

Parallel, Out-of-core methods for N-body Simulation *

John Salmon[†]

Michael S. Warren[‡]

Abstract

Hierarchical treecodes have, to a large extent, converted the compute-bound N-body problem into a memory-bound problem. The large ratio of DRAM to disk pricing suggests use of out-of-core techniques to overcome memory capacity limitations. We will describe a parallel, out-of-core treecode library, targeted at machines with independent secondary storage associated with each processor. Borrowing the space-filling curve techniques from our in-core library, and “manually” paging, results in excellent spatial and temporal locality and very good performance.

1 Motivation

N-body methods are used in the numerical simulation of systems ranging from the atomic to the cosmological. In addition, the mathematical techniques developed in conjunction with the N-body problem have found application in areas as diverse as electromagnetic scattering and stochastic process generation. The papers collected in this mini-symposium [4], and its predecessor [1] offer ample evidence of the breadth and importance of N-body methods.

A family of methods, collectively called “treecodes”, use tree data structures to reduce the time required to approximately evaluate a set of interactions of the form:

$$(1) \quad a_i = \sum_j F(r_i, r_j),$$

where the pairwise interaction F , and the coordinates r_i and r_j are generalizations of the familiar

force laws from Newtonian mechanics. Directly evaluating (1) for N right-hand-sides, with N terms in each summation requires $O(N^2)$ operations. Treecodes typically produce approximate results in $O(N)$, $O(N \log N)$ or $O(N^{1+\epsilon})$ operations, depending on the particular algorithm, and on exactly what is being held constant as N increases. Storage is usually proportional to N .

Very roughly speaking, typical astrophysics applications require about $80N$ bytes of storage and $30000N$ floating point operations per integration timestep. A ten-million body simulation is nearly state-of-the-art, and few (if any) simulations have been done with more than 50 million bodies. The problem is not lack of CPU cycles – a 200MHz Pentium Pro processor can achieve the cycle count in a couple of months, and a small cluster can reduce the time to a few days. The problem is that memory is too expensive, so that systems with 1-10GB of DRAM are still quite rare, even though readily available CPU technology would allow important work to be done on such a system.

As of October 1996, commodity, mass-market memory prices were about \$6/MB, while disk prices were \$0.1/MB, almost two orders of magnitude lower. On the other hand, obtainable data rates from disk are in the range of a few MB/s, approximately two orders of magnitude less than from memory. Even worse, the latency for a typical disk access is five orders of magnitude greater than that for a memory reference (10 million ns vs. 60ns). Using disk as dynamic storage will be a challenge, but one that offers the opportunity to greatly reduce the hardware investment required for very large N-body simulation.

We have implemented out-of-core adaptive oct-trees because of our extensive prior experience with their numerical and computational behavior [6]. Our traversal differs substantially from [2, 3], allowing for a flexible criterion that decides whether a mul-

*This paper will appear in the Proceedings of the Eighth Conf. on Parallel Processing for Scientific Computing, SIAM, 1997.

[†]Center for Advanced Computing Research, California Institute of Technology and the NSF Center for Research in Parallel Computing. johns@cacr.caltech.edu

[‡]Theoretical Astrophysics, Los Alamos National Laboratory. mswarren@lanl.gov

tipole is acceptable based on an error estimate that includes both geometry and the contents of the cell. Groups of bodies are tested for acceptability all at once, and if the multipole is unacceptable, we dynamically decide whether to shrink the group or visit the daughter cells of the moment. This we are able to capture the essential features of $O(N)$, $O(N \log N)$, $O(N^{1+\epsilon})$, $O(N^2)$ and “vectorizing” algorithms all within the same implementation.

2 Latency Tolerance and Bandwidth Economy

The first step is to formulate the algorithms and data structures in a way suitable for out-of-core evaluation. We are used to thinking in terms of floating point operations: counting them, pipelining them, finding ways to eliminate them altogether. The validity of this mindset is questionable in the context of modern computer systems where uncached memory accesses can be an order of magnitude more expensive than floating point operations. It could be disastrous in an out-of-core code where one million “extra” flops might be no more expensive than a single disk access. We must turn our attention primarily to concerns related to data movement, with operation count considerably less important.

Two concerns drive the design of out-of-core algorithms. Latency tolerance and bandwidth economy. The first implies that whenever a datum is required from the disk, we must be prepared to wait hundreds of thousands of cycles for it to be delivered. The second says that whenever we have gone to the trouble of moving a datum from disk to memory, we must use it enough times to amortize the cost of having done so.

