CHAPTER 7

Scalable Memory Allocation

This chapter discusses a *critical* part of any parallel program: scalable memory allocation, which includes use of `new` as well as explicit calls to `malloc`, `calloc`, and so on. Scalable memory allocation can be used regardless of whether we use any other part of Threading Building Blocks (TBB). In addition to interfaces to use directly, TBB offers a "proxy" method to automatically replace C/C++ functions for dynamic memory allocation, which is an easy, effective, and popular way to get a performance boost without any code changes. This is important and workvs regardless of how "modern" you are in your usage of C++, specifically whether you use the modern and encouraged `std::make_shared`, or the now discouraged `new` and `malloc`. The performance benefits of using a scalable memory allocator are significant because they directly address issues that would otherwise limit scaling and risk false sharing. TBB was among the first widely used scalable memory allocators, in no small part because it came free with TBB to help highlight the importance of including memory allocation considerations in any parallel program. It remains extremely popular today and is one of the best scalable memory allocators available.

Modern C++ programming (which favors smart pointers), combined with parallel thinking, encourages us to use TBB scalable memory allocators explicitly with `std::allocate_shared` or implicitly with `std::make_shared`.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_7

Modern C++ Memory Allocation

While performance is especially interesting for parallel programming, *correctness* is a critical topic for *all* applications. Memory allocation/deallocation issues are a significant source of bugs in applications, and this has led many additions to the C++ standard and a shift in what is considered modern C++ programming!

Modern C++ programming *encourages* use of managed memory allocation with introduction of smart pointers in C++11 (`make_shared`, `allocate_shared`, etc.) and *discourages* extensive use of `malloc` or `new`. We have used `std::make_shared` in examples since the very first chapter of this book. The addition of `std::aligned_alloc` in C++17 provides for cache alignment to avoid false sharing but does not address scalable memory allocation. Many additional capabilities are in the works for C++20, but without explicit support for scalability.

TBB continues to offer this critical piece for parallel programmers: *scalable memory allocation*. TBB does this in a fashion that fits perfectly with all versions of C++ and C standards. The heart and soul of the support in TBB can be described as *memory pooling by threads*. This pooling avoids performance degradations caused by memory allocations that do not seek to avoid unnecessary shifting of data between caches. TBB also offers scalable memory allocation combined with cache alignment, which offers the scalable attribute above what one can expect from simply using `std::aligned_alloc`. Cache alignment is not a default behavior because indiscriminate usage can greatly expand memory usage.

As we will discuss in this chapter, the use of scalable memory allocation can be critical to performance. `std::make_shared` does not provide for the specification of an allocator, but there is a corresponding `std::allocate_shared`, which does allow specification of an allocator.

This chapter focuses on scalable memory allocators, which should then be used in whatever manner of C++ memory allocation is chosen for an application. Modern C++ programming, with parallel thinking, would encourage use to use `std::allocate_shared` explicitly with TBB scalable memory allocators, or use `std::make_shared` implicitly with TBB by overriding the default `new` to use the TBB scalable memory allocator. Note, `std::make_shared` is not affected by the `new` operator for a particular class because it actually allocates a larger block of memory to handle both the contents for a class and its extra space for bookkeeping (specifically, the atomic that is added to make it a smart pointer). That is why overriding the default `new` (to use the TBB allocator) will be sufficient to affect `std::make_shared`.

Manner of Use	Summary	Figure listing interfaces
C/C++ proxy	Most popular usage. Automatic replacements of standard memory allocation methods. No code changes required.	**Figure 7.4** has a list of functions replaced by the proxy library.
C++ classes	C++ standard interfaces (`std:allocator`).	List of classes in **Figure 7.12**.
C++ classes	C++ standard interfaces (`std:allocator`).	List of classes in **Figure 7.14**.
Performance optimization tweaks	Ways to tweak performance (across any manner of usage) to meet particular needs, including use of large pages. Useful when optimizing for the ultimate in performance.	Functional interfaces and an environment variable listed in **Figure 7.18**.

