論文 / 著書情報
Article / Book Information

Title	N-body methods
Authors	Mustafa AbdulJabbar, Rio Yokota
Citation	High Performance Parallelism Pearls, , ,
Pub. date	2014, 11
Note	このファイルは著者（最終）版です。 This file is author (final) version.

Chapter 10

N-body Methods

Contributed by Rio Yokota and Mustafa AbdulJabbar.

Optimization techniques discussed in this chapter include the use of compiler options and directives to increase the performance with minimum modification to the code. We show that fused multiply-add and reciprocal square root operations are generated by the compiler without explicitly rewriting the code. We also show that the outer loop of a double loop can be vectorized with the use of a simple directive. We compare this directive based code with a hand-tuned code using intrinsics. We also compare the same code on the Xeon and Xeon Phi. We were able to achieve 1.5 teraFlop/sec single precision performance on the Xeon Phi without rewriting our original C code.

We use a direct N-body method to demonstrate the effect of the above mentioned optimizations. N-body methods have traditionally been used in particle-based simulations such as astrophysical [1] and molecular simulations [2]. They have recently been extended to solve more general forms of partial differential equations as well [3]. The inner kernel of the N-body method has high arithmetic intensity since it performs $20N^2$ operations for every 4N floating point numbers it loads. If we compare this with a matrix-matrix multiplication, which performs $2N^3$ operations for every $2N^2$ numbers loaded, we can see that the flop/Byte ratio of the N-body kernel is higher. This high arithmetic intensity allows N-body methods to remain compute-bound on architectures of the future. In this chapter we focus on the optimization of the N-body method on the Intel® Xeon Phi™ coprocessor, and show the effect of these optimizations on an Intel® Xeon processor as well.

Fast N-body methods and direct N-body kernels

The direct N-body kernel calculates the all-pairs interaction of N bodies against N bodies, which results in an operation count of $O(N^2)$. Fast approximation methods that use hierarchical domain decomposition of the bodies, along with truncated series expansions of the kernel can drive the operation count lower to O(NlogN) [4] or even O(N) [5]. The O(N) method is often referred to as the fast multipole method (FMM). The rationale behind these fast N-body methods is that bodies in the far-field need not be considered individually, but can be grouped into multipoles. However, bodies in the near-field must still be calculated accurately by using the direct N-body kernel. Therefore, the performance of the direct N-body kernel is critical for these fast N-body methods.

The arithmetic complexity refers to the asymptotic amount of arithmetic operations performed as the problem size increases. The arithmetic intensity refers to the amount of arithmetic operations performed per amount of data loaded. The combination of the O(N) optimal complexity of the FMM and the high arithmetic intensity of the direct N-body kernel inside it, makes the FMM an interesting alternative to many algorithms on architectures of the future. This is because conventional algorithms with high arithmetic intensity like dense linear algebra tend to have high arithmetic complexity as well. They are compute-bound and can utilize most of arithmetic units, but are very computationally expensive to begin with. On the other hand, methods with low arithmetic complexity like Fast Fourier Tranform (FFT) and sparse linear algebra have low arithmetic intensity. They are very efficient, but are not able to achieve a high percentage of the peak arithmetic performance of modern processors. FMMs have an exceptional combination of O(N) complexity and an arithmetic intensity that is higher than matrix-matrix multiplication. This means that N-body methods provide a very efficient algorithm that scales favorably with problem size, but should also remain compute bound on future architectures for the next few decades.

The direct N-body kernel is easy to isolate from the FMM, since it is usually being called as a function inside the neighbor finding routine. This function call takes up a majority of the execution time of the FMM. Therefore, if we can optimize the direct N-body kernel it will automatically optimize the hotspot of the FMM. The direct N-body kernel is only a few tens of lines of code, so even optimizing at the assembly level is not so much work. In this chapter, we will show a very specific example of how far we can go with just compiler options, and then

introduce intrinsics to squeeze even more performance out of the direct N-body kernel.

