NASA Contractor Report 182104
ICASE Report No. 90–63

ICASE

INFLATED SPEEDUPS IN PARALLEL SIMULATIONS via malloc( )

David M. Nicol

Contract No. NAS1–18605
September 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Inflated Speedups in Parallel Simulations via `malloc()`

*David M. Nicol**
College of William and Mary

Abstract

Discrete-event simulation programs make heavy use of dynamic memory allocation in order to support simulation's very dynamic space requirements. When programming in C one is likely to use the `malloc()` routine. However, a parallel simulation which uses the standard Unix System V `malloc()` implementation may achieve an overly optimistic speedup, possibly superlinear. An alternate implementation provided on some (but not all) systems can avoid the speedup anomaly, but at the price of significantly reduced available free space. This is especially severe on most parallel architectures, which tend not to support virtual memory. This paper illustrates the problem, then shows how a simply implemented user-constructed interface to `malloc()` can both avoid artificially inflated speedups, and make efficient use of the dynamic memory space. The interface simply caches blocks on the basis of their size. We demonstrate the problem empirically, and show the effectiveness of our solution both empirically and analytically.

*This research was supported in part by NASA grant NAG-1-1060, in part by NASA grant NAS-1-18605, and in part by NSF Grant ASC 8819373.

1 Introduction

Dynamic memory allocation plays an important role in the implementation of discrete-event simulations. For example, in a queueing network simulation blocks of memory are dynamically allocated and freed as the event list changes, and as jobs enter and exit the network. When programming in C one invariably calls `malloc()` to request a block of dynamic memory, and calls `free()` to release it. A programmer may not give great deal of thought to `malloc()`'s underlying implementation. Commonly used implementations search a linked-list of freed blocks for a match. As the length of the list grows, the cost of calling `malloc()` grows. As we will show, this may lead to falsely optimistic speedup measurements of a parallelized simulation (or any other program which makes heavy use of dynamic memory). Other implementations have a cost that is nearly independent of the number of available freed blocks. However, scalable implementations allocate blocks that are significantly larger than the requested block size. Over-allocation poses little problem in a virtual memory system where the effective size of the memory can be measured in gigabytes—but most parallel architectures do not support virtual memory. An implementation that over-allocates physical memory space reduces the size of the simulation model one can run.

This paper illustrates the problem, and offers a simple solution. The solution exploits the fact that there are often only a few different sizes of blocks requested from `malloc()`. A user may easily write an interface to `malloc()` and `free()` that caches freed blocks on the basis of their size. A request to the interface for a block of size n searches the cache for a list of freed blocks of size n. The space-efficient version of `malloc()` is called if the cache fails to satisfy the request. The interface to `free()` rarely calls it. Instead, it places places the freed block into the cache. We show empirically and analytically that this solution "scales"—the cost of requesting a block of memory is nearly independent of the number of freed blocks. We also show that the solution permits the simulation of larger models than is possible using the standard scalable (but space-inefficient) `malloc()`.

This paper is organized as follows. §2 explains how standard implementations of `malloc()` either cause false speedups, or allocate space inefficiently. §3 describes our solution. §4 presents empirical data that demonstrates the problem, and illustrates the effectiveness of our solution. §5 analytically shows that the proposed space management algorithm scales with problem size. §6 summarizes this paper.

2 The Problem

The standard Unix System V implementation of `malloc()` [10] maintains a linked list of all freed blocks, ordered linearly by memory address. Each block records its size and location. Because of the ordering, `free()` can quickly determine whether a newly freed block can be merged with a physically adjacent block. A request to `malloc()` is satisfied by scanning the free list until a block of sufficient size is found. This block is split in two; one subblock is returned to satisfy the `malloc()` request, while the other remains in the free list. The average time required to complete a `malloc()` call depends on the average length of the list. Larger simulation models will tend to demand more dynamic memory and fragment the dynamic memory space more, thereby causing more costly `malloc()` calls.

