CS 5600 Computer Systems Project

SpeedyLoc
A High speed Malloc Library

by

Shubhi Mittal
Sandarsh Srivastav
Rongxuan Liu

under

Professor Kapil Arya

College of Computer and Information Science
Northeastern University

Table of Contents

Abstract

SpeedLoc is a memory allocator on Linux platform. It implements malloc and free routines in a mostly lock-free fashion. SpeedLoc does not suffer from complexity risks such as dead-lock, and can perform better on fast-path operations. It includes a MP-RCS (multiprocessor restartable critical section) sub-module to achieve lock-free operations. It also takes inspiration from TCMalloc's fine-grained size classes to achieve low internal fragmentation rate.

Contribution:

Shubhi Mittal: Upcall routine
Sandarsh Srivastav: Kernel driver
Rongxuan Liu: Malloc library

Introduction

Memory allocation directly affects application's performance. There are various designs for memory allocator. From Solaris' single mutex managed global heap to another extreme, where one allocates a memory arena for each thread, many solutions have been proposed to achieve low operational latency and high memory utilization.

We found lock-free malloc an effective approach to solve latency and concurrency issue via improved data locality and simplified application logic. This paper will discuss how mostly lock-free approach is implemented to improve allocation performance, and how we bring the fine-grained size class idea from TCMalloc to improve memory utilization and scalability.

Novelty

The novelty in our project lies in the mostly-lock free implementation of the malloc library. The program bypasses the traditional mutex based approach by designing a Multi-Processor Restartable Critical Section (MP-RCS) module which permits user level threads to know which processor they are running on and uses upcalls to notify the user process if a thread has been preempted or migrated. When a user level thread is preempted in its critical section, the critical section is restarted.

Design/Approach

System Overview:

SpeedyLoc implementation consists of a Malloc allocator and the Multiprocessor Restartable critical section (MP-RCS) subsystem. The essentials including data structures and algorithms for malloc and free are implemented in the Allocator. MP-RCS subsystem includes a kernel driver, a user-mode notification routine, and the restartable critical sections themselves. Collectively, we refer to the driver and the notification routine as the critical execution manager. The allocator uses MP-RCS to safely operate on CPU-local malloc metadata.

Allocator Design:

SpeedyLoc maintains two types of memory Heaps for malloc and free requests -- local Heap, which belongs to a specific CPU core for its lifetime, and global Heap, one that is shared among threads running on different CPU cores.

Each malloc or free request goes directly to the local Heap of the CPU core where the thread is running on. A local Heap keeps an array of Superblocks, each representing a size class. A Superblock of a particular size class denotes a contiguous memory region that is chopped into multiple chunks of memory regions of that size. For simplicity in design and performance, a local Heap is allowed to have only one Superblock for one size class. Traversing across a linked list or array is avoided.

Global Heap does not deal with allocation or free requests directly. Instead, it serves as a resource pool whenever a Heap belonging to a particular CPU core is in lack of memory. Under such situation, global Heap will either find a non-empty Superblock or create a new Superblock, and swaps it with the exhausted Superblock of that local Heap. Upon successful swap, the local Heap will re-perform the search in the new (now its own) Superblock, and return the address found.

The core idea in this design is to improve data locality. With the pre-indexed size class computation and flat Heap-Superblock design, searching for a free block in the local Heap (where the thread/process is running on) is guaranteed to be fast. Improved locality would ensure the program going through this *fast-path* under most circumstances, therefore minimising the chances of communicating with global shared resource. Since the fast-path takes place on a single CPU core, no lock is needed.

The malloc and free processes are illustrated in Figure 1 and 2 in the appendix.

Implementation / Technical Challenges

Implementing Allocator

LFMalloc has a detailed description on its allocator design. Critical components such as the two restartable critical sections are well explained by pseudo-codes. Implementing it in C on Linux is a straightforward task.

The interesting bits come in when concerns over internal fragmentation and size class computation arose. As part of the allocator design, LFMalloc ensures that the step between two adjacent size classes is no bigger than 20% of the smaller block. The benefit is that smaller number of size classes would in turn speedup size class computation for a coming *malloc* request, since the size class is calculated in a recursive fashion. On the other hand, the fairly large ratio of 20% could severely affect the efficiency when the application scales.

