Efficient Shared Memory Parallelization of FDTD

Alec Hammond

1 Introduction

Large scale photonic integration shows potential to transform fields like quantum computing, machine learning, and telecommunications. In order to design compact and efficient integrated photonic devices, however, current design methodologies must be revamped to encourage faster design cycles. Topology optimization, for example, requires iteratively solving Maxwell's equations, a numerically expensive task that is clearly the bottleneck toward faster device design.

To overcome this challenge, I accelerated an open source finite difference time domain (FDTD) code that I co-developed (1). The codebase already supports MPI for extremely large simulations. That being said, load balancing issues often prevent efficient scaling for even modest simulations and design cycles (2).

While OpenMP is not a substitute for MPI, it offers several advantages for the classes of problems I intend to solve. On the one hand, MPI can scale the simulation domain to any arbitrary compute configuration, which is essential for problems that don't tractably fit inside a single node. That being said, the majority of inverse-design problems are compact and fit in single-node resources. Naively leveraging MPI in this case exhibits exceptional computational waste. This is compounded in a linear fashion as the optimization algorithm calls the solver every iteration. OpenMP enables dynamic load balancing and eliminates the costly communication paradigm required by MPI, thereby accelerating the overall design process.

In summary, I revamped the existing codebase to allow for OpenMP parallelism. **Specifically, I rewrote 11 loop macros, refactored over 106 different parallel loops, adopted a "first-touch" initilization scheme, coalesced memory access to leverage vectorization (basic SIMD operations), and profiled the performance across four different examples**. Unfortunately, the final OpenMP build still underperformed the previous MPI build, despite several weeks of refactoring the original codebase. I performed several numerical experiments on quantities like the OpenMP scheduler and thread chunk size to verify the drawbacks of the OpenMP method.

2 Previous Work & Relevant Background

The most common nanophotonic design tools leverage the finite-difference time-domain method (FDTD). Since the FDTD algorithm is easy to implement, it is straightforward to enable codes that support multiple physics (heterogeneous material types, boundary layers, dispersion, nonlinearities, adjoint methods, etc). Despite it's simplicity, the FDTD algorithm is a fullwave solver that produces accurate solutions to Maxwell's equations, provided the simulation resolution is sufficiently high.

The simulation time itself increases as $O(n^4)$ with resolution. Consequently, accurate simulations require intense computational resources to resolve the requested geometry and physics. Luckily, since the algorithm simply loops over the simulation grid implementing various finite difference stencils, parallilization is also rather straightforward. In fact, several parallel FDTD codes, commercial and open-source, already exist. For example, gprMax is a popular FDTD code geared toward ground penetrating radar and supports both CUDA (3) and OpenMP (4) acceleration. Similarly, B-CALM is a GPU enabled FDTD code designed for plasmonic device simulations (which require extremely high simulation resolutions) (5). Other commercial codes, like AxFDTD solver (6), CST solver (7), and SEMCAD (8) all support both OpenMP and GPU parallelism.

While several parallel codes already exist, they all rely on simple timestepping kernels and lack the flexibility needed to model photonic integrated circuits. For example, None of the codes simultaneously support nonlinear materials, dispersion, and adjoint variable methods. Consequently, I opted to parallelize a code that I developed that I *know* supports the features I need.

MIT Electromagnetic Equation Propagation (MEEP) is an open-source FDTD code that supports everything I need to efficiently perform topology optimization on nanophotonic devices. As previously mentioned, the codebase already supports MPI parallelism for distributed computation. Each of MEEP's kernels iterate over the simulation domain using a series of abstracted loop macros that are defined in a uniform header file. This is largely possible because even if the kernels perform wildly different computations, they all operate over the same fundamental data arrays (electric fields, magnetic fields, etc). The MPI engine then chunks the arrays and allocates these subdomains to particular processes. While all processors will perform time-stepping (the most expensive kernel) on their chunk, not all processors will be required to perform other kernels on their chunk. Even in the simple case where only time-stepping is required for all chunks, it's important to note that the cost of time-stepping heavily depends on the contents of the respective chunk.

Consequently, the current MPI implementation significantly suffers from "load balancing" issues. Deciding how to divide the work among the available processors before the simulation starts is close to impossible. Figure 1 compares the relative time it takes for each processor to complete various

kernels. The computation time for each worker varies wildly. Some processors even sit idly simply because of the uniform chunk splitting required by MPI implementations.