Figure 7-1. *Ways to use the TBB scalable memory allocator*

Scalable Memory Allocation: What

This chapter is organized to discuss the scalable memory capabilities of TBB in four categories as listed in Figure 7-1. Features from all four categories can be freely mixed; we break them into categories only as a way to explain all the functionality. The C/C++ proxy library is by far the most popular way to use the scalable memory allocator.

The scalable memory allocator is cleanly separate from the rest of TBB so that our choice of memory allocator for concurrent usage is independent of our choice of parallel algorithm and container templates.

Scalable Memory Allocation: Why

While most of this book shows us how to improve our programs speed by doing work in parallel, memory allocations and deallocations that are not thread-aware can undo our hard work! There are two primary issues at play in making careful memory allocation critical in a parallel program: contention for the allocator and cache effects.

When ordinary, nonthreaded allocators are used, memory allocation can become a serious bottleneck in a multithreaded program because each thread competes for a global lock for each allocation and deallocation of memory from a single global heap. Programs that run this way are not scalable. In fact, because of this contention, programs that make intensive use of memory allocation may actually slow down as the number of processor cores increases! Scalable memory allocators solve this by using more sophisticated data structures to largely avoid contention.

The other issue, caching effects, happens because the use of memory has an underlying mechanism in hardware for the caching of data. Data usage in a program will therefore have an implication on where data needs to be cached. If we allocate memory for thread B and the allocator gives us memory that was recently freed by thread A, it is highly likely that we are inadvertently causing data to be copied from cache to cache, which may reduce the performance of our application needlessly. Additionally, if memory allocations for separate threads are placed too closely together they can share a cache line. We can describe this sharing as *true sharing* (sharing the same object) or *false sharing* (no objects are shared, but objects happen to fall in the same cache line). Either type of sharing can have particularly dramatic negative consequences on performance, but *false sharing* is of particular interest because it can be avoided since no sharing was intended. Scalable memory allocators avoid false sharing by using class `cache_aligned_allocator<T>` to always allocate beginning on a cache line and maintaining per-thread heaps, which are rebalanced from time to time if needed. This organization also helps with the prior contention issue.

The benefits of using a scalable memory allocator can easily be a 20-30% performance, and we have even heard of 4X program performance in extreme cases by simply relinking with a scalable memory allocator.

Avoiding False Sharing with Padding

Padding is needed if the internals of a data structure cause issues due to false sharing. Starting in Chapter 5, we have used a histogram example. The buckets of the histogram and the locks for the buckets are both possible data structures which are packed tightly enough in memory to have more than one task updating data in a single cache line.

The idea of padding, in a data structure, is to space out elements enough that we do not share adjacent elements that would be updated via multiple tasks.

Regarding false sharing, the first measure we have to take is to rely on the `tbb::cache_aligned_allocator`, instead of `std::allocator` or `malloc`, when declaring the shared histogram (see Figure 5-20) as shown in Figure 7-2.

```
std::vector<tbb::atomic<int>,
tbb::cache_aligned_allocator<tbb::atomic<int>>> hist_p(num_bins);
```

Figure 7-2. *Simple histogram vector of atomics*

However, this is just aligning the beginning of the histogram vector and ensuring that `hist_p[0]` will land at the beginning of a cache line. This means that `hist_p[0]`, `hist_p[1]`, `... , hist_p[15]` are stored in the same cache line, which translates into false sharing when a thread increments `hist_p[0]` and another thread increments `hist_p[15]`. To solve this issue, we need to assure that each position of the histogram, each bin, is occupying a full cache line, which can be achieved using a padding strategy shown in Figure 7-3.

```
struct bin {    // sizeof(bin) = 64 bytes (cache line size)
  tbb::atomic<int> count;  // 4 bytes
  uint32_t padding[15];    // 60 bytes
};
std::vector<bin, tbb::cache_aligned_allocator<bin>> hist_p(num_bins);
for (size_t i = r.begin(); i < r.end(); ++i)
   hist_p[image[i]].count++;
```

