Experience Report: Erlang in Acoustic Ray Tracing

Christian Convey Andrew Fredricks
Christopher Gagner Douglas Maxwell
Naval Undersea Warfare Centers, RI, USA
{conveycj, fredricksaj, gagnercw,
maxwelldb}@npt.nuwc.navy.mil

Lutz Hamel
Dept. of Computer Science and Statistics
University of Rhode Island
hamel@cs.uri.edu

Abstract

We investigated the relative merits of C++ and Erlang in the implementation of a parallel acoustic ray tracing algorithm for the U.S. Navy. We found a much smaller learning curve and better debugging environment for parallel Erlang than for pthreads-based C++ programming. Our C++ implementation outperformed the Erlang program by at least 12x. Attempts to use Erlang on the IBM Cell BE microprocessor were frustrated by Erlang's memory footprint.

Categories and Subject Descriptors D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Design, Performance

Keywords acoustic ray tracing, C++, Erlang

1. Introduction

The U.S. Navy uses a variety of computationally intensive algorithms for research, for system testing and evaluation, and in submarines' and warships' weapons and sensors systems. The Navy constantly seeks computing platforms and programming languages that permit these algorithms to run ever faster. This endeavor is expensive. Evaluating a new computing platform's potential can entail large labor costs as software engineers learn how to optimize software for that particular platform and then must reimplement various algorithms on that platform so they can be benchmarked.

Our project involves identifying a programming language that permits algorithms of Navy interest both to be expressed concisely and to run fast on a variety of hardware, including standard x86 shared-memory systems, distributed message-passing systems, and on-chip message-passing systems such as those based on IBM's Cell Broadband Engine (Cell BE) microprocessor.

We've developed a simple *acoustic ray tracing* algorithm (see section 4.3) as the test-case with which to compare various combinations of programming languages and computing platforms. Acoustic ray tracing has several desirable qualities as a test-case: it is highly parallel, conceptually simple, and is widely enough used within the Navy that our results will have broad relevance. We've implemented that algorithm in C++ and Erlang and attempted to run it on several platforms, comparing the performance and ease of programming.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

2. General Approach

For each of our two initial platforms of interest (a quad-core Intel workstation and a Sony PlayStation 3) we produced a C/C++ reference implementation of the acoustic ray tracing algorithm whose performance we treat as a baseline measure of that platform's performance potential. The widespread use of C++ and its reputation for supporting fast computations makes it the standard against which other language's performance potentials are likely to be compared. We selected Erlang because of its reputation for easy parallelization and for functional languages' reputation for legible code.

We intended to evaluate Erlang's speed potential on our quad-core Intel workstation and on the Sony PlayStation 3. We were only able to properly evaluate Erlang on the Intel workstation, as explained in subsection 5.3.

3. Limitations

Our goal is to make inferences regarding a particular programming-language / computing-platform combination's utility in high-performance Navy computing. Though this work is important, two factors demand a measure of humility regarding general inferences that may be drawn from the work.

The first is that we cannot practically know if we've given a programming language / computing platform an opportunity to demonstrate either its potential runtime speed or expressiveness. There may always have been a better-written program that implemented our algorithms with greater performance or clarity. Additionally, using just one particular algorithm to represent many algorithms of Navy interest may be misleading, because perhaps a different algorithm could have been equally representative but better suited to a particular programming language / computing platform.

The second, related factor is that time and monetary constraints prevent our team from becoming truly expert at developing and tuning software for each language / platform of interest, and our team members have significantly more experience in some languages (C/C++ in this case) than others. This introduces a potential bias in results that must not be ignored. Despite these limitations we feel our approach provides some useful data points regarding these languages and platforms.

