Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2018,5
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Enabling Cross-Event Optimization in Discrete-Event Simulation Through Compile-Time Event Batching

Marc Leinweber, Hannes Hartenstein and Philipp Andelfinger

2018

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Enabling Cross-Event Optimization in Discrete-Event Simulation Through Compile-Time Event Batching

Marc Leinweber
and Hannes Hartenstein
Karlsruhe Institute of Technology
Karlsruhe, Germany

Philipp Andelfinger
Nanyang Technological University
Singapore

***Abstract*—A discrete-event simulation (DES) involves the execution of a sequence of event handlers dynamically scheduled at runtime. As a consequence, a priori knowledge of the control flow of the overall simulation program is limited. In particular, powerful optimizations supported by modern compilers can only be applied on the scope of individual event handlers, which frequently involve only a few lines of code. We propose a method that extends the scope for compiler optimizations in discrete-event simulations by generating batches of multiple events that are subjected to compiler optimizations as contiguous procedures. A runtime mechanism executes suitable batches at negligible overhead. Our method does not require any compiler extensions and introduces only minor additional effort during model development. The feasibility and potential performance gains of the approach are illustrated on the example of an idealized proof-of-concept model. We believe that the applicability of the approach extends to general event-driven programs.**

I. Introduction

In discrete-event simulations (DES), events are executed one after the other in the order of their time stamps. Due to the stochastic nature of most simulation models, the execution order is not known a priori. In particular, the execution order is not available during compilation. Modern compilers for languages such as C++ support powerful optimizations to reduce execution times and executable file sizes [1] such as removal of redundant computations, instruction reordering to improve branch prediction and hide memory access latencies, or inline expansion of function calls to reduce call overheads. Unfortunately, the unpredictable control flow of discrete-event simulations limits the scope of such optimizations to individual event handlers. As an example, consider a situation where an event reverses the state changes performed by its preceding event. Although the execution of the two events has no effect, the situation cannot be detected by the compiler and this unnecessary computation is performed in full.

In this paper, we propose a general method to extend the scope for compiler optimizations in discrete-event simulations implemented in C++ beyond individual events. The approach can be applied both to sequential and parallel simulations. Although our implementation relies on C++ templates, the method only requires suitable metaprogramming facilities that are available in a number of programming languages.

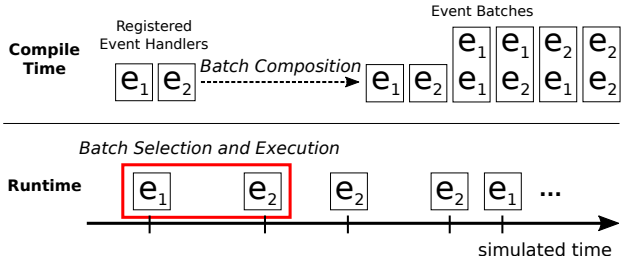

Fig. 1. Overview of the proposed approach: during compilation, batches are composed from registered event handlers. At runtime, batches are identified in the set of pending events and executed as a unit.

The approach is illustrated in Figure 1. During compilation, all combinations of event handlers ("batches") up to a configurable length are composed automatically using metaprogramming constructs. Each batch is a simple concatenation of the code associated with the individual event handlers. The benefit of the approach is that batches can be considered for compiler optimizations as contiguous code fragments, substantially improving the scope for optimizations. During runtime, appropriate batches are selected and executed in place of the original sequence of event handlers. The length of the executed batches varies, since similarly to parallel and distributed simulations [2], dependencies between events must be respected. Model knowledge is applied to guarantee that batches are executable without violating the correctness of the simulation. We make our prototypical implementation of the approach available to the community [1].

The remainder of the paper is organized as follows: in Section II, we discuss related work. In Section III, we present a method to compose event batches at compile time and a mechanism to execute suitable batches at runtime. In Section IV, we evaluate the approach w.r.t. the effects on compilation and runtime performance, as well as the impact on model development. With Section V the results are summarized and the paper is concluded.

