Parallel Ray Tracing Using the Message Passing Interface

Charles B. Cameron, *Senior Member, IEEE*

***Abstract*—Ray-tracing software is available for lens design and for general optical systems modeling. It tends to be designed to run on a single processor and can be very time consuming if the number of rays traced is large. Previously, multiple digital signal processors (DSPs) have been used to perform such simulations. This approach is attractive because DSPs are inexpensive, and the time saved through parallel processing can be significant. In this paper, we report a nearly linear relationship between the number of processors, n, and the rate of ray tracing with as many as 839 processors operating in parallel on the Naval Research Laboratory's Cray XD-1 computer with the Message Passing Interface (MPI). In going from 1 to 839 processors, we achieved an efficiency of 97.9% and a normalized ray-tracing rate of 6.95×10^6 rays $\cdot$ surfaces/(s $\cdot$ processor) in a system with 22 planar surfaces, two paraboloid reflectors, and one hyperboloid refractor. The need for a load-balancing software was obviated by the use of a prime number of processors.**

***Index Terms*—Load balancing, Message Passing Interface (MPI), Moderate Resolution Imaging Spectroradiometer (MODIS), National Aeronautics and Space Administration (NASA), optical ray tracing, parallel computing, parallel processing, prime numbers, ray tracing, reconfigurable computing, Terra.**

I. Background

A. History of Ray Tracing

As EARLY as 1604, the German astronomer Johannes Kepler published *Astronomia pars Optica*, which is a book on astronomy that contained the important principle that light rays travel from an object to an eye and not the other way around. In 1841, Karl Friedrich Gauss published *Dioptrische Untersuchungen* with an analysis of the path light takes through a system of lenses [1]. With the assistance of tables of logarithms, slide rules, calculators, and refinements to handle lens aberrations, the methods that Gauss developed were applied for more than a century after he published them. Lens system designers performed these calculations to predict lens performance *prior* to the expensive and often uncorrectable step of actually grinding a lens.

Since it was sufficiently painful and time consuming to perform such computations without the aid of modern high-speed computers, it was normal to trace only a very small number of rays. However, with the advent of high-speed electronic computers in the 1940s, it became feasible to perform ray tracing more swiftly [2]. As a consequence, it also became feasible to trace more rays to produce superior lens designs. Furthermore, it became feasible to perform ray-tracing simulations repeatedly, i.e., iterating until a satisfactory design had been achieved. Spencer and Murtry presented the equations needed to perform ray tracing by a computer in [3]. These equations also appear in vector notation in [4] and are summarized in [5]. An interest in tracing large numbers of rays in a small amount of time has resulted in programs that are able to trace rays on multiple processors at once [6].

Today, there are numerous commercially available ray-tracing computer programs, including ASAP, CODE V, SYNOPSYS, OSLO, and Zemax. All these programs are tailored to the design of imaging lens systems. Some of them can also handle more general illumination problems for nonimaging systems.

B. Ray Tracing for Computer-Generated Images

In the 1960s, a new application of computer-based ray tracing arose in making computer-generated graphical images. In fact, a large body of the literature that pertains to ray tracing concerns this application. This interest is driven by commercial demands from the entertainment industry and a widespread interest in visualizing complex phenomena that cannot otherwise be seen.

As with ray tracing for lens system design, tracing large numbers of rays in a short time permits the rapid rendering of very complex images. This has led various researchers to apply multiple computer processors operating in parallel to this important application [7].

The objective in computer graphics is to find out which objects generate light rays that reach the viewing screen. The objective in lens system design is to find out which light rays strike the focal plane, which is where the image is formed. It is therefore common in computer graphics ray tracing to start at the viewing screen and trace the light toward the source. It is more usual in lens system design to trace the rays in the direction they actually travel, that is, from source to image plane. Both of these are functionally equivalent.

While ray tracing in computer graphics applications is very similar to ray tracing in lens design, it can be regarded as a distinct application and is not the principal subject of this paper.

Manuscript received September 19, 2006; revised September 13, 2007. This work was supported in part by the National Aeronautics and Space Administration, by the United States Naval Academy, and by a grant of computer time from the DoD High Performance Computer Program at the Naval Research Laboratory.

The author is with the United States Naval Academy, Annapolis, MD 21402 USA, and also with Johns Hopkins University, Baltimore, MD 21218 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIM.2007.909957

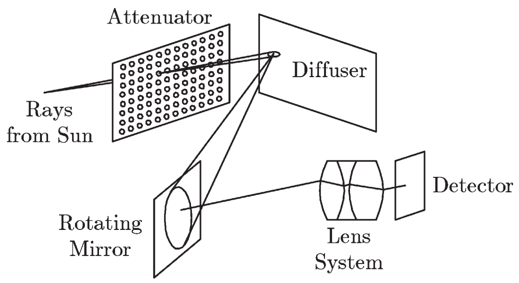

Fig. 1. Schematic showing the MODIS optical system.