We have chosen a simple approach to latency tolerance. We simply ensure that all transfers to and from the disk are sufficiently large that latency is irrelevant. The conventional formula for the time, t , to service a request is:

$$t = t_l + s/b,$$

where t_l is the latency (10ms for a typical commodity disk), b is the bandwidth (1.5MB/s for the same disk) and s is the size of the request. If we ensure that all transfers satisfy $s \gg t_l b$, (15kB for our typical case), then latency will not be a dominant factor in the overall performance. Of course, we have

now made the bandwidth problem somewhat harder. We must design an algorithm that makes optimal use of the available bandwidth but which is restricted to transferring data in chunks no smaller than 15kB. Multi-threading is an alternative approach to latency hiding that allows the processor to do something else while waiting for requests to be processed. Multi-threading is essential if it is not practical to make large requests as we propose to do, but effective multi-threading often relies on special hardware features, and we have not explored its use in treecodes.

To address the bandwidth problem, we must design our data structures and algorithms so that whenever a datum is moved from a disk to memory, we get the maximum possible use of it before returning it to the disk or disposing of it. We must also minimize unnecessary copies, i.e., situations in which data that has not been modified in memory is copied back to disk anyway because the system wasn’t sure that it was safe, or because “dirty” and “clean” data are interspersed in a single page.

In summary, the following rules guide the implementation our out-of-core treecode:

- All disk transfers are in large (\gg 15kB) blocks.
- Every byte transferred must be used many times.
- The total number of transfers must be minimized.

3 The Page Abstraction

In order to use disk for dynamic storage, we must first devise a mechanism for referring to the data that resides on disk, and for transferring that data to and from memory. One option is simply to use virtual memory. It is a simple matter to allocate program space, e.g., with `malloc`, that far exceeds the capacity of memory. The OS is then responsible for paging data to and from a “swap device”, typically a disk partition dedicated to this task. Unfortunately, the programmer has essentially no control over selection of which pages to move, or when to move them. Policies implemented by the OS have usually been designed for large multi-tasking, multi-user systems with load characteristics and requirements far different from our treecode. In addition, the OS generally lacks crucial information that could

guide a better strategy. Unless page-hits are recorded by the hardware, or information is solicited from the program itself, the OS does not know which pages are heavily used and which ones are idle, making it impossible to make an intelligent choice about which pages to swap out. The `madvise` function available on some operating systems is an attempt to address this issue, but it is not universally supported, and even where it is supported, the advice may still be ignored or misunderstood.

The standard POSIX interface provides several calls which can be used to transfer data to and from disk. For example, `read`, and `write` explicitly request data transfers. Alternatively, `mmap` and `munmap` create and destroy correspondences between addressable memory and data on disk. Whichever method we chose, the essential problem remains: devise an abstraction that strikes the right balance between clarity of expression and performance. We want to hide the details of reading and writing from disk, but control the swapping policy so that performance doesn't suffer.

We treat our disk-store as a sequence of fixed-size, sequentially numbered pages. In memory, we maintain a "working set" of copies of some of the disk-pages. The size of the working set and the size of the individual pages (a multiple of the underlying OS page size) are determined during runtime initialization. To refer to data in a page, we use `PgRef`, which returns a pointer to an in-memory copy of the page. The memory at that address will not be replaced by another page from disk until `PgUnref` is called, and even then, it will be kept as long as possible. `PgRefs` accumulate, so a page remains "locked" in memory as long as the number of `PgRefs` exceeds the number of `PgUnrefs`. Much of the data access in our program is read-only, so we wish to avoid unnecessary copies back to disk. `PgUnref` takes an argument which states whether the caller has dirtied the page by modifying any data. Dirty bits from multiple `PgUnref` calls are OR-ed together. Before a dirty page is replaced in memory, its contents must be written back to disk.

We use an *ad hoc*, but effective least-recently-used policy to select pages for replacement. Every time `PgRef` is called, it increments a global counter which acts as a timestamp, and places the value in a structure associated with the page being referenced.

When we need to replace a page, we examine all unreferenced paged, and choose the one with the oldest timestamp. To reduce the number of writes, we preferentially replace clean pages by dividing their age in half before comparison.

This defines an interface to page-sized units of storage. But our program works with `bodies`, `sib_groups`, `moments` and so forth (see Section 4), none of which are the same size, and all of which are far smaller than an entire page. Thus, we introduce a second layer, of "out-of-core pointers", `oocptrs`, which are implemented as a structure containing a page number and an offset. We use `OOCRef` to return a true pointer that corresponds to an `oocptr`, and `OOCUnRef` to signal that we are done with an `oocptr`. `Oocptrs` appear in several places in the library where an in-core implementation might use normal pointers. Explicit page numbers and `PgRefs`, on the other hand, are generally hidden behind the `oocptr` abstraction.