[1] https://github.com/batched-DES/prototype

II. Related Work

Some previous works have considered events as batches to reduce simulation runtimes. However, in contrast to our work, the existing approaches have focused on parallelized simulations and have not considered enabling compiler optimizations across events.

In optimistically synchronized parallel and distributed simulation, a rollback mechanism is required to undo erroneous event executions. Several algorithms have been proposed to reduce the scale and overhead of rollbacks [2]. Zeng et al. proposed an approach to roll back batches of events at once. This is achieved by maintaining sufficient information at each processor so that during a rollback, a correct state can be achieved by rolling back a batch of events locally without inter-processor communication [3].

In 2017, Gupta and Wilsey [4] performed experiments on executing batches of events in optimistically synchronized simulations using the framework *Warped2* to decrease the contention created by concurrent accesses to the shared data structure holding future events. Comparing the performance under different policies for executing events as batches, the authors report a speedup of up to 2.5 over executing events one after the other.

The idea of merging computations into a single step has been explored in general HPC ("superblock technique" [5]) and in the context of computations on graphics cards ("thread coarsening" [6] and "kernel fusion" [7]). In contrast to these works, our approach addresses the challenges given by the limited predictability of the order and dependencies among computations in DES.

III. Proposed Method

In the following section we propose a method for batch composition and scheduling. The main steps of the approach are as follows:

- **Enumeration and composition** of all combinations of event types up to a predefined length during compilation.
- **Selection and execution** of the correct batches during runtime depending on the set of pending events without violation of the causality constraint (non-decreasing time stamp order of the executed events).

The batches of event handlers are composed during compilation. The input for this process is the set of event handling functions which are registered in an array by the modeler. In Section III-A, a compile-time algorithm for the batch composition is described. We developed an event scheduler that maintains the execution order when executing batches instead of single events. Events are extracted as long as it can be guaranteed that the causality constraint is not violated. Subsequently, the corresponding batch is selected and executed. In Section III-B, the scheduling method is presented.

The event batches are constructed during compilation through compile-time metaprogramming using C++ templates. Metaprograms are programs that manipulate executable code [8]. Originally, C++ templates were intended as a mechanism for generic programming by allowing templated functions to be *instantiated* for a specific data type during compilation [9]. However, in 2003 Veldhuizen showed that C++ templates are Turing complete [10].

A. Batch Enumeration and Composition

As mentioned above, an event batch is a concatenation of $n \in \mathbb{N}$ event handlers. To be able to execute the composed batches during runtime, the batches have to be identified uniquely. Hence, a system is needed that constructs identifiers based on the event types that contribute to a batch. In this Subsection, we present an algorithm that enumerates all possible batches up to a configurable length and that uses number system transformation based on a variation of the Horner scheme [11]. We have chosen this approach to achieve a clean implementation and a scheme that can be efficiently evaluated both during runtime (cf. Sec. III-B) and compile-time.

The set of event handlers can be interpreted as the characters of an alphabet Σ. The resulting formal language of all event handler batches is $L = \Sigma^*$, where $*$ is the Kleene closure. For instance, the Kleene closure for the alphabet $\Sigma = \{a, b\}$ starts with $\{\epsilon, a, b, aa, ab, ...\}$, where ϵ is the empty string. Since formal languages are recursively enumerable [12], a bijection f can be found between N and L. If the cardinality $|\Sigma|$ of Σ is interpreted as the base of a number system, a word of Σ^* is a representation of a natural number in the system with base $|\Sigma|$. The characters of the alphabet Σ are interpreted as digits of a number system. However, if the first character a corresponds to the digit 0, it has no effect on the batch identifiers (aba will have the same id as ba). Hence, we introduce an explicit character ν signifying "no event". However, by including the ν-event, redundant batches are generated. If $\Sigma = \{a\}$, it has to be expanded to $\Sigma_\nu = \{\nu, a\}$. If additionally $n = 2$, the words of Σ_ν^* with maximum length 2 are $\nu, a, \nu\nu, \nu a, a\nu, aa$. Obviously, the codes of the batches a, νa and $a\nu$ are equivalent and thus redundant (cf. Sec. IV). Let $|\Sigma_\nu|$ be the number of event handlers (including the ν-event) and n the maximum batch length. Then $B = \sum_{i=1}^{n} |\Sigma_\nu|^i$ event batches exist.