C. Ray Tracing in Lens Design

The design of lens systems remains an important application of ray-tracing simulations. Although large parallel computers are widespread today, they are still quite expensive. Their virtue in performing ray-tracing simulations very swiftly is offset by their high cost, and for this reason they are not much used in commercial ray-tracing software.

As an alternative to using supercomputers, Cameron *et al.* investigated the use of multiple digital signal processors (DSPs) to achieve ray tracing in parallel on a more modest budget [6]. The particular telescopic lens system they considered is part of the Moderate Resolution Imaging Spectroradiometer (MODIS).[1] Fig. 1 shows that MODIS includes a solar attenuator consisting of an array of small pinholes. Calibration of the instrument requires taking an image of the sun. The attenuator reduces the intensity of the sun's radiation during calibration.

Fig. 1 also shows the presence of a nearly Lambertian diffusing surface.[2] This surface is placed in the path of the light when the instrument looks at the sun and it is swung aside when the instrument takes images of the earth.

Waluschka *et al.* reported on anomalies in the performance of this telescope [8]. In their research, they performed ray-tracing simulations on a single Digital Equipment Corporation (DEC) Alpha 3000 series model 800 computer. They achieved a rate of ray tracing of roughly 5160 rays/s, as reported in [6]. Each simulation run took about two weeks to complete; most of the rays traced in the system they studied traversed just 25 of the 28 optical surfaces the instrument contained. Many of them were lost as they passed through various apertures within the system. Cameron *et al.* calculated that about 80.4% as many rays could have traversed the system in the same amount of time had they been chosen in such a manner as to guarantee that none were lost.[3] In effect, the rate of ray tracing quoted above should be reduced by multiplying it by 80.4%. Thus, it is fairer to say that the achieved rate of ray tracing was approximately 4130 rays/s.

Since the number of surfaces varies from one lens system to another, it is easier to compare one ray-tracing simulator to another if this difference is eliminated through normalization. This can be done by multiplying the quoted rate by the number of surfaces in the system studied. The result of this adjustment is to say that the DEC Alpha computer achieved a normalized rate of ray tracing of about 103 000 rays $\cdot$ surfaces/s.[4]

With up to eight DSPs, Cameron *et al.* reported a ray-tracing rate for the same optical system of more than 122 000 rays $\cdot$ surfaces $\cdot$ $(\mathrm{s} \cdot \mathrm{processor})^{-1}$. (This figure already takes into account the 80.4% reduction mentioned above.) However, a systematic error in their analysis caused this ray-tracing rate to be excessive by a ratio of 441/400 due to misstating the number of rays leaving each pinhole in the solar attenuator. Adjusting for this error, the eight DSPs achieved a rate of 111 000 rays $\cdot$ surfaces $\cdot$ $(\mathrm{s} \cdot \mathrm{processor})^{-1}$. This is comparable to the rate achieved on the DEC Alpha, but eight processors instead of one achieved the same job in a little less than one eighth of the time. Most of the processing time entailed number crunching; very little entailed communication between the parallel processors. There was a nearly completely linear relationship between the number of processors used and the speed of the simulations. The two weeks needed on a single DEC Alpha fell to 35.5 h when eight DSPs performed the same simulation.

II. Using a Supercomputer

The use of DSPs avoids the expense of a supercomputer. However, even 35.5 h is longer than the time span one would like to associate with interactive design.

In the rest of this paper, we report on the application of a supercomputer to parallel ray tracing. Simulating ray tracing in the same optical system as before, we used the Naval Research Laboratory's (NRL) Cray XD-1 supercomputer to perform this study.

The NRL's Cray XD-1 has two salient characteristics: 1) It has a large number of high-speed processors (users have access to 420 dual-core AMD Opteron 275 processors operating at 2.2 GHz). 2) It has a large number of field-programmable gate arrays (FPGA), which permit application programs to take advantage of custom hardware designs downloaded into the FPGAs to make the applications execute faster. We are currently researching the use of FPGAs to accelerate the ray-tracing application. In the rest of this paper, we solely focus on the use of multiple sequential processors executing the ray-tracing application in parallel.

As pointed out above, the relationship between the speed s of ray tracing and the number n of processors cooperating in solving the problem was reported by Cameron *et al.* to be highly linear. Kumar *et al.* point out that in general one should not expect such a linear relationship to hold for an indefinitely large number of processors [9]. This idea can be made precise.

Suppose we let r_p be the speed of ray tracing with p processors, and let r_{np} be the speed of ray tracing with np processors.

[1]MODIS is now operating in two similar spacecrafts, i.e., Aqua and Terra.

[2]A Lambertian surface scatters light evenly in all directions.

[3]Although it would be preferable to be able to select only rays that will reach the image plane, in general, there is no way to anticipate whether a given ray will do so or not.