4 Data Structures

The fundamental data structure in a tree code is, of course, the tree, and the fundamental operation is a tree traversal. A two-dimensional version of the basic data structure is shown in Figure 1. In three dimensions, space is divided into cubical `sib_groups`, each of which may contain some terminal `bodies` as well as up eight `moment` structures - i.e., a data structure that contains a pointer to a daughter `sib_group`, as well as numerical quantities like mass, center-of-mass, and higher moments of the distribution within the region. In an in-core version of the algorithm, the arrows in Figure 1 would correspond to normal "C" pointers, but in the out-of-core implementation, they are implemented as `oocptrs`. The depth of the tree is irregular and adapts to local variations in the density of `bodies`. A `moment` is only created when the number of `bodies` in its region of space exceeds some threshold, m . If there are insufficient `bodies` in the region, then the `bodies` become terminal members of the surrounding `sib_group`.

The basic traversal algorithm is shown schematically in Figure 2. When a `moment` is 'visited', a numerical criterion is applied to determine whether to use the `moment` data in the force calculation or

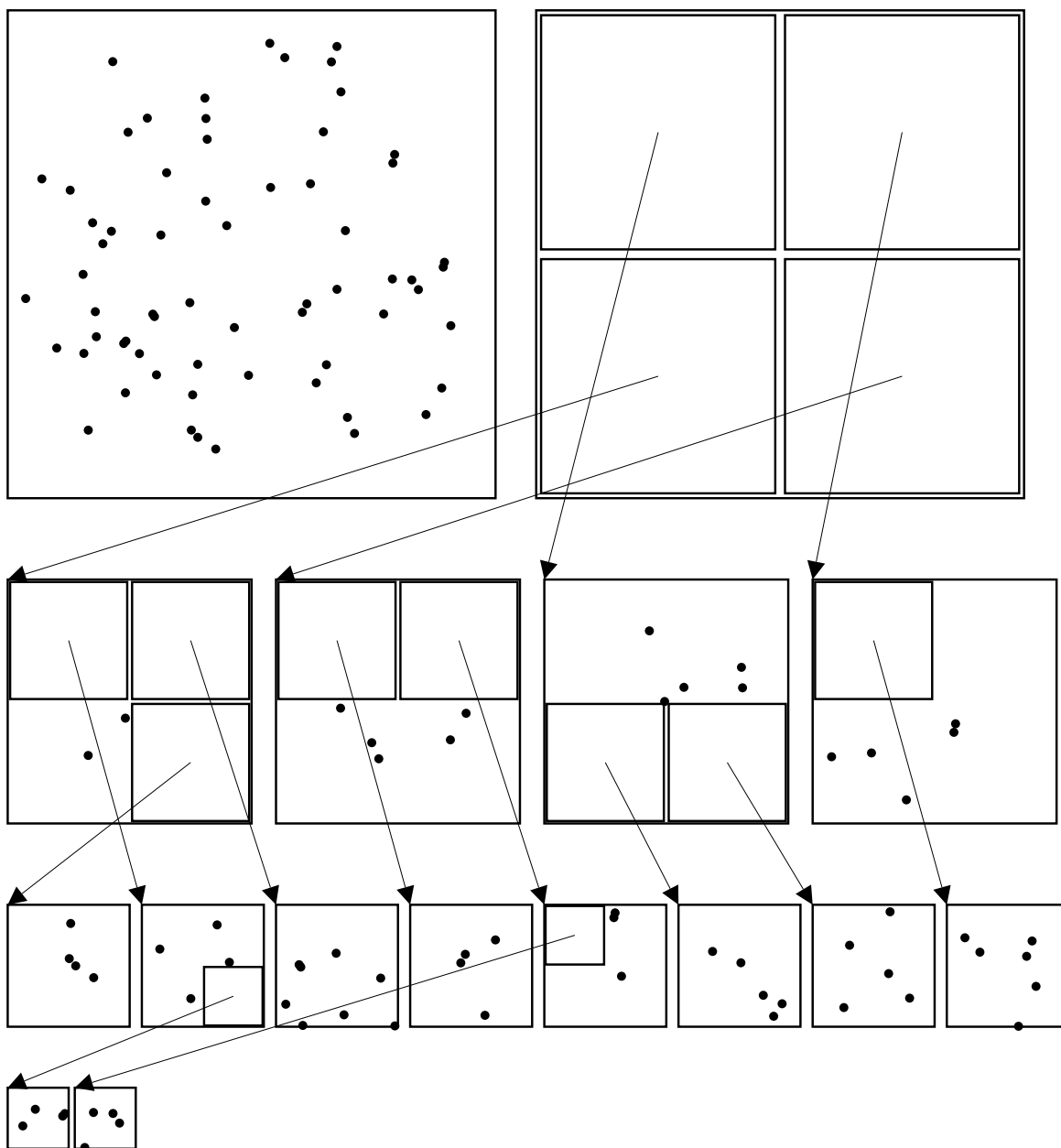


FIG. 1. The elements of the tree data structure are shown for a two-dimensional quad-tree with 64 bodies, and a terminal threshold $m = 3$. The upper left square shows just the bodies without any imposed tree structure. The other squares are *sib_groups*, which contain both moment data (represented as an inset square) and individual bodies, stored together in memory. In addition to cumulative information about the contents the cell, the moment structure also contains a pointer to a daughter *sib_group* where more detailed information may be found. These are shown as arrows in the figure and are implemented as `ooceptors` in the parallel out-of-core code.

```

Traverse(moment m){
  if( !VisitAndTest(m) ){
    sg = OOCRef(DaughterOf(m));
    VisitTerminals(group);
    for(each moment, mm, in sg){
      Traverse(mm);
    }
    OOCUnRef(DaughterOf(m));
  }
}