We assume that during model development, function pointers to all event handlers have been added to a constant array. Now, during compilation, all batch identifiers up to B are enumerated by recursive evaluation of template functions. For each enumerated identifier a template function is invoked that transforms the identifier to the different event types and that batches the corresponding event handling functions in the correct execution order. Function pointers to the composed batch handlers are stored in an array to enable the runtime mechanism (cf. Sec. III-B) to address and execute batches at runtime.

App. A lists the pseudocode for the meta program described above.

B. Batch Selection and Execution

To maintain correct simulation results, a batch should only be executed if the contained sequence of events cannot be

affected by its execution. Here, as in conservatively synchronized parallel and distributed simulations, we require model knowledge to define a *lookahead* [2], i.e., a minimum delta between an event's execution and creation time stamps.

At runtime, our batch scheduler uses a dynamic lookahead window. We assume that a lookahead value is associated with each event type. Events are extracted one by one while their execution time lies within the dynamic lookahead window.

In Fig. 2, the consideration of lookahead during batch extraction is illustrated. We iterate over the future events and compute the minimum of the sum of the events' respective time stamp and lookahead. Once an event's time stamp is larger than the current minimum, the batch extraction terminates. In the figure, the batch $e_1e_2e_3$ is executed. In effect, if t_e is the execution time of event e and l_e is the lookahead of e according to its type, we compute $t_{max} = \min_{e \in E}(t_e + l_e)$ where E is the set of future events up the configured maximum batch length. Following Section III-A, the event type digits contribute, depending on their position in the batch, to the batch's identifier which then is used to execute the before composed batch. In addition to the lookahead window, the number of events in a batch is limited by the configured maximum batch length, which is a tuning parameter.

IV. Evaluation

In the following, we demonstrate the feasibility of the approach by showing successful cross-event compiler optimization and the associated speedup for a synthetic simulation model. Subsequently, we evaluate the increase in compile times incurred by the batching process. Finally, we discuss the deployability as well as limitations and potential improvements of our approach.

A. Proof of Concept

We evaluate our approach using a synthetic simulation model that performs redundant computations across events, providing substantial opportunities for cross-event compiler optimizations. The model is based on two event types: as a computationally intensive event, the `Increment` event performs a million iterations of `sum += sum + 1` on the global variable `sum`. The `Set` event sets the global `sum` variable to the constant value 10. For simplicity, neither of the event types schedules new events. However, our approach poses no limitations on event scheduling at runtime. Since the execution of a `Set` event after an `Increment` event renders the computation of the `for`-loop obsolete, the compiler should remove the loop from batches with this sequence of events entirely.

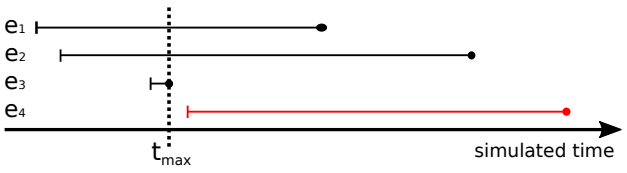

Fig. 2. Dynamic lookahead window (vertical line): according to the next events' time stamps and lookaheads, t_{max} allows for the execution of a batch comprised of $e_1e_2e_3$.