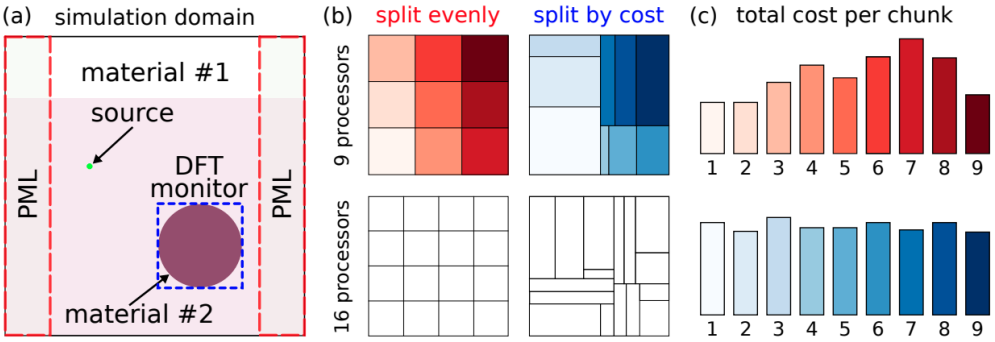

Figure 1: The differences between static and dynamic load balancing. The designer first prescribes a simulation domain with various physics involved. The complexity of the domain is difficult to determine before running the simulation. Consequently most SIMD implementations (e.g. using MPI) split the computational cell at the start of the simulation. A better scheme would allow threads to dynamically shift the size of the domains as some regions require more work than others. This essentially equates to splitting the domain by computational cost.

Alternatively, OpenMP supports various "dispatch" paradigms where several smaller chunks are passed to threads as they become available. In effect, the domain is split by cost, rather than split evenly. This ensures that no processor has to idle and wait while its neighbors finish their respective tasks. In a sense, OpenMP is able to dynamically load balance the simulation domain. Even more important, the chunk division can change between kernels (time-stepping vs Fourier Transforming) and from iteration to iteration.

Also note that almost as much time is spent on MPI communication as time-stepping. While OpenMP will also have some communication overhead, it should be significantly less than the current implementation.

2.1 Computational Complexity and Kernel Analysis

The core time-stepping algorithm is the most expensive kernel and deserves the most attention. We can quantify the total time of the current MPI implementation using

$$T(\vec{p}) = \underbrace{\frac{W(\vec{p}) \times N}{P}}_{\text{computation}} + \underbrace{C(\vec{p}) \times S}_{\text{communication}} \tag{1}$$

where N is the total number of grid points, P is the number of processors, and S is the maximum surface area of any processor's subdomain. (Since time-stepping is synchronous, CS is determined by the process that requires the most communication.) The surface area, S, can be minimized to some extent, but will always increase as the number of processes increases. The actual kernel cost (W) depends on the simulation domain (p) but is predicable to first order. Similar to the communication cost, the slowest processor will dominate the computation time since all other processors must wait.

By looping over our domain using OpenMP, we significantly minimize the overhead caused by the C×S communication cost (since communication overhead is minimized under a shared memory paradigm). Furthermore, we can attempt to perform more FLOPs/data transfer during the main kernel, which further increases the efficiency of the computation term.

That being said, there is only so much we can do to maximize our computation/communication ratio. It is well known that the FDTD algorithm is heavily bound by memory bandwidth. Moving entirely "off the roof" of the roofline model is not feasible without completely rewriting the Yee grid scheme (which is akin to starting from scratch). Instead, we can still enjoy significant performance gains simply by parallelizing with OpenMP and caching where necessary.

3 Parallelization Overview

The code's fundamental looping routines are rather complicated. The FDTD algorithm relies on a staggered grid mechanism, known as the Yee grid (9). Different fields and their respective components are stored at different points in space *and* time. Since the grid is still fundamentally Cartesian, however, we can still use the same loop mechanism for all fields and materials, provided it is general enough to handle all of these extreme cases.