Figure 7-3. *Getting rid of false sharing using padding in the histogram vector of atomics*

As we can see in Figure 7-3, the array of bins, `hist_p`, is now a vector of `structs`, each one containing the atomic variable, but also a dummy array of 60 bytes that will fill the space of a cache line. This code is, therefore, architecture dependent. In nowadays Intel processors, the cache line is 64 bytes, but you can find false sharing safe implementations that assume 128 bytes. This is because cache prefetching (caching line "`i+1`" when cache line "`i`" is requested) is a common technique, and this prefetching is somehow equivalent to cache lines of size 128 bytes.

Our false-sharing-free data structure does occupy 16 times more space than the original one. It is yet another example of the space-time trade-off that frequently arises in computer programming: now we occupy more memory, but the code is faster. Other examples are smaller code vs. loop unrolling, calling functions vs. function inlining, or processing of compressed data vs. uncompressed data.

Wait! was not the previous implementation of the bin struct a bit pedestrian? Well, it certainly was! A less hardwired solution would be this one:

```
struct bin {    // sizeof(bin) = 64 bytes (cache line size)
  tbb::atomic<int> count;            // 4 bytes
  uint8_t padding[64-sizeof(count)]; // 60 bytes
};
```

Since `sizeof()` is evaluated at compile time, we can use the same struct for other padded data structures in which the actual payload (count in this case) has a different size. But we know a better solution that is available in the C++ standard:

```
struct bin {    // sizeof(bin) = 64 bytes (cache line size)
  alignas(64) tbb::atomic<int> count;
};
std::vector<bin, tbb::cache_aligned_allocator<bin>>
    hist_p(num_bins);
```

This warrants that each bin of `hist_p` is occupying a full cache line thanks to the `alignas()` method. Just one more thing! We love to write portable code, right? What if in a different or future architecture cache line size is different. No problem, the C++17 standard has the solution we are looking for:

```
struct bin {  // sizeof(bin) = cache line (implement. defined)
  // available starting in C++17
  alignas(std::hardware_destructive_interference_size)
     tbb::atomic<int> count;
};
```

Great, assuming that we have fixed the *false sharing* problem, what about the true sharing one?

Two different threads will eventually increment the same bin, which will be ping-pong from one cache to other. We need a better idea to solve this one! We showed how to deal with this in Chapter 5 when we discussed *privatization and reduction*.

Scalable Memory Allocation Alternatives: Which

These days, TBB is not the only option for scalable memory allocations. While we are very fond of it, we will introduce the most popular options in this section. When using TBB for parallel programming, it is essential that we use a scalable memory allocator whether it is the one supplied by TBB or another. Programs written using TBB can utilize any memory allocator solution.

TBB was the first popular parallel programming method to promote scalable memory allocation alongside the other parallel programming techniques because the creators of TBB understood the importance of including memory allocation considerations in any parallel program. The TBB memory allocator remains extremely popular today and is definitely still one of the best scalable memory allocators available.

The TBB scalable memory allocator can be used regardless of whether we use any other part of Threading Building Blocks (TBB). Likewise, TBB can operate with any scalable memory allocator.

The most popular alternatives to the TBB scalable memory allocator are `jemalloc` and `tcmalloc`. Like the scalable memory allocator in TBB, there are alternatives to `malloc` that emphasize fragmentation avoidance while offering scalable concurrency support. All three are available open source with liberal licensing (BSD or Apache).

There are some people who will tell you that they have compared `tbbmalloc` for their application with `tcmalloc` and `jeamalloc` and have found it to be superior for their application. This is very common. However, there are some people who choose `jemalloc` or `tcmalloc` or `llalloc` even when using the rest of TBB extensively. This works too. The choice is yours to make.

`jemalloc` is the FreeBSD `libc` allocator. More recently, additional developer support features such as heap profiling and extensive monitoring/tuning hooks have been added. `jemalloc` is used by Facebook.

