Parallel Computation of Cross Beam Energy Transfer

Andrew T. Sexton

University of Rochester

May 1, 2020

Introduction

The Crossbeam Energy Transfer (CBET) code, provided by Dr. Adam Sefkow at the Lab for Laser Energetics, was originally written in Yorick, a C-based scripted language developed for scientific computing. Yorick has many useful features such as array syntax, vectorization, and column major ordering that factored into the complexity of translating the code into. The CBET code has four main parts: initialization, ray launching, energy calculations, and plotting. Each part was recreated in Python as closely as possible to the source code. This was done to enable a more direct comparison between the languages as well as to highlight any improvements made by parallelization.

The software simulates the intersection of two orthogonal laser beams to determine the interactions between them. Each beam is composed of several individual rays, which are tracked through the working space by calculating the trajectory from the initial conditions. The first beam goes from left to right, while the second beam goes bottom to top. This ray tracing is one of the slowest parts of the code, and as such is the primary focus of the parallelization effort. After all the rays have been created, the code iteratively finds all intersections between a given ray and all others. At each intersection, the ray's energy and direction is recalculated to incorporate the interaction between the two rays. The code then

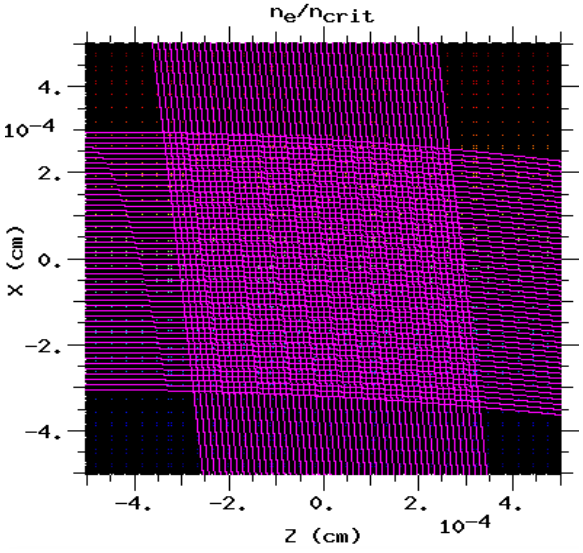


Figure 1a

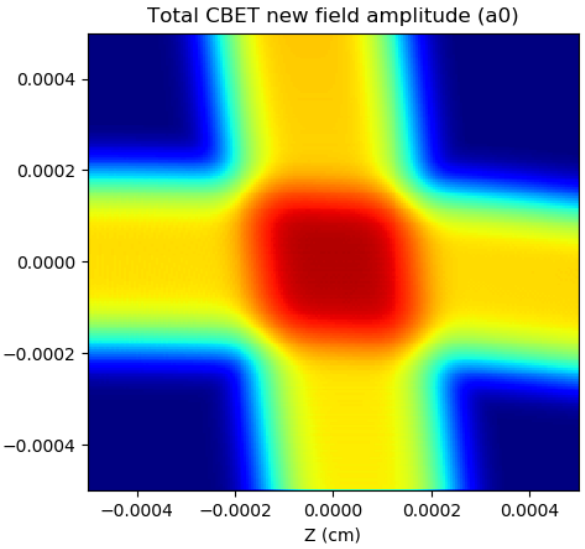


Figure 1b

iterates over the rays again, calculating the resulting intensity change. Finally, the results are plotted in six different plots to show the change in direction and intensity of the beams. Figure 1 is an example of the mesh grid plots made by the software. On the left is the plot showing the path traced by the rays (magenta colored) which shows that each ray of one beam intersects all the rays of the orthogonal beam. The right plot shows the amplitude heatmap of the laser beams as they intersect.

CUDA

Initially, the goal was to implement the code using CUDA libraries and then implement MPI on top of that. This is because CUDA is highly efficient at matrix operations, which means it could be used to compute the entire ray traces in fewer steps, rather than iteratively tracking it across a mesh grid.