This flexibility comes with a price, however. Parallelizing these loop macros requires some sophisticated "trickery" to ensure all of the existing stencils behave correctly. For example, we can collapse some of our nested loops (since we are simply looping over three dimensions of a row-major array) using an OpenMP clause. However, we must first abstract out the initial array starting point calculations by unwrapping a single loop iteration. Tricks like these seem unnecessary, but they let us preserve the variable scope of the existing macro calls without having to change the entire codebase (over 300,000 lines of c++ code). Listing 1 illustrates the rafactoring of one particular loop macro using multiple collapsed loops, unwrapped lops, and SIMD calls.

```
#define PS1LOOP_OVER_IVECS(gv, is, ie, idx)                                    \
_Pragma("unroll(1")                                                            \
for(ptrdiff_t loop_is1 = (is).yucky_val(0), loop_is2 = (is).yucky_val(1),      \
    loop_is3 = (is).yucky_val(2), loop_n1 = ((ie).yucky_val(0) - loop_is1) / 2 + 1,\
    loop_n2 = ((ie).yucky_val(1) - loop_is2) / 2 + 1,                          \
    loop_n3 = ((ie).yucky_val(2) - loop_is3) / 2 + 1,                          \
    loop_d1 = (gv).yucky_direction(0), loop_d2 = (gv).yucky_direction(1),      \
    loop_s1 = (gv).stride((meep::direction)loop_d1),                           \
    loop_s2 = (gv).stride((meep::direction)loop_d2), loop_s3 = 1,              \
    idx0 = (is - (gv).little_corner()).yucky_val(0) / 2 * loop_s1 +            \
           (is - (gv).little_corner()).yucky_val(1) / 2 * loop_s2 +            \
           (is - (gv).little_corner()).yucky_val(2) / 2 * loop_s3,             \
              dummy_first=0;dummy_first<1;dummy_first++)                       \
_Pragma("omp parallel for collapse(2)")                                        \
  for (ptrdiff_t loop_i1 = 0; loop_i1 < loop_n1; loop_i1++)                    \
    for (ptrdiff_t loop_i2 = 0; loop_i2 < loop_n2; loop_i2++)                  \
      _Pragma("omp simd") \
      for (ptrdiff_t loop_i3 = 0; loop_i3 < loop_n3; loop_i3++)                \
        _Pragma("unroll(1") \
    for (ptrdiff_t idx = idx0 + loop_i1 * loop_s1 + loop_i2 * loop_s2 + loop_i3,  \
    dummy_last=0;dummy_last<1;dummy_last++)
```

Listing 1: Sample loop parallelization using complicated macros and folding.

In addition to completely rewriting the main loop macros, I had to modify over 100 other loops that didn't directly call theses loop macros. This was primarily to ensure I was leveraging a "first-touch" policy.

4 Results

4.1 Hardware and Build Details

I ran all simulations using an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz with 40 cores (2 sockets, 20 cores/socket) The node has 190 GB RAM @ 2.933GHz (6 slots, 2 controllers/NUMA at 64 bits each). Hyperthreading is disabled. The L1 cache is 32 kB, the L2 cache is 1 MB, and the L3 cache is 28 MB (shared across the socket).

All software was built using GCC 8.3.0, which ships with OpenMP 4.5. The MPI versions of the code were built using mpicc and mpicxx from MVAPICH 2.3.2. The main FDTD engine was interfaced using SWIG bindings and python scripts to streamline the example batching.

4.2 Examples

In order to thoroughly benchmark any OpenMP performance gains, **I constructed four different example problems** that each involve hetergenous physics stencils. Figure 2 illustrates each example. The first example involves a nanophotonic grating coupler, which is primarily a scattering problem. The second example is a photonic crystal cavity problem. The third example is a CMOS sensor efficiency problem. The final example is an OLED screen light extraction efficiency problem. Some examples focus on absorption while other focus on emission. All examples contain complicated materials, boundary layers, and sources, which require multiple stencils throughout the simulation grid.