```

FIG. 2. *Pseudocode representation of the basic tree-traversal algorithm.*

to traverse the deeper, finer-grained levels of the tree. `Traverse` is called repeatedly by a larger loop that orders the force evaluations and allows the programmer to use `VisitAndTest` routines that consider groups of particles together, possibly implementing a “local expansion”, or deferring force evaluation to a vectorizable loop that is executed once per particle. Notice that `Traverse` visits every member of a `sib_group` whenever it visits one of them. Therefore, by grouping all the moments in a `sib_group` together, we enhance fine-grain data locality, which in-turn improves our cache-hit rate and overall performance. This is helpful to an in-core implementation, but it is particularly important to an out-of-core implementation because it allows us to amortize the cost of the `OOCCRef` and `OOCCUnRef` over the fairly substantial calculations implied by `VisitAndTest` and `VisitTerminals`.

Data locality is further enhanced by the order in which forces are evaluated on bodies. We choose the order in which we compute forces to maximize re-use of `sib_groups` in the tree. If the cache is large enough to hold all the `sib_groups` that contribute to the force on a body, and we then compute the force on a nearby body, we will find almost all the information necessary for the second body already in cache. We have found that by ordering particles along a self-similar, space-filling curve, like the Peano-Hilbert curve shown in Figure 3, we achieve excellent spatial locality and cache hit rates [6].

Again, strategies that favor cache reuse in an in-core implementation are crucial to an out-of-core implementation. Even a very modest working-set

of a few hundred pages is large enough to store the interaction set of a typical body. So the device of ordering evaluations so that sequential bodies substantially share interaction sets leads to very high reuse rates for the working set and correspondingly low swap rates in the out-of-core case.

The placement of `sib_groups` within pages is not an issue for an in-core implementation because there is no reason not to use `m_malloc` to obtain space whenever a new `sib_group` is created. Since the `sib_group` itself is comparable in size to a typical cache-line, there is not much benefit in carefully arranging them with respect to one another. Many `sib_groups`, however, fit on a single out-of-core page, and we must ensure re-use not only of the `sib_group` that has been `OOCCRefed`, but also the others that happen to reside on the same page. Thus, pages should contain `sib_groups` that are likely to be used together. We achieve this by storing `sib_groups` at the same level in the tree together, and ordering them along the same space-filling curve used to order force evaluations. Thus, when a new page is swapped in to satisfy an `OOCCRef` to a particular `sib_group`, the other `sib_groups` on the page are nearby in space and they will probably be accessed shortly, at which time they will already be in memory, and subsequent `OOCCRefs` will impose minimal overhead.

Figure 3 illustrates the spatial clustering achieved by ordering bodies along the Peano-Hilbert curve. Inspection of Figure 3 (in color) reveals that in the vicinity of any given point in space, there are no more than four colors represented (eight in three dimensions). The same spatial locality properties will hold for `sib_groups` at every level of the tree. This suggests that we will need a working set sufficiently large to hold, in the worst-case, eight pages at each level of the tree. Even for systems with many millions of particles, the tree rarely exceeds a depth of 20. so we expect that only 160 or so pages devoted to our tree data structure should be an adequate working set. Results in Section 7 confirm this hypothesis.

5 Tree building and Sorting

Creating a tree of `sib_groups` from a random set of particles, and arranging that the `sib_groups` at

