Effective Parallelization Strategies for Scalable, High Performance Radio Frequency Ray Tracing

Christiaan Gribble Jefferson Amstutz
Applied Technology Operation
SURVICE Engineering Company
Belcamp, MD 21017
Email: {christiaan.gribble,jeff.amstutz}@survice.com

***Abstract*—We present StingRay, an interactive environment for combined RF simulation and visualization based on ray tracing. StingRay is explicitly designed to support scalable, high performance simulation and visualization of RF energy propagation in complex urban environments using modern, highly parallel computer architectures. We explore three strategies for exploiting parallelism in StingRay and provide evaluations of their scalability and performance on a modern workstation-class system. Results show that a more scalable, higher performing version of StingRay is possible with careful attention to the expression of task-level parallelism in OpenMP.**

I. Introduction

Understanding the propagation of radio frequency (RF) energy in the presence of complex outdoor terrain features, such as in urban environments, is critical to planning, optimizing, and analyzing wireless communication and data networks. Unfortunately, simulating and visualizing RF energy propagation can be a difficult and time-consuming task because RF energy propagates throughout these environments via a combination of direct line-of-sight, reflection, and diffraction, all of which must be modeled accurately to obtain high fidelity results. Moreover, features within the environment—trees, buildings, and so forth—typically exhibit different permittivity and absorption properties with respect to the energy being measured and, therefore, impact simulation fidelity. Finally, energy produced from a single transmitter may arrive at a given point in space from a variety of paths, each with a different length and, thus, with different time-delay characteristics.

Many planning, optimization, and analysis tasks are difficult with existing RF simulation models. These models run slowly for moderate to large numbers of transmitter/receiver pairs, in part because they are not designed to take advantage of modern, highly parallel computer architectures. Moreover, these models do not account for noise caused by multi-path scatter, but instead use unacceptably large estimates for its effects.

We present StingRay, an interactive environment for combined RF simulation and visualization based on ray tracing (Fig. 1). StingRay is explicitly designed to support scalable, high performance simulation and visualization of RF energy propagation in complex urban environments by exploiting modern multicore and many-core hardware architectures. Our explicitly parallel codes scale gracefully: for a machine with more cores, either similar quality results are computed in less time or higher fidelity results are computed in similar time. Our ray-based approach also offers high performance: simulations involving 100 million or more rays complete in seconds or minutes using these platforms, thereby allowing users to quickly identify propagation phenomena of interest, ultimately reducing time-to-insight. Finally, our ray-based simulation approach also offers high fidelity results that include important physical effects—including more accurate multi-path scatter—for RF prediction.

We present an overview of radio frequency ray tracing (RFRT) and the StingRay system architecture. We then explore three strategies for exploiting parallelism in StingRay and evaluate their performance on a modern workstation-class system. Results show that a more scalable, higher performing version of StingRay is possible with careful attention to the expression of task-level parallelism in OpenMP.

II. Background

Ray-based methods have been used to approximate the solution of wave equations for electromagnetic fields in non-dissipative media for at least four decades [1], [2]. Typically, these methods proceed in two steps. First, ray paths connecting source and receiver are found; in complex environments, this step is often the most time-consuming. Second, the wave equation is applied to compute field transport along identified ray paths. These classical ray-based approaches typically require hours of run time to simulate areas out to a kilometer or more.

In contrast, our approach to RF modeling uses ray tracing techniques from computer graphics [3], [4], or so-called *optical ray tracing*. Optical ray tracing simulates the propagation of visible light in complex 3D environments and elegantly handles dominant visual phenomena such as reflection, refraction, and shadows. We observe that RF energy is also a form of electromagnetic energy, albeit at a very different frequency than visible light; thus, methods in optical ray tracing offer a possible approach for simulating physical phenomena in the RF domain.