Applications of N-body methods

N-body methods can be naturally applied to problems where the physics itself is described by a collection of discrete points, even before numerical discretization. Many-body problems under gravitational or electrostatic forces is a typical example, where stars and atoms can be represented as point sources of mass and electrostatic charge, respectively. N-body methods can be extended from discrete fields to continuum fields through discretization. This makes it possible to use these methods for solving problems in structural mechanics, fluid mechanics, electromagnetics, acoustics, and even quantum mechanics. However, just because it is applicable does not mean that it is the optimal method to solve that particular problem.

Fast N-body methods have been favored in applications where the geometry information changes dynamically. If the geometry is stationary, it makes much more sense to store this information in the form of a matrix, and to perform sparse/dense linear algebra operations on this same matrix over and over again. N-body methods could be thought of as 'matrix-free' methods, where a matrix is formed on-the-fly before being multiplied to a vector of source points. It is obvious that such methods become advantageous only when the matrix/geometry changes frequently, since storing them would not save any computation in such cases. This is precisely the case for particle-based methods where each particle advances its location every time step. Adaptive mesh refinement may also result in a similar amount of geometry updates if the system is very dynamic.

There are many factors that could influence the comparative advantage of the N-body approach over other elliptic PDE solvers like FFT and multigrid. The asymptotic constant plays a critical role in determining the relative performance of these different O(N)/O(NlogN) algorithms. The FFT is known to have as few as 2NlogN [6] operations and multigrid could have as few as 5N operations. The FMM on the other hand would typically have a much larger asymptotic constant. However, the use of translational and rotational symmetry while prescribing the position of points can reduce this constant significantly.

Comparing FFT against FMM is difficult because FFT has a higher spatial resolution per unknown, so comparing for the same N is not fair.

However, for fields that have local features or discontinuities the homogenous spatial resolution of FFT certainly becomes a disadvantage [7]. Furthermore, when translational and rotational symmetry is utilized in the FMM by prescribing the position of points, it can use BLAS 3 operations much more efficiently than multigrid. Therefore, FMM becomes faster than multigrid to solve the same problem up to the same accuracy in such cases [7].

The FMM in the traditional sense requires a Green's function solution, so the type of scientific applications that it can handle is limited to those that have a Green's function. Generalization of the FMM to hierarchical low-rank approximations of matrices [8] enables the same framework to be applied to a much wider range of applications. These methods use low-rank approximation methods such as rank-revealing QR [9], truncated SVD [10], or adaptive cross approximation [11] instead of multipole expansions. This frees the FMM from its dependence on the existence of Green's functions, and can be applied to problems like variable coefficient Poisson equations or covariance matrices, which were not solvable with the original FMM.

When predicting the performance of FFT, FMM, and multigrid on future architectures, a useful indicator is communication complexity since the bottleneck of any algorithm becomes the communication as it approaches its limit of parallel scalability. FFT has a communication complexity of $O(P^{1/d})$ for a d-dimensional decomposition. Multigrid has a communication complexity of O(log P). We have recently been able to prove that FMM also has a communication complexity of O(log P) [12]. Therefore, we see that both FMM and multigrid are communication optimal. This proof for O(log P) of FMM can be extended to its algebraic variants as well [12].

Direct N-body code

We will now show an example of a direct N-body kernel, first in plain C language, and then using SIMD intrinsics. The plain N-body kernel is shown in Figure 1. We define eight arrays that describe the properties of the body. The arrays *x, y, z* are the coordinates, *m* is the mass/charge, *p* is the potential, and *ax, ay, az* are the acceleration in each direction. The equation we calculate is the smoothed Laplace potential

$$\phi_i = \sum_{j=1}^{N} \frac{m_j}{r_{ij}}$$

and acceleration

$$a_i = \nabla\phi_i = -\sum_{j=1}^{N} \frac{m_j r_{ij}}{r_{ij}^3}$$

where

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 + \varepsilon^2}$$

is the distance between the bodies at *i* and *j*, and ε is the smoothing factor. The arrays *x, y, z, m* are given prescribed values -- in the present example a random number between 0 and 1. These values are used to calculate the potential and acceleration on all *N* bodies, which is induced by all *N* bodies. This results in a double loop from 0 to *N-1* as shown in Figure 1. The loop for *i* goes over the target bodies, while the loop for *j* goes over the source bodies.