[4]A still more useful comparison could be made if this figure were individually specified for each type of optical surface—planar, spherical, paraboloid, hyperboloid, aspheric—and whether it applies for reflection or refraction. The system studied by Waluschka *et al.*, Cameron *et al.*, and in this paper has 28 planar surfaces. Surfaces 1 and 2 are planar transmissive surfaces. Surface 3 is a roughly Lambertian solar diffuser and gives rise to the vast majority of traced rays. Almost all the ray tracing therefore entails surfaces 4–28. These 25 surfaces include 22 planes, two paraboloid reflecting surfaces, and one hyperboloid refracting surface.

For p processors, the speed per processor is given by the ratio r_p/p. Similarly, the speed per processor with np processors is given by the ratio $r_{np}/(np)$. The efficiency of a parallel application is defined by Kumar *et al.* as

$$E = \frac{r_{np}/(np)}{r_p/p}$$
$$E = \frac{r_{np}}{nr_p}. \quad (1)$$

This definition gives an efficiency $E = 100\%$ if increasing the number of processors p by a factor of n also increases the speed by the same factor, that is, if $r_{np} = nr_{\mathrm{p}}$. When the efficiency is 100%, there is a linear relationship between n and r_{np}.

If the efficiency can be held to near 100%, then intuitively we feel we are getting a good return for the extra cost of the extra processors. To the extent that the efficiency declines from 100%, we feel that we are paying for more processors but we are not getting a proportionate increase in speed.

Any assessment of a parallel application should include a calculation of the extent to which the efficiency approaches 100%.

III. Design Changes

In [6], two Bittware Hammerhead 66-MHz peripheral component interconnect boards (each with four Analog Devices 21160 DSPs operating at 80 MHz) ran under the control of an IBM-compatible personal computer (PC) based on an Intel Xeon central processing unit. The PC downloaded to the DSPs the ray-tracing program that ran in them. Apart from this, the PC itself served only to schedule tasks on the DSPs, which filled the role of workhorse processors. The PC passed information about the tasks to the DSPs using shared memory and received information about the results from them by the same means.

This required that the PC set aside shared data areas for each DSP. When the PC wanted to communicate with one of them, it wrote data into that section's shared memory area. To receive messages back from each DSP, it periodically inspected that DSP's shared memory area.

A much less cumbersome interface between multiple processors working on the same task is the Message Passing Interface (MPI), which is described in detail in a two-volume reference manual in [10] and [11]. The book by Pacheco [12] provides an excellent introduction to MPI. Under MPI, all processors execute the same program. The system assigns each processor a unique rank number, i.e., an integer in the range of 0 to $n - 1$, where n is the number of processors participating in a parallel computation. The processors can discover what their rank number is by making a specific system function call. As a result, it is easy for each processor to determine for itself which part of the overall problem it should tackle.

In adapting the ray-tracing program to use MPI, we abandoned the idea of giving to one processor the sole role of a task manager. Instead, the processor with rank 0 bore responsibility for reading inputs, sending copies of them to all the other processors, and receiving their outputs; in all other respects, it did the same kinds of work as all the other processors.

TABLE I
Scheme for Assigning Ray Bundles to Processors

Processor	Bundle numbers
0	$n, 2n, 3n, \ldots$
1	$1, n+1, 2n+1, 3n+1, \ldots$
2	$2, n+2, 2n+2, 3n+2, \ldots$
$\ldots$	$\ldots$
k	$k, n+k, 2n+k, 3n+k, \ldots$
$\ldots$	$\ldots$
$n-1$	$n-1, 2n-1, 3n-1, 4n-1, \ldots$

As described in [6], the ray-tracing simulation for MODIS consisted of tracing the paths of 29 161 rays, each comprising 1 out of 485 pinholes $\times$ 400 ray bundles/pinhole $=$ 194 400 ray bundles. Under MPI, each processor traced one bundle of rays at a time using its own rank number to select the next bundle to trace. The simple scheme we used was to assign to processor i all the bundles whose index number k satisfied the relationship $i = k \bmod n$. Table I shows how this worked. The bundles were numbered from 1 to 194 400.

IV. Observations

We collected measurements of the time each assigned processor devoted to tracing its subset of ray bundles. The initial results were instructive. Plots of the results are shown on the left-hand side of Fig. 2.[5] These results give the average time each processor spent tracing rays. The number n of processors working in each of these three cases was 100, 200, and 400, respectively. The plots show considerable variation in the time each processor spent on its share of the overall computation. The first two of these showed unusually low times for processors $21, 41, 61, \ldots$, and the third also shows similar cases. Inspection of the program showed that the initial rays lay in a 20 $\times$ 20 arrangement, which suggests that one processor was repeatedly tracing rays from the same column of this arrangement. Apparently, these rays did not require too much computation before they were discarded. In other words, the processor loads were very unbalanced.