Let us now characterize the "size" S of a simulation model in terms of the average number of dynamic memory blocks that have been allocated and not yet freed at any given instant. If the simulation has for some time constantly requested and freed blocks randomly, then the number of blocks in the freed list will be proportional to S, and the average cost of a `malloc()` call will be proportional to $g(S)$, where g is some increasing function. Let us also characterize the simulation workload in terms of N, the total number of `malloc()` calls it makes. If N is very large compared to the number of calls that scrambled the freed list, the simulation's execution time will be proportional to $Ng(S)$. Now suppose that the same simulation has been distributed among P processors. If the workload is evenly balanced each processor receives $1/P$-th of the simulation model, and $1/P$-th of the simulation activity. This means that the size of the simulation at one processor is S/P, so that the cost of a `malloc()` call is proportional to $g(S/P)$. Furthermore, the number of `malloc()` calls it performs is N/P. If all P processors execute in parallel, the time required to perform the N `malloc()` calls is proportional to $(N/P)g(S/P)$—a speedup of order $Pg(S)/g(S/P)$ over the serial implementation. Since $g(S) > g(S/P)$ the speedup is superlinear.

It is important to get an accurate measurement of speedup, because only then can we assess the benefit of parallelism to the end user. One may assume that in a serial context the user will execute an optimized implementation; when possible, speedups should be measured against an optimized serial solution. This is not always practical, and so speedups are sometimes measured against one-processor implementations of the parallel algorithm. The research community seems to accept this practice, provided that the complexity of the one-processor solution is the same as the optimized serial solution. In this way the serial solution is asymptotically optimal to within a constant factor. For example, a massively parallel sorting algorithm may require $o(n^2)$ comparisons, but use $o(n^2)$ processors to achieve a fast solution. Computation of speedup based on an $o(n^2)$ serial sorting algorithm is frowned

upon. The speedups we study are computed from one-processor implementations of a parallel approach. Based on experimentation with queueing networks, we estimate that the serial timings we obtain are no more than twice as large as those of highly tuned optimal implementations, at least on queueing networks.

Use of a non-optimal (in terms of complexity) serial implementation is often the underlying cause of overly optimistic measurements of speedup[1]. For example, practitioners of parallel simulation have observed superlinear speedup in early development phases of their algorithms, due to the use of quickly implemented linearly-ordered event lists. Most (but not all—see [3]) realize that performance measurements taken under these conditions are meaningless: any log-time priority list scheme will accelerate both the serial and parallel implementations, and not exhibit superlinear speedups. Careful researchers of parallel simulation ensure that their event list algorithms and synchronization mechanisms scale properly as the number of processors changes.

Observation of superlinear speedup is a clear indication of a problem. More insidious is the case where a non-optimal serial implementation causes speedup to be inflated, but not superlinear. One may be tempted to accept good speedups at face value, without questioning possible inflation. We identify a simple metric, *total workload*, which reveals the presence of inflated speedups. Total workload is simply the sum of the execution time of all "useful" simulation work, in all processors. Inflated speedups of the type induced by `malloc()` are recognized when total workload decreases radically as the number of processors is increased.

In theory one can always defeat inflated speedups by constructing a serial algorithm which emulates the parallel. That option is not so easily chosen for our problem, as delving into the system-level details of dynamic space management is not an activity for the faint-of-heart. One solution exists in the form of a different implementation of `malloc()`, which is standard under Berkeley Unix systems and is usually offered as an option[2] under System V. The size of each block is of the form $2^j - 4$ bytes. This form results from a partitioning of the dynamic memory in blocks of powers of two; `malloc()` reserves four bytes in each block for its own use. A list of free blocks is maintained for each possible size. The size of a requested block is rounded up to the nearest available size, and a free block from that list is returned. Should that list be empty, a larger block is returned. The proper block list is found after a few shifts, and constant-time unlinking operations to release the block. However, if the requested block sizes are uniformly random, the size of an average request will fall half-way between two block sizes. On average, a third of the allocated space will

[1] See [2] for an interesting classification of causes of superlinear speedup.

[2] This is not always the case. At the time of this writing the operating system delivered with the Intel iPSC/860 does not include this option.

be wasted. Node processors on most parallel architectures do not support virtual memory. The over-allocation comes from physical memory, thereby reducing the size of the simulation model that can be evaluated on the machine. However, the cost of calling this version of `malloc()` is nearly independent of the number of outstanding memory blocks. We will refer to this version as the *scalable* `malloc()`.

The problem then is to find a way of managing dynamic space that avoids inflated speedups, and which makes efficient use of space. As is so often the case in computer science, the answer lies in caching.

