Parallel N-Body Simulation Using Problem Space Promotion

By Brandon Farrell

Abstract

N-Body simulations predict the motion of N bodies due to gravitational force given their initial positions and velocities. Cosmologists can use an N-Body simulation to gain insight about the formation of galaxies. However, calculating net gravitational force on N bodies requires calculating on all pairs, an O(N^2) computation, and real world applications may involve millions of bodies. In a gravitation simulation, the individual forces on a body are independent and may be computed in parallel. A parallel algorithm for N-Body simulation could provide a speedup over the typical sequential algorithm, but parallel algorithms may be less efficient if they incur significant communication and synchronization costs. I have researched solving the N-Body problem using a parallel algorithm called Problem Space Promotion, which attempts to reduce communication and synchronization overhead. I have found this new approach leads to efficient parallel performance.

1. Introduction

1.1 The N-Body Problem

The N-Body problem involves using physical laws to produce the motion of N bodies given their initial state. The most common instance of the N-Body problem is modeling the behavior of celestial objects due to gravity. While no general solutions have been found for N > 3, the motion of all N bodies can be simulated by continually taking small time steps. In each time step, the force acting on a body is recalculated, considering the gravitational influence of the other bodies in the system, and its velocity and position are also updated based on this value. This approach requires considering all pairs of bodies in the system to find the total forces, and so the complexity of the simulation grows as N^2. Most applications involve a very large value of N, such that N^2 work cannot be done in a reasonable amount of time. Because of this, much work has gone into developing more efficient algorithms to solve this problem [1].

1.2 Examples

N-Body simulation is useful in cosmology, when simulating the motions of millions of bodies due to gravitation can provide insight into the evolution of star clusters and galaxies. The same techniques can be used to simulate natural phenomena on a much smaller scale. Computing the motions of large numbers of molecules due to van der Waals forces is an N-Body problem as well. Whether simulating stars or molecules, the problem is the same, perhaps only requiring different formulas for computing the influence of one body on another.

1.3 Computational Problem

Computationally the N-Body simulation consists of calculating the results of the pair-wise interactions of all bodies in the system. In classical mechanics, all the forces exerted on an object may be added together to find the net force on that object. To calculate the net force on an object due to gravity, one need only calculate the individual contributions of the other N - 1 objects in the system and sum together all

the results. To calculate the motion of each body, one could iterate over all pairs and add the result of that pair to a running total for each body. Once all pairs have been computed, the net force on a body is equal to its resulting total.

If each pair is considered, this leads to computing N^2 pairs. Since the forces two objects exert on each other are of equal magnitude but opposite direction, you can get away with only calculating N(N-1)/2. However, in a system with a number of bodies on the order of a million, this is still so large the time requirement is prohibitive.

Some simplifications can be made to keep the pair computation fast. In the N-Body gravitation simulation we tend to focus on a simple model of motion, using Newton's laws of universal gravitation and second law of motion (force equals mass times acceleration), assuming gravity to be instantaneous and ignoring general relativity. The influence of these assumptions is small and has a negligible effect on the accuracy of the simulation. This does not improve the asymptotic complexity of the algorithm, however. In order to make the running time of the simulation feasible, N-Body simulations have typically included further approximations.

1.4 Barnes-Hut method

One common technique involves approximating the effect of distant particles. In the example of gravitation, the force between two particles decreases as the square of the distance between them. This means when considering a particle it is possible to group distant particles and treat them as a single, averaged particle without losing a significant amount of accuracy in the simulation. One way of doing this, proposed by Barnes and Hut [1], involves partitioning the space into a hierarchy of regions where each region may contain smaller regions. This hierarchy is represented with a tree structure, such that leaf nodes represent a region containing one particle, and sibling nodes are nearby regions.

The Barnes-Hut method begins by considering the entire space as a single cell and dividing it into smaller, equally sized subcells, which become child nodes of the larger cell in the tree. Subcells containing no particles are ignored, subcells containing exactly one particle are added into the tree, and subcells with more than one particle are recursively divided. The tree can be constructed in this manner in O(N log N) time. At every node of the tree is a virtual particle representing all particles below it in the tree. The virtual particle is located at the center of mass of the represented particles, and has mass equal to the sum of all their masses.