A. Need for Load Balancing

Such an asymmetry often arises when multiple computers operate in parallel to solve a single problem: some processors are more heavily burdened than others. As a result, some processors may have no work left to do while others are still hard at work on the problem. Various means of load balancing to alleviate this problem have been studied. Lee and Lim [13], for example, present a study of load balancing they call *farming* in a system that generates photorealistic renderings of images. It is generally impossible to predict which subsets of rays will

[5]Throughout this paper, when we specify quantities in the form $x \pm \sigma$, σ represents one standard deviation in the measured quantity.

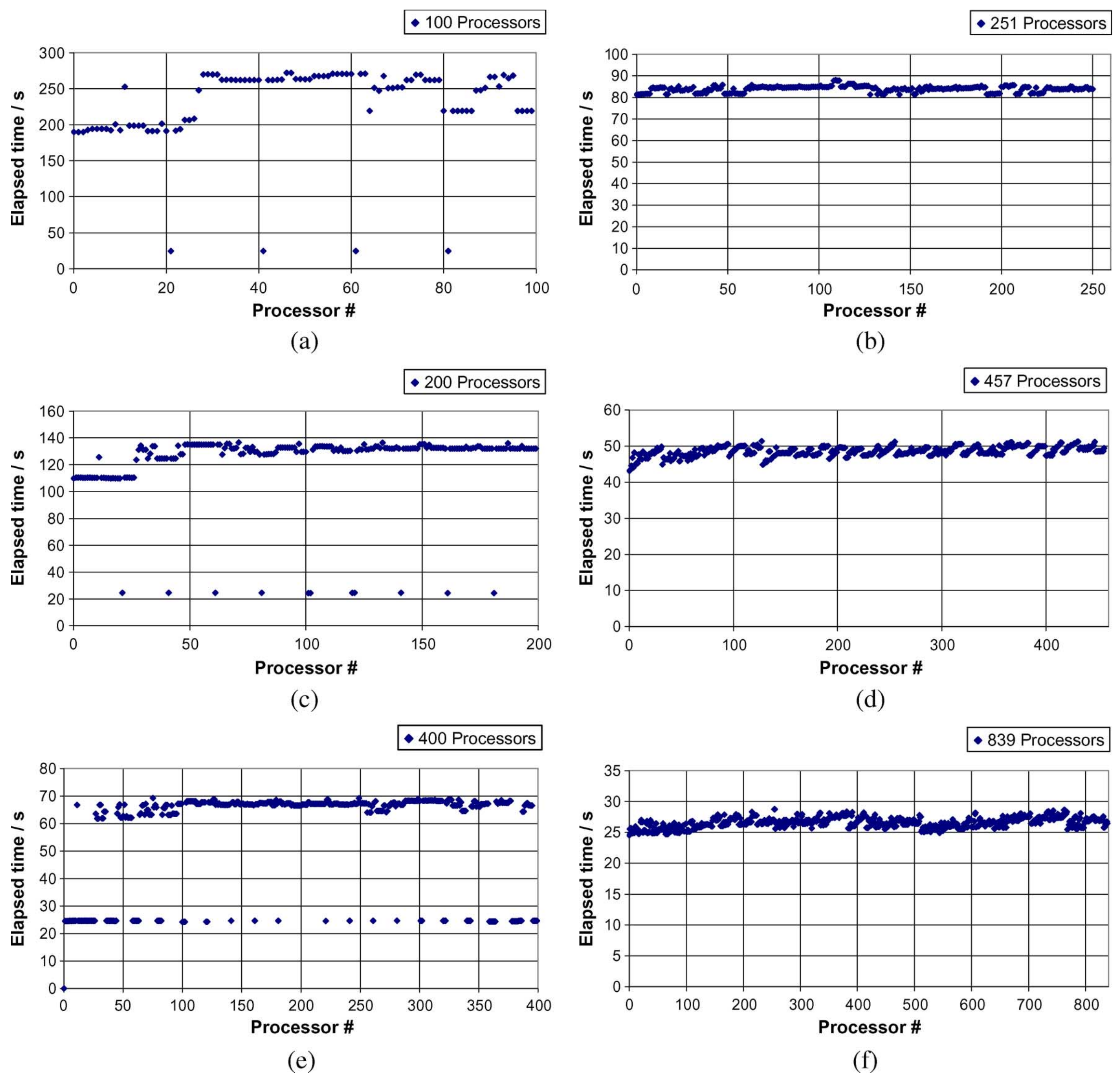

Fig. 2. Elapsed time for each processor for various numbers of processors. The figures on the left-hand side have a nonprime number of processors. Those on the right-hand side have a prime number of processors. The average elapsed time and its standard deviation are given in each case as T. The percentage to the right of the average shown below for each of the graphs is the ratio of standard deviation to average. With a prime number of processors, a reasonable load balancing is achieved without the use of extra software. The fact that the variations are so great with a nonprime number of processors is an indication that load balancing is necessary in the general case. (a) 100 processors: $T = (232 \pm 52)$ s (22%). (b) 251 processors: $T = (83.9 \pm 1.4)$ s (1.7%). (c) 200 processors: $T = (123 \pm 25)$ s (20%). (d) 457 processors: $T = (48.6 \pm 1.3)$ s (2.7%). (e) 400 processors: $T = (58 \pm 17)$ s (29%). (f) 839 processors: $T = (26.56 \pm 0.82)$ s (3.1%).

take the longest to trace, which makes it difficult to avoid a load imbalance problem.