3 Caching Freed Blocks

Any caching scheme relies on some locality property, usually related to memory addresses and the temporal pattern of accesses to them. The locality we exploit is that of *size*—the size of a requested block tends to be one of only a few sizes requested throughout the simulation. Any given simulation will have a number of object types for which it creates and destroys instances; in our experience the number of different types (and hence object sizes) often is not large. We may therefore emulate scalable `malloc()` and maintain (at the application level) a list of freed blocks for each frequently used block size. We will suppose there is a maximum number L of lists we will maintain in the cache. A similar scheme was proposed some years ago for the caching of procedure frames in the Mesa system [5].

A request for a block of size n is handled by first searching to see if a list for size n blocks is present in the cache. If it is, and if there is a free block of size n the block is delinked and returned. If the cache contains an empty list for size n we call `malloc()` to supply a block. Failure to find a size n list results in the creation of one. In our own applications it is very rare to require more than ten different sized blocks. The code we present and the implementations we test all set an upper bound on the number of lists. The interface can be modified to support applications whose size requirements change dynamically: if the number of existing lists equals L at the time a new list is required, we can replace an existing list. An LRU (Least-Recently-Used) policy may govern replacement. The list "touched" most distantly in the past is selected, and all of its blocks may be returned via `free()`.

Figure 1 gives the source code for our implementation of `ssmalloc()` and `ssfree()`—the scalable space-efficient dynamic memory routines. These routines use the first word in the block either as a link (when in the free list), or to store the block size (when allocated). The versions shown are terse; our actual implementations include error and sanity checks. Other implementations may be more efficient when the number of lists is larger, for example, one might hash on the size of the requested block.

```
#define MAXPTRS 10
struct BufferPtrStruct { int length; char **ptr;
                } BufferPtr[MAXPTRS];

char *ssmalloc(size)
        int size;
{
char **ptr,*ans; int i=0;

        while(BufferPtr[i].length && size != BufferPtr[i].length) i++;
        BufferPtr[i].length = size;          /* in case this is new */
        if(BufferPtr[i].ptr)                 /* List non-empty? */
        {  ptr = BufferPtr[i].ptr;           /* get block request */
           BufferPtr[i].ptr = (char **)*ptr; /* delink free block */ ;
        }
        else ptr = (char **)malloc(size + sizeof(char **));
        *ptr++ = (char *)BufferPtr[i].length; /* record size */
        return((char *)ptr);
}

void ssfree(ptr)
        char **ptr;
{
int size,i=0;
        ptr--; size = *(int *)ptr;    /* back up to size field */
        while(BufferPtr[i].length && size != BufferPtr[i].length) i++;
        *ptr = (char *)BufferPtr[i].ptr;
        BufferPtr[i].ptr = ptr;
}
```

Figure 1: Space-Efficient Scalable Dynamic Memory Routines

4 Empirical Studies

We now present empirical evidence that inflated speedups due to `malloc()` can occur. First we demonstrate that in theory speedups can be inflated by as much as an order of magnitude. This extreme case is achieved when dynamic memory management routines completely dominate the computation. We then examine the problem in the context of a working parallel simulation system, YAWNS (Yet Another Windowing Network Simulator) [6, 7]. YAWNS provides a common platform for the parallel simulation of many different types of networks. The demands on dynamic memory come primary from the handling of small "logical messages" passed between network elements, from dynamic event creation/deletion, and from internal bookkeeping activities. The user defines the messages and the message sizes. In the simulation models we have developed the number of different block sizes is less than ten. We examine the performance of YAWNS on two different simulation problems. Both illustrate the phenomenon of inflated speedups due to `malloc()`, one exhibits superlinear speedup.

We are interested in three performance characteristics: raw finishing time, behavior of the speedup curve, and maximal simulatable problem size. We will look at these characteristics as measured using standard System V `malloc()`, using scalable `malloc()`, and using `ssmalloc()`.

4.1 Superlinear `malloc()`

The potential for superlinear speedups is demonstrated by measuring the average cost of calling `malloc()` (or `free()`) as a function of the "size" of the problem. Our experiments show that on large problems, the average cost of calling `malloc()` is over 10 times greater than the average cost on small problems. Therefore, if the problem can be split among enough processors so that each has a "small" problem, speedups that are an order of magnitude larger than linear might be observed.