In the simulation a particle is only considered against single particles in nearby regions. In further regions it is considered with nodes higher up the tree which represent an approximation of the effect of a large number of distant particles, without looking at those particles individually. The total number of interactions considered per particle is O(log N).

The tree structure must be recomputed every time step, due to the movement of particles. But since the tree can be constructed in O(N log N) time and the pair computations number O(N log N), the final running time is O(N log N), a vast improvement over $O(N^2)$.

1.5 The Parallel Computation Approach

Using these methods an N-Body simulation of considerable size can be run within a reasonable amount of time. In many cases, such as the gravitation simulation, each pair can be computed independently of any other. Due to the superposition principle in classical mechanics, only the final sum of the individual influences on each particle matters. This means we can compute pair computations in parallel, as well as the summation of forces on each particle. An efficient parallel algorithm could, when run on P processors, provide a potential factor of P speedup over the sequential implementation.

Finding an algorithm with this characteristic is difficult because of the costs of communication and synchronization in parallel programs. Ideally processors would work completely independently of each other, but occasionally they must communicate data, which costs time. These costs reduce efficiency and the benefits of parallelism versus a sequential algorithm.

A potential parallel implementation of the N-Body simulation, using P processors, might divide up the particles into P subsets, with each processor having one subset of the particles in its memory. The computation would proceed in cycles. Every cycle, each processor transmits its particles to one of the other processors and receives separate group of particles from a different processor. Each time it receives a new group it calculates the influence of each particle in that group upon all particles in its own group. After P-1 cycles all processors will have seen all N particles, and know the solutions for the N divided by P particles they own. Figure 1.5 gives a visualization of this algorithm, for four processors, with data divided into groups N_1 through N_4.

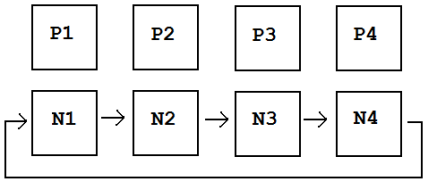

Figure 1.5 – Shifting Algorithm

This solution computes pairs simultaneously, but it does not offer a factor of P speedup. The communication and synchronization costs each cycle will cause slowdowns, making this approach inefficient. If a parallel solution does not speedup by a consistent factor for each processor, it may be faster than a sequential solution but we are not getting the benefit we should expect from using more hardware. An efficient parallel algorithm should provide the same factor of speedup for every extra processor added.

1.6 Introduction to Problem Space Promotion

I have researched how a parallel programming technique called Problem Space Promotion (PSP) can be used to construct an N-Body simulation with good parallel performance. PSP reduces the communication and synchronization problems mentioned in the previous section by increasing the dimensions of the problem. In the case of N-Body, this means increasing the list of N bodies to a two-dimensional, NxN array, in which each new row in the array is a copy of the original. The effect of doing this is the iterative loop running over all pairs has been removed. Instead, the pairs become part of the problem space.

Next the transpose of this array is created. This array contains the original list in each column. Taking each element in the NxN array and comparing it with the corresponding element in the transpose array yields all N^2 pairs. By the time the array has been promoted and its transpose created, these pairs can be accessed without any further communication. An element-wise operation takes each pair of values from the two arrays, computes a result, and stores it in the corresponding location in a third array. A final reduction step takes this array and sums the values in each column. Each of the N columns now contains a final result for the N bodies.

2. Implementation

2.1 ZPL

I implemented my N-Body solution in a language called ZPL, a language designed for parallel programming under development at the University of Washington. I will give an overview of ZPL, explain the PSP algorithm, and analyze tests of its performance. My tests have found that PSP gives a consistent factor of performance increase when the number of processors is increased.

2.1.1 ZPL Overview

ZPL is an array programming language, which means all operations are performed over arrays rather

than single values. The following line of ZPL code shows how to add two arrays together:

```
[1..N] C := A + B;
```

The first part of the statement, `[1..N]`, indicates the range of array indices to be processed by the statement. In ZPL this is known as a region. The equivalent C++ procedure would perform this using a for loop, iterating through the values 1 through N. The remainder of the statement represents the body of the for loop, adding the values of A and B at a given index and storing them in that same index in C.