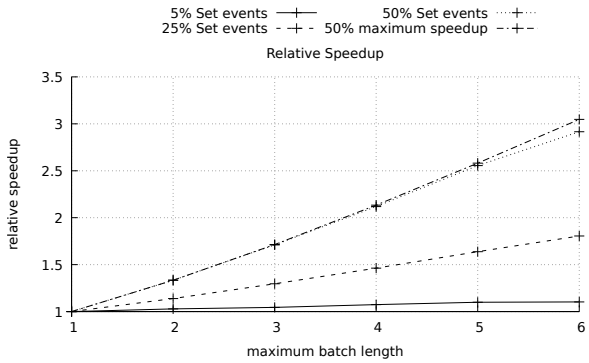

Fig. 3. Speedup by event batching for synthetic proof-of-concept model.

A similar sequence of computations could be given in a model of wireless communication: suppose a node in a simulated network periodically broadcasts messages to nearby receivers. The successful reception depends on whether the receiver is in a power-saving state. If none of the nearby nodes is ready to receive, the computations involved in the creation of the message could be avoided entirely. If a sequence of events in a batch makes it impossible for the results of the message creation to be used, the compiler could remove the message creation code.

The model was compiled with `clang++` version 3.8.0 on an `Intel Core i5-6600K CPU @ 3.5GHz` with 32GB of main memory running Ubuntu 16.04.3.

For an examination of the generated assembly code, we set the maximum batch length to 2 and thus composed $\frac{1-(2+1)^3}{1-(2+1)} - 1 = 12$ batches. The examination confirmed the successful cross-event optimization in this proof-of-concept example. As expected, when an `Increment` event is not followed by a `Set` event within the same batch, the compiler generates assembly code corresponding to the `Increment` event. However, if the `Increment` event is followed by a `Set` event, the loop is omitted entirely, leaving only the assignment of the `Set` event.

B. Speedup

We tested 24 different configurations, varying in the maximum batch length and the proportion p_s of `Set` events. We set p_s to 5%, 25%, 50%, and 75%.

For each simulation run, 1 000 000 initial events were scheduled. No new events were scheduled during the simulation. We scheduled one event at each integer time step. The lookahead was set to 1 000 000 units of time, i.e., all executed batches had the maximum batch length.

Each configuration was run 20 times. For each run, a different seed for the pseudo-random number generator was used. In Fig. 3, the result is plotted. The achieved speedup depends on the chance that the computation-intensive `Increment` event may be omitted.

We can observe that in an idealized case, event batching achieves substantial speedup. Assuming that the impact of `Set`

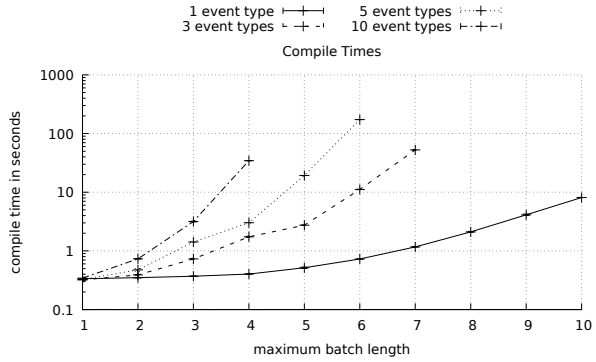

Fig. 4. Effect of event batching on compile times.

events on the runtime is negligible, given a fixed batch length n, the proportion $p_i = 1 - p_s$ of `Increment` events, the maximum possible speedup $s_{\max}$ can be calculated based on the expected number of `Increment` events in each batch, and the probability that an `Increment` event at a certain position in the batch will not be followed by any `Set` events: $s_{\max} = np_i(\frac{1-{p_i}^{n+1}}{1-p_i} - 1)^{-1} = \frac{n(1-p_i)}{1-{p_i}^n}$ (see App. B for a proof). In Figure 3, $s_{\max}$ is plotted for $p_s = 0.5$. With increasing batch lengths, $s_{\max}$ approaches $n/2$. Our performance measurements closely approximate the maximum possible speedup.