We measured the average cost of `malloc()` and `ssmalloc()` on one node of the Intel iPSC/2 [1] in the following way. To create a problem of size S we construct an array of S pointers, which will point to blocks of dynamic memory. Each array position is assigned a block of a given size. The possible sizes (in bytes) are 8, 16, 32, 64, 128, and 256. Assignment of sizes to array positions is cyclic: slots $0, 6, 12, \cdots$ get size 8, slots $1, 7, 13, \cdots$ get size 16, and so on. At initialization, an array position is either filled with a pointer to a block of the appropriate size, or is left empty. The choice is made randomly, with equal likelihood for either possibility. Next we iterate, making many passes over the array. On each pass, at each array position, we randomly decide with equal likelihood whether or not to change the status of the array position. If a decision to change the status is made, a non-empty array

pointer is changed by freeing the indicated block; the status of an empty array pointer is changed by allocating a new block, a pointer to which is stored in the array location.

S models the size of a simulation problem. On average there will be $S/2$ blocks allocated in the array, implying that the average number of freed blocks in `malloc()`'s list is proportional to $S/2$. As S grows we expect the cost of calling `malloc()` to grow.

We can measure the time required to iterate a given number of times over the array, then count the number of calls made to space allocation routines, and compute the average total time per call. However, this measurement includes overhead due to looping, testing, random number generation, and the like. The overhead can be accounted for by performing an identical run that does everything *except* call the space allocation routines. The actual average cost of calling a space allocation routine is computed by taking the difference in timings for the two runs, and dividing by the number of routine calls.

Figure 2 plots the average cost of calling `malloc()` or `free()`, and the average cost of calling `ssmalloc()` or `ssfree()`, as a function of the logarithm (base two) of the array size, S. The cost of the scalable space-efficient routines is seen to be very nearly constant—it rises slightly from 19 μ-sec to 21 μ-sec as S goes from 2^2 to 2^{16}. The cost of `malloc()` and `free()` remains relatively constant at 16 μ-sec for S between 2^2 and 2^6. However, for S larger than 2^6 the cost rises, reaching 186 μ-sec at $S = 2^{16}$. This is over 10 times slower than its average cost at $S = 2^2$. Therefore, in the most extreme case it would be possible to achieve speedup which is a factor of 10 larger than linear. This is unlikely to happen, because other scalable costs are involved in the computation and will serve to mute the effect of a non-scaling `malloc()`. But, as we will see, real applications can suffer inflated and sometimes superlinear speedups due to `malloc()`.

4.2 Inflated Speedups in YAWNS

Next we show how `malloc()` can inflate speedup measurements of a real application, the YAWNS parallel simulation system. We will look at the performance characteristics of YAWNS on two different simulation problems. The first simulates the movement of objects through an abstract hypercube structure. An object resides at a hypercube node for a random period of time, then randomly selects some node connected to its current one and moves there. Nodes do not impose queueing, so any number of objects may reside concurrently at a node. This simulation is interesting because it exhibits superlinear speedup. The second simulation is of Conway's Game of Life. Speedups for this problem become inflated, but are not (usually) superlinear. This problem also reveals how scalable `malloc()` limits the size of problem one can simulate.

The object movement simulation was written as a simple driver to test YAWNS during