ZPL gives explicit control over communication while hiding actual message passing details, and offers high performance on parallel computers [2]. It also provides the exact operations needed for the PSP algorithm: flood, remap, and reduce.

2.1.2 Regions

ZPL provides an abstraction called Regions, which are sets of array indices. By defining regions, a programmer can control which parts of an array operations affect. A region may be defined as follows:

```
region R = [1..N];
```

This region contains all array indices from 1 to the value of N. Arrays and regions can also be multidimensional, as in `region R = [1..N, 1..N]`, which is a NxN area of an array. The first column could be described as `[1..N, 1]` and the first row as `[1, 1..N]`.

2.1.3 Operations

2.1.3.1 Flood

The flood operator is a broadcast, replicating the data in an array across another dimension of the array. This is useful for the initial step of PSP, the promotion of the problem space. The N-Body problem initially involves a one dimensional list of particles. This will be transformed into a two-dimensional list in which each row contains the original array, and columns contain replicated values. This ZPL statement uses the flood operator (>>):

```
[1..N, *] A := >>[1..N, 1] B;
```

`[1..N, *]` is a two-dimensional region, with columns 1 through N, in which each row contains the same data. Each row of A will contain the first row of B.

2.1.3.2 Remap

The next step of PSP involves creating the transpose of the original array, which is done in ZPL using a remap. The remap operator (#) takes arrays containing index values. It maps the values in the current array to new indices given by the index arrays. To create the transpose array, we swap the row index and column index of the value as it goes into the new array. This can be easily done using ZPL's Index1 and Index2 constants:

```
[1..N, 1..N] At := A#[Index2, Index1];
```

Index2 is a constant array in which every value in each row is equal to the index of that row. Likewise, every value in Index1 is equal to the index of its column. By passing the remap operator the row indices for column indices and column indices for row indices, we create the transpose.

2.1.3.3 Reduce

The final step of PSP combines all the intermediate results into a final result using a reduction. In ZPL, the reduce operation (<<) takes another operator with which to reduce the given array:

```
[1..N, 1] B := +<<[1..N, 1..N] A
```

Here the columns of A are summed together so that we only have one row of results, which is stored in the first column of B. This is the computation needed to sum up forces on a particle in the gravitation simulation.

These three operations provide exactly the functionality we need to implement PSP. Additionally, ZPL provides the programmer with explicit control over communication. While some operations involve communication, other operations are guaranteed to never require it. Thus, the performance of an algorithm with respect to communication can be assessed merely by looking at its ZPL implementation. Since the goal of PSP is to confine communication to certain steps of the problem, and have others work with full parallelism, guarantees about communication performance are necessary. The three operations described above may involve communication, but the computation which derives a force vector from a pair of bodies must be fully parallel.

2.2 PSP

Next I will describe PSP, a parallel algorithm that provides efficiency when combinations of items need to be considered. For example, in the N-Body problem we need to calculate on every pair of items. I will demonstrate PSP using the problem of sorting a list of numbers.

2.2.1 Overview

Problem Space Promotion (PSP) is a parallel algorithm whose goal is to maximize parallelism in cases where combinations of items must be considered. It does this by increasing the dimensions of the problem space. For example, the algorithm might involve expanding a linear list of N items to a two-dimensional NxN array. For comparison, the sequential method of generating all pairs is shown in Figure 2.2.1.

```
for (int i = 0; i < N; ++i) {
     for (int j = i; j < N; ++j) {
          // Compute on pair i, j
     }
}
```

Figure 2.2.1 – Sequential pair generation in C++

The doubly nested for loop will generate all pairs, of which there are $N^2/2$. To generate all groups of three items we could add a third loop in a similar fashion, and in general to create all groups of size *k* we could use *k* nested loops.

Since looping is a sequential way of solving problems, we could expect a loss of efficiency if we tried to use this same method in a parallel solution. This is seen with the shifting solution described in section 1.5. PSP attempts to reduce communication and synchronization overhead by removing the loop and making it part of the problem space. When generating all pairs, the promoted NxN problem space represents the N^2 pairs; all work can now proceed on the pairs without using a loop to generate them.

