COMP 464 - High Performance Computing
OpenMP Threading

Loyola University Chicago
Jose Luis Rodriguez

November 2, 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

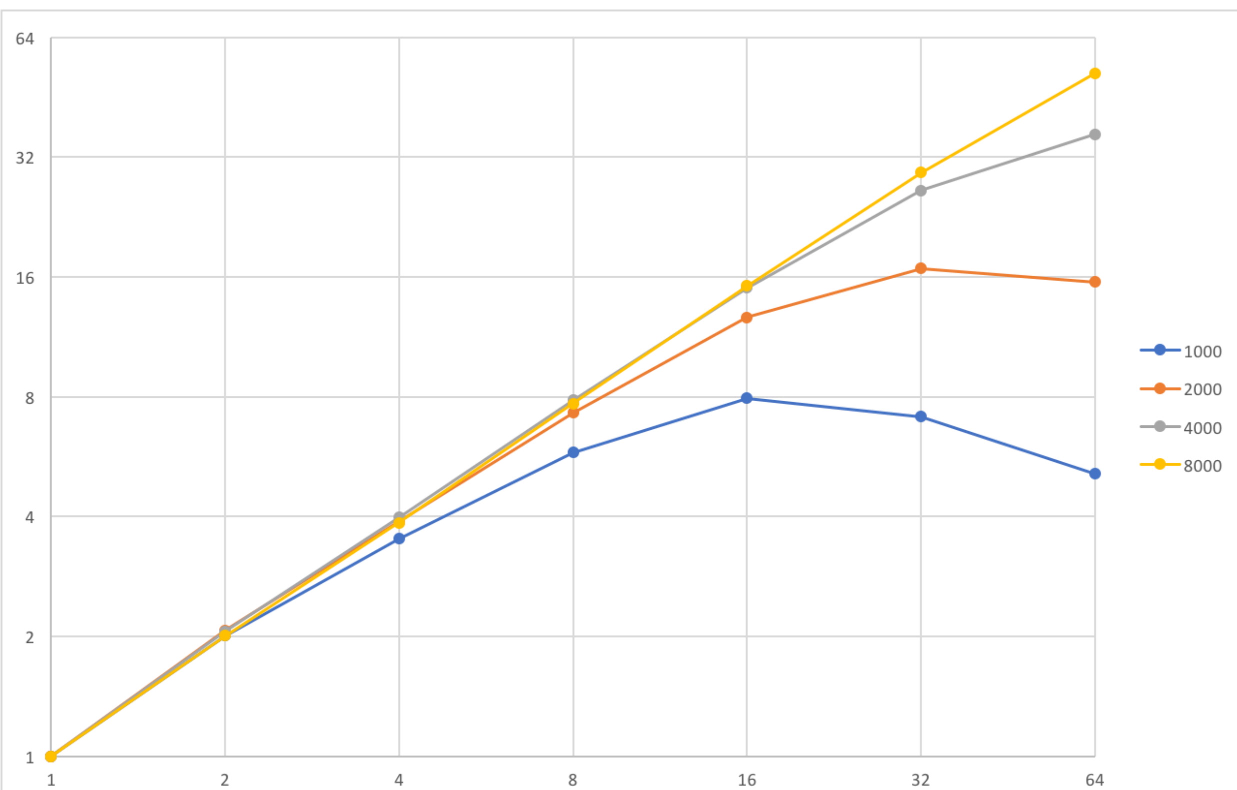
Code Mofications

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

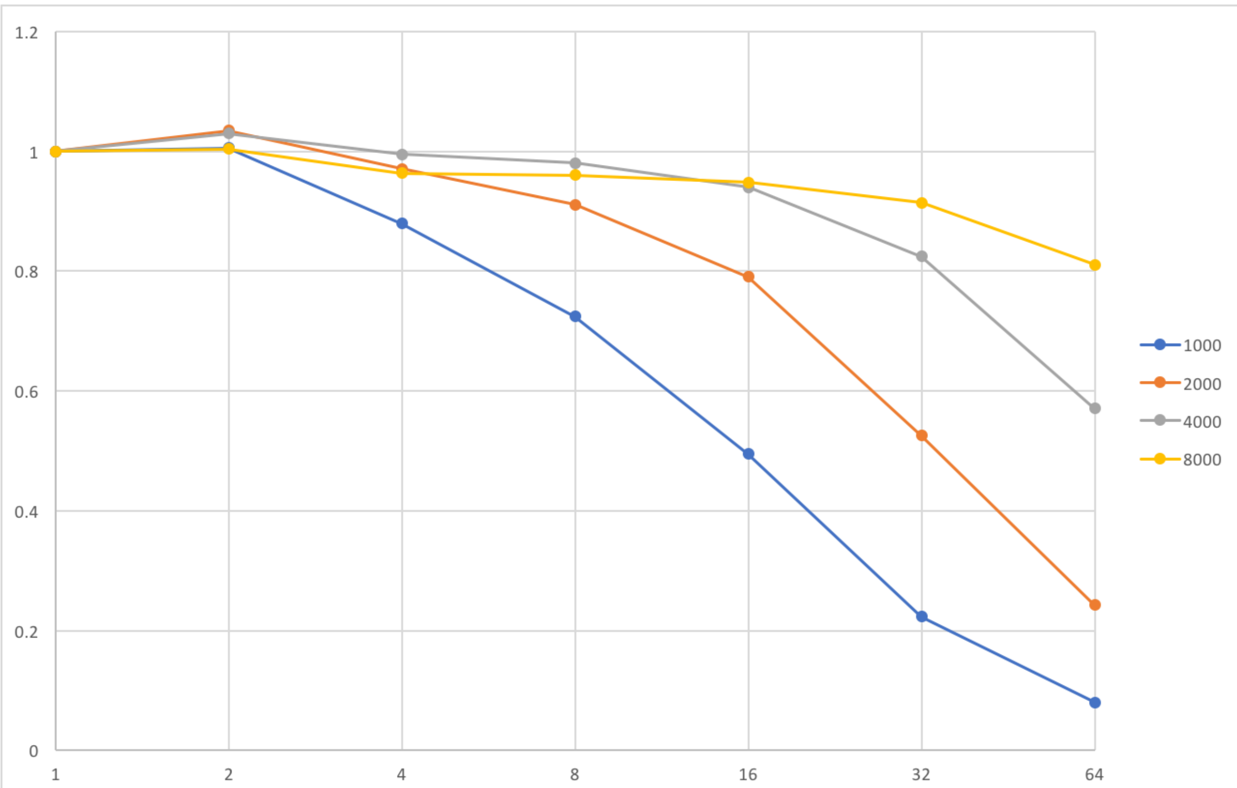
When the nbody problem particles size stays ?xed or increased as the number of processing elements are increased. In Figure.1 Stampede2 compute node using the GNU compiler, we can see how the benchmark that uses most bandwidth is the COPY benchmark, with a top performance at $26kb$

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



In Figure.1 Stampede2 compute node using the GNU compiler, we can see how the benchmark that uses most bandwidth is the COPY benchmark, with a top performance at $26kb$ right before filling the $32kb$ L1 cache then we see another drop at approximately after 534kb array size when L2 cache gets filled. The other benchmarks seem to behave more or less similarly, some of them show some drops but nothing close to what the COPY benchmark experience.

