juliacon

Extrae.jl: Julia bindings for the Extrae HPC Profiler

Sergio Sanchez-Ramirez[1] and Mosè Giordano[2]

[1]Barcelona Supercomputing Center (BSC)
[2]University College London (UCL)

ABSTRACT

The Julia programming language has gained acceptance within the High-Performance Computing (HPC) community due to its ability to tackle two-language problem: Julia code feels as high-level as Python but allows developers to tune it to C-level performance. But to squeeze every drop of performance, Julia needs to integrate with advanced performance analysis tools, also known as profilers. In this work, we present `Extrae.jl`, a Julia package to interface with the Extrae profiler.

Keywords

Julia, HPC, Profiling, Tracing, Performance Evaluation

1. Introduction

Scientific computing applications require code that is both high-level, for expressing complex mathematical ideas, and high-performance, as it usually involves computationally intensive tasks. These requirements derive into a problem known as "the two-language problem"; i.e. programming languages usually focus only on one of these two topics, forcing package authors to use two programming languages: one higher-level for the abstraction and one lower-level for performance. This is characteristic of programming languages such as Python, whose package authors derive to C, C++, or Rust for improving the performance.

In recent years, the Julia programming language [5] has steadily gained adoption within the High-Performance Computing (HPC) community due to its ability to tackle the aforementioned two-language problem: Julia code feels as high-level as Python but allows developers to tune it to C-level performance. Additionally, it supports parallel programming models such as MPI, master-worker distributed computing, and GPU computing, and offers a rich interactive REPL experience, facilitating ease of use and fast prototyping for developers.

While being able to write code so that it can exploit computing resources is key, it is not the full story. In pursuit of higher computational power, hardware vendors have made more parts of their systems available to software. This in turn makes software development more relevant than ever for fully exploiting the performance capabilities of current computers. Driving performance optimization are performance analysis tools that are commonly named as *profilers*. Profilers measure where a program spends its time or which functions consume the most resources. They help identify bottlenecks, optimize code, and improve overall application performance.

One of the most well-known profilers in the field is Extrae. Recognized for its low overhead and large support of programming models and platforms, it is a veteran in the performance analysis field. Its programming model allows for deep customization and tuning, but unfortunately, it also imposes a high barrier for newcomers. In this work, we present `Extrae.jl`: The Julia bindings to the Extrae profiler, with which we try to ease the barrier for Julia users.

This work is organized as follows. In Section 2, we introduce the current state of performance analysis tools in the HPC world to the reader. In Section 3, we describe Extrae's design and how we have mapped it Julia's concepts and parallelism models. In Section 4, we present some examples of `Extrae.jl`running on top of Julia HPC apps. Finally, in Section 5, we summarize our work and point to possible directions of improvement in future works.

2. Background

Depending on the target resource of the analysis and the method used to gather the data, different tools are available. Basically, two methods are used for profiling:

—Tracing: It involves explicitly calling the profiler at the target regions of the code. Although 100% is accurate, the overhead is high. Manual instrumentation of the code or binary interception of symbols can be used.

—Statistical sampling: It entails periodically interrupting the application, collecting data, and resuming. The results are expected to be statistically similar to the ones provided by tracing but with a much lower overhead.

There are a number of commercial and open source profiling tools that can be used to analyze performance in HPC based on these two methods. The following is a list of profilers that are known to work with Julia:

Julia standard library `Profiler.jl`. Julia comes with a statistical profiler called `Profiler.jl`, which can be used to periodically sample the backtrace of each currently running task, capturing the name and file-line information of all Julia functions in the call stack. `Profiler.jl` can also optionally capture information about Julia's core and, optionally, external shared libraries, written in languages such as C, Fortran, etc. The default output format produced by `Profiler.jl` is text-based, but external tools can be used to produce various types of visualization. Some of these tools are `FlameGraphs.jl`, VS Code, `ProfileView.jl`, `ProfileVega.jl`, `StatProfilerHTML.jl`, `ProfileSVG.jl`, `PProf.jl`, and `ProfileCanvas.jl`. It is also possible to profile memory allocations within Julia code with this standard library. Although `Profiler.jl` does not have native support for distributed computing, it is possible to collect separate traces for different workers.

Linux Perf/OProfile. Julia has support for analyzing programs using popular open source system profilers for the Linux operating system OProfile, a statistical sampler, and its more modern replacement Perf. By setting the environment variable `ENABLE_JITPROFILING=1` Julia will create and register an event listener to profile just-in-time (JIT) compiled functions. Furthermore, the package `LinuxPerf.jl` [12] allows getting more fine-grained information about hardware counters using Perf for specified function calls.