4. Acoustic Ray Tracing

4.1 Overview

Developing weapons for submarine warfare requires the accurate modelling of underwater sound propagation. The particular details

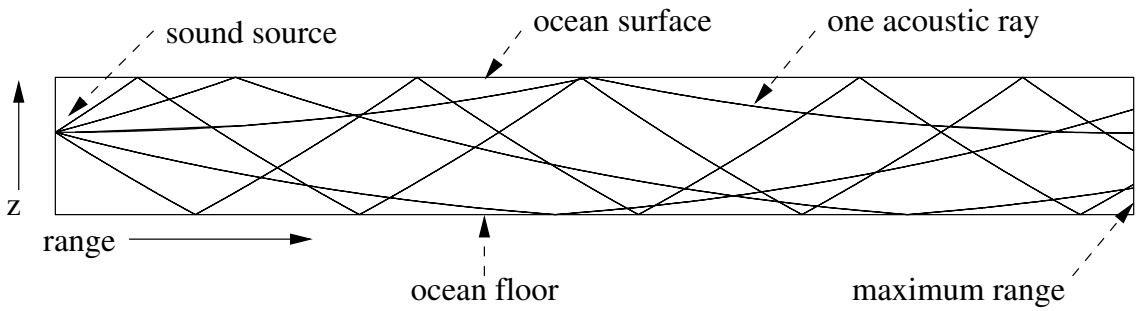

Figure 1. Two-dimensional acoustic ray trace example.

of how sound moves, attenuates, and bounces in a given environment can decide whether or not one's own submarine is detected by an enemy, whether or not a torpedo finds its target, etc.

Acoustic ray tracing is one technique for analyzing the propagation of sound in some particular ocean environment. When sound is emitted into a body of water, the leading edge of the propagating pressure (sound) wave is a surface called a *wavefront*. Researchers are interested in various details about the collisions of a wavefront with objects in the water, such as ships or the ocean floor. Typically a pressure wave is modelled as a set of advancing acoustic rays, in which the head of each ray represents the current location and direction of motion of a point on the pressure wave's surface. Acoustic ray tracing is the process of calculating how a set of acoustic rays advances in a particular ocean environment, noting details such as each ray's location, direction, and intensity at various stages of its propagation. Minimizing computation time is a key requirement for many acoustic ray tracing applications.

4.2 Tracing an Acoustic Ray

4.2.1 Non-linear Paths

Several details differentiate standard acoustic ray tracing from light ray tracing used in computer graphics. Most importantly, acoustic rays don't travel in straight lines (see Figure 1) because the speed of sound varies throughout the ocean. Sound speed is generally a function of the water's temperature, salinity, and (most significantly) depth. Acoustic ray tracers commonly describe the speed of sound using a *sound speed profile* (SSP), a function that maps ocean depth to sound speed (see Figure 2). In our algorithm the SSP is a piecewise-linear function and is specified in a user-supplied text file. Each linear piece in the SSP function is called a *depth band*.

When the speed of sound varies with depth, an acoustic ray will follow a curved path, always bending towards the direction of lower sound speed. The path is an arc whose radius is inversely proportional to the *sound speed gradient*, g. The sound speed gradient is the rate at which the speed of sound changes with depth:

$$g = \frac{dc}{dz} \tag{1}$$

where dc is the change in sound speed and dz is the change in depth. See (Kinsler, Frey, Coppens and Sanders 2000, pp. 138–139) for additional information.

4.2.2 Two-dimensional Tracing

Another difference from light ray tracing is that acoustic ray tracing is most commonly treated as a two-dimensional problem (See Figure 1). The plane in which ray are traced is defined by a *z axis* and a *range axis*. z is the inverse of depth: it is zero at the ocean floor and increases upward. *Range* is the horizontal distance from the sound source.

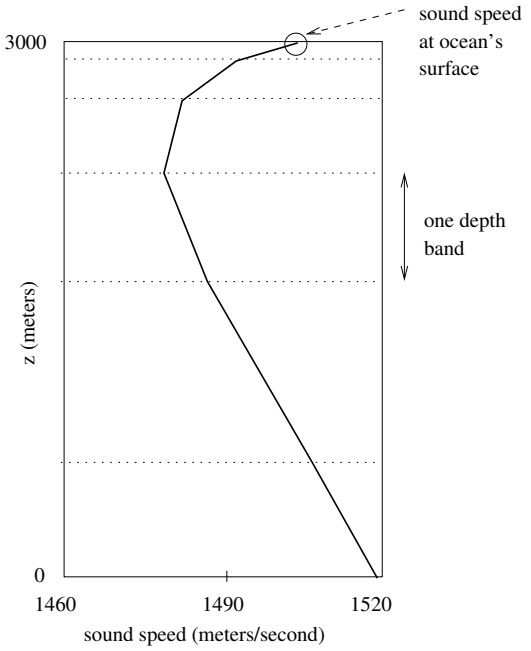

Figure 2. Representative sound speed profile (SSP), drawn from (Kinsler, Frey, Coppens and Sanders 2000, p. 436).

