COMP 464 - High Performance Computing
OpenMP Threading

Loyola University Chicago
Jose Luis Rodriguez

November 9, 2017

1 Overview

This report highlights the procedures and results of running the nbody3 C++ code that predicts the individual motion of a group of celestial objects (particles) interacting with each other gravitationally. The program also runs a series of experiments and generates a benchmark (max, min and average speed) utilizing the Stampede2 supercomputer at The University of Texas at Austin?s Texas Advanced Computing Center (TACC). The nbody3 code utilize Intel C++ library compilers and the OpenMP library in order to parallelize the serial code details and speci?cations to follow.

The n-body problem code performs two major benchmarks in order to compute strong scalability and weak scalability. The program was compiled in serial and parallel each version was tested with 1,000, 2,000, 4,000, and 8,000 particles and 1, 2, 4, 8, 16, 32, and 64 threads, all tests ran with the same number of steps (-s 500). The average time per number of particles was recored in order to compute the strong scalability. This method also allows to compute the weak scalability test as we are keeping a constant workload per-thread when the number of particles increase the number of threads also increased. To ?nd out the number of particles needed per thread the parallel code was executed keeping the number of particles and steps constant (n=1000, s=200) while increasing the number of threats.

2 Benchmark Analysis

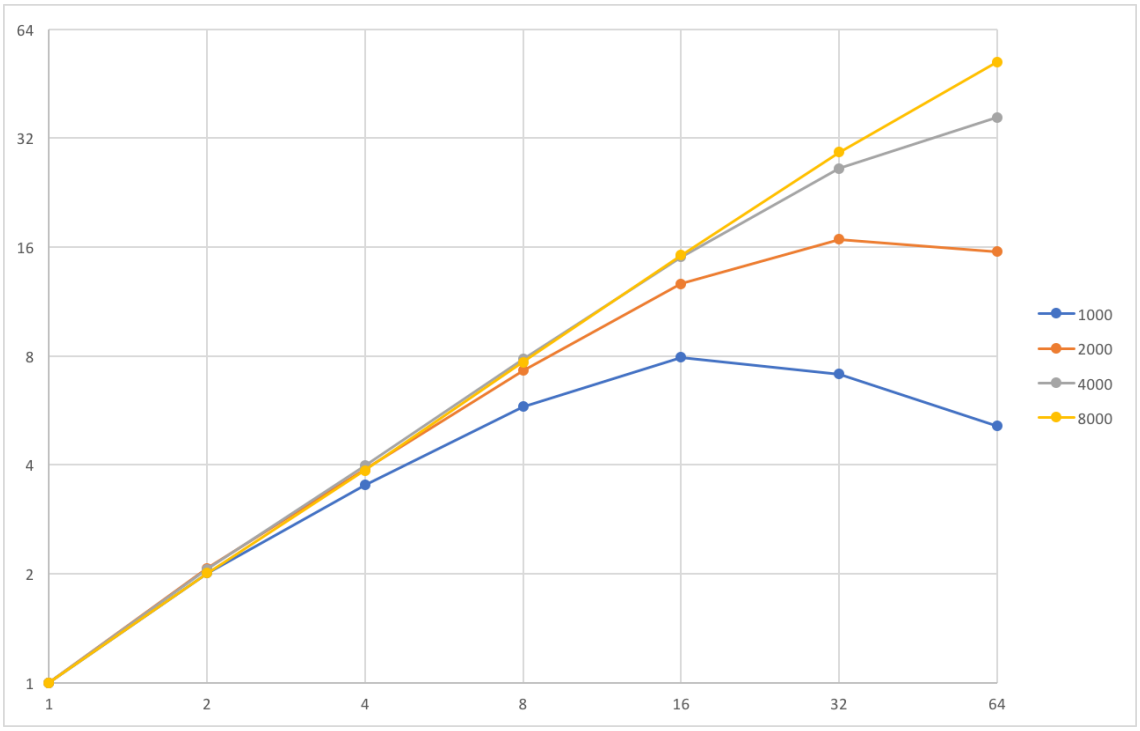
Code Mofications

In order to perform the benchmarks in parallel, pragmas (statements tell the compiler to use the OpenMP framework to parallelize the code) were used on the outer for loops of the accelerate, update and search functions. In the later also a reduction on the min, max and average velocity was used, snippets of the code can be found in the reference.