Nvidia Nsight System. The Nvidia Nsight System is a proprietary statistical sampling profiler with tracing features, which supports MPI and CUDA applications. As such, it can be used to profile also Julia programs, including those distributed using the MPI protocol and/or employing a Nvidia GPU, but JIT functions will have their names mangled by the compiler. The package `NVTX.jl` [10] allows you to instrument the code and annotate specific functions or code regions, optionally also adding a payload (e.g. the value of a local variable) which can be displayed in the Nsight System GUI. The in-development version of Julia can optionally be compiled to automatically instrument with NVTX internal components, such as the compiler and the garbage collector.

Intel VTune. Intel VTune is a proprietary sampling profiler with support for MPI and OpenMP applications. When the environment variable `ENABLE_JITPROFILING=1` is set, Julia JIT functions will be recorded with their correct name on VTune traces. You can instrument Julia code with the package `IntelITT.jl` [11] to get richer information in the traces.

LIKWID. LIKWID is an open-source suite of tools for performance analysis of HPC applications on Linux, using hardware performance counters, rather than sampling or tracing profiling. It supports MPI, OpenMP, and GPU offloading. The package `LIKWID.jl` [4] interfaces with the shared library part of the LIKWID suite to instrument the code and collect important hardware metrics to measure the performance of an application.

Score-P. Score-P is an open-source suite for profiling and event tracing of HPC applications. You can use the package `ScoreP.jl` [3] to instrument Julia code and analyze it with ScoreP.

`MPITape.jl`. `MPITape.jl` [1] is an experimental open-source Julia package which overdubs MPI operations performed through the package `MPI.jl` [8]. This allows you to track communication patterns and timings within a distributed application.

Tracy. Tracy is an open-source frame and sampling profiler, originally designed for gaming applications but can be used for any other kind of program. Julia can optionally be built to instrument internal components, such as the garbage collector and the compiler, so that they can be profiled by Tracy.

3. Design

Extrae generates Paraver traces, which follows two orthogonal object models:

—The **process model** represents the abstract virtual resources used by the most common programming models. Its taxonomy is formed by the WORKLOAD, APPLICATION, TASK, and THREAD objects.

—The **resource model** represents the physical resources where the program is finally executed. Its taxonomy is formed by SYSTEM, NODE and CPU objects.

This separation between virtual and physical resources is key for mapping different parallel programming models to a common model. For example, MPI ranks are assigned to TASK objects, and OpenMP threads are mapped to THREAD objects. The representation of an application with the OpenMP+MPI programming model composes just as one MPI rank (TASK object) contains several OpenMP threads (THREAD objects). Note that, thanks to the aforementioned separation, threads can migrate between cores without invalidating the mapping. Furthermore, new programming models not considered during the development of Extrae/Paraver, such as GPUs, may also map to the Paraver process and resource models.
In this sense, Extrae provides out-of-the-box instrumentation for MPI, OpenMP, OpenACC, PThreads, CUDA, OpenCL, Linux syscalls, Libc's memory-related functions (i.e. `malloc` and `free`) and IO functions. Some of these models require some form of dependency injection mechanism, for which Extrae supports two methods:

—Library symbol interception using `LD_PRELOAD`

—Binary rewrite instrumentation using DynInst

Extrae automatically detects the corresponding resource identifiers. For the process model, the identifiers are taken directly from the underlying API offered by the programming models, but Extrae allows the user to customize the TASK and THREAD identification functions. This is beneficial for custom programming models built on top of other programming models, as is the case of COMPSs [2] which maps workers to TASK objects. We provide access to them through the `set_threadid_function!`, `set_taskid_function!` and derivatives.
Additionally, Extrae provides a statistical call stack and hardware counter sampler. Hardware counters are accessed through the PAPI [6] interface and can be configured to either sample periodically on time or based on accumulative event counters (e.g., sample every 1,000 dispatched instructions). Jitter can be configured to avoid sampling aliasing effects.
It generates three types of annotation: states, events, and communications. States refer to regions of time where the application spends time on user code, on external code, or idle. Extrae provides a routine for annotating the beginning and end of the user-coded region. For the commodity of users, we made it available through a `@user_function` macro. An example of its usage is shown in Listing 1.