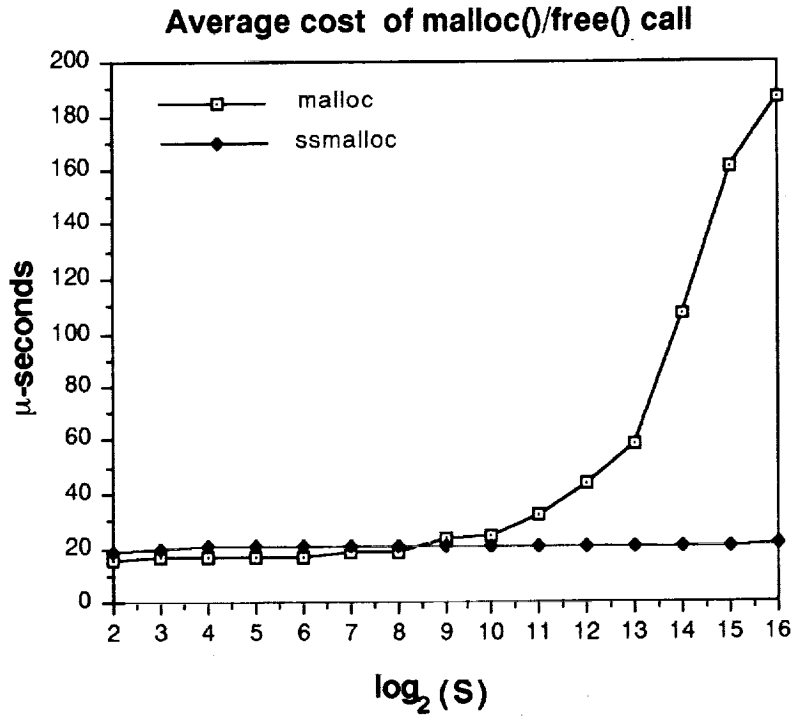

Figure 2: Average cost of space allocation routines, as function of problem size

implementation and debugging. Our discovery of superlinear speedup on this problem led to the inquiry and solution reported in this paper. Our suspicions fell upon `malloc()` only after we had eliminated every other possibility.

We will use one specific problem instance to illustrate superlinear speedup, although this simulation model routinely achieves it under a wide variety of circumstances. The problem instance moves 4096 objects between nodes in an 8 dimensional hypercube structure (256 nodes). Each object resides at the node for a random period of time, composed of the constant 0.25 plus an exponential random variable with mean 1. It chooses a new destination with equal likelihood among all the nodes connected to its current one. The simulation terminates when the simulation time reaches 100. This requires the processing of a few hundred thousand events.

Table 1 presents data taken from a representative problem run. The simulation is written in such a way that given an initial random number seed, exactly the same events occur, independently of the number of processors used. We present data from use of both `malloc()` and `ssmalloc()`. The processor utilization figures come from independent on-processor measurement of the total time spent performing overhead activity—interprocessor commu-

nication, synchronization delays due to blocking, synchronization activity required by the synchronization protocol, etc. The utilization figure thus represents the average fraction of time a processor spends performing "useful" simulation work. This workload should be independent of the number of processors used. In theory, one can always estimate the total time spent executing useful workload by multiplying together the average utilization, the finishing time, and the number of processors. We call this the *Total Workload.* Our tables provide this calculation.

In practice, the true average processor utilization may be difficult to obtain, because lack of good hardware timing mechanisms make it difficult to measure small bursts of overhead activity. An additional problem is faced by programs using optimistic synchronization mechanisms such as Time-Warp [4], because the processing of an event can be overhead, if the event is later rolled back. A method for measuring utilization in a Time Warp System is given in [8] (although they call this the *effective utilization*, allowing the definition of "utilization" to include overhead). The method is based on measuring the time delay associated with processing each event. If the events have small executions times relative to the timer resolution, considerable error may creep into the utilization estimate. A side-effect of YAWNS global style of synchronization is that the overhead occurs in bursts isolated from the processing of useful work. While clock granularity can still be an issue, it is less of a problem than it would be under more asynchronous synchronization protocols.

The data clearly shows superlinear speedup under `malloc()`, and shows *why* the speedup is inflated. The total workload on one processor is 65% larger than the total workload on 16—the serial version appears to be doing more work. The extra work can be attributed to the larger cost of calling `malloc()` and `free()` on larger problems. Caching compares well with `malloc()`. Not only is the total workload relatively constant (less than 5% deviation between the 1 and 16 processor total workloads) so that the speedups behave properly, the raw finishing times are better as well. One shouldn't expect the total workload measurement to be exactly constant, as a fair amount of noise creeps into the timing process—the resolution of the clock available to a program on the Intel iPSC/2 is coarse, at one millisecond.

Checking the relative constancy of total workload is a useful way of detecting whether an application has inflated speedups. Speedup inflation may go unnoticed if the speedups are sublinear. However, if speedups *are* inflated, measurement of the total workload (when possible) will reveal it. This is the case with the second YAWNS application we consider, Conway's Game of Life.