2.1 Strong scalability

When computing the strong scalability for the nbody problem the particles size stays fixed while the number of threads increase. In Figure.1 this benchmark was perform on 1000, 2000, 4000 and 8000 particles and for each set of particles the benchmark was run with 1, 2, 4, ... threads. We can see in the chart how the nbody problem follows shows the strong scalability principle as its almost a perfect speedup (execution time goes down with the number of threads).

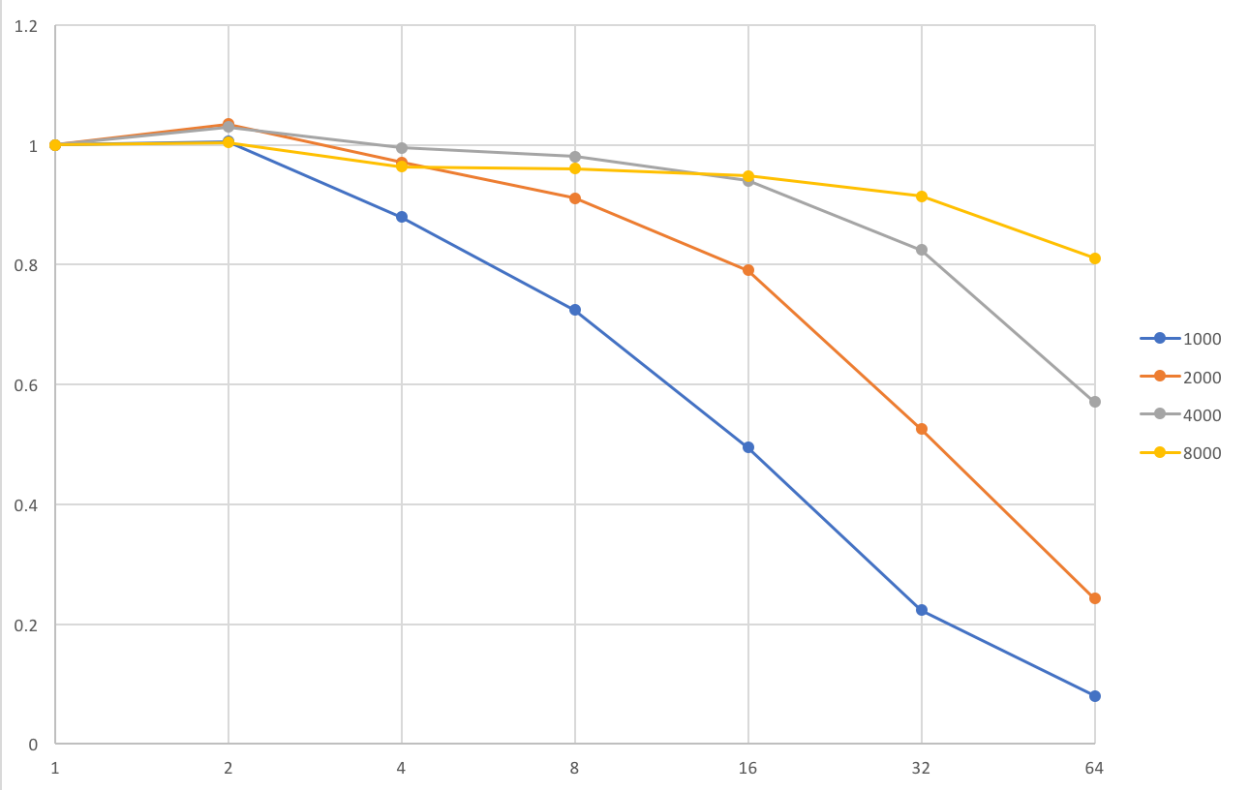
Figure 1: Strong Scaling - Speed-Up (T1/Tp)



The other measure to identify how far the nbody problem is from the ideal speed we compute the efficient. In Figure.2 we can see how as the threads increased depending of the particles sized the efficiency of the parallel program decreased.