Badouel *et al.* [7] discuss two broad classifications of load balancing in the context of ray tracing for graphical rendering, i.e., *data-oriented parallelization* and *control-oriented parallelization*.

In *data-oriented parallelization*, the data are subdivided into distinct subdomains, each of which is assigned to a distinct processor. For example, in a graphics rendering system, each processor might be responsible for handling the rays that lie within a subregion of the 3-D space. As a ray passes from one region to another, the processor handling it then passes it off to a new processor to trace while it is in the new region. This approach requires that different processors exchange messages whenever a ray leaves one region and enters another.

In *control-oriented parallelization*, each processor has its own local copy of the description of the system and performs the same tasks as any other processor but on a different subset of the data. For example, in a ray-tracing simulation of a lens system, each processor might handle its own subset of rays and independently trace them. This approach requires that the system description be passed to all processors at the outset. If the system changes over time, this will require repeated computation. (With the exception of zoom lenses, lens shapes *do not* vary with time. However, the set of objects in a scene-generation system *may* vary with time.)

The above example of a *data-oriented parallelization* approach can lead to load imbalances if one region contains no objects at all, which makes the job of its processor very simple, while another region contains many objects with complicated shapes, which makes its processor's job very much more involved.

The above example of a *control-oriented parallelization* approach can likewise lead to load imbalances if one processor is assigned rays that leave the system quickly, which makes its computational burden very light, while another is assigned a large number of rays that have to be traced all the way to the image plane.

Adding software to even out the load is a general solution to the problem. Such software entails having processors communicate with each other, sharing information about their current loads, and actively working to balance the work.

Rather than designing a load-balancing software, we simply avoided the problem by insisting that the number n of processors be a prime number.[6] This ensured that a processor would never repeat a trace from the same column of the pyramid of rays until it had first performed a trace from every other column.

To understand why this approach is successful, consider a program in which every kth task is particularly simple but other tasks are of comparable and greater complexity. We could say that in this case the periodicity of simple tasks is k. If the simpler tasks are assigned only to a subset of all the processors, then these processors will be underburdened and will have more idle time than the other busier processors will. This situation constitutes an unbalanced load.

To avoid the possibility of this occurring, we can apply some results from abstract algebra as presented by Fraleigh [14].

Theorem 1 (Fraleigh's Theorem 6.3): The set $\{0, 1, 2, \ldots, n-1\}$ is a cyclic group $\mathbf{Z}_n$ of n elements under addition modulo n.

The numbers $0, 1, 2, \ldots, n-1$ can be regarded as identifiers of individual processors; they constitute a cyclic group. The members of this cyclic group can be generated by the member $a = 1$. This entails starting with any member in the group (a itself is an obvious candidate), adding a to it modulo n to get the next member in the group, and repeating the process until all n members of the group have been found.

Theorem 2 (Fraleigh's Theorem 6.4): Let G be a cyclic group with n elements and generated by a. Let $b \in G$ and $b = sa$. Then, b generates a cyclic subgroup H of G containing n/d elements, where d is the greatest common divisor of n and s and is abbreviated as $\gcd(n, s)$.[7]

This theorem implies that if the greatest common divisor of the number n of processors and the periodicity k of simple tasks is *not* equal to 1, then the group member $b = ka = k1 = k$ generates a cyclic subgroup with fewer than n members. Again regarding the numbers $0, 1, 2, \ldots, n-1$ as processor identifiers and supposing that the first simple task is assigned to processor k, we can generate the subgroup H of all processors that are assigned an easy task by taking $2k, 3k, 4k, \ldots$, and this subgroup will have fewer than n members. As a result, some of the n processors available will not be assigned their fair share of these simple tasks.

[6]Except in the case where $n = 1$, which is not considered to be prime.

[7]Fraleigh used multiplicative notation with $b = a^s$. We use additive notation here with $b = sa$, because the specific group we are using entails addition modulo n.