RFRT offers several advantages over traditional RF simulation methods. First, modifications necessary to capture important physical phenomena such as diffraction and interference are fairly straightforward. Second, RFRT generates the full signal trajectory, allowing computation and visualization of signal characteristics that are extremely costly, or even impossible, with other methods. Finally, with careful attention to explicit parallelization, RFRT scales effectively with processor count. Together with with nearly 35 years of research in high performance techniques [5], these characteristics make

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

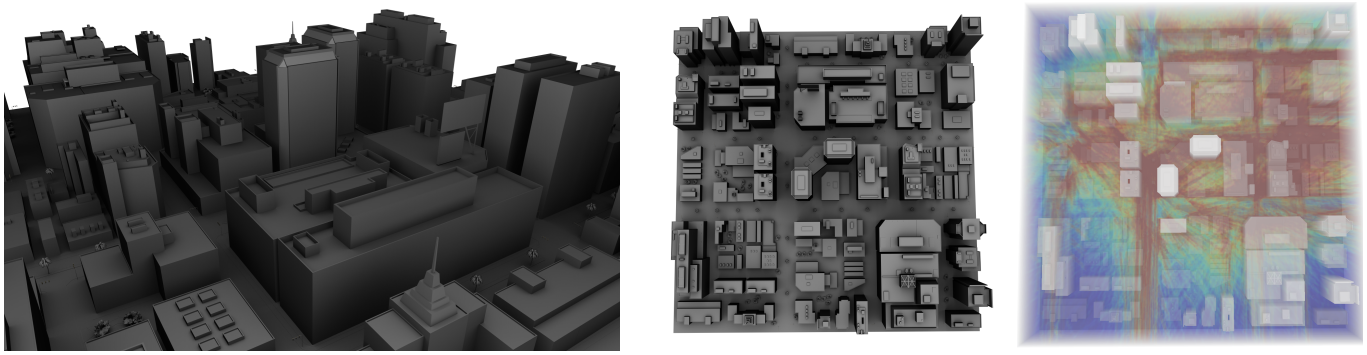

Fig. 1. *High performance radio frequency ray tracing with StingRay.* A high performance ray-based RF simulation engine and a collection of visualization components are integrated via an extensible GUI to support visual analysis of RF energy propagation. Here, StingRay interactively simulates and visualizes RF energy propagation in an urban environment.

optical ray tracing an ideal method on which to base an interactive combined RF simulation and visualization tool for understanding RF propagation phenomena.

In particular, Monte Carlo path tracing [4] formulates a solution to the wave equations for electromagnetic fields using a geometric optics approximation that models interesting visual phenomena. Path tracing probabilistically selects just one path of a (possibly) branching tree at each ray/object interaction. This approach drastically reduces the number of interactions that must be computed, thereby improving computational efficiency. We adapt the path tracing algorithm to compute energy propagation characteristics, including those arising from wave-based phenomena, in the RF domain (Algorithm 1).

Algorithm 1 RF simulation with Monte Carlo path tracing

```
1: function TRACERAY(ray, depth)
2:     event ← OBJECTS.INTERACT(ray)
3:     if event then
4:         ray ← GENERATERAY(newEfield)
5:         TRACERAY(ray, depth + 1)
6:     end if
7: end function
```

Together with our collaborators at the University of Utah, we previously developed the Manta-RF radio frequency ray tracing system [6]–[8]. As in classical ray-based techniques, Manta-RF utilizes the ray concept; however, transport properties are computed directly by launching many rays—on the order of 10^8 to 10^{11} or more—and using the statistical properties of ray distribution and density to represent received power. Whereas classical rays are defined by the order and location of their interactions with environmental features, rays in Manta-RF are more appropriately described as RF photons—discrete packets of electromagnetic energy in the RF portion of the spectrum. Validation against several measured datasets shows that a Monte Carlo approach to ray-based RF simulation offers high fidelity results comparable to those produced by classical ray-based methods (Fig. 2). StingRay builds on the ray-based RF simulation techniques developed for Manta-RF, but pays careful attention to the expression of task-level parallelism in OpenMP to scale effectively with processor count.