We analyzed TCMalloc's size-classing approach. Interestingly, TCMalloc keeps a very fine-grained collection of size classes up to 96 different sizes. For size-classes that could fit some SSE types, it keeps a minimum step of 16 bytes between two adjacent classes. For bigger size classes, it ensures a class is no more than ⅛ bigger than the class before it. The total internal fragmentation rate is kept under 12.5%.

We combined the two approaches to seek a balance between too many size classes and too large waste of space. For size class alignment, we completely adopted TCMalloc's approach, keeping a ⅛ step between adjacent classes. However, in light of the application scenario speedyLoc is targeting at (large number of smaller sized requests), we reduced the biggest size class to 4 kilobytes. The number of size classes in-turn reduced to 48.

Size class computation was also made significantly faster by including a pre-indexed flat array of 377 different sizes. For any allocation request, we calculate the index of the requested size in this array by doing right shift for 3 bits (that is, to divide by 8). The value stored in the array denotes the corresponding size class number. For example, for request of 25 bytes, it gets an index value of 4. The value of index 4 in the array is 2, denoting a size class of 32 bytes. In sum, 2 steps are taken to calculate the size class -- right shift and then fetch a value stored in array.

Implementing Up Call Routine

The upcall routine is essentially a way to switch/transfer the control from the kernel space to the user space. This involves interprocess communication between kernel space and the user space. During our research, we came across various methods of implementing and executing upcalls. The IPC between kernel space and user space can be done using RAM based file systems such as debugfs, procfs; network sockets, creating a kernel device and using signal handlers. The mostly lock-free *malloc* implementation of LFMalloc also refers to changing the instruction pointer of the process monitored in the kernel driver to point to the user-notification routine.

In our implementation, we implemented upcalls using signal handlers. The kernel interrupts the process execution by sending a signal to the process and thus, switches from the kernel space to the user space. The signal handler defined in the user space is also referred to as user-notification routine.

In the user-notification routine, we essentially figure out if the process had been executing in the critical section in either free or *malloc* implementations. This is done by setting a long jump at the beginning of the critical sections for both *malloc* and *free*. We also keep a restartable flag, which gets set to a specific value before the process enters any of the critical sections. This value determines if a process thread was in any of the critical sections or not. Using these, the user-notification routine decides if the process thread was in critical section, if yes then then long jump gets executed at the respective critical section and if not then we simply return from the signal to resume execution.

The challenging aspect of this was to first establish a communication channel between kernel driver and user process. We first implemented the communication mechanism using in memory file systems, which can be accessed by both kernel module and user space program. However, we faced issue to read the content from the file. We solved the issue eventually by implementing a signal handler in SpeedyLoc library that invokes the upcall.

Implementing the Kernel Module

In a nutshell, the kernel module performs the following tasks:

- Register a process to the driver and remember our processes.
- Intercept each process which is going to be executed next using kprobe on the kernel method ***switch_to()***.
- Figure out if this process is registered and if yes, figure out if the process was preempted, or has migrated its CPU.
- Deliver an upcall_signal to the process.
- The upcall routing will now take decisions as mentioned.

The kernel module comprises of two parts.

1. The implementation of a kprobe which is a debugging mechanism used to monitor events. More specifically, you can monitor the specified kernel call and have a function run before the kernel call is executed or/and after the kernel call is executed.
2. The implementation of the kernel module itself which was implemented as a device driver and communicated with using the **ioctl()** system call. Thus the kernel module was visible to the user space as a device inside the **/dev** folder.

The main challenge in this piece was to find a way to recognize the process that was going to be executed next. We needed the details of this process so we could deliver the upcalls to this process. This was done by figuring out that the last kernel method that is called before a process if put on the processor is the ***switch_to()*** method. This method fetches the next method in the schedule_queue and puts it to execute. We put a probe on this kernel call and executed our driver function to figure out if this process was registered with us, was preempted or not, or if this core was a different one compared to where the process previous executed. A decision would be made by the kernel driver based on the above to deliver an interrupt signal to this process. The signal handler in the process then handles the rest.

The second challenge was to identify processes that were ours since we did not want to be initiating interrupt signals for all the processes that were going on the processor (which of course implies that we were listening for everything that went on the processor). This was resolved by implementing the driver as a virtual device present in ***/dev.*** The driver and the process would communicate using ***ioctl()*** system call. In the constructor of our malloc library, we would have a function that would register the process to our driver and thus, any time after that, if the process is scheduled on the processor, we can send (or not) an interrupt.