As an example, suppose we have $n = 6$ processors numbered $0, 1, \ldots, 5$, and that every fourth task is particularly easy, that is, $k = 4$. The greatest common denominator is $d = \gcd(6, 4) = 2$, and the theorem claims that the subgroup H will only contain $n/d = 6/2 = 3$ members. If we assign the first easy task to processor 4, easy tasks will be assigned to the processors as follows:

First easy task:	$4 \bmod 6 = 4$;
Second easy task:	$4 + 4 \bmod 6 = 2$;
Third easy task:	$2 + 4 \bmod 6 = 0$;
Fourth easy task:	$0 + 4 \bmod 6 = 4$;

and so on. It should be clear that the pattern will repeat indefinitely, and only the three processors 0, 2, and 4 are ever assigned the easy tasks. Of the six processors available, only half get the easy tasks, and the other half only get the hard ones, as the theorem predicts. This constitutes an unbalanced load.

The theorem tells us that we can eliminate the imbalance and ensure that simple tasks are assigned to every processor if the number n of processors and the number k expressing the periodicity of simple tasks have greatest common divisor equal to 1. This will certainly be true if we choose n to be prime and if $k < n$. Otherwise, we must ensure that n and k have no common factors apart from the number 1. To do this, we must know both n and k.

Knowing n is easy, as it is the number of processors we choose to assign to a problem. Knowing k is not so easy, as it depends on the internal details of the algorithm used to tackle the problem.

Continuing the above example, where every fourth task is easy and the first easy task is assigned to processor 4, but with the number n of processors now equal to the prime number 7, the processors are assigned as follows:

First easy task:	$4 \bmod 7 = 4$;
Second easy task:	$4 + 4 \bmod 7 = 1$;
Third easy task:	$1 + 4 \bmod 7 = 5$;
Fourth easy task:	$5 + 4 \bmod 7 = 2$;
Fifth easy task:	$2 + 4 \bmod 7 = 6$;
Sixth easy task:	$6 + 4 \bmod 7 = 3$;
Seventh easy task:	$3 + 4 \bmod 7 = 0$;
Eighth easy task:	$0 + 4 \bmod 7 = 4$;

and so on. It is apparent that the easy tasks are assigned to all the processors in turn. Every processor gets a more or less even distribution of the easy tasks.

In this paper, k appeared to depend on the number of columns in a certain array used by the algorithm. The need to pin down the value of k through a close analysis of an algorithm can be avoided by simply choosing n to be prime, for then, the greatest common divisor is guaranteed to be 1, no matter what k is, provided that $k < n$.

If $k \geq n$, then it is possible that $k = an$ for some positive integer a. In this case, easy tasks would always be assigned to processor 0, although that processor would also be assigned more complicated tasks if $a > 1$. This situation can be detected by comparing the times consumed by each processor in a small test run. Finding that one of them is substantially smaller than all the others is evidence of this problem. Once the situation has been discovered, it is possible to avoid it by picking a different prime number for n.

TABLE II
EXPERIMENTAL OBSERVATIONS WHEN TRACING RAYS ON VARYING NUMBERS OF PARALLEL PROCESSORS ON NRL'S CRAY XD-1 SUPERCOMPUTER

# of Processors	Speed / (10^6 rays/s)	Time / s	StdDev of Time / s	Fractional % Error in Time
1	0.28	20113.93	—	—
2	0.57	10200.94	—	—
5	1.34	4287.16	55.28	1.29
11	3.14	1874.67	63.40	3.38
19	5.10	1113.51	3.65	0.33
51	13.33	424.46	0.26	0.06
79	20.75	272.74	0.42	0.15
101	28.39	206.90	4.02	1.94
197	56.20	106.70	3.05	2.86
251	69.53	83.92	1.38	1.64
457	131.17	48.58	1.33	2.73
839	230.44	26.56	0.82	3.08

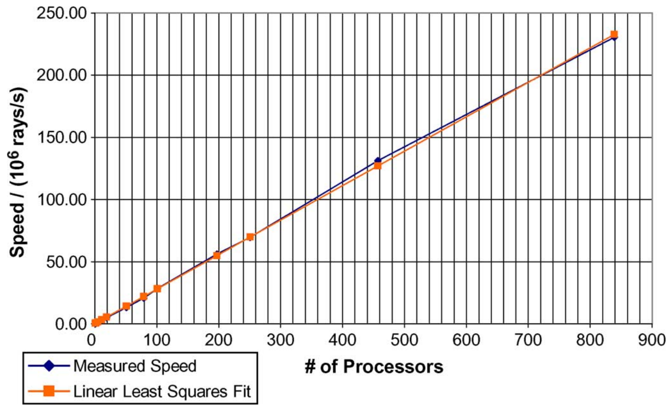

Fig. 3. Measured speeds s of ray tracing with various numbers n of processors. The least number of processors assigned to the task was 1. All the other choices are prime numbers that are chosen to avoid the need for a load-balancing software. The numbers are 1, 2, 5, 11, 19, 51, 79, 101, 197, 251, 457, and 839.