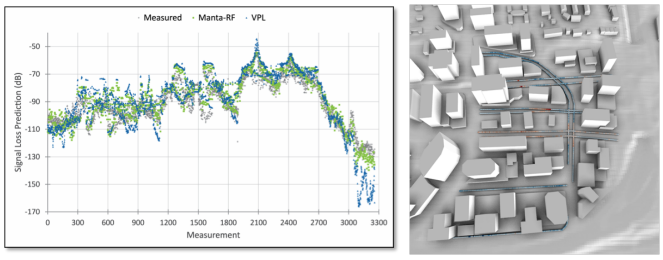

Fig. 2. *High fidelity RF simulation via Monte Carlo path tracing.* Comparison of signal loss predictions using Manta-RF and VPL [9] (left), a classical ray-based RF simulation approach, against measured data from Rosslyn, VA (right). The data show that a Monte Carlo path tracing approach to RF modeling provides high fidelity results comparable to those produced by classical ray-based methods. (Data and image courtesy of Konstantin Shkurko, University of Utah.)

III. StingRay System Architecture

Combined simulation and visualization of RF energy propagation promotes deeper understanding of these phenomena, thereby reducing time-to-insight for critical analysis tasks. However, the complexity of typical RF analysis scenarios, including the underlying physical environment and the sheer number of ray/object interactions, can lead to issues with scale and visual clutter. Such issues necessitate a flexible, interactive environment in which users control both inputs and results at runtime.

StingRay satisfies these constraints via an extensible, loosely coupled plug-in architecture. The simulation and visualization components promote flexibility with user-controlled features, while an extensible GUI enhances a user's ability to perform debugging and analysis tasks by enabling easier navigation and exploration of the data in real-time.

The key components of the StingRay system architecture (Fig. 3) combine to form an analysis process that is functional, flexible, and extensible. The design of StingRay leverages the following concepts:

- **Plug-in architecture.** StingRay is built around a set of configurable components that follow a specific design pattern to create a flexible infrastructure in which to implement RF simulation and visualization. We provide a set of core components to perform common tasks, but the plug-in architecture enables a

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

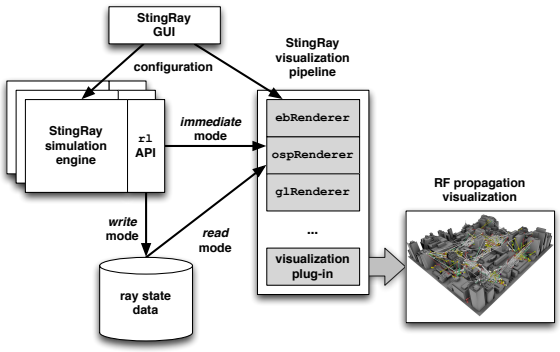

Fig. 3. *StingRay system architecture.* Interactive simulation and visualization coupled with a ray state recording and processing API and a loosely coupled plug-in architecture creates a functional, flexible, and extensible system for visual analysis. A set of core components implements RF simulation and visualization, but the plug-in architecture enables a programmer to extend the core facilities with arbitrary functionality. These components are integrated via a GUI that enables comprehensive control of the entire analysis process.

programmer to create new components and extend the core facilities with arbitrary functionality.

- **Pipelined operation.** StingRay uses a pipeline model for simulation and visualization, coupled with lazy evaluation for necessary values to avoid recomputation in later stages. The pipeline model leads to a layered visualization approach in which results of individual components are combined, under control of the user and at runtime, to achieve the desired result.
- **Extensible GUI.** The simulation and visualization components are integrated via an extensible GUI to enable comprehensive control of the entire analysis process. These components can tailor their user interface, exposing input parameters in a manner consistent with the functionality they provide.

This design enables fine-grained control of the entire analysis process, making StingRay ideally suited to a wide variety of RF simulation and visualization tasks.