To investigate the overhead incurred purely by the runtime batch selection, we compared the runtime with a one-by-one event execution as in common sequential discrete-event simulators. When executing m `Set` events in the unbatched case and with our batching approach, we observed that the runtime batch selection adds overhead of about 5% at an average batch length of 2.

C. Compilation

To benchmark the compile-time algorithm (cf. Sec. III-A), four configurations, varying in the number of different event types, were compiled 15 times. Once the compilation time for a single configuration exceeded 240 seconds, the compilation was stopped. The compile times are plotted in Fig. 4. As expected from the increase in the batch count, the compile times increase exponentially. With ten different event types and a maximum batch length of 5, the compilation time exceeds 240 seconds drastically.

Although these numbers seem enormous, large C++ projects like simulation models can easily reach compile times in ranges of several minutes. As long as the number of event types is small, the relative impact on compilation times may be acceptable. Further, the compilation times can be reduced by choosing a smaller maximum batch length. Additionally, as discussed in Section III-A, due to the need for a ν-event, a substantial number of batches will never be used by the scheduler. Overall, $(\frac{1-(|\Sigma|+1)^{n+1}}{1-(|\Sigma|+1)} - 1) - (\frac{1-|\Sigma|^{n+1}}{1-|\Sigma|} - 1)$ redundant batches are composed. As an example, with five different event types and a maximum batch length of five, 9331 batches (i.e., 58%) are redundant. In the future, a refined enumeration scheme could eliminate these redundant batches.

D. Discussion

In this section, we first state a number of limitations and technical aspects of the batching approach. Subsequently, we discuss potential avenues for future work.

Overall, the approach imposes only minor restrictions on the model development. A marginal burden on modelers is given by the assumption made in the batch composition that function pointers to all event handlers are initially stored in an array.

A key avenue for future work lies in reducing the overhead during compilation and at runtime. Major opportunities for improvements are given within the batch composition process: since a large proportion of the composed batches are redundant, an improved enumeration scheme could reduce the number of batches.

During the execution of a batch, each event may generate new events immediately. By postponing the scheduling of all new events to the end of a batch execution, it may be possible to improve performance through a reduction in accesses to the data structure holding the scheduled events.

Currently, the runtime mechanism executes batches conservatively, i.e., never violating the simulation correctness. If the batch lengths achievable in this manner are small, a speculative approach could improve performance. Similarly to optimistically synchronized parallel simulations [2], if the execution of a batch generates new events with time stamps earlier than the last event in the batch, a rollback mechanism could restore a correct simulation state.

Whether significant opportunities for cross-event compiler optimizations exist in real-world models is still to be investigated. For instance, in real-world simulation models, the assignment of events to simulated entities is typically only known at runtime. Thus, the compiler may not be able to determine whether a variable change by an event will be overwritten by a successor event. It may be possible to formulate implementation guidelines for simulation models to maximize opportunities for optimizations.

A wide range of modern applications is implemented in an event-driven manner, i.e., events representing computational tasks are dynamically scheduled and extracted according to their priorities. We believe that the proposed batching approach is generic enough to be applied to event-driven applications beyond simulations.

Finally, since our approach relies on C++ template metaprogramming, the feasibility of the approach beyond C++-based simulators should be explored. Generally, the approach requires compile-time evaluated metaprogramming facilities that support code transformation, code generation and reflection. However, reflection is not needed if constructs such as function pointers exist. Beyond ahead-of-time compilation, modern just-in-time compilers achieve remarkable optimization results through runtime profiling. Hence, just-in-time compilation could focus on the creation of relevant batches according to the observed runtime behavior of the considered simulation model.