Communicating back to the user space from kernel space was a challenge as described above and that was successfully solved using the ***ioctl()*** system call.

Evaluation

Time efficiency and Memory consumption were the two factors incorporated to evaluate the performance of the speedyLoc library. The load testing was done to evaluate time efficiency by iteratively increasing the number of threads. Libc malloc was used as a benchmark to compare the performance of our library. Figure 4 to Figure 8 are our test results for the performance evaluation.

Conclusion and Future Work

Kernel driver and upcall routines significantly speed up the malloc operation. However, during evaluation we noticed two issues of these submodules that can severely damage the performance of SpeedyLoc. The first issue is the lack of re-routing mechanisms for upcall. In the current implementation, kernel driver redirects every thread it recognizes as SpeedyLoc consumer to the upcall routine. With the number of concurrent theads increases, the frequency of upcalls will increase along. This communication overhead is bad for performance and should be avoided in most situations.

The second issue is a programmatic bug in upcall routine. Upcalls are routines executed in user space and can be interrupted any time. Therefore, threads that were preempted during upcall would re-execute upcall when rescheduled. This can lead to lead to infinite loop and starve other threads.

The first issue can be solved by introducing a data structure in kernel driver that maps a specific thread to its previous CPU and last *onproc* time. A thread that has not been preempted and has not changed CPU during its lifetime will not have contention issue, and thus does not need to trigger the upcall to restart its critical section. The kernel driver would then only initiate communication to upcall when the thread has been preempted before.

To solve the second issue, the way upcall being executed must change. Currently it is implemented using a signal handler. However, instead of the kernel driver sending a signal to invoke user space program, it can simply change the instruction pointer to the address of the upcall. If during the registration phase the kernel driver is made aware of the address of the upcall, it can simply determine whether the about-to-be-run thread is in the upcall routine or not by comparing its IP with the upcall address.

Reference

1. Dice, David & Garthwaite, Alex. (2002). *Mostly lock-free malloc*. SIGPLAN Notices. 38. 269-280.
2. Google. *TCMalloc*. http://goog-perftools.sourceforge.net/doc/tcmalloc.html
3. Linux Documentation Project Wiki. *Kernel Space, User Space Interfaces*. http://wiki.tldp.org/static/kernel_user_space_howto.html
4. Xavier Calbet. *Writing Device Drivers in Linux: A Brief Tutorial*. http://freesoftwaremagazine.com/articles/drivers_linux/