`tcmalloc` is part of Google's `gperftools`, which includes `tcmalloc` and some performance analysis tools. `tcmalloc` is used by Google.

`llalloc` from Lockless Inc. is available freely as an open-source lockless memory allocator or can be purchased for use with closed-source software.

The behavior of individual applications, and in particular patterns of memory allocations and releases, make it impossible to pick a single fits-all winner from these options. We are confident that any choice of `TBBmalloc, jemalloc, and tcmalloc` will be far superior to a default `malloc` function or `new` operator if they are of the nonscalable variety (FreeBSD uses `jemalloc` as its default malloc).

Compilation Considerations

When compiling with programs with the Intel compilers or `gcc`, it is best to pass in the following flags:

`-fno-builtin-malloc` (on Windows: `/Qfno-builtin-malloc`)

`-fno-builtin-calloc` (on Windows: `/Qfno-builtin-calloc`)

`-fno-builtin-realloc` (on Windows: `/Qfno-builtin-realloc`)

`-fno-builtin-free` (on Windows: `/Qfno-builtin-free`)

This is because a compiler may make some optimizations assuming it is using its own built-in functions. These assumptions may not be true when using other memory allocators. Failure to use these flags may not cause a problem, but it is not a bad idea to be safe. It might be wise to check the compiler documentation of your favorite compiler.

Most Popular Usage (C/C++ Proxy Library): How

Using the proxy methods, we can globally replace `new/delete` and `malloc/calloc/realloc/free/etc`. routines with a dynamic memory interface replacement technique. This automatic way to replace `malloc` and other C/C++ functions for dynamic memory allocation is by far the most popular way to use the TBB scalable memory allocator capabilities. It is also very effective.

We can replace `malloc/calloc/realloc/free/` etc. (see Figure 7-4 for a complete list) and `new/delete` by using the `tbbmalloc_proxy` library. Using this method is easy and sufficient for most programs. The details of the mechanism used on each operating system vary a bit, but the net effect is the same everywhere. The library names are shown in Figure 7-5; a summary of the methods is shown in Figure 7-6.

	Linux	macOS	Windows
Replaceable global C++ operators `new` and `delete`	YES	YES	YES
Standard C library functions: `malloc`, `calloc`, `realloc`, `free`	YES	YES	YES
Standard C library functions (added in C11): `aligned_alloc`	YES		
Standard POSIX* function: `posix_memalign`	YES	YES	
Depending on the platform, other functions are also replaced (a list, current as of publication, follows below – any additions/changes will be in the TBB Developer Guide)			
GNU C library (glibc) specific functions: `malloc_usable_size`, `__libc_malloc`, `__libc_calloc`, `__libc_memalign`, `__libc_free`, `__libc_realloc`, `__libc_pvalloc`, `__libc_valloc`	YES		
Microsoft* C run-time library functions: `_msize`, `_aligned_malloc`, `_aligned_realloc`, `_aligned_free`, `_aligned_msize`			YES
`valloc`			
`malloc_size`		YES	
`memalign`, `pvalloc`, `mallopt`	YES		

Figure 7-4. *List of routines replaced by proxy*

	Release version library	**Debug version library**
Linux	`libtbbmalloc_proxy.so.2`	`libtbbmalloc_proxy_debug.so.2`
macOS	`libtbbmalloc_proxy.dylib`	`libtbbmalloc_proxy_debug.dylib`
Windows	`tbbmalloc_proxy.dll`	`tbbmalloc_proxy_debug.dll`

Figure 7-5. *Names of the proxy library*

	Injection "by proxy"
Linux	`LD_PRELOAD` (**Figure 7.7**)
macOS	`DYLD_INSERT_LIBRARIES` (**Figure 7.7**)
Windows	See **Figure 7.8**