There are two CUDA libraries that were recommended for Python, PyCUDA and Numba. PyCUDA utilizes a wrapper interface around the C++ CUDA libraries and uses a "kernel" to perform the CUDA operations. The "kernel" used by PyCUDA is a function to be used on the GPU, written in C code to be passed to the CUDA libraries. This required that larger sections of the code be rewritten in order to be more memory efficient, but also to allow for what is essentially a section of C code to be inserted into the middle to be run on the GPU. This required the code be refactored from the ground up to properly design these kernels and the backing data structures.

```
mod = SourceModule("""
        __global__ void doublify(float *a)
        {
                Int idx = threadIdx.x +threadIdx.y * 4;
                a[idx] = 2 * a[idx];
        }
        """)
func = mod.get_function("doublify")
```

Example 1: PyCUDA kernel (mod) creating a function for CUDA usage

Alternatively, Numba was also recommended for utilizing CUDA. Numba is a library that utilizes a Just-In-Time compilers (JIT) to achieve speed ups and CUDA integration. JIT compilers are used to compile small sections of code at run time in order to achieve the speed ups usually associated with compiled code. Since the code is already being compiled, it can be utilized by a GPU if properly set up. Numba allows for "JIT-ing" pure Python code, compared to PyCUDA which required the kernel to be written in C code. However, Numba also required a thorough refactoring of the code for it to work correctly. Things like Python classes have some support in Numba, but that support is not fully integrated with the rest. This also made it prohibitive to use at this time due to the need for the code to be rebuilt from scratch.

```
@cuda.jit
def matmul(A, B, C):
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Example 2: Numba CUDA compiled code using @cuda.jit decorator

Message Passing Interface

Message Passing Interface (MPI) is a parallel computing standard which allows for a single program to be executed on multiple processors, either locally as processes on CPU cores or networked as processes on a compute node. Data synchronization is achieved by Sending and Receiving of messages, as well as large scale messaging with broadcasts and scatter/gather functions. Python has only a single MPI library available for it, mpi4py. This library provides a Python wrapped interface to the C/C++ implementation that is the base of the core of the MPI standard. The library also provides built-in support for Numpy arrays, which are optimized more than the function for generic Python objects.

MPI works by creating copies of the program to be run where each process views the same code as all the other processes. In order to obtain parallelism, the code must be written so that certain actions are explicitly taken by certain processes. For example, in the CBET code only the root process (defined by having a rank of 0) computes the ray tracing for the first beam. This is done with an if-statement that only allows Rank 0 to enter the statement body.

Message passing was implemented in two different ways in the code. The first was to pass the entire ray generation loop to the first two processes (ranks 0 and 1). This allows for the ray generation of each beam to be done simultaneously, which should result in a proportional decrease in computation time. The crux of the implementation is in using the optimal message sending protocols. For example, each process requires copies of certain data structures in order to modify them. Each process could initialize all these structures independently, or one process can initialize and then send the structures to all other processes as a broadcast. The other is to send each small segments of data to a process which then computes or collates the data as needed. This second option allows finer control of the data, but at the cost of increased communications between processes which can result in slowdowns as the problem size and complexity increases, or when the number of processes communicating increases.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD          # initialize communications
rank = comm.Get_rank()
ary = None
if rank == 0:
        ary = numpy.zeros(5, 5)  # initialize 5x5 array
        comm.Send(ary, dest=1)  # send ary to process 1
if rank == 1:
        comm.Recv(ary, source=0)  # receive array from root
```

Example 3: Sending an array from rank 0 to rank 1

The CBET codes structure again precluded the ability to utilize the more general message functions. Arrays in the CBET code are initialized as "Fortran-ordered Numpy Arrays", which means they are represented as column-major ordered arrays in memory (although the actual memory layout is not changed). This allowed the Yorick source code to be translated to Python more easily, as loops and arrays did not have to be redesigned for row major ordering. This does mean that the functions such as Scatter/Gather required a significant amount of shuffling and reshaping in order to work correctly. Instead, the finer grain Send/Recv functions were utilized. This should result in slower parallel computation, but without the need to significantly refactor the code. A secondary benefit of using Send/Recv functions is that they block the process from running until the message has been received. This enforces a certain amount of synchronization between the processes without the need for explicit synchronization barriers.