We will use the code in Figure 1 as a base case to see how much performance we can achieve on the coprocessor without changing the code. The combination of the two pragmas for SIMD and OpenMP makes it possible to parallelize the outer loop both over threads and SIMD vectors. The performance results when using different compiler options is shown in the following section. This same code runs on both the Xeon processor and the Xeon Phi coprocessor.

The coprocessor has 512-bit wide SIMD intrinsics, which are similar to Intel® SSE and Intel® AVX. Since these intrinsic instructions directly map to assembly instructions, it leaves less ambiguity in what the compiler is doing to the code. To be more specific, it tells the compiler to explicitly perform *load, store, fmadd*, and *rsqrt* operations, and also the loop to be vectorized is also explicitly specified.

```
#pragma simd
#pragma omp for
for (i=0; i<N; i++) {
  float pi = 0;
  float axi = 0;
  float ayi = 0;
  float azi = 0;
  float xi = x[i];
  float yi = y[i];
  float zi = z[i];
  for (j=0; j<N; j++) {
    float dx = x[j] - xi;
```

```
    float dy = y[j] - yi;
    float dz = z[j] - zi;
    float R2 = dx * dx + dy * dy + dz * dz + EPS2;
    float invR = 1.0f / sqrtf(R2);
    float invR3 = m[j] * invR * invR * invR;
    pi += m[j] * invR;
    axi += dx * invR3;
    ayi += dy * invR3;
    azi += dz * invR3;
  }
  p[i] = pi;
  ax[i] = axi;
  ay[i] = ayi;
  az[i] = azi;
}
```

Figure 1 Code listing – Direct N-body kernel. Targets (outer loop) are vectorized.

The direct N-body kernel has an outer loop for the targets and an inner loop for the sources. With the use of SIMD intrinsics it is possible to specify that the outer loop should be vectorized by putting 16 array elements into the SIMD registers and using a stride of 16 for the loop, as shown in Figure 2. The general structure of the direct N-body code has not changed, but all the operations are now written in *_mm512* intrinsics, and all intermediate values are declared as *__m512* registers. When possible the *fmadd* instruction is explicitly specified.

An alternative form of vectorizing the inner loop is shown in Figure 3. The code is almost identical to the one in Figure 2, except the stride of 16 is now in the *j* loop, and a *reduce_add* operation must be performed at the end instead of a simple *store*.

```
#pragma omp for
for (i=0; i<N; i+=16) {
  __m512 pi = _mm512_setzero_ps();
  __m512 axi = _mm512_setzero_ps();
  __m512 ayi = _mm512_setzero_ps();
  __m512 azi = _mm512_setzero_ps();
  __m512 xi = _mm512_load_ps(x+i);
  __m512 yi = _mm512_load_ps(y+i);
  __m512 zi = _mm512_load_ps(z+i);
  for (j=0; j<N; j++) {
```

```
    __m512 xj = _mm512_set1_ps(x[j]);
    xj = _mm512_sub_ps(xj, xi);
    __m512 yj = _mm512_set1_ps(y[j]);
    yj = _mm512_sub_ps(yj, yi);
    __m512 zj = _mm512_set1_ps(z[j]);
    zj = _mm512_sub_ps(zj, zi);
    __m512 R2 = _mm512_set1_ps(EPS2);
    R2 = _mm512_fmadd_ps(xj, xj, R2);
    R2 = _mm512_fmadd_ps(yj, yj, R2);
    R2 = _mm512_fmadd_ps(zj, zj, R2);
    __m512 mj = _mm512_set1_ps(m[j]);
    __m512 invR = _mm512_rsqrt23_ps(R2);
    mj = _mm512_mul_ps(mj, invR);
    pi = _mm512_add_ps(pi, mj);
    invR = _mm512_mul_ps(invR, invR);
    invR = _mm512_mul_ps(invR, mj);
    axi = _mm512_fmadd_ps(xj, invR, axi);
    ayi = _mm512_fmadd_ps(yj, invR, ayi);
    azi = _mm512_fmadd_ps(zj, invR, azi);
  }
  _mm512_store_ps(p+i, pi);
  _mm512_store_ps(ax+i, axi);
  _mm512_store_ps(ay+i, ayi);
  _mm512_store_ps(az+i, azi);
}
```