Figure 7-6. *Ways to use the proxy library*

Linux: malloc/new Proxy Library Usage

On Linux, we can do the replacement either by loading the proxy library at program load time using the LD_PRELOAD environment variable (without changing the executable file, as shown in Figure 7-7), or by linking the main executable file with the proxy library (-ltbbmalloc_proxy). The Linux program loader must be able to find the proxy library and the scalable memory allocator library at program load time. For that, we may include the directory containing the libraries in the LD_LIBRARY_PATH environment variable or add it to /etc/ld.so.conf. There are two limitations for dynamic memory replacement: (1) glibc memory allocation hooks, such as __malloc_hook, are not supported, and (2) Mono (an open source implementation of Microsoft's .NET Framework based) is not supported.

macOS: malloc/new Proxy Library Usage

On macOS, we can do the replacement either by loading the proxy library at program load time using the DYLD_INSERT_LIBRARIES environment variable (without changing the executable file, as shown in Figure 7-7), or by linking the main executable file with the proxy library (-ltbbmalloc_proxy). The macOS program loader must be able to find the proxy library and the scalable memory allocator library at program load time. For that, we may include the directory containing the libraries in the DYLD_LIBRARY_PATH environment variable.

```
# On Linux:
export LD_PRELOAD=libtbbmalloc_proxy.so.2

# On macOS:
export DYLD_INSERT_LIBRARIES=$TBBROOT/lib/libtbbmalloc_proxy.dylib

# There is no simple "DLL injection" on Windows, although
# there is a Wikipedia article "DLL injection" discussing
# more complicated ways to inject a DLL.  We recommend using
# the methods we outline in Figure 7.8 instead.
```

Figure 7-7. *Environment variables to inject the TBB scalable memory allocator*

Implementation insight for the curious (not required reading): TBB has a clever way of overcoming the fact that using `DYLD_INSERT_LIBRARIES` requires using flat namespaces in order to access the shared library symbols. Normally, if an application was built with two-level namespaces, this method would not work, and forcing usage of flat namespaces would likely lead to a crash. TBB avoids this by arranging things such that when the `libtbbmalloc_proxy` library is loaded into the process; its static constructor is called and registers a *malloc zone* for TBB memory allocation routines. This allows redirecting memory allocation routine calls from a standard C++ library into TBB scalable allocator routines. This means that the application does not need to use TBB `malloc` library symbols; it continues to call standard `libc` routines. Thus, there are no problems with namespaces. The macOS *malloc zones* mechanism also allows applications to have several memory allocators (e.g., used by different libraries) and manage memory correctly. This guarantees that Intel TBB will use the same allocator for allocations and deallocations. It is a safeguard against crashes due to calling a deallocation routine for a memory object allocated from another allocator.

Windows: malloc/new Proxy Library Usage

On Windows, we must modify our executable. We can either force the proxy library to be loaded by adding an `#include` in our source code, or use certain linker options as shown in Figure 7-8. The Windows program loader must be able to find the proxy library and the scalable memory allocator library at program load time. For that, we may include the directory containing the libraries in the `PATH` environment variable.

Including `tbbmalloc_proxy.h>` to a source of any binary (which is loaded during application startup):

```
#include <tbb/tbbmalloc_proxy.h>
```

or add the following parameters to the linker options for the binary (which is loaded during application startup). They can be specified for the EXE file or a DLL that is loaded upon application startup:

For win32:
```
tbbmalloc_proxy.lib /INCLUDE:"___TBB_malloc_proxy"
```
For win64:
```
tbbmalloc_proxy.lib /INCLUDE:"__TBB_malloc_proxy"
```

Figure 7-8. *Ways to use the proxy library on Windows (note: win32 has an additional underscore vs. win64)*

Testing our Proxy Library Usage

As a simple double check to see that our program is taking advantage of a faster allocation, we can use the test program in Figure 7-9 on a multicore machine. In Figure 7-10, we show how we run this little test and the timing differences we saw on a quadcore virtual machine running Ubuntu Linux. In Figure 7-11, we show how we run this little test and the timing difference we saw on a quadcore iMac. On Windows, using the Visual Studio "Performance Profiler" on a quadcore Intel NUC (Core i7) we saw times of 94ms without the scalable memory allocator and 50ms with it (adding `#include <tbb/tbbmalloc_proxy.h>` into `tbb_mem.cpp`). All these runs show how this little test can verify that the injection of the scalable memory allocator is working (for `new`/`delete`) and yielding nontrivial performance boosts! A trivial change to use `malloc()` and `free()` instead shows similar results. We include it as `tbb_malloc.cpp` in the sample programs download associated with this book.