B. Results With Balanced Loads

The results from three of the runs we performed after applying Theorem 2 are shown on the right-hand side of Fig. 2. These runs were made with 251, 457, and 839 processors, respectively (all prime numbers). While there is still some variation in the measured times, it is dramatically reduced from between 20% and 30% to around 3% or less.

We then performed a more complete set of simulations and chose n to be prime numbers logarithmically distributed more or less from 1 to 840.[8] The average times a processor spent on simulation and the standard deviation of each of these times are shown in Table II.

To assess the linearity of these data, we performed a linear regression. Fig. 3 shows the results.

The linear equation describing the best least-squares fit for the data gives the speed s as a function of the number n of assigned processors

$$s = \big((277.3 \pm 2.0) \times 10^3 \text{ rays/(s} \cdot \text{processor)}\big)\, n + (164 \pm 598) \times 10^3 \text{ rays/s}) \quad (2)$$

with a correlation coefficient of 0.9997. The large s-intercept is significant for a small n only and may be high because of the use of a logarithmic spread for n. It indicates that this linear equation does not accurately predict speed for low values of n. However, it does show a very strongly linear relationship.

As an alternative to this straightforward linear regression, we instead sought a linear relationship between $\log n$ and $\log s$. A linear regression yielded

$$\log_{10} s = (0.9891 \pm 0.0030) \log_{10} n + (5.4440 \pm 0.0055) \quad (3)$$

which can be rewritten as

$$s = 10^{(5.4440 \pm 0.0055)} n^{(0.9891 \pm 0.0030)}. \quad (4)$$

Using the upper and lower error bounds for the power of 10, we can approximate this expression as

$$s = (278\,000 \pm 3500) n^{0.9891}. \quad (5)$$

The exponent of n is very close to unity, which again reflects the nearly linear relationship between speed and the number of processors. The slope in (5) can be seen to be close to that in (2), as should be expected. The absence of a constant coefficient is intuitively satisfying: with zero processors, we expect zero speed. Furthermore, the error bounds in (3) are tighter than those in (2) (0.3% compared to 0.7%), which suggests a better fit. Finally, the regression coefficient is even closer to 1. It is 0.99995 for (3) compared to 0.9997 for (2), which is another indication that this method of prediction is more accurate.

The data predicted using (3) are plotted along with the observed values in Fig. 4. A comparison with a similar plot in Fig. 3, which is based on the predictions in (2), shows the

[8]The selection of $n = 251$ was made so that a comparison with another system that only had 256 processors might be made.

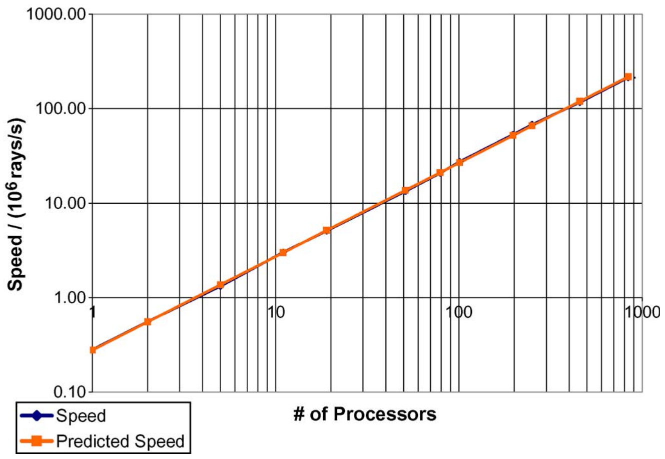

Fig. 4. Predicted speeds s compared to actual speeds for various numbers n of processors when a linear regression is performed using $\log n$ and $\log s$.

closer fit of (3). The gap between observed and predicted values for $n = 457$ in Fig. 3 is not present in Fig. 4. The principle difference applies for a very small n, and this is hard to discern by mere visual inspection of the figures.

As mentioned earlier, only a fraction of the rays traced struck the image plane. Reducing the speed shown here to 80.4% of the calculated value compensates for what amounts to a poor choice of rays to trace. Using only the linear proportionality coefficient of (5), we can estimate our ray-tracing speed as

$$278.0 \times 10^3 \text{ rays}/(\text{s} \cdot \text{processor}).$$

Because there are 25 surfaces for the vast majority of rays, multiplying by this factor gives the normalized ray-tracing speed

$$6.95 \times 10^6 \text{ rays} \cdot \text{surfaces}/(\text{s} \cdot \text{processor}).$$

A single Opteron 275 processor traced 5 657 234 000 rays at a rate of 280×10^3 rays/s. With 839 of these processors, the rate increased to 230×10^6 rays/s. The efficiency in using 839 processors can be calculated using (1) as

$$\begin{aligned} E &= \frac{r_{np}}{nr_p} \\ &= \frac{230 \times 10^6 \text{ rays/s}}{839 \times 280 \times 10^3 \text{ rays/s}} \\ &= 97.9\%. \end{aligned}$$

This is quite close to 100%, which is yet another indication that the ray-tracing application is highly linear in the number of processors brought to bear on the problem.