PSP works in four steps, data orientation, data replication, computation, and data collapse. To illustrate each of these steps, I will use sorting as an example. Sorting is similar to N-Body in that we start

with a list of N items. PSP will create an NxN array representing all pairs, and perform the N^2 comparisons from this array in order to put the initial array in sorted order. The example list to be sorted is [5,2,9,7,1].

2.2.2 Orientation

The first step of PSP is to create a second copy of the initial array of items that is the transpose of the first array. If we envision the data, a list of N items, as an array containing one row and N columns, this mean creating a new array with N rows and 1 column, containing all the same values and in the same order. Figure 2.2.2 shows the result of the orientation step.

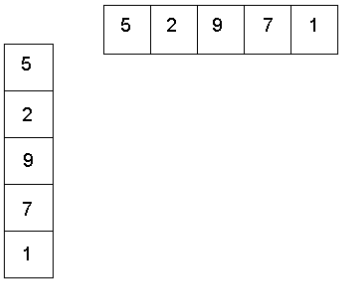

Figure 2.2.2 – Orientation: The initial array and its transpose

2.2.3 Replication

The next step is to replicate the data in the array and its transpose. The original array, with the data stored in one row with N columns, becomes an NxN array in which all N rows are copies of the original. The transpose array is replicated similarly, becoming an NxN array in which the columns are copies of the original array.

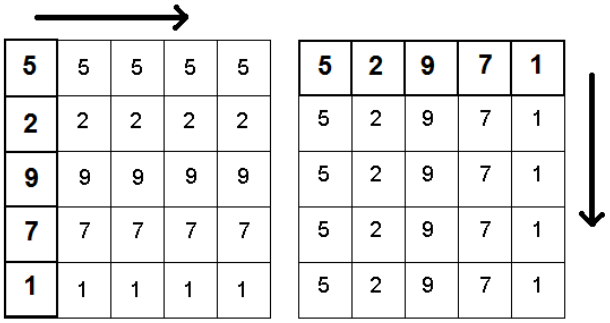

Figure 2.2.3 - Replication

2.2.4 Computation

The procedure we have seen thus far for the sorting example will be identical for the N-Body simulation. The difference comes in the operations used in the next step. Once we have the data in an NxN array, and we have the transpose of that NxN array, we can begin with the computation. For sorting, the operation to be performed on each pair is greater-than-or-equals. This returns a 1 if the first value is greater than or equal to the second value, and 0 otherwise. This will sort the array in descending order (to sort into ascending order, simply use less-than-or-equals instead). This example will assume 1-indexed arrays, but for 0-indexed arrays we just use a strictly greater-than operation. In this step, the N-Body simulation will perform an operation which takes two bodies and returns a vector representing the force of gravity on the first body from the second body.

This will result in the creation of a third NxN array, containing the result for each pair, as shown in Figure 2.2.4. Each cell in this array contains the result from computing the corresponding cells in the two

other arrays. Note how each value in this array can be computed separately from any of the others. The computation step is fully parallel, which is the advantage PSP provides.

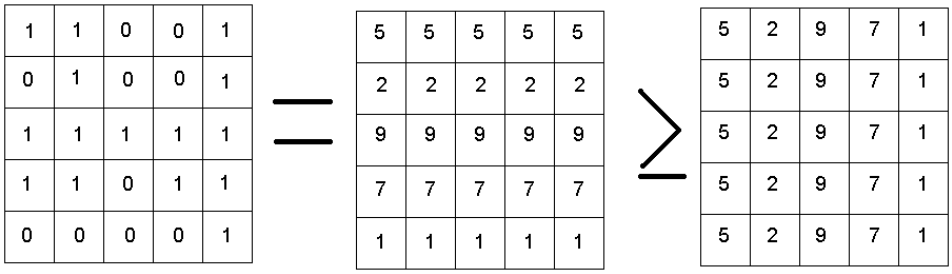

Figure 2.2.4 - Computation

2.2.5 Reduction

In the last step, we obtain the final results through a reduction on the array created by the computation step. This entails adding up the values in each column to get a single value for that column, returning to an Nx1 array. Figure 2.2.5 shows the result of this reduction. The values in this array are new indices for the values in the old array. For each value in the original array, we place it into a new array at a different index, given by its corresponding value in the result array. For example, the first value in the original array, 5, goes at index 3, the first value in the result array.