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

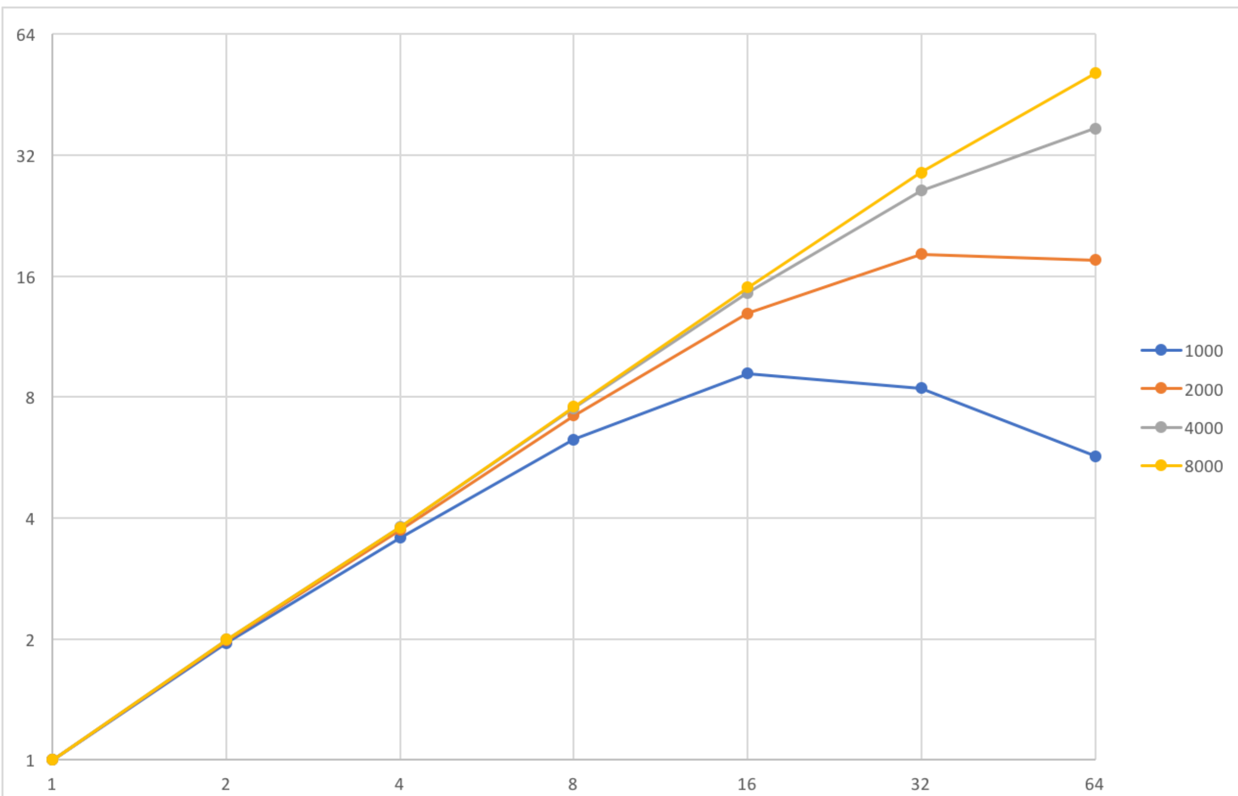


2.2 Weak scalability

When the nbody problem size particles assigned to each processing element stays constant and additional elements are used to solve the nbbody problem. Now in Figure.2 we change the compiler to Intel's C++ compiler and we can see significant changes on all the benchmarks. The most prominent change from the previous figure is the AXPY