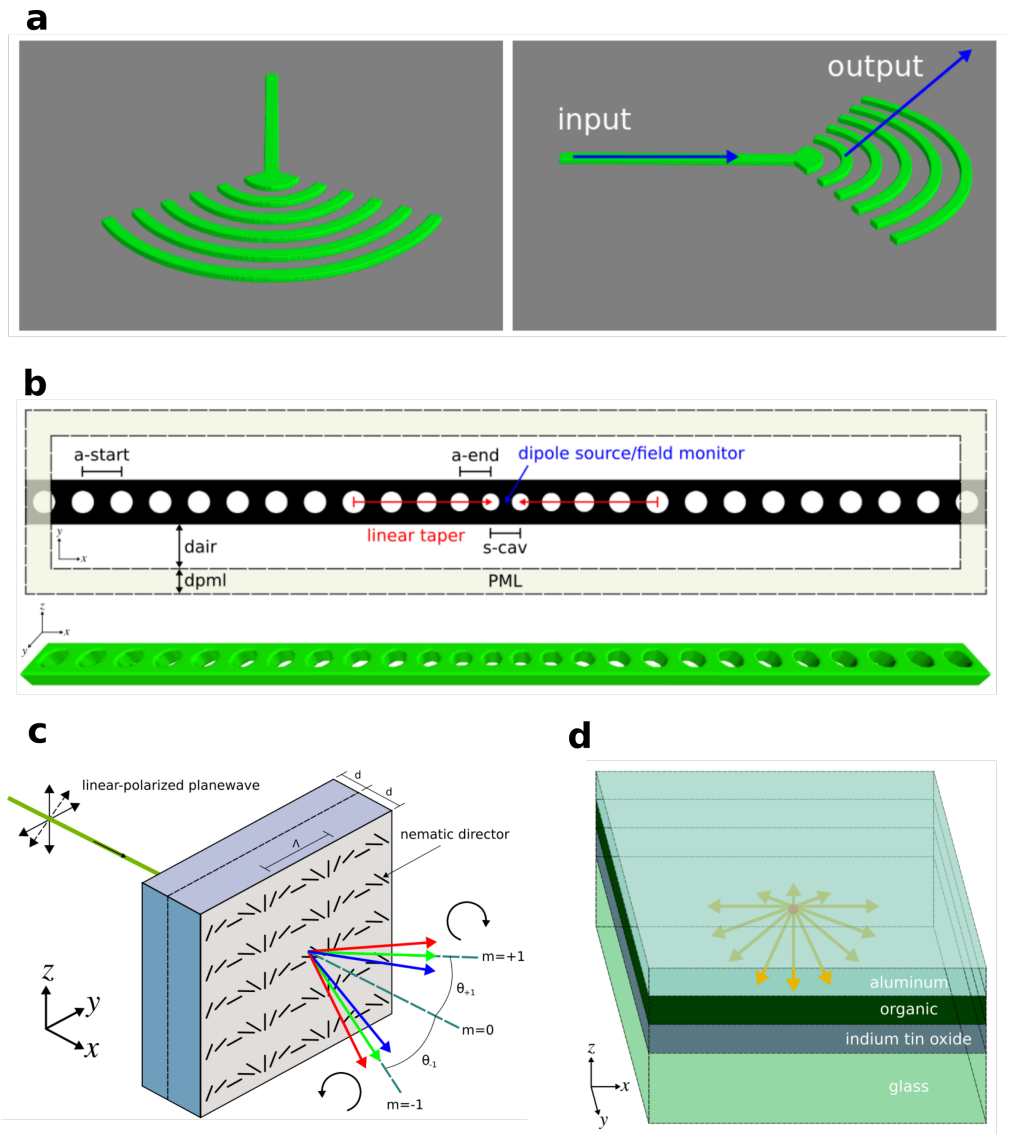

Figure 2: The four different example problems used to thoroughly test the new OpenMP implmentation. (a) A nanophotonic grating coupler. (b) A nanophotonic crystal cavity. (c) A liquid crystal display diffraction grating. (d) An organic light emitting diode stack.

Each example was run on the Xeon 6248 node at various thread counts. Figure 3 compares the raw execution time, speedup, and efficiency for each example. As expected, each example performs rather differently. For example, the nanophotonic grating example (nanobeam) experiences very little maximum speedup (2.5x) whereas the liquid crystal display example (liq) experiences almost 25x maximum speedup. The other two examples topped out at about 5x speedup.

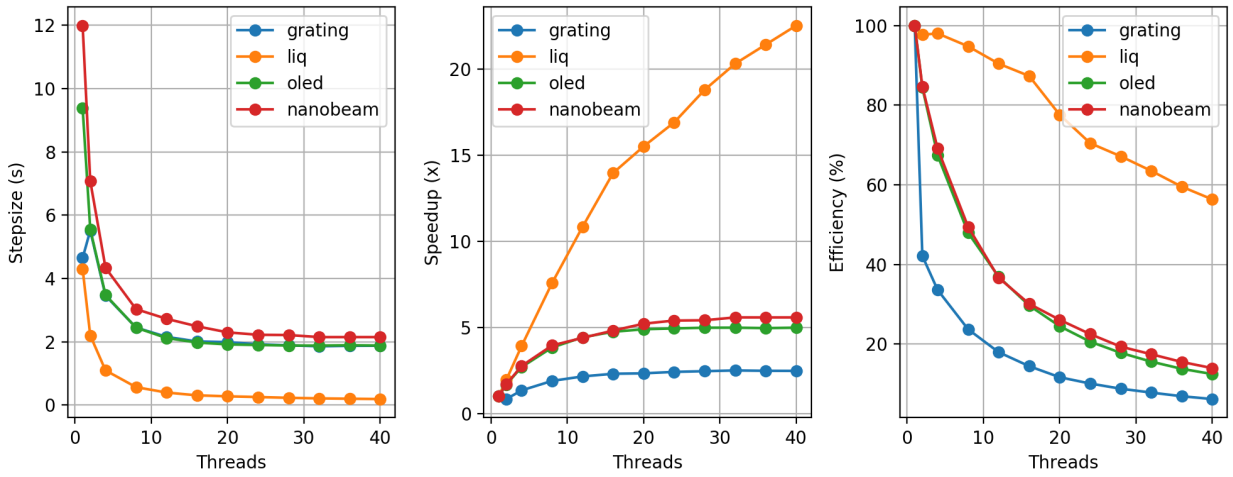

Figure 3: Timing results, speedup and efficiency for all four examples running with the OpenMP acceleration. The liquid crystal display example (liq) experienced the most speedup, followed by the nanobeam cavity, the OLED example, and then the grating coupler.