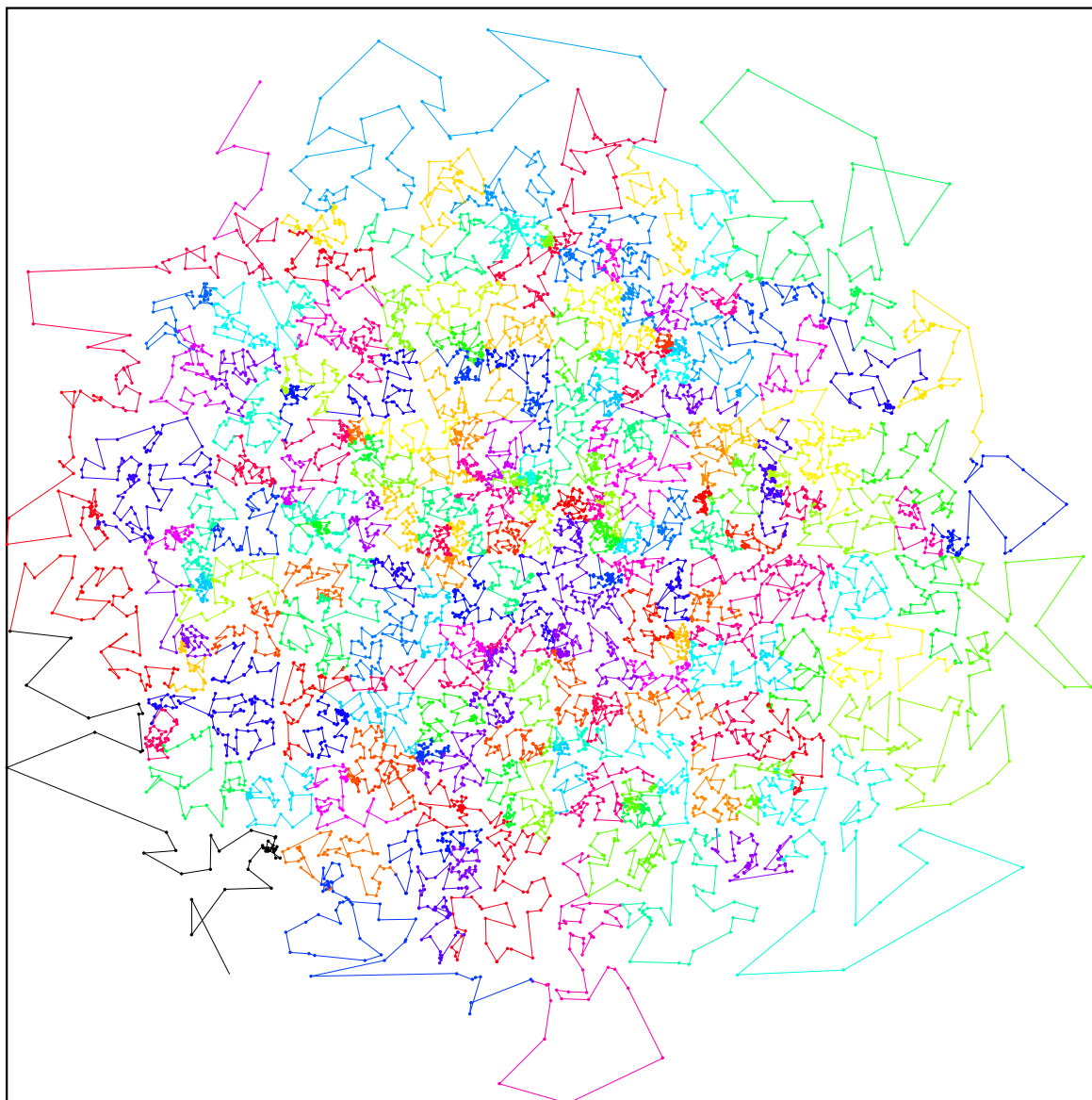


FIG. 3. A Peano-Hilbert curve induced ordering of 10000 bodies. Bodies are assigned the same color in groups of 50 at a time, indicating how they would be grouped on pages in out-of-core storage.

each level are themselves ordered along a space-filling curve is our next problem. Fortunately, both tree-building and the organization of `sib_groups` within pages is easily accomplished if the bodies themselves are first sorted along the curve. It is complicated, but not time-consuming, to compute a key corresponding to path-length along the Peano-Hilbert curve by interleaving the bits from an integer representation of the spatial coordinates in each dimension. The sorted list of bodies may then be examined sequentially and `sib_groups` in the tree may be allocated and completed in the desired order with no out-of-core pointer-chasing or backtracking. As new bodies are examined, a “current” group is maintained in normal memory. (The storage required is only proportional to the depth of the tree, not the number of bodies). If the new body is outside the current group, we are certain to have seen every member of the current group, so appropriate computations to fill in the moment data structures may be executed, and out-of-core space can be reserved at the appropriate level to hold a `sib_group` corresponding to the current group. Parents of the current group are examined and completed until the new body falls within the new current group. Now, the new body is either added directly (if the number of bodies already contained is less than m) to the current group, or a new sub-group is created and the bodies already in the group are moved to the sub-group, which becomes the current group.

Thus, we have reduced the problem of out-of-core tree-building to the problem of out-of-core sorting. We use a bucket-sort, in which we first sort into buckets which are themselves many pages in length, but small enough to fit in memory. Then the buckets are paged back in, and an in-core quicksort algorithm is applied to each bucket. Rather than implement sorting as a separate, atomic, stand-alone procedure, we integrate the “bucketizing” phase into the position-update from the “previous” timestep. Thus, when we first learn the new position of a particle, we immediately select which bucket it will go in, and place it there directly, rather than first copying it to disk, and then reading it back in from disk only to select a bucket and ship it back out.