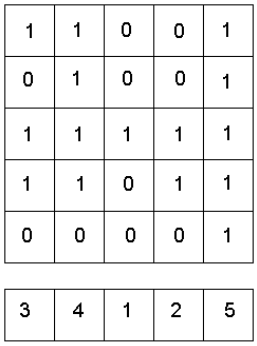

Figure 2.2.5 - Reduction

For the N-Body simulation, we will also be performing an addition operation in this step, but it will be vector addition instead of scalar addition. The vector sums will be the net force on each body in the original list. The final step will be to integrate to get the change in position and velocity of each body.

2.3 PSP Performance

PSP has been shown to offer efficient parallel performance for a variety of problems [2]. That is, each additional processor given to the algorithm gives roughly the same advantage in running time. There are a couple more issues to note about its performance.

First, the algorithm, as described in the previous section, always performs N^2 work. Comparison sort algorithms can complete this same task in O(N log N) time, which is significantly better. More generally, other all pairs algorithms may be able to stop work at some point before having to use all N^2 pairs. PSP may not be the best parallel algorithm for the job if a different algorithm appears to do it with less work. However, the communication complexity of parallel algorithms may be significantly greater than their work complexity [2]. In these cases, PSP may be more efficient.

Another apparent drawback of PSP comes from the increase in the problem space. However, this increase does not actually affect the space complexity of the algorithm. The flooding of the array into an

extra dimension is only a virtual flood; the physical storage of the array does not increase. In addition, the reduction operation can be combined with the creation of pairs, and the summation represented as a running total rather than storing individual results. This means the storage needed is only O(N).

2.4 Applying PSP to N-Body

The features of PSP are useful for the N-Body simulation, since they allow the N^2 pairs to be processed completely in parallel. To test the performance of this method, I developed my solution in ZPL, a parallel programming language described in section 2.1.

PSP has been shown to be fast on computations like the N-Body problem [2], but it could still be aided by reducing the required number of computations. In our gravitation example, most particles will be a very large distance away from any given particle. Due to the fact gravitational force is inversely proportional to squared distance, the effects of distant particles are small and can be approximated. As described in section 1.4, other N-Body simulations have used the Barnes-Hut tree to exclude distant regions from the particle to particle comparison [1]. The same can be done in the PSP solution, with the same increase in performance.

To do this effectively, we must divide the space into regions containing the same number of particles. One way to do this is to first divide the objects into two groups based on whether their x-coordinate is greater or less than the median. Next divide both groups into subgroups based on their y-coordinate, and continue dividing them, alternating axes each time, until the each group contains a small number of particles. Once this has been done, the PSP computation can proceed on each group, ignoring the particles in any other group.

Combining this space partitioning technique with PSP gives us an algorithm that reduces communication for maximum parallel performance and also prunes away small factors in order to reduce the final workload.

3. Tests

In order to test the performance of the PSP solution to the N-Body problem, I implemented it in ZPL (described in section 2.1). The code for this implementation is given in the Appendix.

3.1 Experiment Details

I tested my implementation on a Sun Fire T2000 server, described in Figure 3.1. This computer runs with 16 virtual processors on 8 physical cores.

Sun Fire T2000 Server

- 8 core 1.2GHz UltraSPARC T1 processor
- 16GB DDR2 memory (16 * 1GB DIMMs)
- 2 * 73GB 2.5" 10K rpm SAS hard disk drives
- Solaris 10 OS

Figure 3.1 – SunFire specs

I ran the ZPL implementation on a problem set containing 2048 bodies, obtaining running times for different numbers of threads, to check the factor of speedup gained per thread. I also wrote a sequential implementation in C++, which uses nested for loops to calculate all pairs and was compiled using aggressive optimizations.

3.2 Experimental Results

In Figure 3.2a, the factor of speedup over the sequential implementation is plotted on the Y-axis and number of threads used on the X-axis.