The differences in speedup/efficiency between each example problem are somewhat expected as they involve different physics stencils and call each stencil a different number of times. For example, the grating coupler has several complicated material profiles that require calling the corresponding stencil several times. The liquid crystal display, however, has much simpler material profiles and consequently calls each stencil fewer times. The relative memory footprint of each example is relatively uniform. That being said, it is important to investigate the impact of memory footprint and other OpenMP parameters that could impact the performance of the code.

4.3 OpenMP Parameter Tuning

In order to better understand the limitations of the current OpenMP implementation, I ran several more simulations by sweeping through various OpenMP hyperparameters. In particular, I analyzed the impact of the simulation footprint in memory, the efficiency dependency on various schedulers, and the effect of thread chunking. To be consistent, I used the same OLED example for all test cases since it neither performed the best nor the worst compared to all four examples.

With regard to memory footprint, we would expect the actual timestepping rate to depend directly

on the size of the data arrays in memory, but we wouldn't expect any changes in efficiency/speedup. Figure 4 illustrates the numerical experiment I performed to quantify this relationship. As expected, the increase in timestepping was linear with memory size, but the speedup was consistent between the different sizes. We can rule out any possibility that our initial example simulations were too small to benefit from any accelerated parallelism.

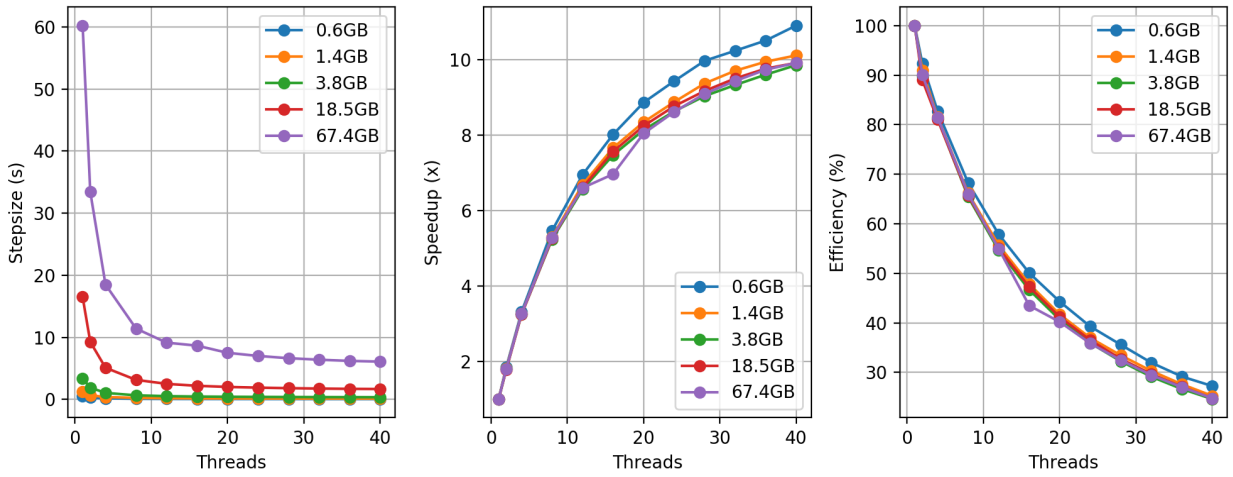

Figure 4: Effect of simulation size on OpenMP speedup. The OLED example with different sized domains (600 MB, 1.4 GB, 3.8 GB, 18.5 GB, and 67.4 GB in memory) was run with different thread counts. While the actual runtime is directly dependent on the memory footprint (left plots), the actual speedup and efficiency are practically independent (right two plots).

Next I opted to run the OLED example (18.5 GB) using different runtime schedulers. In practice, we would expect significant difference in performance from scheduler to scheduler. As shown in Figure 5, however, there is practically no difference between the dynamic, static, guided, or auto schedulers. This is rather troublesome for two reasons. First, we would expect significant changes between them as they operate very differently. Next, the whole point of our OpenMP implementation is to leverage the dynamic load balancing facilitated by the scheduler. Consequently, we can conclude that there must be significant computational overhead somewhere else.