V. Conclusion

We presented an approach that extends the scope of compiler optimizations in discrete-event simulation beyond individual events. Using C++ template metaprogramming, all possible sequences of events up to a configurable sequence length are composed at compile time. A runtime mechanism selects and executes event batches, relying on model-specific temporal properties to maintain correctness. We showed that the approach is feasible, achieves substantial speedup in a proof-of-concept example and does not add immense runtime overhead. Only minor additional effort is required by modelers to apply the approach. The main limitation is a substantial increase in compilation times and executable sizes, both of which may be improved on in future work by an advanced batch composition approach. Further, we consider experiments with real-world discrete-event simulation models and general event-driven applications the most interesting avenues for further exploration.

References

[1] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern Compiler Design*. Springer Science & Business Media, 2012.

[2] R. M. Fujimoto, "Parallel and distributed simulation systems," in *Proceedings of the Winter Simulation Conference*. IEEE, 2001, pp. 147–157.

[3] Y. Zeng, W. Cai, and S. J. Turner, "Batch based cancellation: a rollback optimal cancellation scheme in time warp simulations," in *Proceedings of the Workshop on Parallel and Distributed simulation*. ACM, 2004, pp. 78–86.

[4] S. Gupta and P. A. Wilsey, "Quantitative driven optimization of a time warp kernel," in *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2017, pp. 27–38.

[5] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab *et al.*, "The superblock: an effective technique for vliw and superscalar compilation," *Instruction-Level Parallelism*, pp. 229–248, 1993.

[6] A. Magni, C. Dubach, and M. F. O'Boyle, "A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–11.

[7] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU Computing Gems Jade Edition*, vol. 2, pp. 359–371, 2011.

[8] R. D. Cameron and M. R. Ito, "Grammar-based definition of metaprogramming systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, pp. 20–54, 1984.

[9] B. Stroustrup, *The C++ Programming Language*, 4th ed. Pearson Education, 2013.

[10] T. L. Veldhuizen, "C++ templates are turing complete," *Available at citeseer. ist. psu. edu/581150. html*, 2003.

[11] W. G. Horner, "A new method of solving numerical equations of all orders, by continuous approximation," *Philosophical Transactions of the Royal Society of London*, vol. 109, pp. 308–335, 1819.

[12] I. M. Chiswell, *A Course in Formal Languages, Automata and Groups*. Springer Science & Business Media, 2009.

Appendix A
Batch Composition Meta Program Pseudo Code

In Algorithm 1 the meta program pseudo code for the batch composition procedure, which takes place during compilation, is given. The function enumerateBatches() enumerates all possible batch IDs in a recursive manner. For each ID genBatch() is invoked where the different event types are identified with the modified Horner's scheme. The event types' handling functions are concatenated with appendFuncCall() into a single function body and the pointer to the resulting function is stored.

Algorithm 1 Batch Composition Metaprogram

$eventHandlers$: Array of $Functionpointer$
$batchedHandlers$: Array of $Functionpointer$
$maxBatchSize : \mathbb{N}$
$eventTypeCount = sizeof(eventHandlers) : \mathbb{N}$
$batchCount = \frac{1-eventTypeCount^{maxBatchSize+1}}{1-eventTypeCount} - 1 : \mathbb{N}$

function GENERATEBATCHES
  $enumerateBatches(0)$

function ENUMERATEBATCHES($batchID : \mathbb{N}$)
  if $batchID = batchCount$ **then return**
  $pointer =$ pointer to new empty function : $Functionpointer$
  $batchedHandlers[batchID] := genBatch(batchID, pointer)$
  $enumerateBatches(batchID + 1)$ ▷ enumerate all indexes

function GENBATCH($batchID : \mathbb{N}$, $pointer : Functionpointer$)
  if $batchID = 0$ **then return** $pointer$ ▷ batch is complete
  $eventIndex := batchID \mod eventTypeCount$ ▷ factor of kth exponent
  if $eventIndex > 0$ **then** ▷ check for ν-event
    $appendFuncCall(pointer, eventHandlers[eventIndex - 1])$