Figure 2 Code listing – Direct N-body kernel with intrinsics. Targets (outer loop) are vectorized

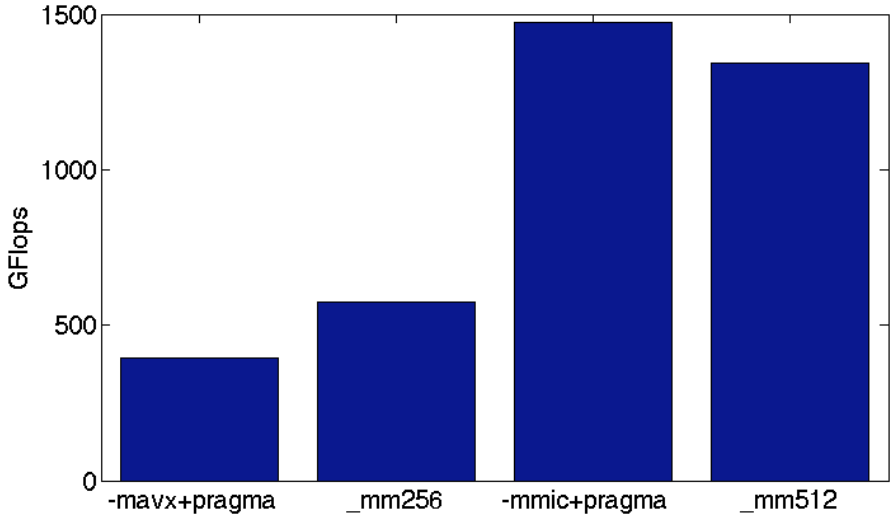
The code for AVX intrinsics is very similar to the ones shown in Figures 2 and 3 excpet the ``fmadd" are changed to separate ``mul" and ``add" operations and the ``reduce_add" operation in Figure 3 becomes a combination of `` permute2f128", ``add" and ``hadd" operations.

```
#pragma omp for
for (i=0; i<N; i++) {
  __m512 pi = _mm512_setzero_ps();
  __m512 axi = _mm512_setzero_ps();
  __m512 ayi = _mm512_setzero_ps();
  __m512 azi = _mm512_setzero_ps();
  __m512 xi = _mm512_set1_ps(x[i]);
  __m512 yi = _mm512_set1_ps(y[i]);
```

```
    __m512 zi = _mm512_set1_ps(z[i]);
    for (j=0; j<N; j+=16) {
      __m512 xj = _mm512_load_ps(x+j);
      xj = _mm512_sub_ps(xj, xi);
      __m512 yj = _mm512_load_ps(y+j);
      yj = _mm512_sub_ps(yj, yi);
      __m512 zj = _mm512_load_ps(z+j);
      zj = _mm512_sub_ps(zj, zi);
      __m512 R2 = _mm512_set1_ps(EPS2);
      R2 = _mm512_fmadd_ps(xj, xj, R2);
      R2 = _mm512_fmadd_ps(yj, yj, R2);
      R2 = _mm512_fmadd_ps(zj, zj, R2);
      __m512 mj = _mm512_load_ps(m+j);
      __m512 invR = _mm512_rsqrt23_ps(R2);
      mj = _mm512_mul_ps(mj, invR);
      pi = _mm512_add_ps(pi, mj);
      invR = _mm512_mul_ps(invR, invR);
      invR = _mm512_mul_ps(invR, mj);
      axi = _mm512_fmadd_ps(xj, invR, axi);
      ayi = _mm512_fmadd_ps(yj, invR, ayi);
      azi = _mm512_fmadd_ps(zj, invR, azi);
    }
    p[i] = _mm512_reduce_add_ps(pi);
    ax[i] = _mm512_reduce_add_ps(axi);
    ay[i] = _mm512_reduce_add_ps(ayi);
    az[i] = _mm512_reduce_add_ps(azi);
}
```