The Game of Life consists of a toroidal mesh of cells, each of which is either dead or alive. Time progresses in unit steps. The state of a cell at time n is determined by a simple rule. If the cell was alive at time $n-1$, then it remains alive at n if and only exactly three

Processors	secs	utilization	Speedup	Total Workload
1	851	99%	1.00	842
2	414	88%	2.05	728
4	171	86%	4.97	588
8	83	81%	10.2	537
16	45	71%	18.9	511

Performance data using `malloc()`

Processors	secs	utilization	Speedup	Total Workload
1	493	99%	1.00	488
2	250	96%	1.97	480
4	132	89%	3.73	470
8	71	82%	6.94	466
16	41	71%	12.02	466

Performance data using `ssmalloc()`

Table 1: Performance measurements for moving object simulation

of its immediate neighbors (at all 8 points of the compass) were alive at time $n-1$. The rationale is that if fewer than three neighbors are alive the cell dies of loneliness, while if more than three neighbors are alive it dies of overcrowding. Similarly, a cell which was dead at time $n-1$ springs to life spontaneously at time n if it has exactly three live neighbors at time $n-1$.

One usually thinks of the Game of Life in the context of cellular automata, but discrete-event simulation provides an efficient mechanism for performing the computation. The events are re-evaluation of a cell's state. Whenever a cell changes state it sends a message to each of its 8 neighbors informing them of the change. A cell which receives a change of state message must re-evaluate its own state, as its environment has changed.

The Game of Life consumes a great deal of dynamic memory on large boards, owing to the high number of messages that a cell sends when its state changes. It is therefore a good problem for illustrating the short-comings of scalable `malloc()`. We measured the largest board size that could be simulated for 25 time-steps without exhausting memory, given a

random initial assignment of cell states where each cell is alive with probability 0.2. Of course, the largest board size possible depends on the initial assignment, but one run on 16 processors is fairly representative of the others. On the representative run, the `malloc()` implementation was able to simulate a 357×256 cell board. Under `ssmalloc()` it handled board sizes up through 315×256 cells, while under scalable `malloc()` it failed after a board of size 147×256.

Standard `malloc()` permits the simulation of a board which is 13% larger than the largest one permitted under `ssmalloc()`. This is explained almost exactly by the fact that intercell messages are 7 words long. The space for these messages always comes originally from `malloc()`, but `ssmalloc()` asks it for 8 words—the extra one is used by `ssmalloc()` and `ssfree()` for linking, and for storing of the message size. Thus, on this problem `ssmalloc()` suffers a 12.5% space overhead. It is possible to eliminate this overhead, for a price. `malloc()` writes its own secret information in the first word before the one returned. We could overwrite that word for own purposes, but then could never return the block to `free()` as we need to if `ssmalloc()` were modified to support more than L different list sizes.

The failings of scalable `malloc()` are clear–over a 50% reduction in the size of the maximum simulatable problem . That the degradation is so much greater than the 33% "average" one infers from a quick average case analysis is largely due to the primary message size. Scalable `malloc()` uses a block of 15 words to satisfy `ssmalloc()`'s request for 8 words. Also, the quick average case analysis does not account for the fact that a failure to find the smallest block which contains a request causes the selection of blocks that are 4 or more times larger than the size of the request.

Table 2 presents performance measurements obtained using `malloc()`, scalable `malloc()` and `ssmalloc()` on a 64 cell board, simulated for 50 steps, where the initial probability of a cell being alive is 0.2. 64×64 is the largest power-of-two sized board that a single processor is capable of handling using `malloc()`, or `ssmalloc()`. Scalable `malloc()` requires the memory of four processors to simulate a board of that size, for that long. Lacking a serial timing, we omit speedup calculations for scalable `malloc()`. However, it is clear that `ssmalloc()` is approximately 20% faster than scalable `malloc()`, as well as being more space efficient.