Appendix

Figure 1 Malloc Flow Diagram

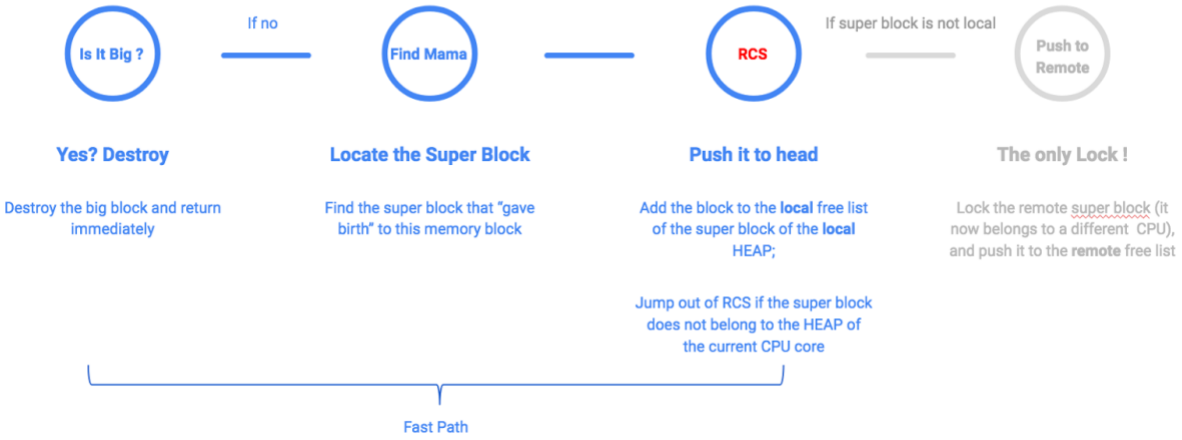

Figure 2 Free Flow Diagram

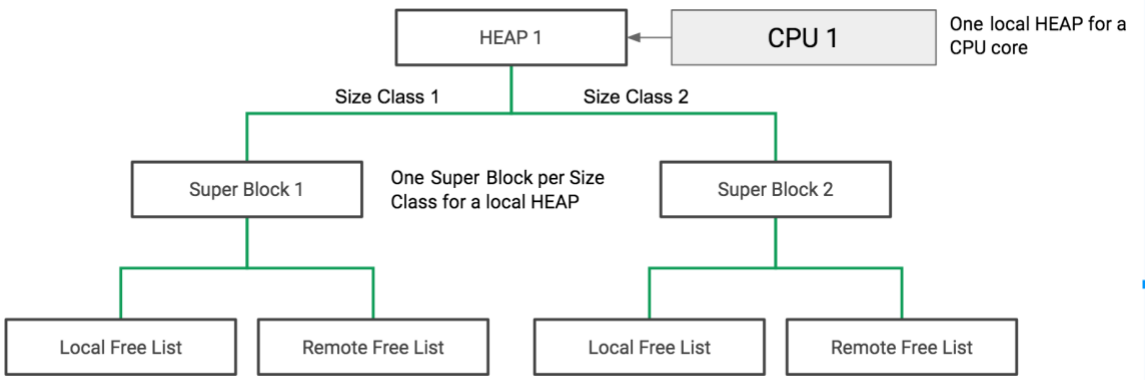

Figure 3 Local Heap Design

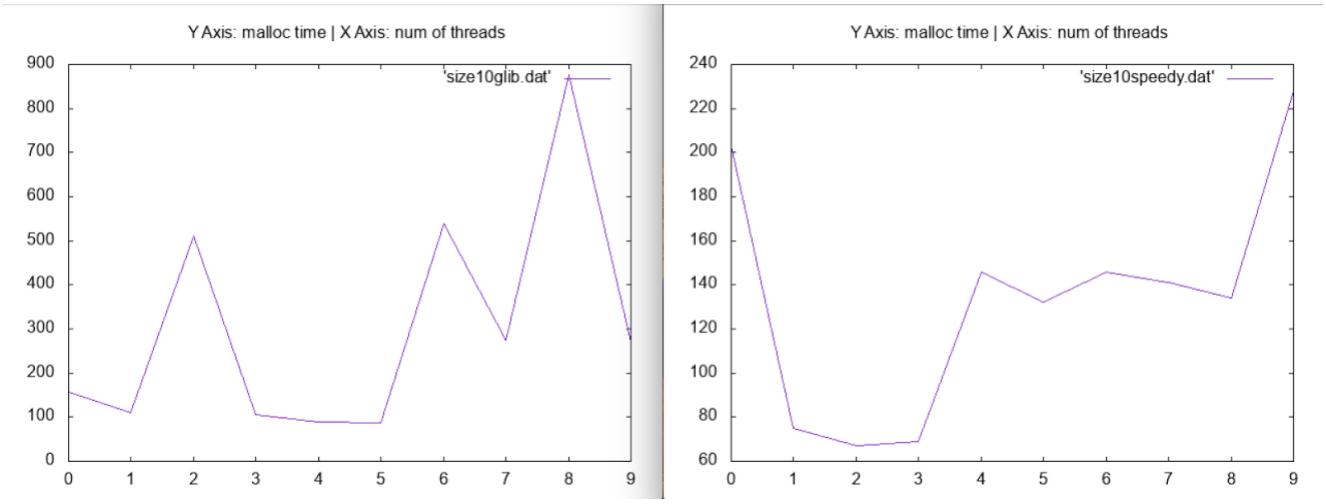

Figure 4 Speed Comparison up to 10 threads (left: glibc, right: speedyLoc)