The second way MPI was implemented in the code was by changing the for-loop iteration counters so that each process computes a subset of the for loop, and then messages the data back to the root for collation. This allows for more processes to be utilized to aid in the computation, but at the cost of increased traffic between processes. For example, each process begins the loop at its rank. So, rank 0 does the zeroth iteration, rank 1 does the next iteration and so on.

Multiprocessing

Python has a standard library for multiprocessing which includes a Thread Pool Class. This operates similarly to the OpenMP standard. It is different from OpenMP in that the Pool class cannot explicitly parallelize for-loops, but rather requires the loop to be replaced with a function and a parameter list. For example, in C/C++ code a for-loop could be parallelized with a compiler directive, or "pragma". The following shows how a simple loop would be parallelized in C and Python.

```
#pragma omp parallel {
        For(int i = 0; i < 5; ++i) {
                printf("%d", i);
        }
}
```

Example 4: C/C++ OpenMP Parallel for loop

```
def example(n):
        print(n)
Pool.map(example, [0, 1, 2, 3, 4])
```

Example 5: Python Multiprocessing for loop

The Python code shows that the parameters to be passed to the example function must be contained in a list, and each element of the list is then mapped (using the Pool classes map method) to the example function. This required some modification of the code by making each "loop" its own function, and passing parameters as copies instead of as references, otherwise data races between threads may occur.

This method has the best performance increase and requires the least amount of code modification. However, the drawbacks are that it only works on a single computer and not over networks and that each thread requires complete copies of each required array. This memory requirement for each thread to have a copy of each array quickly becomes the limiting factor for this implementation, as seen in the result section. This memory limitation can be overcome using shared memory buffers, but the multiprocessing library does not support two dimensional arrays in shared memory, nor does it support Numpy array objects. This means that implementing a shared memory structure would require significant refactoring of the code in order to achieve.

Experiment

Testing the performance of the CBET code came with several challenges. The first of which is timing. The Yorick code, which is single threaded, utilizes Yorick's built in elapsed timer function to track each individual code section's timing. This gave a good view into where the slowest parts of the code were, namely ray launching and gain calculation. However, Python does not have a similar timer and attempts to create one that mimicked the Yorick version failed. Also, timing individual sections of parallelized code is challenging and for the most part unhelpful. Instead, the code was timed in total over multiple samples in order to obtain an average run time.

The code was timed with three different initial conditions, described in figure 2, which are the default values, doubling the number of rays, and doubling several parameters and setting many rays. For the default values, each test was performed five times to determine an average run time. The ray doubling test was performed three times each, due to the time required per run. The area doubling test was only performed once for each version, the results of which are discussed in the next section. For MPI and the

Multiprocessing implementation, each test was also performed for processes in multiples of two (2, 4, 6, 8, 10, and 12). Twelve is the limit of the hardware these tests were run on. The multiprocessing test was also tested with a single thread, but due to the program design, MPI could not be tested with only one process.

The number of rays to be used by the program is calculated by the following equation. The values "beam_max_z" and "beam_min_z" represent the upper and lower bounds of the beam, or together represent the beam diameter. "z_min" and "z_max" represent the left and right edges of the working space.

$$nrays = int\left(\frac{rays\_per\_zone * (beam\_max\_z - beam\_min\_z)}{\left(\frac{z\_max - z\_min}{nz - 1}\right)}\right)$$

For the default parameters, this results in 599 rays, as casting to an integer rounds the value down. However, nrays can be set to any arbitrary value. For the second test, the number of rays was multiplied by two, and for the third test the numbers of rays were set to be 6000 directly.