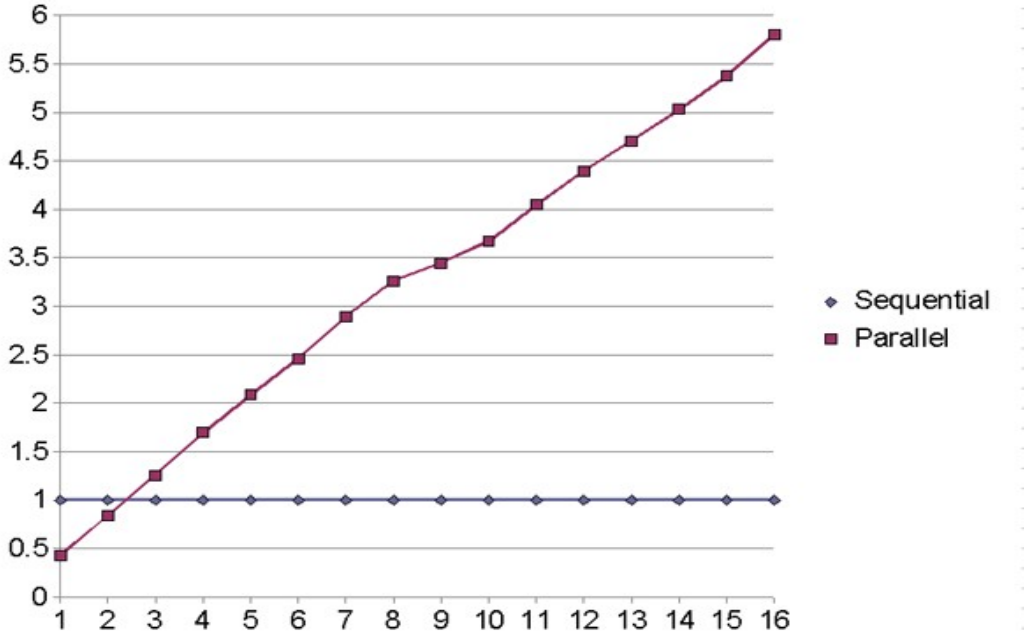

Figure 3.2a – Factor of speedup vs. Number of threads

The factor of speedup is fairly consistent with each added thread. At nine threads this starts to decline, a consequence of there only being eight physical cores on the Sun Fire computer. Based on these results, the parallel performance of the algorithm is quite strong; the speedup over the sequential program should scale well with the number of processors.

Another performance measure is parallel efficiency, which is defined as the speedup offered per number of processors. The efficiency results are given in Figure 3.2b.