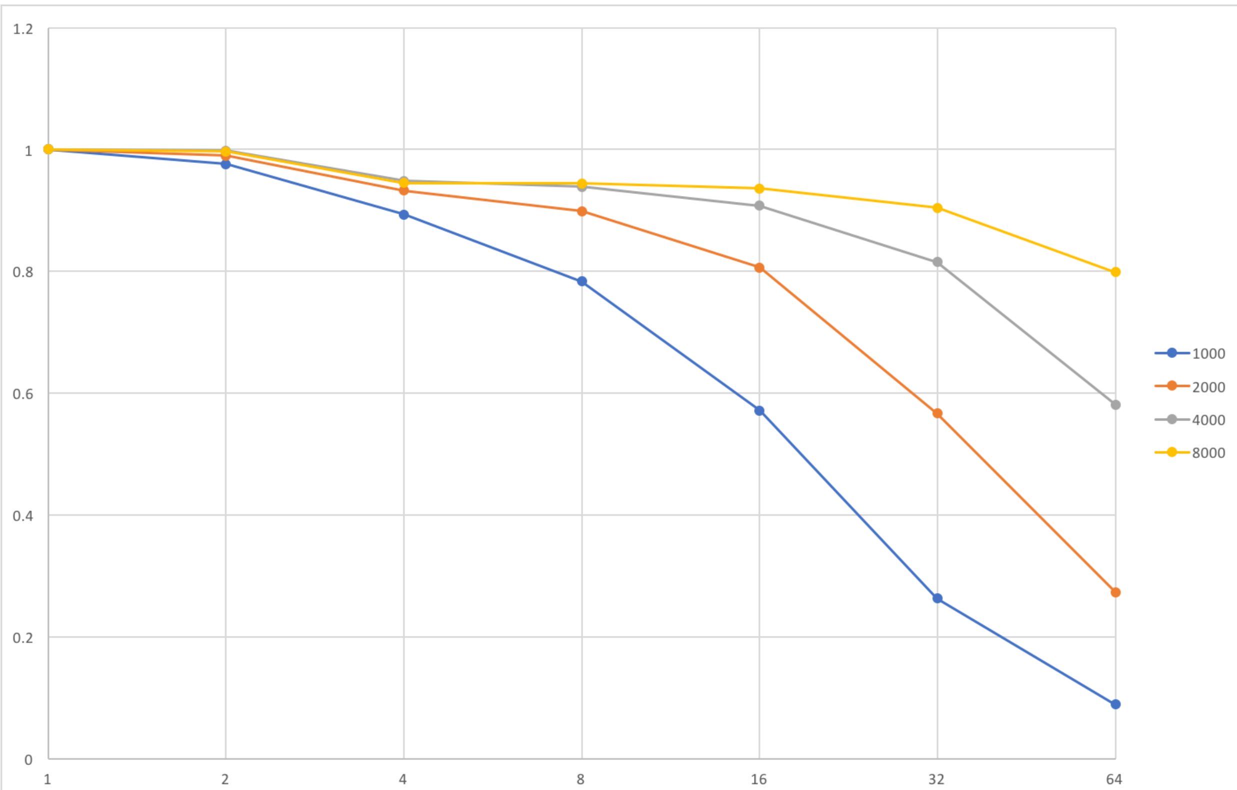
benchmark that peeks approximately around $40kb$ with a much higher bandwidth almost 120 times higher compared with the GNU compiler. The FILL benchmark also shows signs of a significant drop with a peek at 32kb and 100Gbs to 40Gbs after the $L1$ gets filled.

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



Now in Figure.2 we change the compiler to Intel's C++ compiler and we can see significant changes on all the benchmarks. The most prominent change from the previous figure is the AXPY benchmark that peeks approximately around $40kb$ with a much higher bandwidth almost 120 times higher compared with the GNU compiler. The FILL benchmark also shows signs of a significant drop with a peek at 32kb and 100Gbs to 40Gbs after the $L1$ gets filled.

Figure 4: Weak Scaling - Efficiency (Sp/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	2.551973	1.00	1.00
2	1.307288	1.95	0.98
4	0.713981	3.57	0.89
8	0.407256	6.27	0.78
16	0.27913	9.14	0.57
32	0.303005	8.42	0.26
64	0.447266	5.71	0.09

Table 6: Weak Scaling Benchmark - Number of Particles: 2000

Threads	runtime-n2000	speed-up-n2000	efficiency-n2000
1	10.067885	1.00	1.00
2	5.085486	1.98	0.99
4	2.698663	3.73	0.93
8	1.399622	7.19	0.90
16	0.78047	12.90	0.81
32	0.555237	18.13	0.57
64	0.574328	17.53	0.27

Table 7: Weak Scaling Benchmark - Number of Particles: 4000

Threads	**runtime-n4000**	**speed-up-n4000**	**efficiency-n4000**
1	39.98665	1.00	1.00
2	20.032609	2.00	1.00
4	10.537717	3.79	0.95
8	5.324176	7.51	0.94
16	2.753726	14.52	0.91
32	1.53249	26.09	0.82
64	1.075133	37.19	0.58

Table 8: Weak Scaling Benchmark - Number of Particles: 8000

Threads	**runtime-n8000**	**speed-up-n8000**	**efficiency-n8000**
1	158.33655	1.00	1.00
2	79.404148	1.99	1.00
4	41.873768	3.78	0.95
8	20.95203	7.56	0.94
16	10.574536	14.97	0.94
32	5.472768	28.93	0.90
64	3.098319	51.10	0.80

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 9: 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)	