Parameter	Default	High Intensity
nx/nz	201	401
x range	$\pm 5.0x10^{-4}$	$\pm 10.0x10^{-4}$
z range	$\pm 5.0x10^{-4}$	$\pm 10.0x10^{-4}$
rays per zone	5	25
nrays	599	6000
beam min/max	$\pm 3.0x10^{-4}$	$\pm 6.0x10^{-4}$

Figure 2

In order to more accurately assess the timing of the code, all print statements and plotting functions were removed in order to remove any inconsistencies that I/O or graphical rendering may introduce. Although the Yorick code creates six plots, which were recreated in Python with the Matplotlib library, the internal workings of the plotting functions are not known, and so cannot be accounted for.

All the tests were performed on a desktop computer utilizing an AMD Ryzen 5 1600 CPU with 6 physical cores and 12 logical cores operating up to 3.6 GHz, 16 GB of RAM, and the Ubuntu 18.04LTS operating system. The programs were written in Python 3.7, utilizing mpi4py 3.0.3.

Results

The average run time for the Yorick code with default parameters, sans output, was $147.890s \pm 4.926s$ (see Appendix: Table 1). Although not recorded, this was approximately 15 seconds faster than with print statements and plotting enabled. The Python version with no parallel implementation, recorded as Sequential in the timing data tables, was slightly slower with an average time of $176.316s \pm 4.324s$. In Table 1, the last row shows the percent difference between the Yorick implementation and the given implementation. Figure 3 shows the average times for all default tests.

For MPI, as each ray launching loop was implemented in parallel, the optimal speedup would be half that of the Sequential code. Process creation incurs some amount of overhead, which slows the total time down. For MPI with two threads, the speedup was approximately 48% over the Sequential version. This is nearly an ideal speedup, however as more threads were included, the performance gain begins to drop slightly reaching 100 seconds with 8 processes. This is expected as the overhead to create the processes and communicate messages between them begins to slow the program down. More processes mean more messages being passed, and each message blocks the sending and receiving processes from progressing until the message is completed.

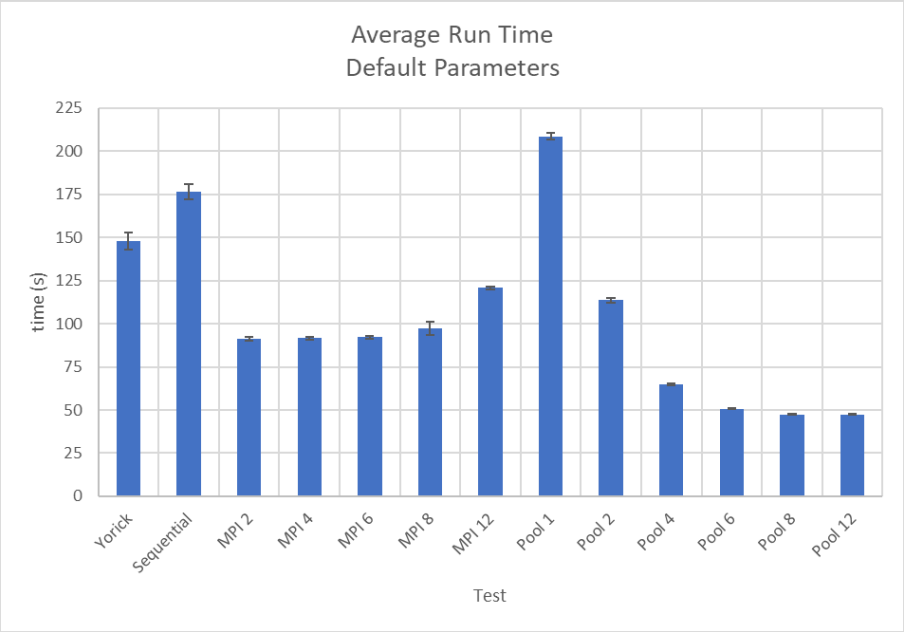


Figure 3: Average times for each test.