5 Analysis

A simple analytic model supports the observed near-constant cost of `ssmalloc()`. We model the behavior of a single list of commonly sized blocks as a probabilistic birth-death process, and show that the average number of transitions between expensive calls to `malloc()` grows exponentially fast. Even if the cost of calling `malloc()` is linear in the number of outstanding

Processors	secs	utilization	Speedup	Total Workload
1	197	99%	1.00	195
2	91	85%	2.16	155
4	50	68%	3.91	137
8	29	61%	6.87	139
16	17	51%	11.7	137

Performance data using `malloc()`

Processors	secs	utilization	Total Workload
4	36	79%	133
8	20	72%	116
16	12	63%	120

Performance data using scalable `malloc()`

Processors	secs	utilization	Speedup	Total Workload
1	97	99%	1.00	96
2	50	90%	1.95	99
4	28	78%	3.45	88
8	16	68%	5.93	89
16	10	58%	9.63	93

Performance data using `ssmalloc()`

Table 2: Performance measurements for Game of Life simulation

memory blocks, there are exponentially many "cheap" `ssmalloc()` calls between the linearly expensive ones. The expensive calls are therefore amortized over so many cheap calls that the average cost of calling `ssmalloc()` is nearly constant.

Let $T_1, \ldots, T_L$ be the set of lists, holding blocks of size $s_1, \ldots, s_L$ respectively. Consider the sequence of `ssmalloc()` and `ssfree()` calls to all lists on one processor. We will call this the *complete* sequence. We can always filter the complete sequence and consider only those calls for blocks of size s_j. We suppose that this stream forms a simple Markovian birth-death process whose state is the number of dynamic blocks allocated by `ssmalloc()`,

but not yet released. From a non-zero state, with probability $p_j < 0.5$ a call in the filtered stream for T_j will be to `ssmalloc()`, with probability $q_j = 1 - p_j$ it will be to `ssfree()`. If all requested blocks have been returned, then the state is zero and the next call must be for `ssmalloc()`.

A non-constant cost of calling `ssmalloc()` occurs whenever the appropriate list is empty. This event coincides with the Markov chain achieving a *record*, or new maximal state. The *ith* record $R_{i,j}$ for list T_j is defined simply to be the number of chain transitions that occur before state i is reached for the first time. We are interested in $E[R_{i,j}] - E[R_{i-1,j}]$, for $i = 2, 3 \ldots,.$, as these differences indicate how often, on average, `ssmalloc()` must call `malloc()`.

Let $S_{n,j}$ denote the total number of blocks associated with T_j, either explicitly in the list or still allocated at the *nth* complete `ssmalloc()` or `ssfree()` call. $S_{n,j}$ is just the index of the last record defined for this list, e.g. $S_{n,j} = k$ if k is the largest record $R_{k,j}$ such that $R_{k,j} < k$. We assume that the average cost of calling `malloc()` at the *nth* complete call is an increasing sublinear function g of the total number of allocated but unfreed blocks at the *nth* complete call: $O(E[g(\sum_{j=1}^{L}(S_{n,j})]])$. The point we will establish is that this non-constant cost increases by only $O(1)$ every time `malloc()` is called, or equivalently, every time some list achieves a new record. We will show that for each T_j, the expected number of calls between records ($E[R_{i,j} - R_{i-1,j}]$) grows exponentially in i. This implies that the number of references between calls to `malloc()` grows exponentially as i increases, so that each "expensive" `ssmalloc()` call is amortized over exponentially many constant-cost calls.

We now derive an expression for $E[R_{i,j} - R_{i-1,j}]$, for any list T_j. The only way to reach state i the first time is through state $i-1$. It requires $R_{i-1,j}$ transitions to reach $i-1$ for the first time. Then, a Bernoulli trial with probability p_j determines whether state i is achieved in the next transition. In fact, the number of times after the first that the chain touches state $i-1$ before stepping up to state i is a geometric random variable G minus 1, where $E[G] = 1/p_j$. Each time the chain fails to step up from $i-1$ to i it wanders off in the lower indexed region of the state-space before returning. The number of transitions involved in each wandering away is a random variable with mean μ_{i-1}. Each wandering is independent and identically distributed as any other, and $G-1$ is a "stopping time" for the sequence of wanderings. If the number of transitions in the *kth* wandering is denoted by W_k, then

$$R_{i,j} = R_{i-1,j} + \sum_{k=1}^{G-1} W_k + 1.$$

Applying Wald's lemma[9] to the random sum and rearranging we find that

$$E[R_{i,j}] - E[R_{i-1,j}] = \left(\frac{1}{p_j} - 1\right)\mu_{i-1} + 1. \tag{1}$$

We will derive μ_{i-1} from the fact that the mean time between visits to a state k in an ergodic Markov chain is equal to the reciprocal of the limiting occupancy probability of state k : $1/\pi_k$ [9].