The StingRay RF simulation engine implements the key RF propagation phenomena using a collection of C++ objects, including scene geometry, diffraction edge proxy geometry, and ray path loggers. Simulation functionality is exposed to client applications through a multi-threaded controller object that provides a straightforward API.

StingRay adopts a layered visualization approach in which elements from separate visualization components are composited to generate the final image. StingRay currently supports several visual elements for RF visualization, including underlying scene geometry, proxy geometry, ray glyphs, and scalar volume data generated from simulation results (Fig. 4).

IV. Parallel RF Simulation with OpenMP

Certain ray tracing operations benefit from vectorization with single instruction, multiple data (SIMD) instruction sets. Effective vectorization requires careful attention to *ray coherence*, or the tendency of spatially local rays traveling in similar directions to perform similar operations during rendering—to traverse the same nodes in an acceleration structure, for example, or to test the same primitives for intersection.

Similarly, ray tracing applications benefit from parallelization by exploiting task-level parallelism among rays. For example, in ray-based rendering, pixel color calculations are embarrassingly parallel: the computations along each ray are independent of those required by every other ray. In the same way, each path traced from a transmitter in RFRT is independent of every other path.

In this section, we explore three strategies to exploit task-level parallelism in ray-based RF simulation, and we evaluate the scalability and performance of each strategy on multicore and many-core hardware architectures.

A. Motivation

Modern ray tracing engines—for example, Intel's Embree ray tracing kernels [10]—exploit ray coherence to achieve high performance. In a typical ray-based renderer, primary rays, in particular, exhibit a high degree of coherence, and much of the recent work in high performance ray tracing examines techniques to identify and exploit coherence for other types of rays, including shadow rays, reflection rays, refraction rays, and so forth.

In this case, vector processing with modern SIMD instruction sets can accelerate low-level traversal, intersection, and shading operations to improve performance. However, paths traced by the RF simulation engine vary in both path length and computational demand. As path length increases, even initially coherent rays tend to diverge, both spatially and in terms of path length itself. This behavior is often exaggerated in RF simulation due to the nature of path generation—in this case, even primary rays tend not to exhibit coherence.

Vectorization is particularly difficult in this context, and simply relying on a high performance ray tracing engine to accelerate RFRT is not sufficient to achieve our performance goals. Other opportunities to exploit parallelism—for example, the task-level parallelism noted above—must be identified and expressed carefully to maximize simulation performance.

B. Task-Level Parallelism

For example, one way to parallelize the basic RF path propagation loop (Fig. 5) is to simply assign live path segments to each thread in a thread pool.[1] This approach attempts to balance work across threads by assigning the same type of task to each one: threads simply trace the next segment of the assigned path in the core simulation loop.

This path-level parallelism is expressed in OpenMP by allocating a collection of live paths among threads using an `omp parallel for` construct over the core loop (Fig. 6). Each iteration of this loop traces one segment of the corresponding path; if any path completes, a new path is generated, and propagation begins with its first segment in the next iteration. The `omp parallel for` construct is embedded in a `while` loop that stops only after the simulator leaves the `running` state.

[1]Complete source code for the RF simulator featured here is available at http://www.rtvtk.org/~cgribble/research/rfrt-bench/.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