The thread pool version showed the best performance overall, but only at higher thread numbers. For single thread performance, it performed significantly slower than both Yorick (41.1% slower) and Sequential (18.3% slower). However, the speed up climbs significantly with two and four threads, before appearing to converge at about 47 seconds for eight and twelve threads. With eight threads the speedup over the Yorick implementation is 67.9%, and 73.0% over the Sequential version.

	Yorick	Sequential	MPI 4	Pool
Run 1	438.722	362.347	194.414	N/A
Run 2	430.599	357.708	195.023	N/A
Run 3	419.326	359.290	194.608	N/A
Average	429.549	359.782	194.682	N/A
Std Dev	9.741	2.358	0.311	N/A
Difference	0.0	-16.2	-54.7	N/A

Table 1: Double Ray Timing Data

The second test, where the number of rays was doubled (from 599 to 1158), shows that for longer running programs, Python is faster than Yorick. This is not apparent in the data in Table 1. The reason for

this is not quite clear, as it could be many things that are inherent to the language design and implementations. With more time, a program to compare performance on a simple and computational heavy problem would be used to determine where this speed up may be occurring. Note that the Pool column has no times since the program would quickly utilize all 16 GB of RAM and would be killed by the operating system.

	Yorick	Sequential	MPI 4	Pool
Run 1	16360.36	3804.466	2120.566	N/A

Table 2: High Intensity Timing Data

The last test was what was dubbed the Super High Intensity Test, in order to stress the software with an excessively large number of rays (Figure 2). The Yorick code took 4 hours and 32 minutes to complete, while the Sequential Python code took 63.4 minutes for a speedup of 76.7%. The discrepancy in speed between the two is indicative of language specific optimizations happening. MPI was run with four processes, as this was the one of the fastest of the default MPI tests. MPI displayed a speedup of 44.3% over the Sequential Python and 87.0% over the Yorick code. Again, the Multiprocessing Pool version failed to run by consuming all the system memory before being killed by the OS.

Discussion

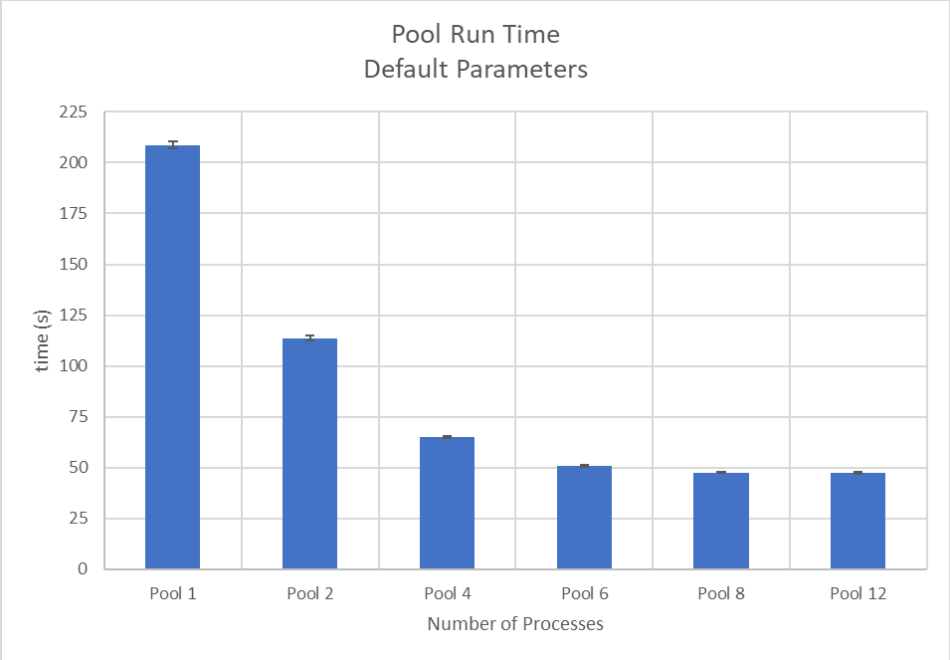
This project started with many possible implementations such as Pytorch, MPI with CUDA, or MPI with Threading. However, due to the complexity of the code and limitations on time, these were not possible to achieve. This resulted in a comparison of four different versions of the software, each of which has its own pros and cons. From a performance and ease of implementation, utilizing Pythons' Multiprocessing library was the best. However, massive memory requirements made it unusable with anything but the smallest of problems. MPI required a more considerate approach to implement but did not suffer from any significant drawbacks. However, it did not show any significant improvement when increasing the number of processes. This is likely due to the choices made on how to divide the workload between processes. A redesign of the code would allow for more efficient workload distribution, leading to better performance with more processes.