The subchain in which the wandering occurs is simply a birth-death process with reflecting states at 0 and $i-1$. The occupancy probability of state $i-1$ is derived using standard techniques. First, the local balance equations are set up:

$$\begin{aligned} \pi_0 &= q_j \pi_1 \\ p_j \pi_k &= q_j \pi_{k+1} \qquad \text{for } k = 1, \ldots, i-3 \ , \\ p_j \pi_{i-2} &= \pi_{i-1} \end{aligned}$$

from which it follows that

$$\pi_{i-1} = \left(\frac{p_j}{q_j}\right)^{i-2} \pi_0,$$

or equivalently,

$$\mu_{i-1} = \frac{1}{\pi_{i-1}} = \frac{\left(\frac{q_j}{p_j}\right)^{i-2}}{\pi_0}.$$

As i increases, the limiting probability π_0 decreases. Furthermore, since $(q_j/p_j) > 1$, it follows that μ_{i-1} increases exponentially fast in i. Applying this observation to equation (1) we see that the expected number of transitions between records grows exponentially in i.

The cost of calling `malloc()` grows at most linearly in the total number of records achieved by all the lists. However, for each list the expected number of transitions between records is growing exponentially in the number of records. Consequently, on average the sublinear cost of `malloc()` is amortized over sufficiently many constant-cost calls that the asymptotic average cost of calling `ssmalloc()` is nearly constant.

6 Summary

Dynamic space management is an important component of many discrete-event simulations. When programming in C, one is likely to use `malloc()` to acquire blocks of free space. However, commonly used versions of `malloc()` either induce inflated speedups, or overallocate memory by as much as 50%. This paper gives empirical evidence of the problem, and then proposes that dynamic memory blocks be cached on the basis of their size. We demonstrate empirically and analytically that the proposed solution is effective.

Acknowledgments

Scott Riffe is commended for his efforts in programming the `malloc()` measurement code. We thank Phil Kearns for showing us the source code for scalable `malloc()`.

References

[1] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, Sept. 1989.

[2] D.P. Helmbold and C.E. McDowell. Modeling speedup (n) greater than n. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):250–256, 1990.

[3] N.J. Davis IV, D.L Mannix, W. Shaw, and T Hartrum. Distributed discrete-event simulation using null message algorithms on hypercube architectures. *Journal of Parallel and Distributed Computing*, 8(4):349–357, April 1990.

[4] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[5] B.W. Lampson. Fast procedure calls. In *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 66–76, 1982.

[6] D. Nicol, C. Micheal, and P. Inouye. Efficient aggregaton of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

[7] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. Technical Report 90-20, ICASE, 1990. Available from ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

[8] P.L. Reiher and D. Jefferson. Virtual time based dynamic load management in the time warp operating system. In *Distributed Simulation 1990*, volume 22, pages 103–111. SCS Simulation Series, 1990.

[9] H.S. Ross. *Stochastic Processes.* Wiley, New York, 1983.

[10] AT & T. *Unix System V/386, Release 3.2.* Prentice-Hall, Englewood Cliffs,NJ, 1989.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-182104 ICASE Report No. 90-63		

4. Title and Subtitle	5. Report Date
INFLATED SPEEDUPS IN PARALLEL SIMULATIONS VIA MALLOC()	September 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David M. Nicol	90-63
	10. Work Unit No.
	505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered
	Contractor Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to International Journal of Simulation

Final Report

16. Abstract

Discrete-event simulation programs make heavy use of dynamic memory allocation in order to support simulation's very dynamic space requirements. When programming in C one is likely to use the malloc() routine. However, a parallel simulation which uses the standard Unix System V malloc() implementation may achieve an overly optimistic speedup, possibly superlinear. An alternate implementation provided on some (but not all) systems can avoid the speedup anomaly, but at the price of significantly reduced available free space. This is especially severe on most parallel architectures, which tend not to support virtual memory. This paper illustrates the problem, then shows how a simply implemented user-constructed interface to malloc() can both avoid artificially inflated speedups, and make efficient use of the dynamic memory space. The interface simply caches blocks on the basis of their size. We demonstrate the problem empirically, and show the effectiveness of our solution both empirically and analytically.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parallel simulation, superlinear speedup, dynamic space allocation	61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	17	A03