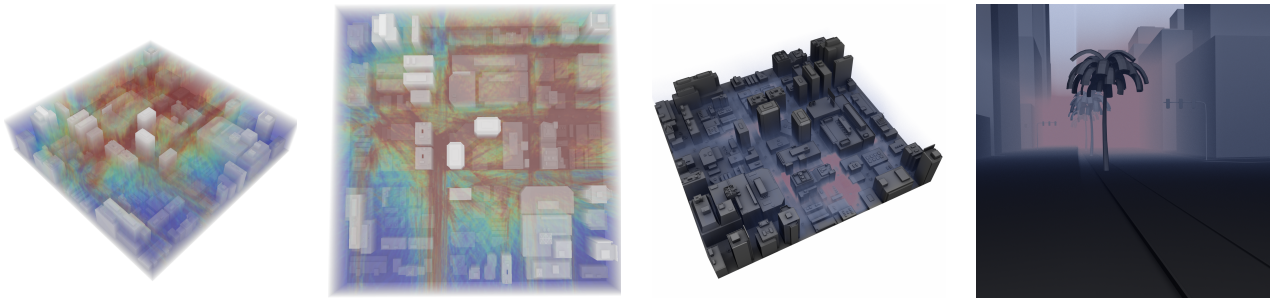

Fig. 4. *Scalar volume data generated from RF simulation results.* By capturing detailed information about the full path generated for each transmitter sample, StingRay enables computation and visualization of signal characteristics that are extremely costly, or even impossible, with other RF simulation methods. Here, RF energy characteristics are visualized using traditional volume rendering (left) and as a participating medium (right).

```
while (running)
{
  // Generate path originating from current transceiver
  Transceiver* tr = transceivers[txrx%ntxrx];
  Path path(tr->samplePrimaryRay(), tr->getPower());

  // Trace path to completion
  while (!path.traceNextRay(scene))
    ;

  // Record state, advance to next transceiver
  ...

} // End while
```

Fig. 5. *Single-threaded RF simulation loop.* In each iteration of the outer `while` loop, this implementation traces to completion the next path from the current transceiver. Importantly, this version does not exploit opportunities for task-level parallelism inherent to RF path propagation; however, it serves as the starting point for expressing task-level parallelism and as the baseline for understanding the impact of the parallelization strategies we explore. (Code snippet from `StingRay/Simulation.cpp:408--422`.)

```
while (running)
{
  #pragma omp parallel for num_threads(nthreads)
  for (auto i = 0; i < npaths; ++i)
  {
    Path& path = livePaths[i];

    // Trace next segment in path... if current path
    //   completes, create a new one
    if (path.traceNextRay(scene))
    {
      // Create a new path
      ...
    }

  } // End for

} // End while

// Simulation has been stopped, finish in-flight paths
//   and terminate
...
```

Fig. 6. *Using OpenMP `parallel for` to express path-level parallelism.* The first approach to parallel RF path propagation simply assigns live path segments to each thread in a thread pool, expressed in OpenMP using an `omp parallel for` construct. However, this approach induces unnecessary synchronization among threads and hinders parallel performance. (Code snippet from `StingRay/Simulation.cpp:451--476`.)

To understand the impact of parallelization using the OpenMP `parallel for` construct, we measure performance of the RF simulator on a modern multicore workstation using a varying number of threads. The test machine features

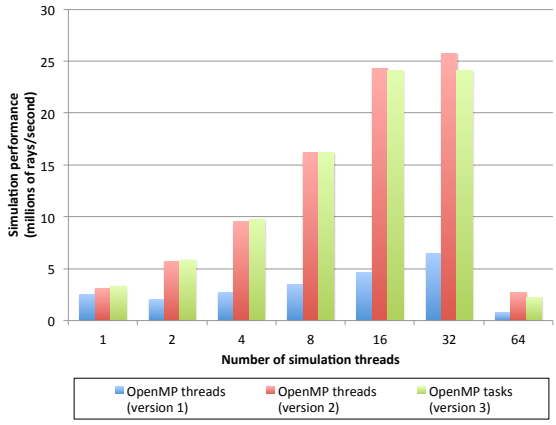

Fig. 7. *Parallel RF simulation with OpenMP.* Tracing RF propagation paths is an embarrassingly parallel operation; however, careful attention to the expression of task-level parallelism in this context is required to maximize performance. These results depict scalar simulation performance, measured in terms of millions of rays per second, on an Intel Xeon E5-2699 processor using each of the three OpenMP parallelization strategies we explore.

an Intel Xeon E5-2699 processor with 36 cores running at 2.30 GHz and 64 GB of RAM. The machine is also equipped with a Intel Xeon Phi 7120A coprocessor with 61 cores running at 1.24 GHz and 16 GB of RAM. To establish baseline performance, we first execute the serial RF path propagation loop (Fig. 5) in a single thread for five seconds using the city model (Fig. 1 and Fig. 4) with a collection of eight transmitters distributed throughout the environment. The naive implementation achieves about 3.74 million rays per second (Mrps) on the test machine.