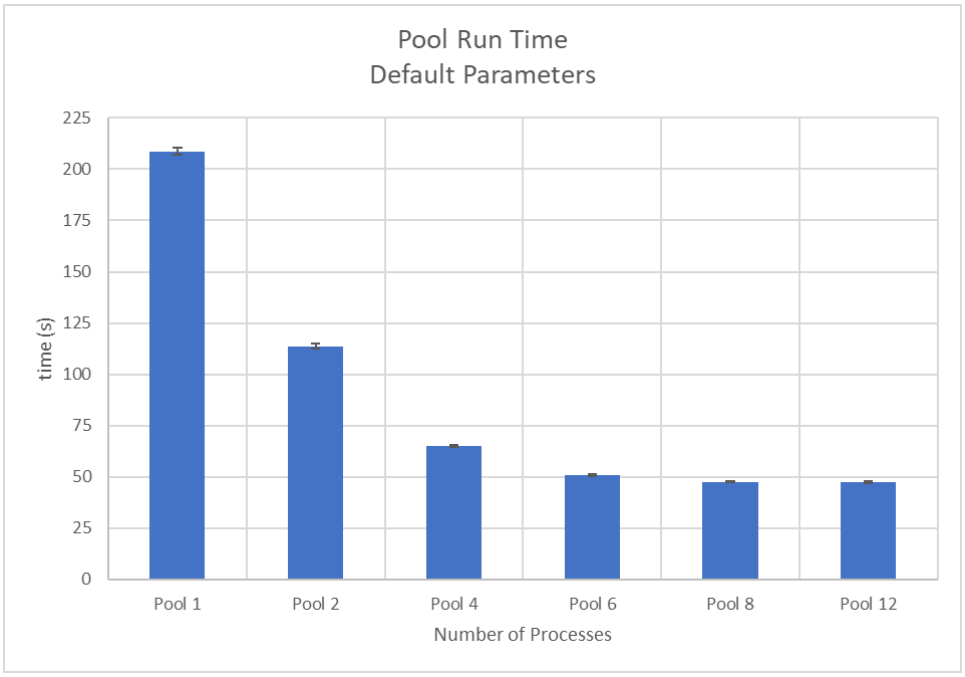
The purpose of this project was to become familiar with the Python libraries that allow parallel computing. In this aspect, much was learned about the requirements and implementation of these libraries. However, the primary lesson learned was that adapting code directly from one language to another is challenging and time consuming. It also precludes that ability to modify the code to implement efficient design decisions that are required for some libraries (such as CUDA). Instead, it is much better to design the software from the ground up with a single implementation in mind.