Figure 3 Code listing – Direct N-body kernel with intrinsics. Sources (inner loop) are vectorized

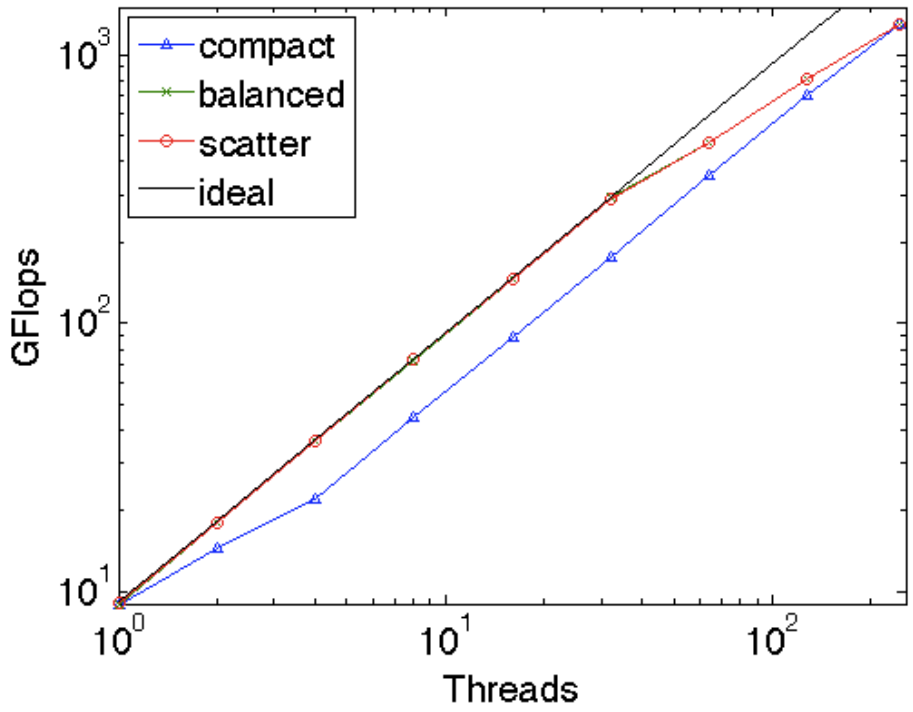
Performance results



Art File=compiler_options.pdf

Figure 4 Single precision gigaFlop/sec for the direct N-body kernel using directives and intrinsics on an Ivy Bridge processor and Xeon Phi coprocessor