Next, we execute the simulator using the `omp parallel for` parallelization strategy and scale number of threads from one to 64. Each configuration runs for five seconds, and we again measure the resulting performance in Mrps (Fig. 7).

As can be seen, when implemented in this way, simulator performance does not scale well with thread count. In fact, this version requires at least 16 threads before even a modest increase in performance is evident. This behavior is a result of an implied barrier at the end of the `for` loop imposed by the `omp parallel for` construct. Each path segment will likely have a different computational cost due to variations in complexity of ray/object interactions or path length. Unfortunately, the barrier prevents other threads from making forward progress if any one segment is expensive to trace or if any one

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

```
#pragma omp parallel num_threads(nthreads)
{
  while (running)
  {
    #pragma omp for nowait
    for (auto i = 0; i < npaths; ++i)
    {
      Path& path = livePaths[i];

      // Trace next segment in path... if current path
      //   completes, create a new one
      if (path.traceNextRay(scene))
      {
        // Create a new path
        ...
      }

    } // End for

  } // End while

} // End omp parallel

// Simulation has been stopped, finish in-flight paths
//   and terminate
...
```

Fig. 8. *Using OpenMP `parallel` and `for nowait` to express path-level parallelism.* The first approach to parallel RF path propagation induces unnecessary synchronization among threads and hinders parallel performance. This version more carefully expresses the available parallelism using a combination of `omp parallel` and `omp for`, together with the `nowait` clause, to maximize performance. (Code snippet from `StingRay/Simulation.cpp:498--537`.)

thread must create a new path. This parallelization strategy, though straightforward, induces unnecessary synchronization.

More careful use of OpenMP threads alleviates this issue (Fig. 8).[2] As before, this implementation spawns some number of threads using the `omp parallel` construct, but does so at the level of the `while` loop—not the nested `for` loop—so that each thread now executes the entire simulation loop. Note that there is an implied barrier at the end of the `omp parallel` construct.

Within the `while` loop, the `omp for` construct indicates that the current team of threads works together to execute the iterations of the corresponding loop. However, the `nowait` clause at the end of the `omp for` construct allows a thread to execute additional iterations without waiting for other threads: this clause removes the unnecessary synchronization that was implied by the `omp parallel for` construct in the first OpenMP parallelization strategy. As before, threads run until the simulator is stopped, at which point threads complete their current path and the simulator terminates.

This approach improves performance by as much as $6.9\times$ relative to single-threaded RF path propagation and recovers a factor of about $3.6\times$ (on average) relative to the first OpenMP parallelization strategy (Fig. 7). In this case, each thread actually executes independently of every other thread—without synchronization—and performance scales effectively.

[2]Modfications to the OpenMP `parallel for` approach other than use of `nowait` as described here—for example, the `schedule` clause and various chunk sizes—show no benefit over `nowait`. In particular, the performance impact of combining `static` or `dynamic` scheduling with various chunk sizes and live path counts (in the range [36, 9180]) is either negligible or negative, as additional paths increase cache pressure in an already incoherent problem. We thus present details of only the `nowait` approach here.

```
#pragma omp parallel num_threads(nthreads)
{
  #pragma omp single
  {
    for (auto i = 0; i < npaths; ++i)
    {
      #pragma omp task
      {
        Path& path = livePaths[i];

        while (running)
        {
          // Trace next segment in path... if current
          //   path completes, create a new one
          if (path.traceNextRay(scene))
          {
            // Create a new path
            ...
          }

        } // End while

        // Simulation has been stopped, finish in-flight
        //   paths and terminate
        ...

      } // End omp task

    } // End for

  } // End omp single

} // End omp parallel
```

Fig. 9. *Using OpenMP tasks to express path-level parallelism.* A third approach to parallel RF path propagation generates tasks using a single thread, expressed by the OpenMP `omp single` construct, and defers synchronization until the end of the surrounding `omp parallel` construct. As in the second OpenMP parallelization strategy, each path-trace task actually executes independently of every other task, and performance scales nicely. However, the task generation loop imposes unnecessary serialization, so absolute performance is generally not as high as with the second approach. (Code snippet from `StingRay/Simulation.cpp:549--593`.)