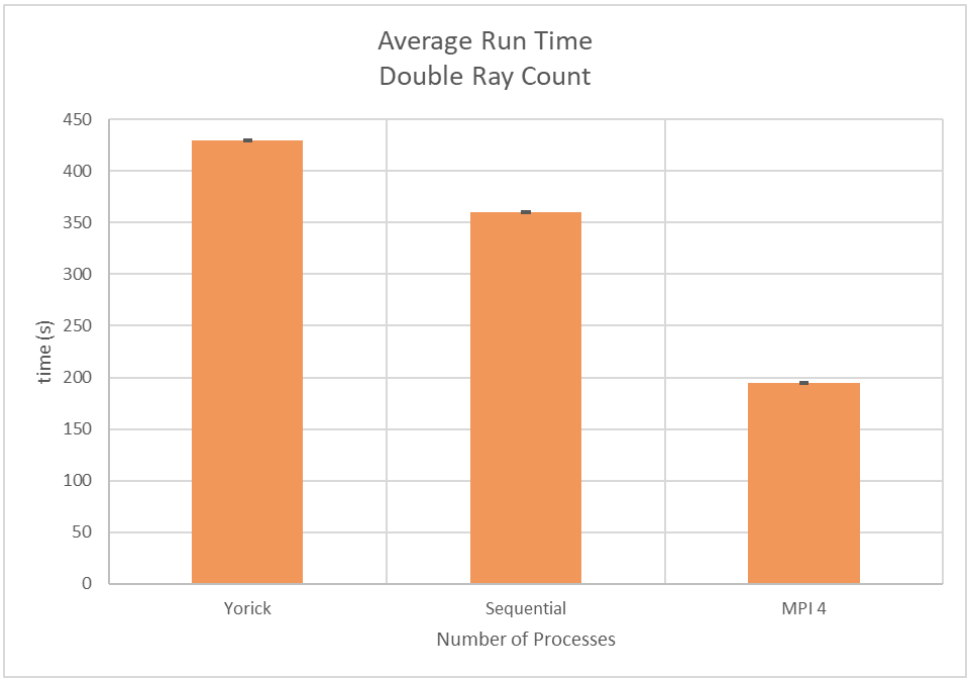
Appendix

	Yorick	Sequential	MPI 2	MPI 4	MPI 6	MPI 8	MPI 12	Pool 1	Pool 2	Pool 4	Pool 6	Pool 8	Pool 12
Run 1	142.655	181.507	91.834	90.372	91.930	94.947	119.515	210.394	115.438	65.412	51.501	47.193	47.201
Run 2	142.424	170.905	93.260	91.904	91.607	96.796	121.502	210.014	114.659	64.587	50.741	47.793	47.810
Run 3	150.509	176.170	90.606	92.090	91.985	102.465	120.425	209.364	113.222	65.109	50.717	47.422	47.597
Run 4	151.590	179.539	90.644	92.246	93.541	100.200	120.624	207.062	111.928	65.376	50.971	47.466	47.856
Run 5	152.273	173.460	91.284	91.504	91.758	92.157	121.451	206.472	113.187	64.465	50.131	47.786	46.992
Average	147.890	176.316	91.526	91.623	92.164	97.313	120.703	208.661	113.687	64.990	50.812	47.532	47.491
Std Dev	4.926	4.324	1.094	0.752	0.784	4.103	0.820	1.780	1.376	0.441	0.494	0.257	0.381
% Diff	0.0	19.2	-38.1	-38.0	-37.7	-34.2	-18.4	41.1	-23.1	-56.1	-65.6	-67.9	-67.9

Table 3: Default Parameter Timing Data (all times are in seconds)




Pool Run Time
Default Parameters
225
200
175
150
125
100
75
50
25
0
time (s)
Pool 1
Pool 2
Pool 4
Pool 6
Pool 8
Pool 12
Number of Processes


Average Run Time
Double Ray Count
450
400
350
300
250
200
150
100
50
0
time (s)
Yorick
Sequential
MPI 4
Number of Processes