The example programs do use a lot of stack space, so "`ulimit -s unlimited`" (Linux/macOS) or "`/STACK:10000000`" (Visual Studio: Properties > Configuration Properties > Linker > System > Stack Reserve Size) will be important to avoid immediate crashes.

```
#include <cstdio>
#include <tbb/tbb.h>

const int N = 1000000;
double *a[N];

int main() {
  tbb::parallel_for( 0, N-1, [&](int i) { a[i] = new double; } );
  tbb::parallel_for( 0, N-1, [&](int i) { delete a[i];       } );
  return 0;
}
```

Figure 7-9. *Small test program (`tbb_mem.cpp`) for speed of `new`/`delete`*

```
% g++ -o tbb_mem tbb_mem.cpp -std=c++11 -ltbb -O3
% time ./tbb_mem

real    0m0.160s
user    0m0.072s
sys     0m0.048s
%
% export LD_PRELOAD=$TBBROOT/lib/libtbbmalloc_proxy.so.2
or alternativelly
% g++ -o tbb_mem tbb_mem.cpp -std=c++11 -ltbb -O3 -ltbbmalloc_proxy
% time ./tbb_mem

real    0m0.043s
user    0m0.048s
sys     0m0.028s
```

Figure 7-10. *Running and timing `tbb_mem.cpp` on a quadcore virtual Linux machine*

```
% time ./tbb_mem

real    0m0.046s
user    0m0.078s
sys     0m0.053s
%
% export DYLD_INSERT_LIBRARIES=$TBBROOT/lib/libtbbmalloc_proxy.dylib
%
% time ./tbb_mem

real    0m0.019s
user    0m0.032s
sys     0m0.009s
```

Figure 7-11. *Running and timing tbb_mem.cpp on a quadcore iMac (macOS)*

Family 1	`void *scalable_malloc (size_t size)`	`malloc` analogue.
	`void scalable_free (void *ptr)`	`free` analogue.
	`void *scalable_realloc (void *ptr, size_t size)`	`realloc` analogue.
	`void *scalable_calloc (size_t nobj, size_t size)`	`calloc` analogue complementing `scalable_malloc.`
	`int scalable_posix_memalign (void **memptr, size_t alignment, size_t size)`	`posix_memalign` analogue.
Family 2	`void *scalable_aligned_malloc (size_t size, size_t alignment)`	`posix_memalign` analogue.
	`void *scalable_aligned_realloc (void *ptr, size_t size, size_t alignment)`	`realloc` analogue complementing `scalable_malloc`
	`void scalable_aligned_free (void *ptr)`	`free` analogue for a previously allocated by `scalable_aligned_malloc` or `scalable_aligned_remalloc`
All Families	`size_t scalable_msize (void *ptr)`	`msize/malloc_size/malloc_usable_size` analogue. Returns the usable size of a memory block previously allocated by `scalable_x`, or `0` (zero) if `ptr` does not point to such a block.
	`int scalable_allocation_mode (int param, intptr_t value)`	Set TBB allocator-specific allocation modes. *Discussed in a section titled "Performance Tuning: some control knobs" near the end of this chapter.*
	`int scalable_allocation_command (int cmd, void *param)`	Call TBB allocator-specific commands. *Discussed in a section titled "Performance Tuning: some control knobs" near the end of this chapter.*

Figure 7-12. *Functions offered by the TBB scalable memory allocator*

C Functions: Scalable Memory Allocators for C

A set of functions, listed in Figure 7-12, provide a C level interface to the scalable memory allocator. Since TBB programming uses C++, these interfaces are not here for TBB users – they are here for use with C code.