An alternative expression of path-level parallelism using OpenMP tasks can also be used to avoid unnecessary synchronization (Fig. 9). Here, the implementation spawns threads using the `omp parallel` construct, but creates `npaths` tasks, one for each live path, using a single thread from the pool. This arrangement is expressed by embedding the `omp single` construct immediately within the `omp parallel` construct. In this case, the first thread that arrives at the `omp single` construct executes the task creation loop. A task created in this manner can be executed immediately by any thread in the pool. Finally, once task creation is complete, the corresponding thread joins the other threads in executing the newly created tasks. As before, threads run until the simulator is stopped, at which point threads complete their current path and the simulator terminates.

This approach improves performance by as much as $6.5\times$ relative to single-threaded RF path propagation and recovers a factor of about $3.5\times$ (on average) relative to the first OpenMP parallelization strategy (Fig. 7). As in the second strategy, each task actually executes independently of every other task and performance scales with thread count. However, the task generation loop imposes unnecessary serialization in the RF path propagation loop, so absolute performance is generally not as high as with the second approach.

Task-level parallelism also improves performance of RF simulation on Intel Xeon Phi coprocessors (Fig. 10). Behavior

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

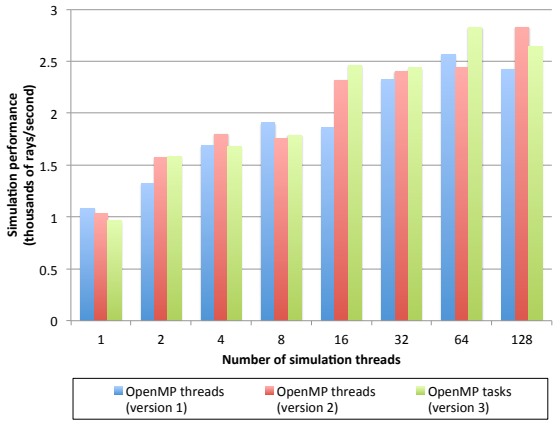

Fig. 10. *Parallel RF simulation on an Intel Xeon Phi coprocessor.* These results depict scalar simulation performance, measured in terms of thousands of rays per second, on an Intel Xeon Phi 7120A coprocessor using task-level parallelism with one to 128 threads. Behavior of these codes is similar to that of the corresponding CPU versions; however, absolute performance suffers due to scalar code paths and memory bandwidth bottlenecks.

of these codes on Xeon Phi is similar to that of the corresponding codes on the CPU: the second and third OpenMP parallelization strategies best express the available parallelism and generally achieve the best performance among the strategies we consider here. However, absolute performance suffers due to scalar code paths and memory bandwidth bottlenecks. As noted above, vectorization of RF path propagation is particularly difficult, and is an important area of future work.

V. Conclusions and Future Work

Radio frequency simulation and visualization is critical to planning, analyzing, and optimizing wireless networks. An interactive tool supporting visual analysis of RF propagation characteristics in complex environments will enable users to better understand RF propagation phenomena. StingRay provides an interactive combined RF simulation and visualization environment that satisfies these constraints. Derived from first principles, StingRay's ray-based RF simulation engine provides a high fidelity approach for RF prediction in complex environments. StingRay is explicitly designed for modern multicore and many-core hardware architectures and provides scalable, high performance simulation and visualization capabilities for a variety of RF analysis tasks.