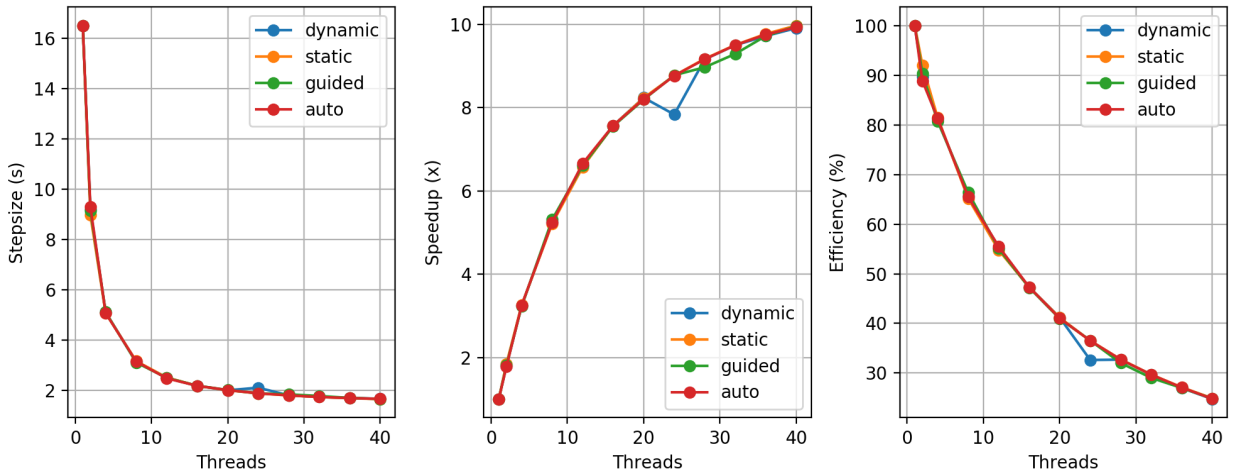

Figure 5: Effect of OpenMP scheduler on overall speedup. The OLED example was run with the four basic OpenMP schedulers (dynamic, static, guided, and auto). Interestingly, there is no performance dependence on any of the schedulers.

Finally, I ran an experiment to determine the influence of the chunk size for each thread's workload. Since our build supports AVX-512 SIMD instructions, and our code explicitly leverages that instruction set, we should expect some speedup relative to chunk sizes that fit in cache and can leverage this form of local parallelism. Figure 6 once again demonstrates, however, a relative independence on chunk size. Once again we can conclude that there is significant overhead somewhere else in the code impeding the gains we would hope to accomplish.

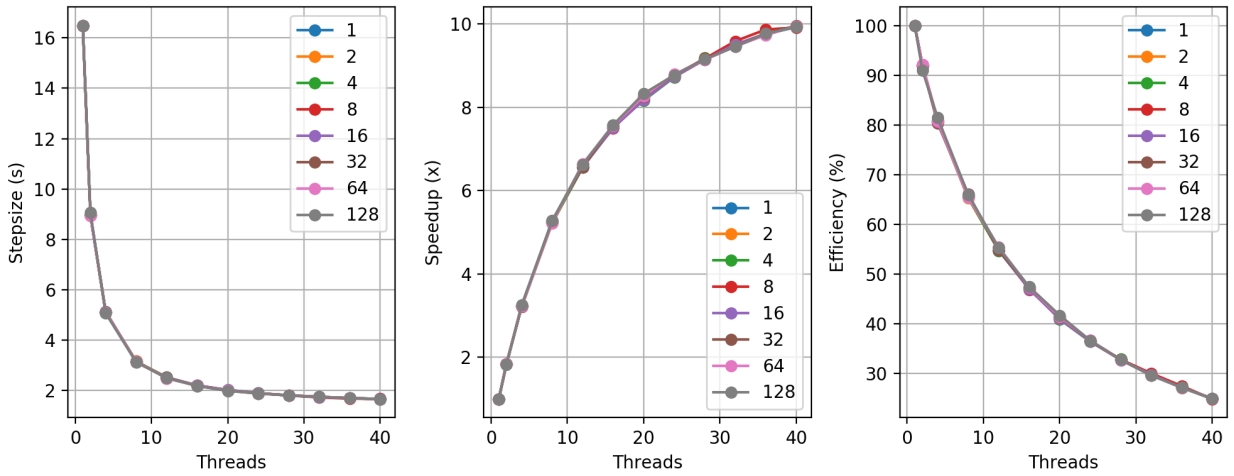

Figure 6: Effect of thread chunk size on speedup. There is no net performance dependency on the OLED example.

4.4 Comparison with MPI

While our speedup isn't as high as we would hope, it still might be better than the current state of the MPI build. To test this, I ran each example on multiple threads/processes for the MPI build and the OpenMP build separately. Figure 7 illustrates the comparison.