Listing 1: Usage example of `@user_function` in a benchmark for the `axpy!` routine.

```
using Extrae: Extrae, user_function

Extrae.init()

function axpy!(a, x, y)
   @user_function @simd for i in eachindex(x, y)
       @inbounds y[i] = muladd(a, x[i], y[i])
   end
   return y
end

for T in (Float16, Float32, Float64)
   benchmark(axpy!, T, "julia")
end

Extrae.finish()
```

An omnipresent type of annotation is events. Events are punctual annotations of 2-tuple integer values on a specific processor and time location. Many automatically collected annotations are events; e.g. hardware counters. `Extrae.emit` allows the user to emit custom events, where the first argument is the event type or code, and the second value is the event value. Event types and values can be given some string descriptions with `Extrae.register`. Listing 2 shows an example of how to use both the `emit` and `register` functions.

Listing 2: Usage example of event registration and emition.

```
# on setup
const CODE_VEC_LEN::UInt32 = 84210
Extrae.register(CODE_VEC_LEN, "Vector length")

# on workload
Extrae.emit(CODE_VEC_LEN, UInt64(N))
```

Finally, the last type of annotation is communication. One communication mark represents a message sent between two processes during execution. Custom emission of communication marks is part of the extended API, for which the support is experimental, but MPI instrumentation already uses it automatically. An example can be shown in Figure 2b, where yellow lines represent MPI messages.

3.1 Mapping Julia's programming models to Paraver's process model

In the case of `MPI.jl` [8], MPI calls are automatically instrumented if Extrae is loaded before the MPI library, but loading it before Julia can lead to some errors due to library version incompatibility issues. The solution is then to use the `preloads` setting of `MPI.jl` preferences (using `MPIPreferences.jl`). This will load Extrae after Julia but before the MPI library.
In the case of `Distributed.jl`, workers can map to TASK objects. If MPI is used as the communication layer between distributed workers using the `MPIClusterManagers.jl` package, then the mapping is performed automatically using the procedure described above. In the case of other cluster managers, the user needs to manually remap the `taskid` and `numtasks` routines for Extrae. To simplify this process, we have added a helper method to `init` so the user only needs to do something similar to Listing 3.

Listing 3: Manual initialization of Extrae on a Distributed.jl workflow.

```
using Distributed

addprocs(4)

@everywhere using Extrae
@everywhere Extrae.init(Val(:Distributed))
```

In the case of Julia shared-memory programming model, threads map to THREAD objects automatically since they use PThreads underneath. However, Julia tasks (i.e. coroutines) are impossible to map, as they add another level of virtual indirection because tasks can migrate between threads when they yield execution. In this sense, a Julia task id event could be emitted if Extrae could hook to Julia's BPF points but it is something that is not yet supported. Meanwhile, if task tracing is critical to the user, we propose using manual instrumentation, as the template code in Listing 4.

Listing 4: Template code for instrumenting Julia tasks.

```
taskid() = objectid(current_task())

@spawn begin
    Extrae.emit(TASKID_EVENT, taskid())
    # workload
    yield()

    EXTRAE.emit(TASKID_EVENT, taskid())
    # other workload
    Extrae.emit(TASKID_EVENT, 0)
end
```

Finally, thanks to the Yggdrasil repository of binary artifacts, users can fetch the latest Extrae library based on their current platform, MPI ABI and CUDA version. As an interesting fact, this was the first Yggdrasil recipe to combine both CUDA and MPI platform augmentation, further exemplifying the success of Julia's artifact system and BinaryBuilder.
For a deeper insight into Extrae and Paraver, we point the reader to [7, 9, 13].

4. Evaluation

With the aim of testing the integration of `Extrae.jl`with Julia, we evaluate the performance of a Taylor-Green vortex simulation parallelized with MPI using `Trixi.jl` [16, 17] and `OrdinaryDiffEq.jl` [15]. The source code can be found in `https://github.com/trixi-framework/performance-2024-trixi_taylor-green_vortex`. The application was run on the Marenostrum 5 supercomputer and the analysis was performed with the Paraver trace viewer [9, 14].
First, Paraver is able to compute some general metrics about the run. For example, in Figure 1 the evolution of instantaneous parallelism is shown, which reveals parallelization problems during the main workload.