4.3 Top-level Algorithm

Our overall algorithm is summarized in Figure 3. Acoustic rays originate from the same point and begin propagation at the same time. What initially differentiates the rays is the depth/elevation (D/E) angle at which each initially propagates. A D/E of 0° indicates horizontal propagation; a positive D/E indicates propagation towards the ocean surface; a negative D/E indicates propagation towards the ocean floor. The user provides a minimum D/E angle, a maximum D/E angle, the number of rays to be traced (`num_rays`), the number of worker threads/processes (`num_threads`), and the height (`z0`) of the sound source.

Our algorithm[1] evenly distributes the rays' initial D/E angles within the range `[min_de_angle, max_de_angle]`. Rays are assigned to worker threads using a simple modulo division scheme (see variable i in Figure 3). This approach minimizes the communication needed to distribute the workload, and in practice

[1] Implementations available. Please e-mail conveycj@npt.nuwc.navy.mil

```
sub main(min_DE_angle, max_DE_angle, num_rays,
      num_threads, z0) {
   delta_DE = (max_DE_angle - min_DE_angle) /
      (num_rays - 1);
   SSP = load_ssp_from_disk();
   for i in 0...(num_threads-1) {
      spawn(thread_function,
         (min_DE_angle, max_DE_angle, delta_DE,
          SSP, i, z0));
   }

   wait_until_threads_done();
}

sub thread_function(min_DE_angle, max_DE_angle,
      delta_DE, SSP, i, z0, max_range) {
   de = min_DE_angle + (i * delta_DE);
   while de <= max_DE_angle {
      trace_one_ray(SSP, z0, de);
      de += i * delta_DE;
   }
}
```

Figure 3. Pseudocode for top-level acoustic ray tracing algorithm.

```
sub trace_one_ray(SSP, z0, de) {
   range = 0; z = z0; bounces = 0;
   depthband = get_depthband(SSP, z0);
   progress = [(range, z)];
   while (range <= MAX_RANGE) and
         (bounces <= MAX_BOUNCES) {
      (range2, z2, de2, dband2, bounced) =
         adv_thru_dband(dband, range, z, de);
      if bounced { bounces++; }
      range=range2;z=z2;de=de2;dband=dband2;
      progress.append( (range, z) );
   }
}
```

Figure 4. Pseudocode for single ray tracing algorithm.

seems to balance the workload fairly evenly amongst the worker threads/processes.

4.4 Tracing a Ray

Figure 4 summarizes the algorithm for tracing an individual acoustic ray. Tracing an individual ray requires calculating its progression through a series of depth bands until a stopping condition occurs: either the ray has been traced to range of `MAX_RANGE`, or it has bounced `MAX_BOUNCES` times. A bounce occurs when the ray encounters the ocean's surface or floor.

Within a depth band the `adv_thru_dband` function determines details about the ray's exit from the depth band including: its new position (`range2`, `z2`), propagation angle (`de2`), the depth band into which it will next enter (`dband2`), and whether or not it has just encountered the ocean's surface or floor (`bounced`). The function compares the geometric intersection of that band's curvature circle with the rectangle bounding the depth band. The intersection point with the lowest range greater than `range` is the point at which the ray exits the depth band. `adv_thru_dband` makes several trigonometry function invocations but contains a significant amount of branching.

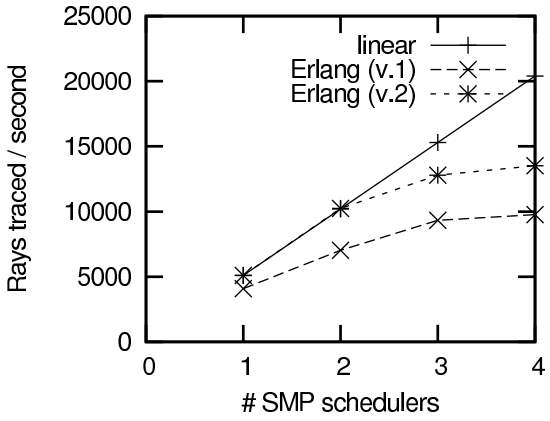

Figure 5. Speed scaling of ray tracer Erlang implementations.

