages 177–184. Springer Berlin / Heidelberg, 2012.

[12] F. Degerlund, R. Grönblom, and K. Sere. Code generation and scheduling of Event-B models. Technical Report 1027, Turku Centre for Computer Science (TUCS), 2011.

[13] F. Degerlund, M. Waldén, and K. Sere. Implementation issues concerning the action systems formalism. In *Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'07)*, pages 471–479. IEEE Computer Society Press, 2007.

[14] Eclipse platform. `http://www.eclipse.org/` .

[15] A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, 2010.

[16] A. Edmunds and M. Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. In *PLACES 2011*, 2011.

[17] GNU Compiler Collection (GCC) web site. `http://gcc.gnu.org/` .

[18] R. Grönblom. A framework for code generation and parallel execution of Event-B models. Master's thesis, Åbo Akademi University, 2009.

[19] S. Hallerstede. Structured Event-B models and proofs. In *Abstract State Machines, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 273–286. Springer-Verlag, 2010.

[20] S. Hallerstede. On the purpose of Event-B proof obligations. *Formal Aspects of Computing*, 23:133–150, January 2011.

[21] A. Iliasov. On Event-B and control flow. Technical Report CS-TR-1159, School of Computing Science, Newcastle University, 2009.

[22] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *International Journal of Software Tools for Technology Transfer (STTT)*, 10:185–203, February 2008.

[23] D. Méry and N. K. Singh. Automatic code generation from Event-B models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188. ACM, 2011.

[24] Message Passing Interface Forum. `http://www.mpi-forum.org/` .

[25] C. Métayer. AnimB web site. `http://www.animb.org/` .

[26] MPICH2 web site. `http://www.mcs.anl.gov/research/projects/mpich2/` .

[27] Rodin platform web site. `http://www.event-b.org/` .

[28] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, Åbo Akademi University, 1990.

[29] T. Servat. BRAMA: A new graphic animation tool for B models. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 274–276. Springer Berlin / Heidelberg, 2006.

[30] M. Waldén and K. Sere. Reasoning about Action Systems using the B-Method. *Formal Methods in System Design*, 13:5–35, May 1998.

[31] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, April 1971.

[32] S. Wright. Using EventB to create a virtual machine instruction set architecture. *Abstract State Machines, B and Z*, pages 265–279, 2008.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2765-3
ISSN 1239-1891