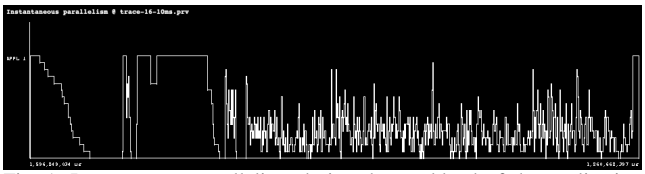
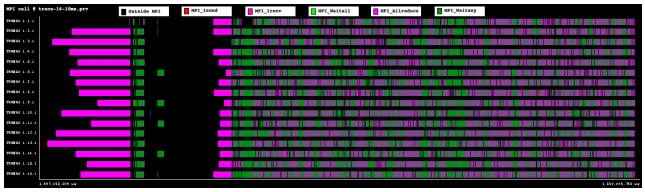
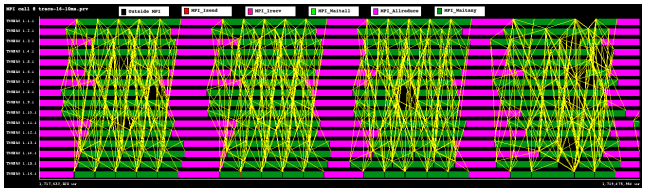


Fig. 1: Instantaneous parallelism during the workload of the application measured as the number of MPI ranks not being idle at the given moment. Maximum value is 16 and minimum is 1.

For a more detailed and finer view of the execution, Paraver can plot a timeline of MPI calls per rank, as shown in Figure 2a. By sight, we can correlate the region of loss of parallelization with the region where there is a high density of `MPI_Waitany` and `MPI_Allreduce` calls. Zooming in on that region, shown in Figure 2b, we can confirm that execution is dominated by these two MPI routines.
If we take the connectivity pattern between MPI ranks, shown in Figure 3, we can observe no communication imbalance.
Paraver also allows for external post-processing of the collected data. It is clear from Figure 4 that the bottleneck is `MPI_Waitany` (around 60% of the total time), followed by `MPI_Allreduce` (around 30% of the total time). Furthermore, the variability is small enough compared to the amount of time to discard load imbalance problems.



(a) Original



(b) Zoom

Fig. 2: Timeline of MPI calls for the (a) runtime of the application and (b) zoom on a couple of iterations of the main workload. Horizontal axis is time and each row shows the events that took place on each MPI rank. Blocks represent MPI calls where color maps to MPI routines and the start and end positions fit the duration of the call. Yellow lines represent messages sent betweens ranks.

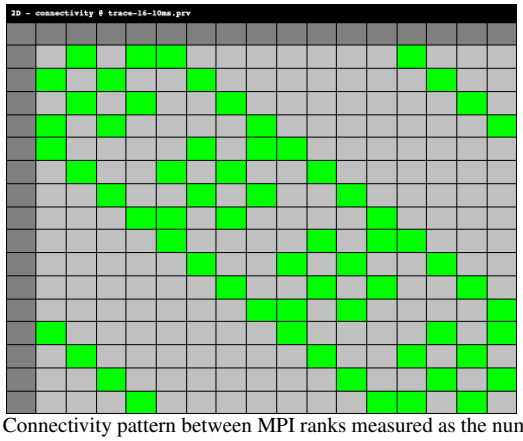


Fig. 3: Connectivity pattern between MPI ranks measured as the number of messages sent from MPI rank x to rank y. Grey cells mean no communication while green cells mean represent a value of 2,016 messages sent.

Finally, Paraver can also estimate the node bandwidth by taking the communication annotations, as shown in Figure 5. In correlation with Figure 2b, we can see that the bandwidth increases during `MPI_Waitany` calls and decreases and even stops communication during `MPI_Allreduce` calls. In addition, the bandwidth peaks at around 188.73 MB/s, far away from the theoretical peak bandwidth of 12.5 GB/s (100 Gb/s).

5. Summary

In this work, we have presented `Extrae.jl`, a Julia package to interface with the Extrae profiler. Furthermore, we have demonstrated the integration through the use of Extrae to evaluate the performance of a Taylor-Green vortex simulation using MPI, `Trixi.jl` and `OrdinaryDiffEq.jl`. Thanks to the Yggdrasil binary repository, we provide binary artifacts for the target platform, MPI ABI and CUDA version, so users can use it without requiring an administrator to install it in their cluster.