  $genBatch((batchID/eventTypeCount), pointer)$ ▷ continue with quotient

APPENDIX B
PROOF OF s_{max}

$p_I \hat{=}$ probability of an *Increment* Event
$p_S = 1 - p_I \hat{=}$ probability of a *Set* Event
$n \hat{=}$ batch length
$T_1 \hat{=}$ standard DES time
$T_p \hat{=}$ batched DES time
$S = \frac{T_1}{T_p} \hat{=}$ speedup
$E[S] = \frac{E[T_1]}{E[T_p]}$
$E[T_1] = n \cdot p_I$
$E[T_p] = \sum_{j=1}^{n-1}(j \cdot {p_I}^j \cdot p_S) + n \cdot p_I$

Lemma 1. *Closed form of* $E[T_p]$.

$$E[T_p] = \sum_{j=1}^{n-1}(j \cdot {p_I}^j \cdot p_S) + n \cdot p_i = \frac{1 - {p_I}^n}{\frac{1}{p_I} - 1}$$

Proof.

$$\begin{aligned}
E[T_p] &= \sum_{j=1}^{n-1}(j \cdot {p_I}^j \cdot p_S) + n \cdot {p_I}^n \\
&= \sum_{j=1}^{n-1}(j \cdot {p_I}^j \cdot (1 - p_I)) + n \cdot {p_I}^n \\
&= (1 - p_I)\sum_{j=1}^{n-1}(j \cdot {p_I}^j) + n \cdot {p_I}^n \\
&= \sum_{j=1}^{n-1} j \cdot {p_I}^j - p_I \sum_{j=1}^{n-1} j \cdot {p_I}^j + n \cdot {p_I}^n \\
&= \sum_{j=1}^{n-1} j \cdot {p_I}^j + n \cdot {p_I}^n - p_I \sum_{j=1}^{n-1} j \cdot {p_I}^j \\
&= \sum_{j=1}^{n} j \cdot {p_I}^j - p_I \sum_{j=1}^{n-1} j \cdot {p_I}^j \\
&= \sum_{j=1}^{n} j \cdot {p_I}^j - \sum_{j=1}^{n-1} j \cdot {p_I}^{j+1} \\
&= \sum_{j=1}^{n} j \cdot {p_I}^j - \sum_{j=2}^{n} (j-1) \cdot {p_I}^j \\
&= \sum_{j=1}^{n} j \cdot {p_I}^j - \sum_{j=1}^{n} (j-1) \cdot {p_I}^j \\
&= \sum_{j=1}^{n} (j - j + 1) {p_I}^j \\
&= \sum_{j=1}^{n} {p_I}^j \\
&= p_I \sum_{j=1}^{n} {p_I}^{j-1} \\
&= \frac{(1 - p_I) \cdot (\sum_{j=0}^{n-1} {p_I}^j) \cdot p_I}{1 - p_I} \\
&= \frac{(1 - {p_I}^n) \cdot p_I}{1 - p_I} \\
&= \frac{1 - {p_I}^n}{\frac{1}{p_I} - 1}
\end{aligned}$$

□

Corollary 1. *Maximum speedup* s_{max}. *The expected value for the maximum speedup is*

$$s_{max} = E[S] = \frac{n \cdot (1 - p_I)}{1 - {p_I}^n}.$$

Proof.

$$\begin{aligned}
E[S] &= \frac{E[T_1]}{E[T_p]} \\
&= \frac{n \cdot p_I}{\frac{1 - {p_I}^n}{\frac{1}{p_I} - 1}} \\
&= (n \cdot p_I) \cdot \frac{\frac{1}{p_I} - 1}{1 - {p_I}^n} \\
&= \frac{n \cdot p_I \cdot \frac{1}{p_I} - n \cdot p_I}{1 - {p_I}^n} \\
&= \frac{\frac{n \cdot p_I}{p_I} - n \cdot p_I}{1 - {p_I}^n} \\
&= \frac{n - n \cdot p_I}{1 - {p_I}^n} \\
&= \frac{n \cdot (1 - p_I)}{1 - {p_I}^n}
\end{aligned}$$

□