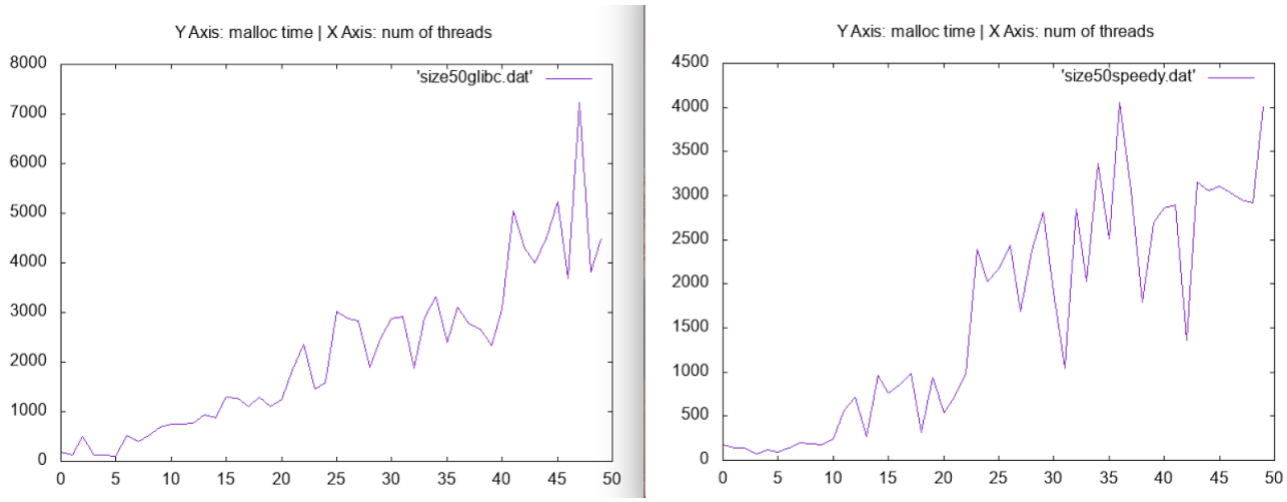

Figure 5 Speed Comparison up to 50 threads (left: glibc, right: speedyLoc)

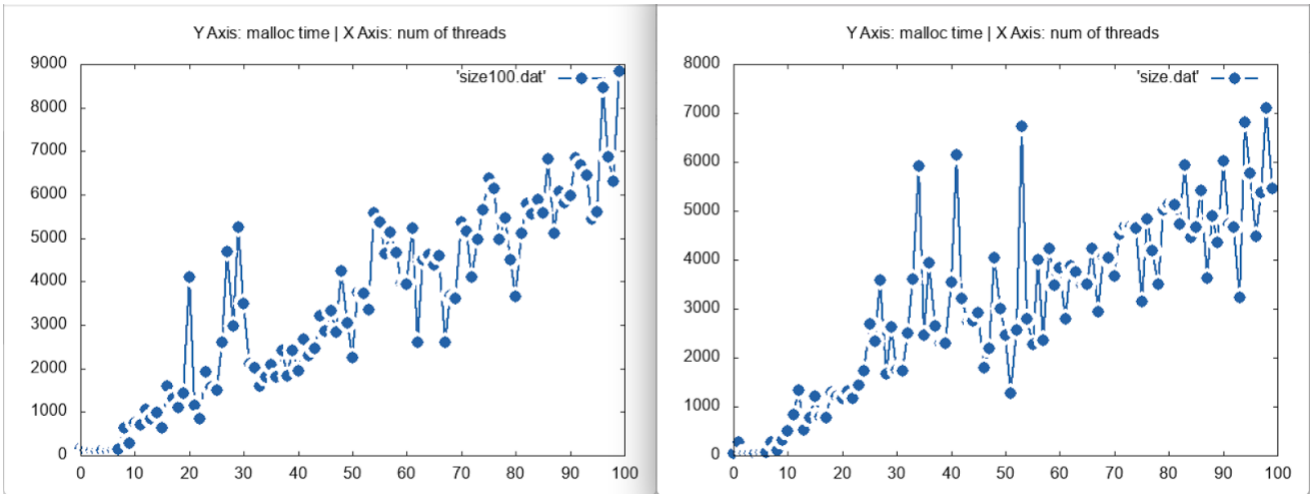

Figure 6 Speed Comparison up to 100 threads (left: glibc, right: speedyLoc)