We explore three strategies to express task-level parallelism in RF simulation using OpenMP. Careful attention is required to achieve a scalable, high performance version of the core RF path propagation loop. Results show that using OpenMP `parallel` and `for nowait` permits effective scaling on both Intel Xeon CPUs and Intel Xeon Phi coprocessors, improves performance by nearly $7\times$ over the serial implementation, and recovers as much as $3.6\times$ over a naive use of OpenMP `parallel for`.

Though we explore only task-level parallelism in this work, ray tracing applications can benefit from modern engines' ability to use vectorization in low-level operations such as traversal, intersection, and shading. Effective use of vector processing in RF path propagation will be required to overcome bottlenecks on current Intel Xeon Phi coprocessors, and these techniques would likely benefit an implementation on graphics processing units (GPUs) as well. We would like to investigate techniques to effectively exploit vectorization for the highly incoherent ray distributions generated by RFRT on Xeon Phi coprocessors and modern GPUs.

StingRay implements a novel RFRT methodology based on Monte Carlo path tracing from computer graphics. Additional efficiency improvements could be gained by leveraging more sophisticated ray tracing algorithms from the computer graphics literature. In particular, bidirectional path tracing (BDPT) is a Monte Carlo ray tracing algorithm that generalizes the classical path tracing algorithm. Paths originating from both source and receiver are first computed using the classic path tracing algorithm; path vertices are then connected using occlusion rays, and energy is accumulated at the receiver for unoccluded paths. Results in computer graphics show that BDPT performs better than classical path tracing for environments in which indirect (non-LOS) contributions are most significant, as is the case for typical RF simulation scenarios. We would therefore like to investigate the application of BDPT to RF simulation to further improve the performance and accuracy of StingRay.

Acknowledgments

We thank James Reinders (Intel), who first suggested the OpenMP `for nowait` variation. We also thank the Intel Parallel Visualization Engineering Team, including Jim Jeffers, Ingo Wald, Carsten Benthin, Greg P. Johnson, Greg S. Johnson, Brad Rathke, and Sven Woop, for their contributions. Finally, we thank Erik Brunvand, Thiago Ize, and Konstantin Shkurko (University of Utah); Lee Butler (US Army Research Laboratory); and Henry Bertoni (NYU Polytechnic) for their contributions to the early research and development efforts that led to the methodology on which StingRay is based.

References

[1] G. A. Deschamps, "Ray techniques in electromagnetics," *Proceedings of IEEE*, vol. 60, no. 9, pp. 1022–1035, 1972.

[2] L. B. Felsen and N. Marcuvitz, *Radiation and Scattering of Waves*. Englewood Cliffs, NJ: Prentice Hall, 1973.

[3] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[4] J. T. Kajiya, "The rendering equation," in *SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986, pp. 143–150.

[5] A. Keller, I. Wald, T. Karras, S. Laine, J. Bikker, C. Gribble, W.-J. Lee, and J. McCombe, "Ray tracing is the future and ever will be..." in *ACM SIGGRAPH 2013 Courses*, 2013.

[6] K. Shkurko, T. Ize, C. Gribble, E. Brunvand, and L. A. Butler, "Simulating radio frequency propagation via ray tracing," Poster, *GPU Technology Conference 2013*. Available at http://www.gputechconf.com/page/posters.html, 2013, last accessed 3 April 2013.

[7] E. Brunvand, T. Ize, and E. Ribble, "Ray trace applications to radio frequency (RF) propagation," Final report, US Army Research Office, Contract No. W911NV-07-D-0001, TCN 09-063, 2009.

[8] E. Brunvand and T. Ize, "Radio frequency (RF) ray trace propagation analysis," Final report, US Army Research Office, Contract No. W911NF-07-D-001, TCN 10-218, 2011.

[9] G. Liang and H. L. Bertoni, "A new approach to 3D ray tracing for site specific propagation modeling," in *IEEE Vehicular Technology Conference*, 1997, pp. 853–863.

[10] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree - a kernel framework for efficient CPU ray tracing," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 143:1–143:8, July 2014.