V. Conclusion

The change from Analog Devices AD21160 DSPs clocked at 80 MHz to AMD Opteron 275 processors clocked at 2.2 GHz represented an increase in clock frequency of 2650%.[9] It resulted in increasing the speed of ray tracing from 111×10^3 to 6.95×10^6 rays $\cdot$ surfaces/(s $\cdot$ processor), which is an increase of over 6100%. The switch also permitted an increase in the maximum number of processors available from 8 to 839. As a result, the simulation time fell from 284.4 h on a single AD21160 to under 27 s on 839 Opteron 275 processors, which is a reduction by a factor of 38 000. The efficiency achieved using 839 processors was 97.9%, which indicates that every doubling of the number of processors nearly doubles the speed. The highly linear relationship between processors and speed is a well-known characteristic of ray-tracing applications. This paper has shown that the relationship persists even when the number of processors reaches over 800.

We have introduced a novel approach to avoiding the need to write elaborate load-balancing software. By using a prime number of processors, we reduced the chance that one processor would get an inordinately heavy or light share of the processing load.

[9]The Analog Devices AD21160 has no hardware for performing floating-point-division or square-root operations, whereas the Opteron 275 does.

References

[1] K. F. Gauss, *Dioptrische Untersuchungen*. Göttingen, Germany: Dieterich, 1840.
[2] F. Williams and T. Kilburn, "The University of Manchester computing machine," in *Proc. Manchester Univ. Comput. Inaugural Conf.*, 1951, pp. 5–11.
[3] G. H. Spencer and M. V. R. K. Murtry, "General ray-tracing procedure," *J. Opt. Soc. Amer.*, vol. 52, no. 6, pp. 652–678, Jun. 1962.
[4] P. Mouroulis and J. Macdonald, *Geometrical Optics and Optical Design*. New York: Oxford Univ. Press, 1997.
[5] C. B. Cameron, R. N. Rodriguez, N. Padgett, E. Waluschka, and S. Kizhner, "Optical ray tracing using parallel processors," *IEEE Trans. Instrum. Meas.*, vol. 54, no. 1, pp. 87–97, Feb. 2005.
[6] C. B. Cameron, R. N. Rodriguez, N. Padgett, E. Waluschka, S. Kizhner, G. Colon, and C. Weeks, "Fast optical ray tracing using multiple DSPs," *IEEE Trans. Instrum. Meas.*, vol. 55, no. 3, pp. 801–808, Jun. 2006.
[7] D. Badouel, K. Bouatouch, and T. Priol, "Distributing data and control for ray tracing in parallel," *IEEE Comput. Graph. Appl.*, vol. 14, no. 4, pp. 69–77, Jul. 1994.
[8] E. Waluschka, X. Xiong, B. Guenther, W. Barnes, and V. Salomonson, "Modeling studies of the MODIS solar diffuser attenuation screen and comparison with on-orbit measurements," *Proc. SPIE*, vol. 5542, no. 47, pp. 342–353, 2004.
[9] V. Kumar, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. Redwood City, CA: Benjamin Cummings, 1994.
[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI—The Complete Reference*, 2nd ed, vol. 1. Cambridge, MA: MIT Press, 2000.
[11] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphire, and M. Snir, *MPI—The Complete Reference*, 2nd ed, vol. 2. Cambridge, MA: MIT Press, 2000.
[12] P. S. Pacheco, *Parallel Programming With MPI*. San Francisco, CA: Morgan Kaufmann, 1997.
[13] H. J. Lee and B. H. Lim, "Parallel ray tracing using processor farming model," in *Proc. IEEE Int. Conf. Parallel Process. Workshops*, 2001, pp. 59–63.
[14] J. B. Fraleigh, *A First Course in Abstract Algebra*. Reading, MA: Addison-Wesley, 1967.

Charles B. Cameron (SM'05) received the B.Sc. degree in computer science from the University of Toronto, Toronto, ON, Canada, in 1977 and the M.S.E.E. and Ph.D. degrees from the Naval Postgraduate School, Monterey, CA, in 1989 and 1991, respectively.

He is currently an Assistant Professor with the United States Naval Academy, Annapolis, MD, and a Lecturer with the Johns Hopkins University, Baltimore, MD. As a Commander in the United States Navy, he qualified as a Mission Commander in E-2C Hawkeye airborne early warning aircraft in 1983. His research and teaching interests are in massively parallel computing, computer architecture, reconfigurable computing, and sensors. He holds a patent for an electronic demodulator used to recover signals from interferometric fiber-optic sensors.

Dr. Cameron is a licensed Professional Engineer in the State of Maryland.