This is due to that in many cases starting the parallelize regions creates an overhead that when the problem size doesnt scale proportionally to the number of threads the program efficiency decreased significantly.

Figure 2: Strong Scaling - Efficiency (Sp/P)

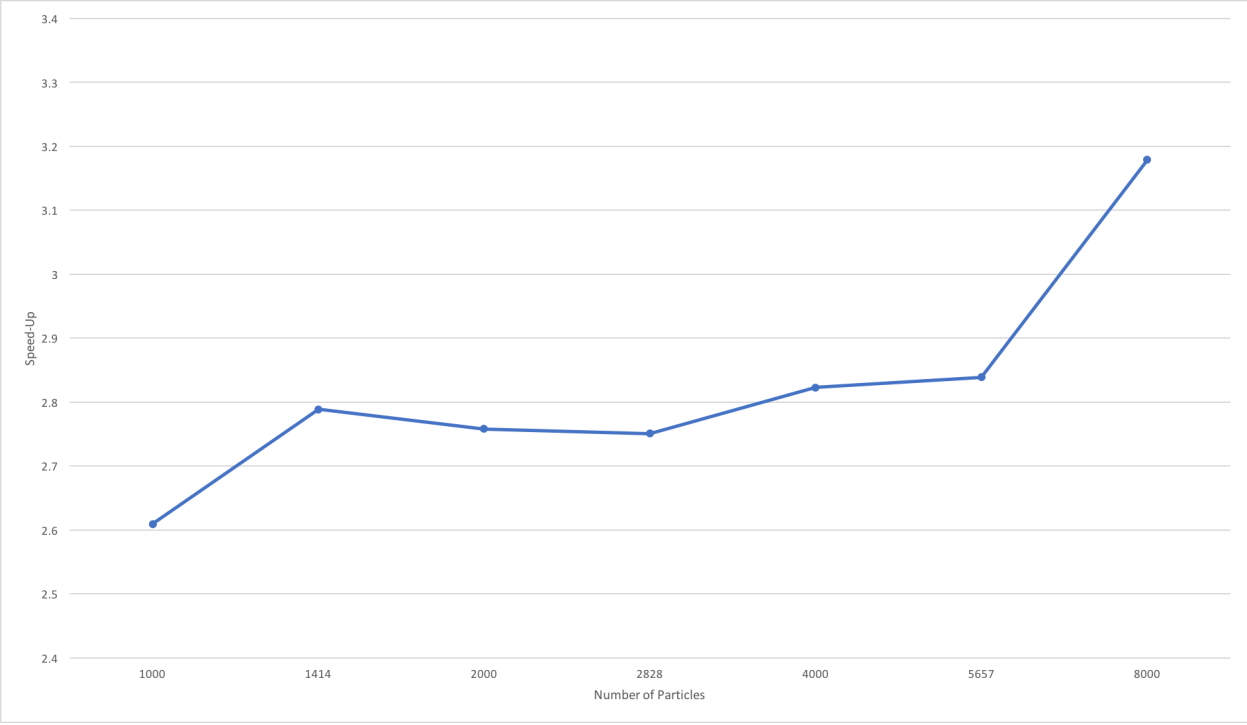


2.2 Weak scalability

A more realistic accurate way to measure the speed-up, efficiency of a parallel program, would distribute the data amount all threads in order to avoid idling due to some threads having less data to process. To summarize when we use more than thread communications between threads increased which creates an overhead. Now when the data is not distributed evenly between threads then we would have some threads idling. These are some of the reason why the actual speedup and efficiency of the parallel program decreased. For the case of weak scalability on the nbody problem as the particles and threads increase we want to keep the amount of data per thread to stay constant or as close to constant as possible to achieve constant execution time.