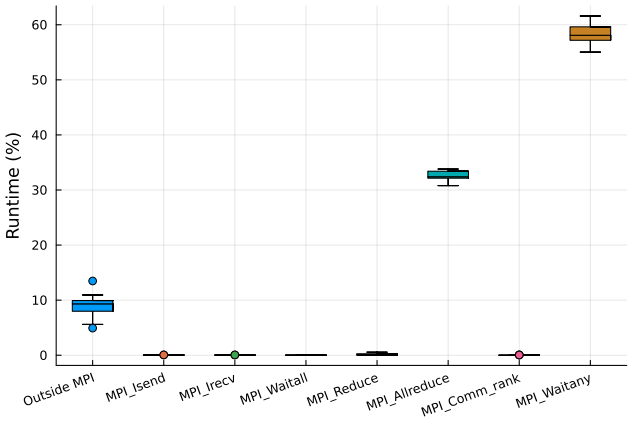


Fig. 4: Fraction of time spent on each MPI routine. Dispersion comes from the time spent by different MPI ranks.

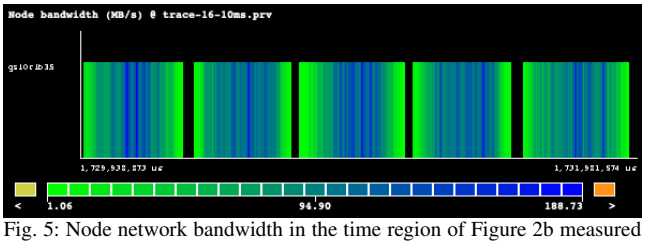


Fig. 5: Node network bandwidth in the time region of Figure 2b measured in MB/s. Only one node is shown because all MPI ranks run on the same node.

In the future, we aim to integrate with other BSC's performance modeling tools such as Folding and Dimemas. Alternatively, there may be a reimplementation of this functionality in Julia through the use of the Paraver parser. A Terminal User Interface (TUI) is planned for interactive configuration of Extrae. Moreover, there is an ongoing effort to parse Paraver tracefiles and convert them to Open Trace Format 2 (OTF2) which would ensure compatibility with other trace visualization tools like ScoreP.

6. Acknowledgments

MG's work was funded by the UCL ARC "Fostering International Collaborations in Advanced Digital Research" program. Authors would like to thank Marc Clascà, for help instrumenting `Distributed.jl` and general Extrae advice, and Carsten Bauer, for interest and suggestions on future work directions.

7. References

[1] *MPITape.jl*. Available at `https://github.com/pc2/MPITape.jl`.

[2] Rosa M Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque, Daniele Lezzi, Francesc Lordan, Cristian Ramon-Cortes, and Raul Sirvent. COMP Superscalar, an interoperable programming framework. *SoftwareX*, 3:32–36, 2015.

[3] Carsten Bauer. *ScoreP.jl*. JuliaPerf. Available at `https://github.com/JuliaPerf/ScoreP.jl`.

[4] Carsten Bauer and Valentin Churavy. *LIKWID.jl*. JuliaPerf. Available at `https://github.com/JuliaPerf/LIKWID.jl`.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[6] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications*, 14(3):189–204, 2000.

[7] BSC-CNS. *Extrae 4.2.3 documentation*, 2024. Available at `https://tools.bsc.es/doc/html/extrae/index.html`.

[8] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. MPI.jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68, 2021.

[9] CEPBA-UPC. *Paraver: Reference Manual (Version 3.1)*, October 2001. Available at `https://tools.bsc.es/doc/1364.pdf`.

[10] JuliaGPU. *NVTX.jl*. Available at `https://github.com/JuliaGPU/NVTX.jl`.

[11] JuliaPerf. *IntelITT.jl*. Available at `https://github.com/JuliaPerf/IntelITT.jl`.

[12] JuliaPerf. *LinuxPerf.jl*. Available at `https://github.com/JuliaPerf/LinuxPerf.jl`.

[13] Jesus Labarta. Introduction to Paraver. POP Webinar 22, July 2021. Available at `https://youtu.be/R8_EhVpOzb0`.

[14] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31, 1995.

[15] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl: A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[16] Hendrik Ranocha, Michael Schlottke-Lakemper, Andrew Ross Winters, Erik Faulhaber, Jesse Chan, and Gregor Gassner. Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing. *Proceedings of the JuliaCon Conferences*, 1(1):77, 2022.

[17] Michael Schlottke-Lakemper, Gregor J Gassner, Hendrik Ranocha, Andrew R Winters, and Jesse Chan. Trixi.jl: Adaptive high-order numerical simulations of hyperbolic PDEs in Julia. `https://github.com/trixi-framework/Trixi.jl`, 09 2021.