There appears to be considerable tension between optimization for out-of-core performance (and perhaps to a lesser extent, cache performance) and

generally accepted standards of good programming practice. Good program design stresses modularity and encapsulation with each software component having a clear, limited interface and behavior. Applied naively, these ideas can lead to many more page-swaps (or cache misses) than necessary. For example, a standard high-level program design for an N-body code would probably separate sorting, tree-building, force computation, diagnostic accumulation and time integration into separate, independent modules. Each of these modules requires a complete scan of the list of particles, so the entire data set would be paged in and out many times. To recover performance, one combines the time integration, diagnostic accumulation and force computation together in a single pass. More dramatically, the sort is split into two phases, as described above. This leaves only two cycles of paging, which may result in a threefold speedup if the program is bandwidth limited. But it also results in a code where modularity is far less apparent. Sorting, in particular, no longer exists as a clear, separable component so it is more difficult to apply the results of other research efforts in parallel sorting.

6 Parallelism

Interprocessor communication in a parallel system has performance characteristics similar to communication between disk and memory. Large latencies favor long messages and interprocessor bandwidth is typically far lower than internal memory bandwidth. If an implementation is capable of tolerating latencies and bandwidths to local disk, it is likely that it can also tolerate latencies and bandwidths to remote processors.

In fact, we have implemented a parallel out-of-core treecode primarily through a small extension of the uni-processor paging abstraction. In addition to a page number, the identifier that is passed to, e.g., `PgRef`, also contains a processor number. The implementation of `PgRef` et al distinguishes between local pages, which are accessed with `read` and `write`, and remote pages, which are accessed via message exchange with the a “server” on the remote processor. There is no problem with coherence because once the tree is built (a purely local operation involving writing data), all further access is

read-only. We must be certain to purge all remote pages between timesteps, though. The “server” is simply an occasional poll for page requests during force evaluation. It is remarkable that this simple, blocking, synchronous approach is successful. At the outset, we assumed multiple servers would have to run asynchronously in separate threads but performance is entirely adequate without such sophisticated infrastructure.

With this minor extension to the underlying paging system, the implementation of the basic tree traversal, force evaluation and time integration algorithms was completely unchanged. Some additional code was added to the tree-build phase to allow processors to build local trees independently, and then merge those `sib_groups` whose spatial domain extends over multiple processors. There is also a requirement for parallel data decomposition code which assigns processors to regions of space and moves bodies appropriately, but both of these are essentially identical to the equivalent components in in-core parallel codes.

7 Results

We have implemented the ideas above in portable ANSI C. Interprocessor communication is through a simple API of our own design which, in turn, may directly use Unix sockets, or many of the open or vendor-specific communication libraries, e.g., MPI, NX, etc. The results here are for a cluster of 16 200MHz Pentium Pro systems running Linux each with 128MB of memory and a 1GB EIDE disk partition devoted to out-of-core scratch space. The peak performance of this system is 200Mflops/processor, but in practice, a highly optimized vector of in-cache, gravitational interactions runs at approximately 86Mflop/processor. The communication system is very modest – consisting of a single 100baseT ethernet switch, delivering a bi-directional bisection bandwidth of 80MB/s and latency of $150\mu s$ to user-level code. The entire system is constructed from mass-market commodity parts and is extremely cost-effective, with a total purchase price in autumn 1996 of under \$60000.

All benchmarks reported here are for uniformly distributed collections of particles. The tree has a maximum terminal occupancy, $m = 3$. We simply

have not had time to run highly non-uniform cases but we believe, based on our in-core results, that the non-uniformities typical of astrophysical systems will not introduce significant new overheads. The runs involved between 1000 and 1350 monopole interactions and approximately 150 MACs per body. Higher order moments could be employed without substantially changing the code, but the bandwidth vs. operation-count tradeoffs need to be carefully considered. Measured times are for an entire, realistic timestep, i.e., they include sorting, tree building, force evaluation and time-integration. In the parallel case, they do not include redistribution of particles to processors, but we expect this to be a small additional overhead which will be substantially offset by the fact that the particles begin subsequent timesteps almost sorted.

Figure 4 shows overall performance of the system for a fixed size problem (5 million bodies) and for a problem that grows with the number of processors (5 million $\times P$ bodies). The abscissa has been scaled to factor out the average work per processor, so that the departure from horizontal indicates parallel overhead in the fixed-size case. Note, though, that for a fixed size problem, as we add processors, more and more of the data fits in memory, so the observed “super-linear” speedup is not unreasonable. The largest system is an 80 million body model which took 5363 sec/timestep on 16 processors. The single-processor, 5 million body model took 4346 sec/timestep. In contrast, our in-core code integrates a 500000 body model with comparable accuracy in 378 sec/timestep, so the net performance cost of using disk achieve a factor of ten increase in dynamic storage capacity is in the neighborhood of 15%.