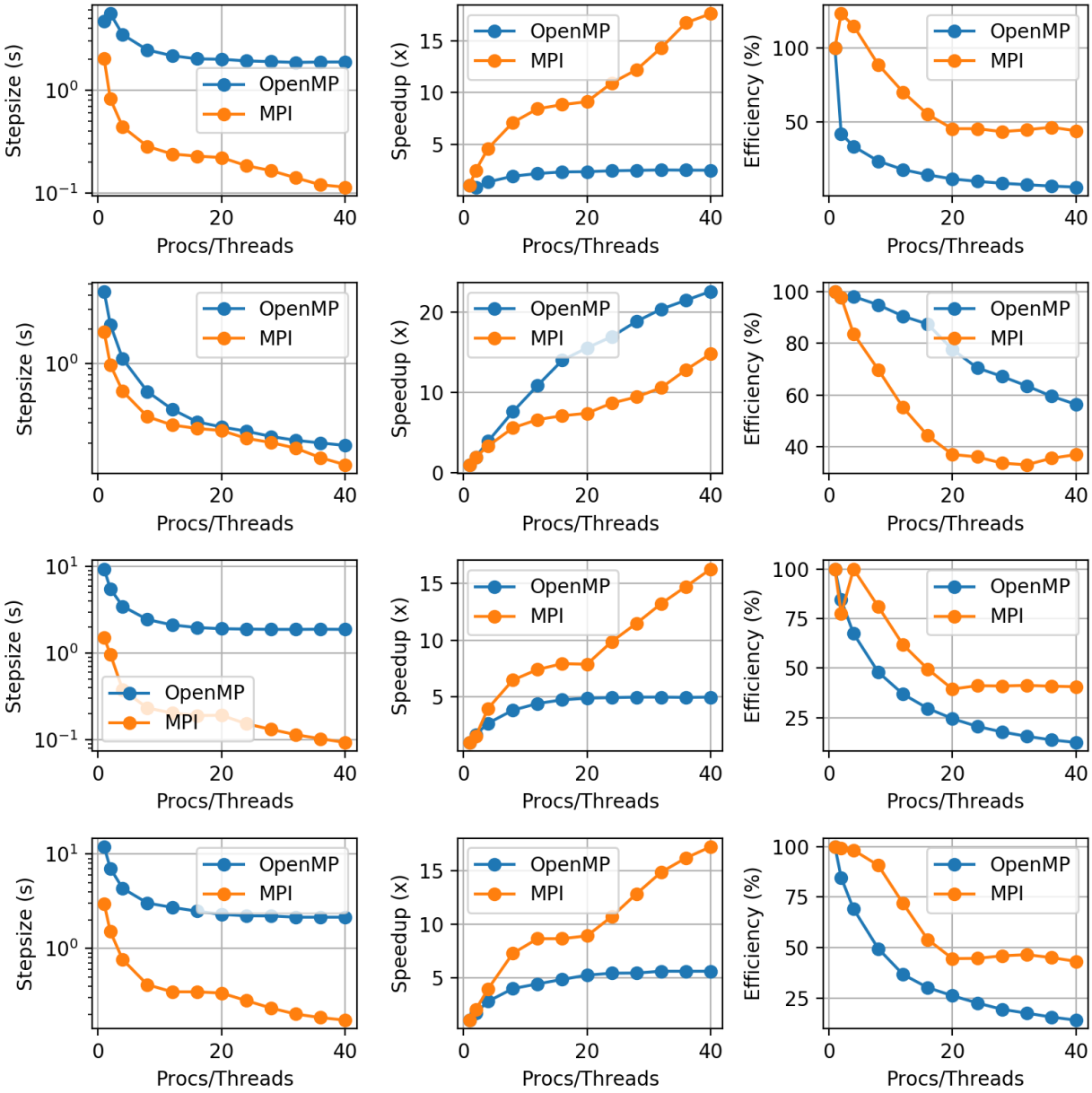

Figure 7: Comparison of the OpenMP builds and the MPI builds for the grating coupler example (row 1), the liquid crystal display example (row 2), the OLED example (row 3) and the nanobeam example (row 4). For each test case, I tracked the net runtime in seconds (column 1), the relative speedup vs threads/processes (columns 2), and the corresponding efficiency (column 3).

This last experiment revealed many important facts about both the MPI build and our new

OpenMP build. First, some of the MPI builds experienced superlinear speedup (efficiency greater than 100%) for smaller processor counts. This implies that these particular configurations are better leveraging the SIMD instruction sets. The MPI communication cost quickly drowns out these performance gains as the processor count increases, however. That being said, the communication cost never fully surpasses the computational cost as there is still strong speedup when using all of the processor cores.