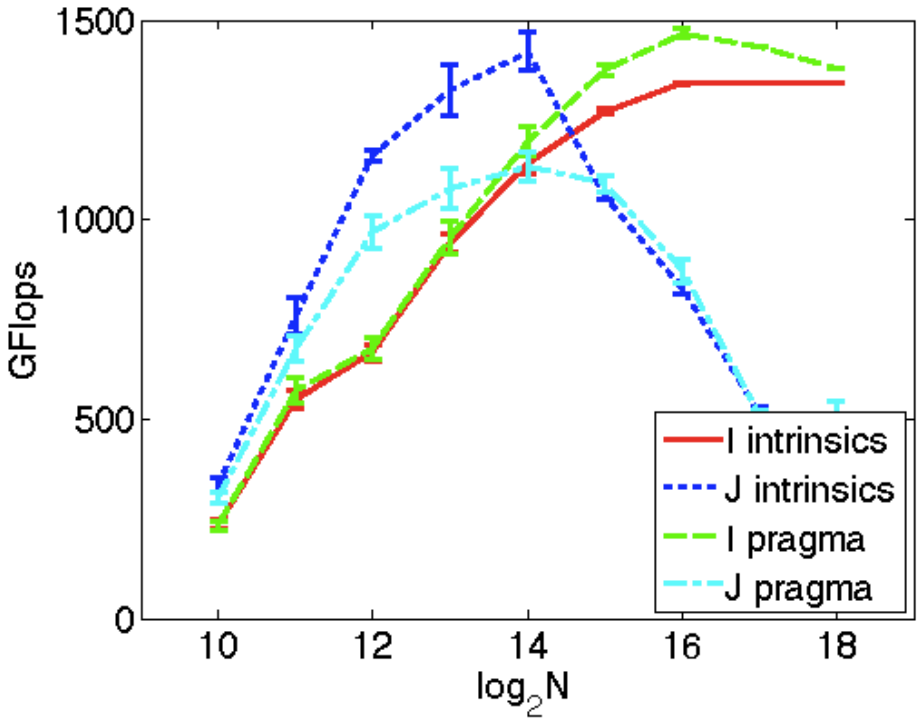
In this section, we will report the performance of the direct N-body kernel shown in Figures 1 and 2 on an Ivy Bridge processor and Xeon Phi coprocessor. The tests are performed on a Intel Xeon Phi 7120P coprocessor with Intel® MPSS 3.2.1 and Intel® C++ Composer XE version 14.0.1 in native mode, and two Intel Xeon E5-2680 v2 (Ivy Bridge) processors with Intel® C++ Composer XE version 13.0.1. The runs were performed with N=65,536, and we counted 20 Flop/sec per pair of bodies. The compiler options ``-mavx -openmp -O3" were used on the Ivy Bridge and ``-mmic -openmp -fimf-domain-exclusion=15 -O3" were used on the Xeon Phi. Figure 1 shows the comparison against Ivy Bridge and Xeon Phi with directives and with intrinsics.



Art File=strong_scaling.pdf

Figure 5 Single precision GFlops of the direct N-body kernel without intrinsics on different number of cores

We use all 40 threads on the two socket Ivy Bridge and all 244 thread on the coprocessor. The thread scalability on the coprocessor is very good, as we will show in more detail in Figure 5. The legends ``-mavx+pragma", ``_mm256", ``-mmic+pragma", ``_mm512" stand for Ivy Bridge with directives, Ivy Bridge with intrinsics, Xeon Phi with directives, Xeon Phi with intrinsics, respectively. On the Ivy Bridge processor, the intrinsics-based version outperforms the directive-based version, but on the Xeon Phi the directive-based version is faster. Both our directive-based and intrinsics-based code out perform the previous direct N-body work on the coprocessor [13].



Art File=problem_size.pdf

Figure 6 Single precision gigaFlop/sec for the direct N-body kernel with intrinsics and without intrinsics with vectorization of the outer loop for *i* or inner loop for *j* for different problem sizes.

In Figure 5, we show the strong scalability of the direct N-body kernel when different numbers of threads are used. The Intel Xeon Phi 7120P coprocessor has 61 cores running 4 threads each so the thread count goes up to 244. We test the effect of the thread affinity by changing ``KMP_AFFINITY" to *compact*, *balanced*, and *scatter*. Both the *scatter* and *balanced* options show ideal scalability up to 61 cores, while there is one thread running per core. The *compact* option scales ideally when all four threads are running on a core. This means that the scalability of the core count is very good, while the scalability among the four threads within a core are not as good. In the end, when all 244 threads are utilized the different ``KMP_AFFINITY" options do not make any difference.

So far the experiments shown in Figures 4 and 5 were for a specific problem size N=65,536. Figure 6 shows the single precision gigaFlop/sec of the direct N-body kernel for different problem sizes. In the legend, ``I intrinsics", ``J intrinsics", ``I pragma", ``J pragma", represent the target loop vectorization with intrinsics, source loop vectorization with intrinsics, target loop vectorization with directives, and source loop vectorization with directives, respectively. The error bars show the standard deviation in the gigaFlop/sec among the 256 runs that we performed for each case. We show these error bars because we noticed that some cases had very large variation in the runtime. First, we see that the source vectorized versions peak at $N=2^{14}=16,384$. With the use of SIMD intrinsics and the directive *#pragma simd* we are able to explicitly tell the compiler to vectorize the outer loop, and this results in higher performance for N>16,384. The use of SIMD intrinsics is faster for the source vectorization, but the directive-based version is faster for the target vectorized case. The optimal choice between vectorizing the outer loop or inner loop is dependent on the problem size.