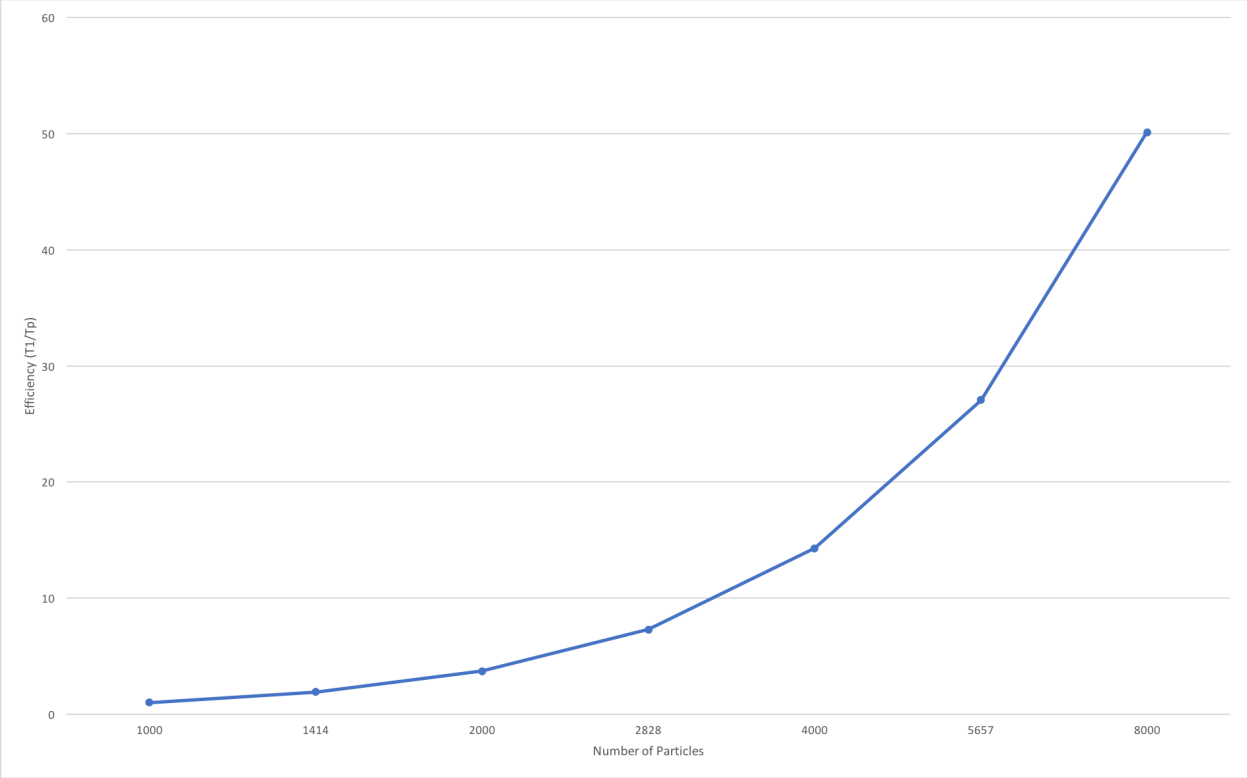
In Figure.3 we can see how as the number of particles and threads increase the speed-up stays more or less constant, with some variation at the tails of the chart.

Figure 3: Weak Scaling - Speed-Up (T1/Tp)



For the case of weak scalability efficiency in Figure.4 the numbers of particles are given by the recursive function $P_i = \sqrt{2} \times P_{i-1}$. We can see how as the number of particles and threads increase efficiency also increases.