The Linux kernel maintains a “buffer cache” of disk pages in OS-controlled memory which grows and shrinks in opposition to user-memory. This feature significantly improves normal operation, but makes benchmarking difficult because if we restrict ourselves to a small working set, and then ask to move pages to and from disk, there is a very good chance that the kernel will find the pages in a buffer cache and not need to query the disk controller at all. The net result is that wall-clock times for I/O operations are often far less than one might expect based on hardware disk latencies and bandwidths, and they are strongly influenced by the vagaries of

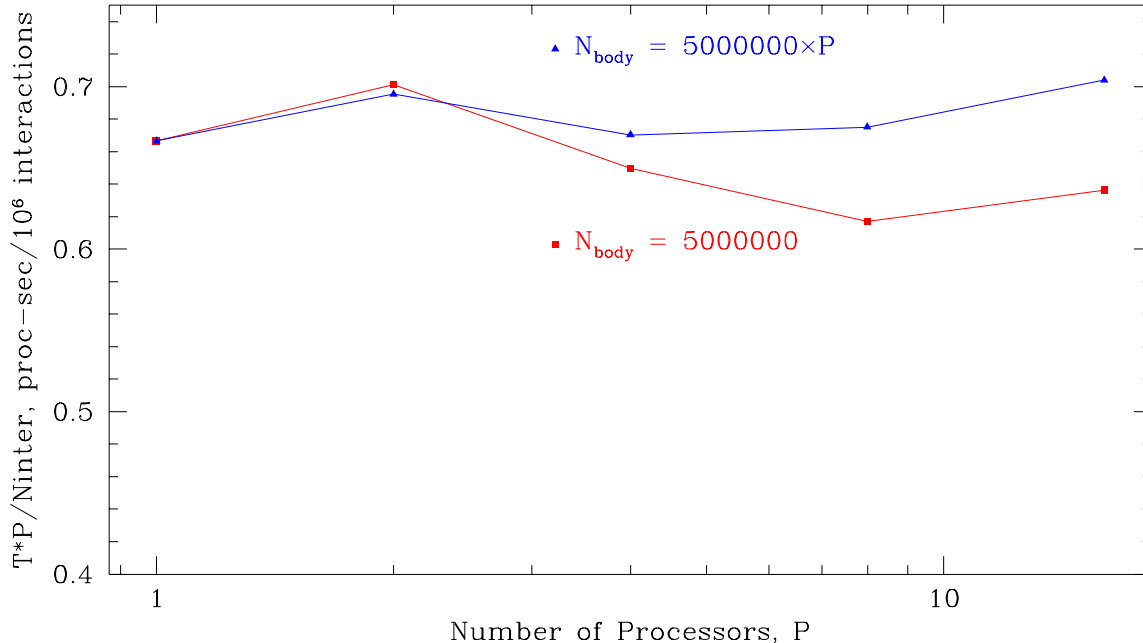


FIG. 4. Scaling behavior up to 16 processors for a fixed size problem (5 million bodies) and a problem that grows with the number of processors (5 million \times P bodies).

the kernel’s buffer caching policies and the size of the system’s DRAM. Rather than attempt to isolate these effects, we simply report the number and size of pages swapped, with the assumption that this provides a lower bound on performance.

Figure 5 shows paging behavior for a simulation with 1 million bodies on a single processor. The model requires about 72 MB of storage altogether. Runs were made with different combinations of the page size and the number of in-core pages. The amount of swapped data is flat over a large range of in-core sizes, and falls dramatically as the in-core size approaches the size of the entire data set. Furthermore, once the number of in-core pages exceeds about 200, there are diminishing returns in making it larger, allowing one to increase the page size instead. Thus, one can amortize disk latency as long as the in-core memory exceeds about $200t_{lb}$, i.e., only 3MB of DRAM is needed to effectively run out-of-core with a commodity EIDE disk. It is tempting to try to fit this in cache, but unfortunately it is almost impossible to get explicit control over cache behavior on modern processors.

8 Conclusions and the future

We have demonstrated that disk can be used for dynamic storage in an “out-of-core” implementation of an astrophysical treecode. An 80 million body model can run on a cluster of 16 PC-class systems. Simulating such a model over the age of the Universe will take a couple of months, but one should recall that the computer is extremely economical, costing under \$60000. One can use cost-effective processors, modest amounts of DRAM, and much larger amounts of disk to address N-body problems that had heretofore been accessible only on the largest of parallel supercomputers. On the other hand, one can now imagine integrating extraordinarily large systems (billions of particles) on large MPPs with independently addressable disks.

