Approximating Euler's Number

/* By Jaret Varn, Thomas Laggan, and Zach Pallotta */

Programs

C

- MPI
- Pthreads
- Open MP

Python

-Multiprocessing library

Algorithm

Euler's number can be approximated by summing inverse factorials

$$\sum_{n=0}^{\infty} \frac{1}{n!} \approx e$$

$$\frac{1}{1} + \frac{1}{1*2} + \frac{1}{1*2*3} + \frac{1}{1*2*3*4} \dots \approx e$$

Challenges

- **Workload assignment matters**

1, 2, 3, 4, 5 <- Thread 0 (less work)

6, 7, 8, 9, 10 <- Thread 1 (more work)

- **Large numbers**

Factorials get BIG

Unsigned long long int max is 18,446,744,073,709,551,615

22! = 1,124,000,727,777,607,680,000 (already over 60x larger)

- **Floating point precision**

IEEE double can only accurately store 15 decimal places

The sum quickly converges to 15 decimal places of e with only 18 iterations

Comparison Measurements

Runtime Measurements for n = 16000

# of threads	Python (s)	MPI (s)	Pthreads (s)	OpenMP (s)
1	1.06e+02	1.07e+00	1.11e+00	1.09e+00
2	5.48e+01	5.46e-01	5.53e-01	5.96e-01
4	3.24e+01	2.77e-01	2.81e-01	2.96e-01
8	2.19e+01	1.39e-01	1.59e-01	1.45e-01

Speedup Measurements for n = 16000

# of threads	Python	MPI	Pthreads	OpenMP
1	1.00	1.00	1.00	1.00
2	1.94	1.95	2.00	1.83
4	3.28	3.85	3.94	3.67
8	4.85	7.67	6.96	7.50

Comparison Measurements (cont.)

Efficiency Measurements for n = 16000

# of threads	Python (%)	MPI (%)	Pthreads (%)	OpenMP (%)
1	100	100	100	100
2	97	98	100	91
4	82	96	99	92
8	61	96	87	94

C Code

MPI

```c
double local_start, local_stop, local_elapsed, elapsed;

MPI_Comm comm = MPI_COMM_WORLD;
MPI_Init(NULL, NULL);
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);

//retrieve n from user
if (rank == 0)
{
        printf("Enter value for n\n");
        scanf("%d", &n);
}

//broadcast n
MPI_Bcast(&n, 1, MPI_INT, 0, comm);

local_start = MPI_Wtime();

//calculate block
for (long long i = rank; i < n; i += size)
{
        local_sum += 1.0 / factorial((double)i);
        ifdef DEBUG
        printf("local sum is %0.15lf for process %d on index [%lld]\n", local_sum, rank, i);
        endif
}

local_stop = MPI_Wtime();
```

Factorial function

```c
//calculate factorial of double
double factorial(double num)
{
        if (num == 0.0)
        {
                return 1.0;
        }
        else
        {
                return num * factorial(num - 1.0);
        }
}
```

C Code

PThreads

```c
long thread;
pthread_t* thread_handles;
thread_count = strtol(argv[1], NULL, 10);
thread_handles = malloc(thread_count * sizeof(pthread_t));

//retrieve n from user
printf("Enter value for n\n");
scanf("%lld", &n);

if (n % thread_count != 0)
{
        printf("n must be evenly divisible by number of threads\nExiting...\n");
        exit(0);
}


//calculate block
clock_gettime(CLOCK_MONOTONIC, &start);
pthread_mutex_init(&mutex, NULL);
for (thread = 0; thread < thread_count; thread++)
{
        pthread_create(&thread_handles[thread], NULL, pthreadE, (void*) thread);
}
for (thread = 0; thread < thread_count; thread++)
{
        pthread_join(thread_handles[thread], NULL);
}
pthread_mutex_destroy(&mutex);
clock_gettime(CLOCK_MONOTONIC, &stop);

elapsed = stop.tv_sec - start.tv_sec;
elapsed += (stop.tv_nsec - start.tv_nsec) / 1000000000.0;
```

OMP

```c
        //omp parallel code
        start = omp_get_wtime();
#   pragma omp parallel for schedule(dynamic) num_threads(thread_count) \
      reduction(+: sum)
        for (long long i = 0; i < n; i++)
        {
                sum += 1.0 / factorial((double)i);
        }
        stop = omp_get_wtime();

        elapsed = stop - start;
```

Python

Multiprocessing

```python
thread_count = 8
iterations = 1000

# Sets the precision to 15 decimal places because we are using doubles for our C implementation.
mp.dps = 15

def euler_approximation(n, rank, return_val):
    local_chunk = math.floor(n / thread_count)
    local_sum = 0

    for i in range(1, local_chunk + 1):
        chunked_n = thread_count * (i - 1) + rank
        factorial = mpf(math.factorial(chunked_n))
        local_sum = mpf(local_sum + factorial ** -1)
    return_val[rank] = mpf(local_sum)


if __name__ == '__main__':
    threads = []
    manager = multiprocessing.Manager()
    return_val = manager.dict()
    start_time = time.time()

    for i in range(0, thread_count):
        new_thread = multiprocessing.Process(target=euler_approximation, args=(iterations, i, return_val))
        threads.append(new_thread)
        new_thread.start()
    for i in threads:
        i.join()

    end_time = time.time()
    run_time = end_time - start_time
    sum = 0
    # Serial sum each thread's part
    for i in range(0, thread_count):
        sum += return_val[i]

    print(f"Estimated e:        {sum}")
    print(f"Runtime (seconds):  {run_time}")
```

Conclusions

- All 3 C implementations shared similar scalable performance with near 100% efficiency.
- Our C code out-performed Python as expected, with an order of 100x in speed comparison.