Figure 4: Weak Scaling - Efficiency (T_1/T_p)



3 Reference

Stampede2 User Guide – Managing Memory
`https://portal.tacc.utexas.edu/user-guides/stampede2#managingmemory`

Introduction to High Performance Scientific Computing – Victor Eijkhout
`http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html`

Table 1: Strong Scaling Benchmark - Number of Particles: 1000

Threads	**runtime-n1000**	**speed-up-n1000**	**efficiency-n1000**
1	2.699547	1.00	1.00
2	1.342266	2.01	1.01
4	0.767273	3.52	0.88
8	0.465877	5.79	0.72
16	0.340848	7.92	0.50
32	0.378338	7.14	0.22
64	0.527209	5.12	0.08

Table 2: Strong Scaling Benchmark - Number of Particles: 2000

Threads	**runtime-n2000**	**speed-up-n2000**	**efficiency-n2000**
1	10.702174	1.00	1.00
2	5.170901	2.07	1.03
4	2.755932	3.88	0.97
8	1.468389	7.29	0.91
16	0.846486	12.64	0.79
32	0.636906	16.80	0.53
64	0.689989	15.51	0.24

Table 3: Strong Scaling Benchmark - Number of Particles: 4000

Threads	**runtime-n4000**	**speed-up-n4000**	**efficiency-n4000**
1	42.174895	1.00	1.00
2	20.477445	2.06	1.03
4	10.593958	3.98	1.00
8	5.375854	7.85	0.98
16	2.804787	15.04	0.94
32	1.598818	26.38	0.82
64	1.155382	36.50	0.57

Table 4: Strong Scaling Benchmark - Number of Particles: 8000

Threads	**runtime-n8000**	**speed-up-n8000**	**efficiency-n8000**
1	161.399121	1.00	1.00
2	80.410162	2.01	1.00
4	41.872722	3.85	0.96
8	21.005236	7.68	0.96
16	10.634796	15.18	0.95
32	5.517657	29.25	0.91
64	3.109065	51.91	0.81

Table 5: Weak Scaling Benchmark - Number of Particles: 1000

Threads	**runtime-n1000**	**speed-up-n1000**	**efficiency-n1000**
1	2.609648	1.00	1.00
2	1.346056	1.94	0.97
4	0.759546	3.44	0.86
8	0.471636	5.53	0.69
16	0.356043	7.33	0.46
32	0.393292	6.64	0.21
64	0.57277	4.56	0.07

Table 6: Weak Scaling Benchmark - Number of Particles: 1414

Threads	**runtime-n1414**	**speed-up-n1414**	**efficiency-n1414**
1	5.316273	1.00	1.00
2	2.788752	1.91	0.95
4	1.497149	3.55	0.89
8	0.825781	6.44	0.80
16	0.538592	9.87	0.62
32	0.499729	10.64	0.33
64	0.667798	7.96	0.12

Table 7: Weak Scaling Benchmark - Number of Particles: 2000

Threads	**runtime-n2000**	**speed-up-n2000**	**efficiency-n2000**
1	10.186014	1.00	1.00
2	5.148521	1.98	0.99
4	2.757893	3.69	0.92
8	1.468954	6.93	0.87
16	0.852953	11.94	0.75
32	0.647286	15.74	0.49
64	0.688234	14.80	0.23

Table 8: Weak Scaling Benchmark - Number of Particles: 2828

Threads	**runtime-n2828**	**speed-up-n2828**	**efficiency-n2828**
1	20.007112	1.00	1.00
2	10.115954	1.98	0.99
4	5.331802	3.75	0.94
8	2.751072	7.27	0.91
16	1.495437	13.38	0.84
32	0.960981	20.82	0.65
64	0.854334	23.42	0.37

Table 9: Weak Scaling Benchmark - Number of Particles: 4000

Threads	**runtime-n4000**	**speed-up-n4000**	**efficiency-n4000**
1	40.344163	1.00	1.00
2	20.077003	2.01	1.00
4	10.612086	3.80	0.95
8	5.409125	7.46	0.93
16	2.822763	14.29	0.89
32	1.659283	24.31	0.76
64	1.192224	33.84	0.53