We also note that for one process/thread, the MPI build is always significantly faster than the corresponding OpenMP build. While this is surprising, it helps us understand why the scheduler and chunk size don't seem to be impacting the performance of our OpenMP build. As we earlier predicted, there is significant overhead initializing the OpenMP thread pool. Unlike most codes that leverage OpenMP, our code reopens a thread pool between 40 and 120 times *each timestep*, depending on the physics of the example problem. This is largely because of the way that the heterogeneous stencils must be solved. They depend on earlier states of themselves at different grid points. Consequently, we must loop over the entire grid several times. This is why we see somewhat better speedup with the liquid crystal display example. There are fewer stencils that require a new spawning of the thread pool. That being said, even this example performs worse than the raw MPI build.

Ideally, we would spawn one thread pool at the start of our simulation and keep that thread pool alive, handing work to each thread as it became available. However, that sort of paradigm would require a significant refactoring of the current codebase.

The MPI implementation is truly SIMD. From the beginning, each process is handed a subset of the grid and only loops over that subset. Communication between processes only occurs at boundary (ghost) pixels. While there is still overhead in spawning these processes and communicating the respective surface areas, it is significantly less than the overhead of spawning multiple thread workers dozens of times each timestep.

5 Conclusion

I've refactored an existing FDTD code to leverage OpenMP acceleration. In all, I rewrote 11 loop macros, refactored over 106 different parallel loops, adopted a "first-touch" initialization scheme, and coalesced the memory access to leverage vectorization. I generated four different examples that use various aspects of the available physics to test the robustness of the solution. I found surprisingly different results for each example. However each example under-performed the MPI equivalent. This is largely due to the number of thread groups that must be spawned over and over.

I tried refactoring the code even further to reduce the number of thread spawns, but realized

this would not sufficiently limit the thread overhead. In fact, the very multi-stencil paradigm itself inhibits the benefits of dynamic load balancing, since so many different stencils must be called. Future efforts should focus on other SIMD forms of parallelism, i.e. with a GPU. While this is still a static load distribution determined before the simulation starts, it allows for several more workers to split the load on a hardware platform with significantly more memory bandwidth (which is important since the algorithm is heavily memory bound). The stenciling algorithm itself could be refactored to allow caching of values, making the routine less memory bound and more compute bound. However, the diversity of the stencils once again makes this a difficult problem to solve.

Regardless of the somewhat disappointing outcome, I am satisfied that my current codebase is performing exceptionally well. I can continue to launch parallel jobs knowing that my implementation is efficient. My OpenMP branch is documented in (10) and all of my experimental results are found in (11).

References

[1] A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. D. Joannopoulos, and S. G. Johnson, "Meep: A flexible free-software package for electromagnetic simulations by the fdtd method," *Computer Physics Communications*, vol. 181, no. 3, pp. 687–702, 2010.

[2] A. Oskooi, C. Hogan, A. M. Hammond, M. Reid, and S. G. Johnson, "Factorized machine learning for performance modeling of massively parallel heterogeneous physical simulations," *arXiv preprint arXiv:2003.04287*, 2020.

[3] C. Warren, A. Giannopoulos, A. Gray, I. Giannakis, A. Patterson, L. Wetter, and A. Hamrah, "A cuda-based gpu engine for gprmax: Open source fdtd electromagnetic simulation software," *Computer Physics Communications*, vol. 237, pp. 208–218, 2019.

[4] C. Warren, A. Giannopoulos, and I. Giannakis, "gprmax: Open source software to simulate electromagnetic wave propagation for ground penetrating radar," *Computer Physics Communications*, vol. 209, pp. 163–170, 2016.

[5] P. Wahl, D.-S. Ly-Gagnon, C. Debaes, D. A. Miller, and H. Thienpont, "B-calm: An open-source gpu-based 3d-fdtd with multi-pole dispersion for plasmonics," *Optical and Quantum Electronics*, vol. 44, no. 3-5, pp. 285–290, 2012.

[6] *xFDTD solver*. https://www.remcom.com/xfdtd-3d-em-simulation-software.

[7] *CST solver*. https://www.3ds.com/products-services/simulia/products/cst-studio-suite/.

[8] *SEMCAD solver*. https://speag.swiss/products/semcad/solutions/.

[9] K. Yee, "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media," *IEEE Transactions on antennas and propagation*, vol. 14, no. 3, pp. 302–307, 1966.

[10] A. Hammond, *Multithreading branch*, 2020. https://github.com/smartalecH/meep/tree/multithreading.

[11] A. Hammond, *Experimental data*, 2020. https://github.com/smartalecH/meep$_{openmp}$.