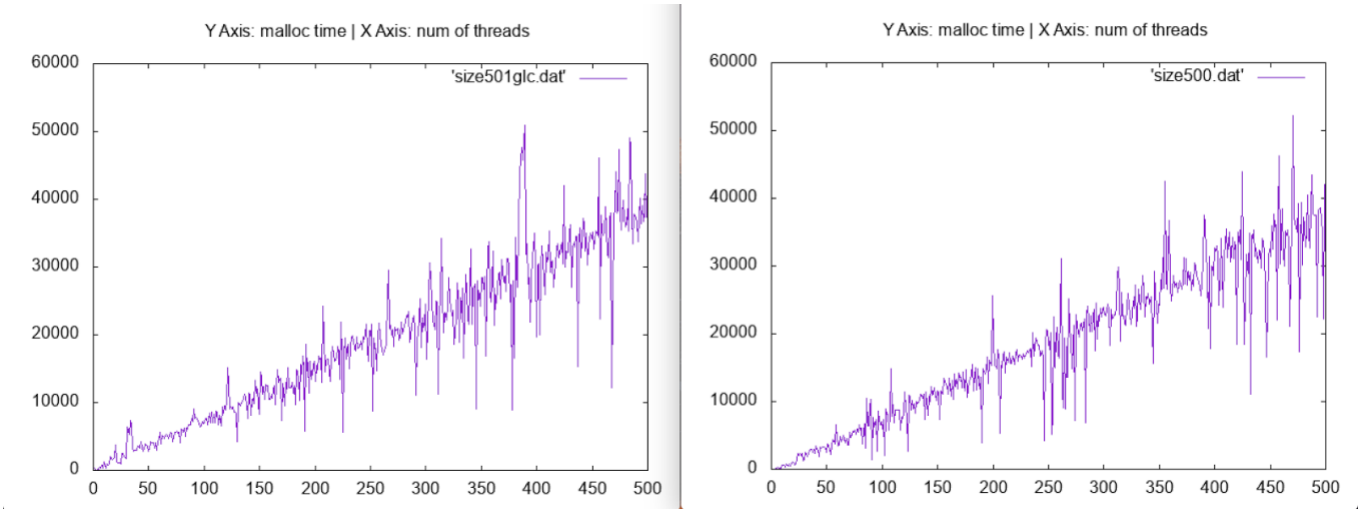

Figure 7 Speed Comparison up to 500 threads (left: glibc, right: speedyLoc)

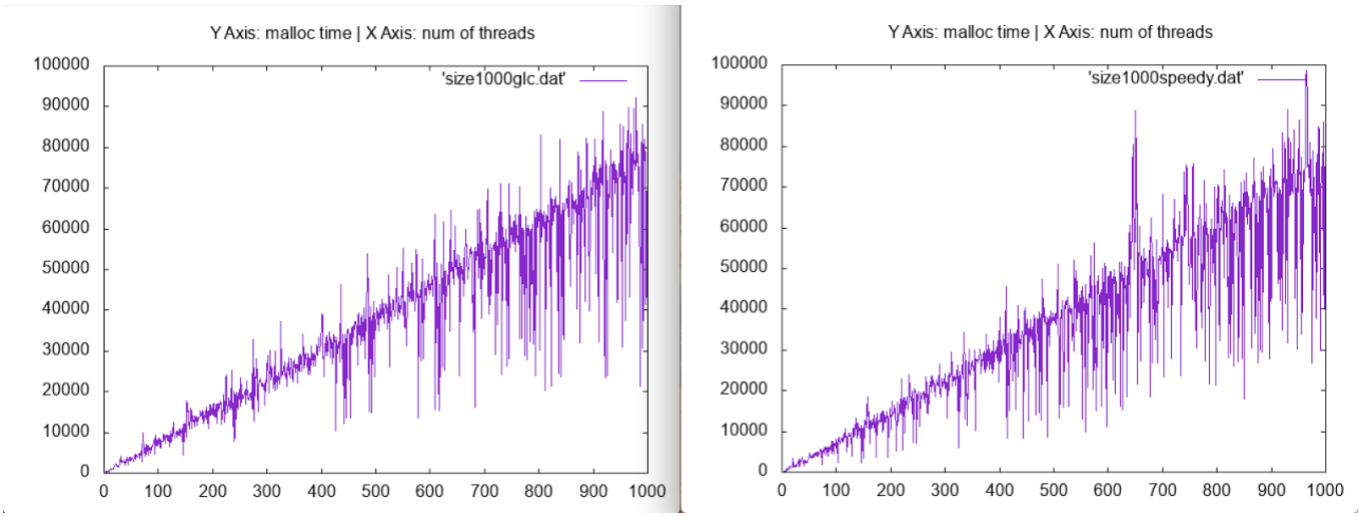

Figure 8 Speed Comparison up to 1000 threads (left: glibc, right: speedyLoc)