5. Runtime Performance: C++ vs. Erlang

5.1 Methodology

On our x86 SMP system we implemented the ray tracing algorithm in C++ and Erlang. Neither implementation used platform-specific coding styles, libraries, or pragmas. 100,000 rays were traced in each benchmark run.

Our reported program runtimes give the time duration from just before the work is distributed to the worker threads until after all worker threads complete their work. To measure performance scalability with respect to the number of CPU cores, we used two approaches. For our C++ program we varied the number of worker threads from one to four. For Erlang we varied both the number of scheduler threads (via `erl`'s `+S n` command-line parameter) and the number of worker processes. Our Erlang results show the runtime from the best-performing number of worker processes for a given number of scheduler threads. Each reported runtime is the mean of three runs' times; variance was negligible.

Our benchmark computer contained a single Intel Core2 Quad microprocessor clocked at 2.4 GHz, running Ubuntu Linux 7.04. C++ code was compiled with the Gnu g++ compiler version 4.1.2 and `O3`-level optimization. Erlang code was compiled with the HiPE native compiler from the OTP R12B-1 release of Erlang.

After prototyping the ray tracing algorithm in Python, our initial C++ implementation required an estimated 30 hours of labor. The port from C++ to Erlang (of which we had no former knowledge) required an estimated 20 hours of reading (Armstrong. 2000) and 30 hours of porting, debugging, profiling, and tuning. Performance of the untuned (v.1) and tuned (v.2) versions of the Erlang program are shown in Figure 5. Tuning efforts were stopped when they appeared to stop yielding runtime improvements. The performance of the equivalent C++ program is shown in Figure 6.

5.2 Performance Results

Figures 5 and 6 show that in general both the C++ and Erlang implementations benefitted from multiple cores. However, comparing the two figures shows that not only did the C++ implementation significantly outperform the Erlang implementation, but it also scaled better than the Erlang implementation as the number of threads / worker processes was increased to take advantage of the CPU's four cores.

5.3 Inability to run Erlang on Cell

The IBM Cell BE microprocessor (Chen, Raghavan, Dale, Iwata 2005) is an appealing platform for acoustic ray tracing. The Cell processor found in Sony PlayStation 3 game consoles is inexpen-

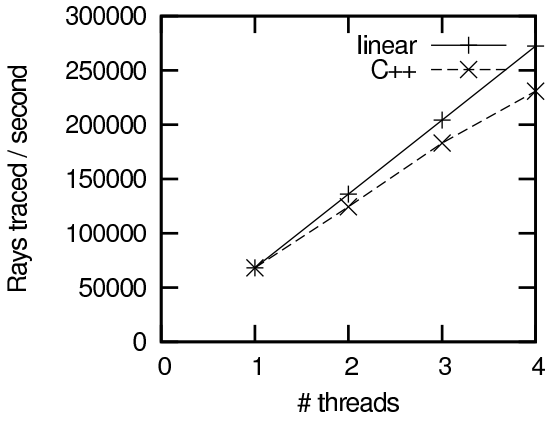

Figure 6. Speed scaling of ray tracer C++ implementation.

sive and offers six Synergistic Processor Element (SPE) cores, potentially providing a good price to performance ratio for acoustic ray tracing. Implementing our acoustic ray tracing algorithm in C/C++ on the Cell processor showed approximate performance parity on a per-core basis with an Intel Core2 4400 2.0 GHz microprocessor. Cell-specific tuning yielded about a 10% improvement in runtime compared to our initial naive porting.

Cell programming requires a significant learning curve due to unusual compilation/linkage requirements, restricted support for C++, the small working memory on each SPE core, and lack of shared memory between processor cores. We investigated using Erlang to hide these barriers from application programmers by having one Erlang process run on each SPE core. Erlang's parallelism via message passing maps cleanly onto the Cell processor's use of message passing for inter-SPE communication. We abandoned this effort when it appeared that Erlang runtime memory requirements exceeded an SPE core's 256KB working memory by an order of magnitude.