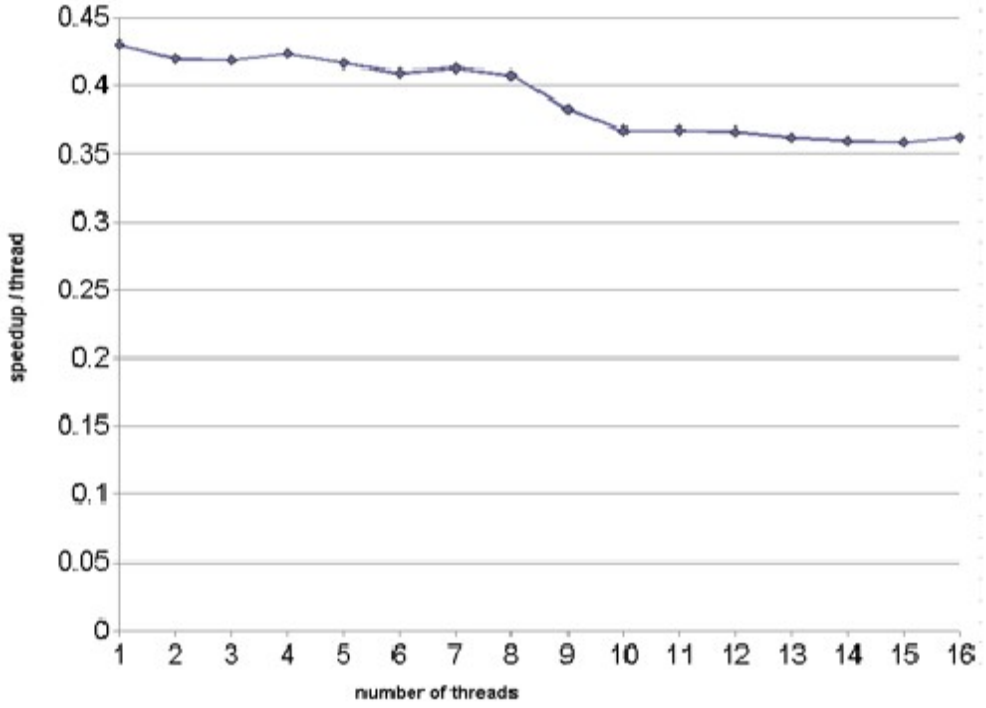

Figure 3.2b – Parallel Efficiency vs. Number of threads

The decrease in parallel efficiency is probably due to a starvation effect, where increasing the number of processors decreases the workload for each processor. Factors other than the pure computation, such as communication, begin to influence performance. Again, the drop in efficiency at nine threads is due to the existence of only 8 cores in the physical machine. Other than this, the efficiency drop off is not serious, and represents a favorable result for the PSP algorithm.

The data from these tests indicate an N-Body simulation using PSP can have very efficient parallel performance. While the algorithm does more work than necessary, it reduces communication and synchronization overhead to the point that a consistent factor of speedup per processor is achieved.

3.3 Summary

Computing the N-Body problem is difficult in practice due to the need to work with $O(N^2)$ pairs. Since the computations on the pairs are independent, the problem lends itself to a parallel solution. My experiments have shown a parallel solution using PSP has good performance that scales with the number of processors added. This allows for running an N-Body simulation involving a large number of bodies within a reasonable time frame.

4. References

1. Josh Barnes and Piet Hut. *A hierarchical O(N log N) force-calculation algorithm.* Nature 324, 446-449 (04 December 1986)

2. Bradford L. Chamberlain, E Christoper Lewis, and Lawrence Snyder. *Problem Space Promotion and Its Evaluation as a Technique for Efficient Parallel Computation*. Department of Computer Science and Engineering, University of Washington, 1999.

5. Appendix

5.1 ZPL Implementation

The following section contains the ZPL procedure used to run the PSP algorithm for the N-Body simulation. This procedure takes three arguments, a float indicating the size of the time step, and two bounds indicating which bodies are to be animated. This allows the client to only run PSP over a subsection of the array rather than the whole, enabling the space partitioning method described in section 2.4.

`universe` is an array containing all bodies in the system. The PSP algorithm (described in section 2.2) is then run, returning updated results for all bodies indexed from `min` to `max`. Comments in ZPL are preceded with two dashes.

```
procedure animate(dTime : float; min : integer; max : integer)
: [min..max, min] vector3;
var
     bodies1 : [min..max,min..max] body;
     bodies2 : [min..max,min..max] body;
     force : [min..max,min..max] vector3;
     distSq : [min..max,min..max] double;
     magnitude : [min..max,min..max] double;
```

```
    result : [min..max,min] vector3;
[min..max, min..max] begin

    -- flood
    bodies1 := >>[,min] universe;
    -- calculate transpose
    bodies2 := bodies1#[Index2,Index1];

    -- gravitation calculation
    force.x := bodies2.pos.x - bodies1.pos.x;
    force.y := bodies2.pos.y - bodies1.pos.y;
    force.z := bodies2.pos.z - bodies1.pos.z;
    distSq := force.x*force.x + force.y*force.y + force.z*force.z;

    if (distSq != 0.0) then
        magnitude := ((grav_const * bodies1.mass * bodies2.mass) / distSq);
        force.x := force.x * magnitude / sqrt(distSq) / bodies1.mass;
        force.y := force.y * magnitude / sqrt(distSq) / bodies1.mass;
        force.z := force.z * magnitude / sqrt(distSq) / bodies1.mass;
    else
        force.x := 0.0;
        force.y := 0.0;
        force.z := 0.0;
    end;

    -- sum up each factor
    [,min] begin
    result.x := +<<[min..max,min..max] force.x;
    result.y := +<<[min..max,min..max] force.y;
    result.z := +<<[min..max,min..max] force.z;

    return result;

    end;

end;
```