Finally, we observe that memory hierarchies are getting deeper, with the gap between processor clock rates and memory latency continuing to widen. Out-of-core methods are designed to tolerate the extreme latencies of disk systems, but they may also be adapted to make effective use of caches and memory hierarchies in more traditional systems. Some ap-

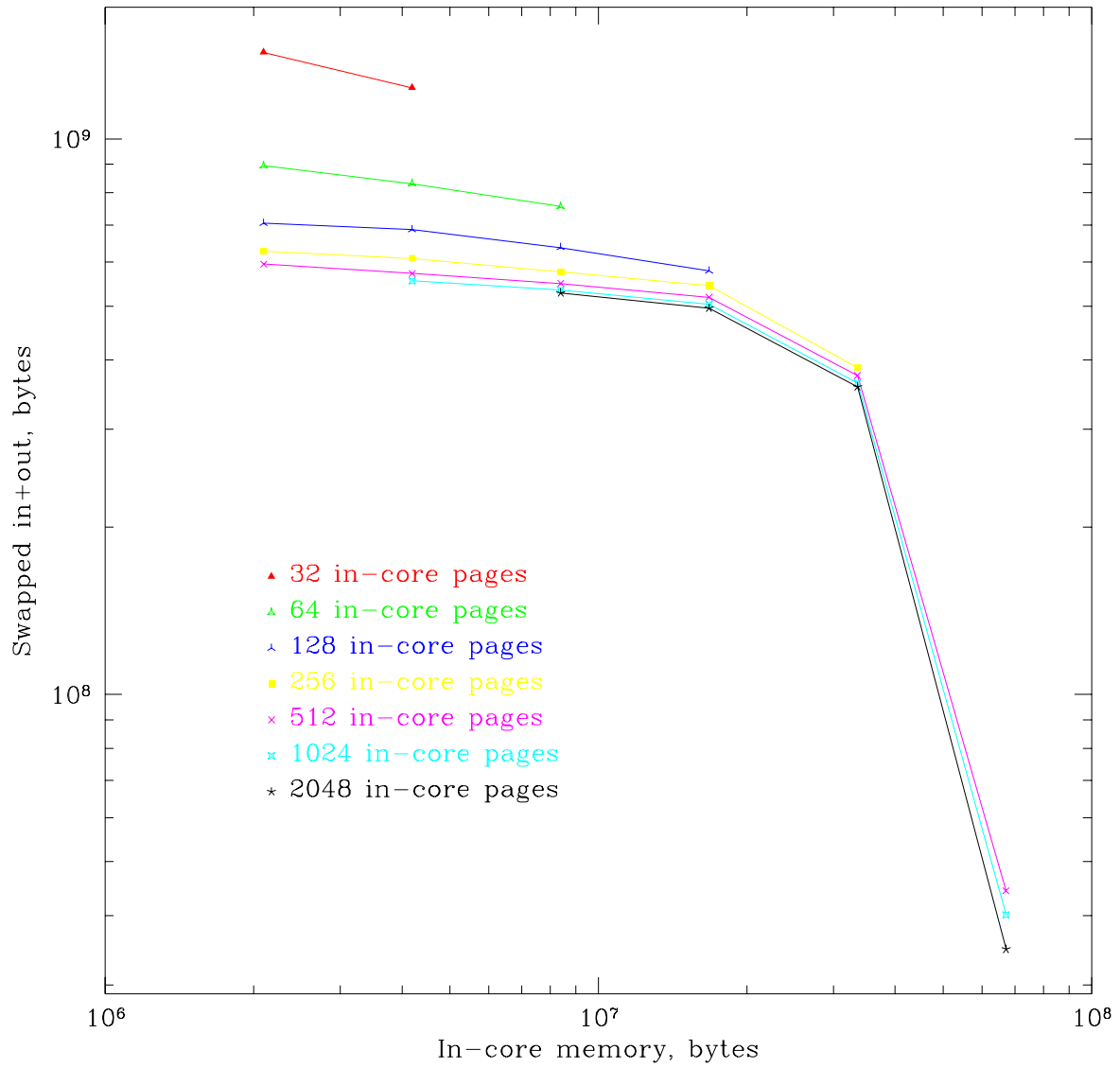


FIG. 5. Aggregate swapped data vs. in-core storage for a 1 million body model and various parameter choices of page size and number of in-core pages.

proaches to the next generation of “petaflop” computers [5] will display latencies (measured in clock ticks) as large as those we observe today in disk systems, so we might expect that optimal algorithms on those systems will be closely related to the out-of-core algorithms of today.

References

- [1] D. H. BAILEY ET AL., eds., *Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1995.
- [2] J. E. Barnes and P. Hut, *A hierarchical $O(N\log N)$ force-calculation algorithm*, *Nature*, 324 (1986), pp. 446–449.
- [3] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, PhD thesis, Yale University, 1987.
- [4] M. HEATH, V. TORCZON, ET AL., eds., *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1997.
- [5] IEEE COMPUTER SOCIETY, *Frontiers '96*, 1996.
- [6] M. S. Warren and J. K. Salmon, *A portable, parallel, versatile N-body tree code*, in Bailey et al. [1].