Table 10: Weak Scaling Benchmark - Number of Particles: 5657

Threads	**runtime-n5657**	**speed-up-n5657**	**efficiency-n5657**
1	76.810418	1.00	1.00
2	38.630002	1.99	0.99
4	20.254395	3.79	0.95
8	10.229366	7.51	0.94
16	5.252118	14.62	0.91
32	2.839027	27.06	0.85
64	1.781306	43.12	0.67

Table 11: Weak Scaling Benchmark - Number of Particles: 8000

Threads	**runtime-n8000**	**speed-up-n8000**	**efficiency-n8000**
1	159.344296	1.00	1.00
2	79.801012	2.00	1.00
4	41.869284	3.81	0.95
8	21.055475	7.57	0.95
16	10.666953	14.94	0.93
32	5.562885	28.64	0.90
64	3.178754	50.13	0.78

Figure 5: OpenMP Pragmas for Acceleration Function

```
template <typename ValueType>
void accel_register (    ValueType * __RESTRICT pos,
                         ValueType * __RESTRICT vel,
                         ValueType * __RESTRICT mass,
                         ValueType * __RESTRICT acc,
                         const int n)
{
   #pragma omp parallel
    {
    #pragma omp for
       for (int i = 0; i < n; ++i)
       {
          ValueType ax = 0, ay = 0, az = 0;
          const ValueType xi = pos_array(i,0);
          const ValueType yi = pos_array(i,1);
          const ValueType zi = pos_array(i,2);
          for (int j = 0; j < n; ++j)
          {
             /* Position vector from i to j and the distance^2. */
             ValueType rx = pos_array(j,0) - xi;
             ValueType ry = pos_array(j,1) - yi;
             ValueType rz = pos_array(j,2) - zi;
             ValueType dsq = rx*rx + ry*ry + rz*rz + TINY2;
             ValueType m_invR3 = mass[j] / (dsq * std::sqrt(dsq));

             ax += rx * m_invR3;
             ay += ry * m_invR3;
             az += rz * m_invR3;
          }

          acc_array(i,0) = G * ax;
          acc_array(i,1) = G * ay;
          acc_array(i,2) = G * az;
       }
    }
}
```

Figure 6: OpenMP Pragmas for Update Function

```
template <typename ValueType>
void update (ValueType pos[], ValueType vel[], ValueType mass[], ValueType
    acc[], const int n, ValueType h)
{
    #pragma omp parallel
    {
    #pragma omp for
       for (int i = 0; i < n; ++i)
          for (int k = 0; k < NDIM; ++k)
          {
             pos_array(i,k) += vel_array(i,k)*h + acc_array(i,k)*h*h/2;
             vel_array(i,k) += acc_array(i,k)*h;
          }
    }
}
```

Figure 7: OpenMP Pragmas for Search Function

```
template <typename ValueType>
void search (ValueType pos[], ValueType vel[], ValueType mass[], ValueType
    acc[], const int n)
{
  ValueType minv = 1e10, maxv = 0, ave = 0;
  #pragma omp parallel default(none) shared(minv,maxv,ave)
  {
  #pragma omp for reduction(+:ave) reduction(max:maxc) reduction(min:minv)
   for (int i = 0; i < n; ++i)
   {
         ValueType vmag = 0;
         for (int k = 0; k < NDIM; ++k)
                vmag += (vel_array(i,k) * vel_array(i,k));
         vmag = sqrt(vmag);
         maxv = std::max(maxv, vmag);
         minv = std::min(minv, vmag);
         ave += vmag;
    }
   printf("min/max/ave velocity = %e, %e, %e\n", minv, maxv, ave/n);
  }
}
```

Table 12: Stampede2 - Compute Node (knl) - System Information

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	272
On-line CPU(s) list	0-271
Thread(s) per core	4
Core(s) per socket	68
Socket(s)	1
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	87
Model name	Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
Stepping	1
CPU MHz	1255.132
BogoMIPS	2793.44
L1d cache	32K
L1i cache	32K
L2 cache	1024K
NUMA node0 CPU(s)	0-271
NUMA node1 CPU(s)	