6. Programmer Productivity

Erlang significantly outperformed C++ in the duration of its learning curve, program conciseness, and debugging time.

6.1 Learning Curve

Considering the countless hours we required to become competent C++ and pthreads programmers, the short time required for our initial port to Erlang is remarkable.

The design of our C++ code may have helped to keep the porting time to a functional language low. The C++ implementation contains a function to trace a single acoustic ray, without any needed communication or cross-thread coordination during that ray's tracing. All required details for tracing a ray are provided as explicit parameters and the results are returned as a C++ vector; the function does not read or modify any mutable, shared data structure. This design was intended to make parallel execution simpler and more efficient, but we found that it played well into the functional programming paradigm as well, likely reducing our porting time.

6.2 Conciseness

The Erlang implementation was more concise, requiring 591 non-comment lines of code compared to the C++ implementation's 966 lines. This size difference may be partially attributable to a tendency to write unnecessarily verbose C++ code, including the definition of classes when mere functions would have sufficed. One might also consider the C++ code more engineered with careful error checking. For example, the C++ implementation gives meaningful error messages when parsing an ill-formed Sound Speed Profile (SSP) input file. The Erlang code is more in the style of a research prototype with less rigorous error handling, and one would therefore expect fewer lines of code. It's unclear if this entirely explains the large discrepancy in line counts.

6.3 Debugging

Single-assignment variables and Erlang's standard graphical debugger significantly helped our debugging efforts. In the C++ implementation, a set of working variables records a ray's current status as tracing progresses. Sometimes when a ray was found to have reached an obviously invalid state, the information describing where things had first gone wrong in the ray's progression had already been overwritten. This led to protracted and frustrating debugging efforts. With Erlang's single-assignment variables and well-implemented graphical debugging tool, a ray's tracing history was easy to examine whenever an error was discovered.

6.4 Memory Use

One weakness we did encounter with Erlang was memory use. When an Erlang-based benchmarking program repeatedly ran our ray-tracing function the Erlang shell would crash without reporting the cause of the crash. We eventually noticed that the Erlang shell's memory usage grew monotonically as the benchmarking function repeatedly invoked our ray tracing function. With a lucky guess we were able to fix the problem by explicitly invoking Erlang's garbage collector between timed invocations of the ray tracing function. (Tuning Erlang's generational garbage collection policy via `erlang:system_flag(fullsleep_after, ...)` did not solve the problem.) We believe that Erlang would be improved by addressing this issue.

7. Conclusions and Future Work

We investigated the relative suitability of C++ and Erlang for implementing mathematically intensive algorithms of interest to the U.S. Navy on a several computing platforms. On an industry-standard Intel x86 SMP system, Erlang proved easier to debug and may have permitted more compact programs. However C++'s 12+ times performance advantage and better scalability seem to be an insurmountable barrier to using current implementations of Erlang for these performance-critical Navy applications.

Despite Erlang's poor speed we feel that functional programming shows considerable promise for our domain. One candidate for future examination is concurrent ML (Reppy 2007).

Acknowledgments

Funding was made possible by the Office of Naval Research under a Naval Undersea Warfare In-house Laboratory Independent Research Project. The ONR Sponsor for this work is Kirk Jenne.

References

Lawrence E. Kinsler, Austin R. Frey, Alan B. Coppens, and James V. Sanders. *Fundamentals of Acoustics, 4th Ed.* Danvers, MA, USA: John Wiley & Sons, Inc., 2000. ISBN 0-471-84789-5

Joe Armstrong. *Programming Erlang* Raleigh, NC, USA: The Pragmatic Bookshelf, 2007. ISBN-13: 978-1-934356-00-5

Thomas Chen, Ram Raghavan, Jason Dale, Eiji Iwata. *Cell Broadband Engine Architecture and its first implementation* http://www.ibm.com/developerworks/power/library/pa-cellperf/

J.H. Reppy. *Concurrent Programming in ML* Cambridge University Press, 2007. ISBN-13: 978-0521714723