Summary

In this chapter, we described how to optimize a direct N-body kernel on the coprocessor. We were able to achieve about 1.5 teraFlop/sec single precision performance by simply using the compiler option ``icc –mmic –openmp –fimf-domain-exclusion=15" and adding a ``#pragma simd" to the original C code for CPUs. The strong scalability of multi-core execution was close to ideal, while the scalability of the intra-core threading was less efficient. The choice of ``KMP_AFFINITY" did not have any effect when all 244 threads were utilized.

By using the *_mm512* intrinsics, we were able to increase the performance to about 1.4 teraFlop/sec even at smaller problem sizes. We found a strong dependence of the performance on the problem size and the relation was not monotonic. Up to a certain problem size vectorizing the inner loop gave better performance, but after N>16,384 vectorizing the outer loop gave better performance. The vectorization of the outer loop was made possible by use of a *#pragma simd* directive or *_mm512* intrinsics.

For More Information

Here are some additional reading materials we recommend related to this chapter.

- Mini N-body kernels, https://github.com/harrism/mini-nbody
- Test-driving Intel Xeon Phi Coprocessors with Basic N-body Simulation
  http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx
- Download the code from this, and other chapters, http://lotsofcores.com

References

[1] S. Aarseth, Dynamical Evolution of Clusters of Galaxies, *Monthly Notices of the Royal Astronomical Society*, Vol. 126, pp. 223—255, 1963.

[2] B. J. Alder, Phase Transition for a Hard Sphere System, *The Journal of Chemical Physics*, Vol. 27, pp. 1208—1209, 1957.

[3] S. R. Sambavaram, V. Sarin, A. Sameh, and A. Grama, Multipole-based Preconditioners for Large Sparse Linear Systems, *Parallel Computing*, Vol. 29, pp. 1261—1273, 2003.

[4] J. Barnes and P. Hut, O(NlogN) Force-Calculation Algorithm, *Nature*, Vol. 324, pp. 446—449, 1986.

[5] L. Greengard, V. Rokhlin, A Fast Algorithm for Particle Simulations, *Journal of Computational Physics*, Vol. 73, pp. 325—348, 1987.

[6] S. Johnson, M. Frigo. A Modified Split-radix FFT with Fewer Arithmetic Operations, *IEEE Transactions on Signal Processing*, Vol. 55, 111—119, 2007.

[7] A. Gholami, D. Malhotra, H. Sundar, G. Biros, FFT, FMM, or Multigrid? A Comparative Study of State-of-the-art Poisson Solvers, *SIAM Journal on Scientific Computing*, submitted.

http://users.ices.utexas.edu/~hari/files/pubs/sisc14.pdf

[8] L. Greengard, D. Gueyffier, P. G. Martinsson, V. Rokhlin, Fast Direct Solvers for Integral Equations in Complex Three-dimensional Domains, *Acta Numerica*, Vol. 18, pp. 243—275, 2009.

[9] M. Gu and S. C. Eisenstat. Efficient Algorithms for Computing a Strong Rank-revealing QR Factorization. *SIAM Journal on Scientific Computing*, Vol. 17, pp. 848—869, 1996.

[10] L. Grasedyck and W. Hackbusch. Construction and Arithmetics of H-matrices. *Computing*, Vol. 70, pp. 295—334, 2003.

[11] S. Rjasanow. Adaptive Cross Approximation of Dense Matrices. *In International Association for Boundary Element Methods*, UT Austin, TX, USA, May 28-30 2002.

[12] R. Yokota, G. Turkiyyah, D. Keyes, Communication Complexity of the Fast Multipole Method and its Algebraic Variants, *arXiv:1406.1974*, 2014.

[13] A. Vladimirov, V. Karpusenko, Test-driving Intel Xeon Phi Coprocessors with Basic N-body Simulation, Colfax International, 2013

http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx