An Evaluation of Software Cacheing in Astrophysical n-body Simulation Codes for the Fujitsu AP1000

Tony Field    Simon Boothroyd

Department of Computing, Imperial College
180 Queens Gate, London SW7 2BZ, UK
E-mail: {ajf, sjb}@doc.ic.ac.uk

Abstract

This paper summarises recent work aimed at developing efficient message-passing solutions to irregular problems in computational science and engineering, using the Fujitsu AP1000 as the target platform. Specifically we focus here on a well-known problem in astrophysical modelling, namely the study of a many-body system in which the bodies are mutually attracted by gravitational forces. We describe an implementation of the Barnes-Hut algorithm for solving this problem which exploits software caching to provide a form of shared-memory abstraction within the context of a message-passing system. We demonstrate that this greatly simplifies the programming aspects of the problem whilst introducing very low overheads compared with hand-coded alternatives. The software caching approach is applicable to other n-body problems and problems using tree-like data structures.

1 The N-Body Problem

The N-body problem models physical domains where there is a system of N separate bodies, each of which is influenced by all the others by, for example, a force such as gravity. The idea is to study the evolution of such systems over time.

Simulations of these systems proceed by calculating the effects on each body due to the others, then updating the state of the bodies before recalculating the effects again and so on. In this way, the behaviour of the system over time is broken down into a series of discrete time steps.

A drawback of N-body problems is the amount of computation required as the number of bodies increases. To calculate the influences on each body, the interactions between it and all the other bodies must be considered. In the context of a galaxy simulation, each star or planet exerts a gravitational pull on all the others. The force on a single star is a combination of the gravitational attraction caused by all the others. As there are N bodies, and each must consider the effects of the other N – 1, the number of interactions in each time step is N * (N – 1), or O(N^2). To simulate galaxies of stars, tens of thousands of bodies are required. This becomes clearly impracticable in a reasonable time span.

Fortunately, an insight made as far back as 1687 by Isaac Newton can be used to reduce the computation needed. If the interaction force falls off rapidly with distance (e.g. as the inverse square of the distance like gravitation), the effect of a large group of particles far enough away can be approximated by a single equivalent particle with very little loss of accuracy. So, a hierarchical approach can be used, where particles close by are considered separately, but groups of particles at a distance are considered as one equivalent particle.

This is the motivating idea behind the Barnes-Hut algorithm, and others, which reduces the computational complexity to O(N log N) by recursive subdivision of the space domain into an oct-tree data structure. In this case the internal nodes of the tree represent a volume of space with an associated total mass and centre of mass, whilst the leaves store the individual bodies, or particles, each with a mass, position and velocity vector. An example in two dimensions is shown
in Figure 1.

1.1 The Barnes Hut Algorithm

The Barnes Hut algorithm [1] breaks the problem into a series of steps that are performed each time step.

1. Subdivide space and build the tree of particles.

2. Starting from the bottom of the tree, work towards the root, calculating the centres of mass of each cell. This is the “single equivalent particle” for each cell that will be used to approximate all the particles below that cell in the tree.

3. Calculate the force on each particle. This is done by starting at the root of the tree, which contains the entire system and proceeding recursively. A simple calculation determines whether to use the centre of mass of the cell (the single equivalent particle), or to consider each of the children of the cell separately (the opening criteria).

4. Update positions and velocities of particles.

1.2 Parallelisation

There is potential for very good parallel performance in N-body problems, especially in the force computation phase, which is the dominant phase computationally. The goal is to distribute the particles or cells across processors and calculate the forces on them at the same time.

Most implementations of the algorithm have been targeted at shared memory computers. This is because it is far easier to achieve low communication overheads by the above method, scheduling tasks that access the same data to the same processor. A key factor in obtaining an efficient solution is that of load balancing and here we use the so-called “costzones” partitioning and load balancing method [2] [3] which has proven to yield very impressive speed-ups on shared-memory platforms.

The costzones method of partitioning is based on the fact that the tree of particles has a correspondence with the physical distribution of particles because of the way it is constructed. The costzones method simply allocates to each processor a contiguous, equal zone of costs. In effect, the leaves of the tree are allocated evenly from left to right. How well this corresponds to physical locality depends on how well each cell’s location in the tree corresponds to its location in physical space. To achieve a good correspondence, the costzones method changes the order that the children of each cell are numbered and stored within the tree. The ordering used has been shown to be equivalent to the Peano-Hilbert ordering.

Message-passing implementations of Barnes-Hut [4] also exist, but these are based on so-called “locally essential trees” (LET). The idea is to introduce a new phase each time step, where processors determine which other processors might need cells they own and send the details to those processors. Each processor then builds its LET which is a subtree of the global tree containing all the nodes that processor will need to perform the force computations on the particles allocated to it. After each processor has done this, it can proceed and calculate the forces without any further communication.

This method has been studied and compared with the shared address space costzones method [3] It was found to be substantially more difficult to program and debug. It was also found to have serious execution time overheads due mainly to the building of the locally essential trees. There are also severe memory overheads for the locally essential trees as they tend to pre-load far more data than is required in practice.

Our goal here is to provide the benefits of the shared-memory programming model without incurring the performance overheads which can accrue from the naive simulation of shared on a
message passing machine. This enables use to code the algorithm as if for a shared-memory system, thus avoiding the need for expensive additional phases such as LET calculation.

2 Implementation Details

The algorithm we describe has been implemented on the Fujitsu AP1000 [7] at Imperial College using the MPI library [6] developed for the AP1000 by ANU. Since the AP1000 cells have no IO capability, the host-cell model is used, with the host responsible for distributing the input data, and collecting 3D output data and the final states of the particles if required.

2.1 Input

Input particles are generated randomly according to the Plummer model, which is widely used to generate spherical galaxies made up of equal mass bodies. The code for the particle generator was lifted from the SPLASH2 Barnes application, and modified for use with this simulation.

2.2 Constructing the Tree

Tree construction proceeds assuming that the particles have been distributed to the processors by the costzones load balancing scheme. Each processor begins by building its own particles into a local tree. These are then merged to form a global tree. To merge trees, a recursive algorithm is used. Starting with local cell being the root of the local tree and global the root of the global tree, the algorithm takes an appropriate action depending on the types of local cell and global cell:

1. local cell is internal, global cell is empty. The local cell is inserted into the tree.

2. local cell is internal, global cell is a leaf. The global leaf is removed from the tree, inserted into the local tree, which is then inserted into the global tree.

3. local cell is internal, global cell is internal. This means a spatial equivalent of the local cell already exists in the global tree (because the local trees all have a root node of the same size as the global tree). The algorithm is called recursively on each of the children of the local cell and their corresponding children in the global tree.

4. local cell is a leaf, global cell is empty. Same as case 1.

5. local cell is a leaf, global cell is a leaf. Same as case 2.

6. local cell is a leaf, global cell is internal. The local cell is subdivided, which pushes the particle in it one level deeper in the local tree. Case 3 now applies.

The algorithm is more efficient than a single global construction phase because large sub-trees are merged in a single operation, rather than single particles. This requires less locking, less contention and hence less communication. The property of large sub-trees being merged in a single operation depends on processors having groups of particles that have physically locality. Without this property, this algorithm would perform significantly worse than the original due to the extra work of creating the local trees and then merging them.

2.3 Calculating Masses

Calculation of the centre of mass of each cell and the square of the critical radius for each cell proceeds using a diffusing algorithm. The general operation is as follows. The root cell starts the computation (this is known as the environment in standard diffusing computation terminology). At any stage, if a node is internal a mass request message is sent to the processes that have each of its children (possibly including the local process). If a node is a leaf, then a mass reply message is sent to the originator of the request, containing the mass and position of the node. As mass reply messages are received the positions and masses are accumulated. When all replies have been received for a node, a reply is sent to the originator of the mass request for that node, containing the position and mass as before. When all replies have been received for the original root cell, the computation is over.
2.4 Calculating Forces

Force calculation proceeds with all processors calculating the forces on their particles at the same time, starting at the root of the global tree constructed previously. The opening criteria determine whether the collection of particles beneath a node on the tree can be aggregated for the purposes of the interaction with the particle in question (long-range interaction). At each stage tree nodes and leaves may have to be fetched from remote processors and to effect this transparently and efficiently, we use a form of simulated shared memory using virtual pointers and software caching.

2.5 Software Caching

Our approach creates the illusion of a form of virtual shared memory using so-called “virtual pointers” to link the nodes and leaves of the global tree. Whereas an ordinary pointer simply gives the address in memory of the object it is pointing to, a virtual pointer gives the identity of the processor that has the object in memory, and the address of the object on that processor.

The communication arising from the use of such virtual pointers consists of relatively short messages arising in an unstructured way. With increasing numbers of processes, the proportion of objects located on a remote process increases, and every access to such objects requires a message to be sent, the target process to probe for messages and respond, and the requesting process to receive the reply before computation can proceed.

Shared-address space computers address the problem of contention for the interconnection network using caches attached to the processors. A cache-coherency protocol ensures that copies of a data item in multiple caches are all consistent. This idea can be adapted when simulating shared memory and is particularly straightforward in this case because the global tree is not updated once it has been constructed. However, there are potentially large search overheads as software-cached data may have to be searched for explicitly (real caches have hardware search logic).

A solution is possible in this case by extending the virtual pointer mechanism to allow it to be redirected to locally (software) cached data using the following structure:

```c
struct virtual_pointer
{
    int processor_number;
    datatype *data;
    datatype *cache_address;
}
```

The second pointer field will contain the address of the object pointed to if it is present in the local processes cache, otherwise it will contain NULL. When a virtual pointer is dereferenced, and the processor number is remote, the cache address is checked. If it is NULL, the desired object is not in the cache and a request is issued as normal. When the reply is received, the object is stored in an area of memory reserved for use as a cache, and the virtual pointer is modified to set the cache address to point to the objects location. If the object is accessed again, this must be via the same virtual pointer (by the assumption that each object has only one pointer pointing to it). The cache address will not be NULL in this case, and can be returned without any messages being exchanged.

Virtual pointers are dereferenced by using a simple test. If the identity of the processor in the virtual pointer matches that of the dereferencing processor, the address can be used as normal. If not, a request message must be sent to the relevant processor. Figure 2 shows an example of the software cache in use.

The software caching scheme requires processors to respond to incoming requests at all times, even when awaiting a reply, otherwise deadlock could result. This is awkward to code using MPI but can be done by sprinkling message checking code (implemented using using MPI_Iprobe) throughout each node process. Some form of active messaging would have greatly simplified the programming effort at this point.

We remark that global synchronisation between phases of the algorithm cannot be achieved using normal barriers as this would result in deadlock. Instead, a new synchronisation message type is used. Each process has a counter which is initially set to the number of cells. Every time a synchronisation message is received, this counter is decremented. When
Figure 2: Software cache example.

a process wishes to synchronise it sends a synchronisation message to every other cell, then repeatedly receives and processes messages until the synchronisation counter reaches 0. The counter contains, in effect, the number of other cells that have not performed a synchronisation. To achieve complete synchronisation, a conventional barrier operation is then performed. This is to eliminate potential problems caused by processes resuming computation at different times due to the communication times of the synchronisation messages.

2.6 Cache Replacement

Eventually the cache area may become full and an existing object in the cache must be displaced in order to place a new object.

Cache replacement using a software cache is complicated by the fact that removing an object from the cache may result in pointers to other objects in the cache becoming lost; these descendant objects would also have to be removed from the cache. Since the data structure involved is a tree, the problem of multiple replacements can be solved by only replacing nodes at the bottom level of the cache. A list is therefore maintained of the nodes that are at the bottom level of the cache. Two events could change the contents of this list:

- When a new node enters the cache, its parent is removed from the list.
- When a node is replaced, it is removed from the list. Its parent may be added to the list if the parent has no other children in the cache.

Figure 3 shows the “bottom” list for the software cache example in Figure 2.

Removing a node from the cache consists of:

- Setting the cache address of the virtual pointer that points to the node to be removed to NULL.
- Removing the node from the bottom of cache list (see above).
- Adding the parent of the node to the bottom of cache list if it has no other children in the cache and is in the cache itself.

This is shown in Figure 4.

Any node that is in the “bottom list” is a potential candidate for removal. The method used to select which one to replace is to replace the first item on the list. This corresponds to the
first eligible node to have been cached. If the “bottom list” is empty, the only node at the bottom level of the cache must be the parent of the node that is about to be cached. This means the cache is not large enough to contain a path though the tree from root to leaf. In this case the contents of the cache are preserved, and operations proceed as if the cache were not active.

In order to make cache management possible, it is necessary to know the parent of a node that is entering the cache, and also the parents of nodes in the cache, and the location of the relevant virtual pointers within these parents. This is required to remove a node from the cache, and to add and remove nodes from the “bottom list” described above. To achieve this, virtual pointers have an additional field containing a pointer to the tree node that contains the virtual pointer. This is used to remove parents of incoming nodes from the “bottom list”. If the tree node that contains the virtual pointer is not in the cache (i.e. it is a local node) then this field will be NULL. Tree nodes themselves contain a field `cache_parent`, which gives the location of the virtual pointer that points to that tree node. This is used when replacing nodes from the cache.

2.7 Load Balancing and Redistribution

The list of particles for a processor consists of a list of structures of type `vp_list` which is defined as follows:

```c
typedef struct vp_list {
    virtual_pointer vp;
    vector accel;
    int load;
    struct vp_list *next;
} vp_list;
```

`vp` is a virtual pointer pointing to the particle, `accel` and `load` are used during force computation to store the partially accumulated gravitational field and cost of the particle, and `next` is a pointer to the next object in the list.

After loadbalancing, each processor has a list of particles which it is responsible for. These particles could be located on any processor. Redistributing particles (that is moving the particles so that the data for the particles each processor is looking after is located on that processor) allows tree building to be done by merging locally constructed trees as described.

To redistribute particles, each processor steps through the list of particles generated by the loadbalance phase. If a particle is located on a different processor, an exchange request message is sent to the owning processor. Exchange request messages are identical to normal request messages used in the virtual pointer mechanism, but upon receiving an exchange request message, a processor deletes the particle from memory before replying. When the requesting processor receives the reply, it allocates memory for the incoming particle and stores it. This causes the particle to move from one processor to the other.

2.8 Output

The output is recorded as a series of frames on the DDV (Distributed Disk and Video) system [8] which can be replayed in real time in the form of an animation.
3 Performance

We have performed a number of experiments on the AP1000 in order to evaluate the performance of the software caching scheme. The problem was shown to scale as $N \log N$ as expected with approximately 86% of the total execution time being consumed by the force calculation phase. Approximately 5% of the time was spent equally performing mass calculation, load balancing, tree construction, and particle update. The redistribution time was negligible in comparison.

3.1 Software Cache Performance

We first measured the cache hit rate as the number of particles per processor increased. For anything but very small problem sizes the hit rates were very high - typically 99% as the number of particles per node increases beyond $10^5$.

We next measured the effect of the cache size on the execution time. Figure 5 shows the results for 32 processors and $N=8192$; for a sufficiently large cache the processor is essentially constructing and storing its locally essential tree, but here it is doing so dynamically rather than by a separate computation phase as described in [5].

Finally we report on the effect of the cache size on the cache hit rate, and the problem size on the cache occupancy. These are summarised in Figures 6 and 7.

Note that the cache is used during the force computation phase, which is the dominant phase computationally, and has a critical effect on performance. Performance degrades very quickly with cache sizes under approximately 600 nodes. This corresponds to the working set size for this problem. The average amount of cache used per processor if no limit is imposed on its size by this problem is approximately 1500 nodes. The cache occupancy figures were produced by varying the size of the problem on 32 processors.

4 Summary and Conclusions

We have described the implementation and evaluation of an efficient parallel algorithm to perform N-body simulation using the message passing paradigm. The basic algorithm is a version of the Barnes-Hut algorithm, one variant from a family of algorithms known as hierarchical N-body algorithms. The algorithm designed is based around the concept of virtual pointers, which have been suggested in the literature, but generally rejected because of the poor perfor-
mance due to large communication overheads.

The key concept proposed and developed in this project is the software cache scheme. Again, software cache has been suggested before, but suffers from a large search overhead and coherency problems. These have been overcome by using a direct indexing system and avoiding using the cache during update phases. Using virtual pointers enabled the costzones load-balancing algorithm to be implemented straightforwardly and with high efficiency.

The algorithm was implemented in C using the MPI message passing standard, which makes it versatile and portable. It has been run on several platforms, including the original design target Fujitsu AP1000, and networks of Sun workstations and PCs running Linux. Performance on workstation networks is generally fairly poor as accurate load balancing is impossible and communication costs are very high.

The software cache virtual pointer system developed here is applicable to other problems, particularly those that rely on tree structures, such as many search-based methods. We look forward to exploring other applications of this kind in the near future.

Bibliography