Each allocation routine `scalable_x` behaves analogously to a library function `x`. The routines form the two families shown in the Figure 7-13. Storage allocated by a `scalable_x` function in one family must be freed or resized by a `scalable_x` function in the same family, and not by a C standard library function. Similarly, any storage allocated by a C standard library function, or C++ `new`, should not be freed or resized by a `scalable_x` function.

These functions are defined by the specific `#include <tbb/scalable_allocator.h>`".

Family	Allocation Routine	Deallocation Routine	Analogous Library

1	`scalable_malloc`
	`scalable_calloc`
	`scalable_realloc`
		`scalable_free`	C standard library

	`scalable_posix_memalign`		POSIX
2	`scalable_aligned_malloc` `scalable_aligned_realloc`	`scalable_aligned_free`	Microsoft C run-time

Figure 7-13. *Coupling of allocate-deallocate functions by families*

`tbb::aligned_space< T, N >`	Block of space aligned sufficiently large to construct an array `T` with `N` elements of type `T`. The elements are not constructed or destroyed by this class. Somewhat analogous, but not semantically compatible, with the C++ `aligned_storage` class.
`tbb::cache_aligned_allocator< T >`	Scalable memory allocation, aligned to begin on a cache line. Helps avoid false sharing, but alignment can come at some cost in additional memory consumption.
`tbb::memory_pool_allocator< T, P >`	The class is mainly intended to enable memory pools within STL containers. This is a "preview" feature as we write this book (likely to promote to a regular feature in the future). Use `#define TBB_PREVIEW_MEMORY_POOL 1` to enable.
`tbb::scalable_allocator< T >`	Scalable memory allocation. Calling this directly will fail if the `TBBmalloc` library is not available.
`tbb::tbb_allocator< T >`	The class selects `tbb::scalable_allocator` when available, and falls back on standard `malloc` when the `TBBmalloc` library is not available. Calling this will work even if the `TBBmalloc` library is not available.
`tbb::zero_allocator< T [ , Allocator ] >`	Forwards allocation requests to `Allocator` (defaults to `tbb_allocator`) and zeros the allocation before returning it.

Figure 7-14. *Classes offered by the TBB scalable memory allocator*

C++ Classes: Scalable Memory Allocators for C++

While the proxy library offers a blanket solution to adopting scalable memory allocation, it is all based on specific capabilities that we might choose to use directly. TBB offers C++ classes for allocation in three ways: (1) allocators with the signatures needed by the C++ STL `std::allocator<T>`, (2) memory pool support for STL containers, and (3) a specific allocator for aligned arrays.

Allocators with std::allocator<T> Signature

A set of classes, listed in Figure 7-14, provide a C++ level interface to the scalable memory allocator. TBB has four template classes (`tbb_allocator, cached_aligned_allocator`, `zero_allocator, and scalable_allocator`) that support the same signatures as `std::allocator<T>` per the C++ standards. This includes supporting `<void>` in addition to `<T>`, per the C++11 and prior standards, which is deprecated in C++17 and will likely be removed in C++20. This means they can be passed as allocation routines to be used by STL templates such as `vector`. All four classes model an allocator concept that meets all the "Allocator requirements" of C++, but with additional guarantees required by the Standard for use with ISO C++ containers.

scalable_allocator

The `scalable_allocator` template allocates and frees memory in a way that scales with the number of processors. Using a `scalable_allocator` in place of `std::allocator` may improve program performance. Memory allocated by a `scalable_allocator` should be freed by a `scalable_allocator`, not by a `std::allocator`.

The `scalable_allocator` allocator template requires that the `TBBmalloc` library be available. If the library is missing, calls to the `scalable_allocator` template will fail. In contrast, if the memory allocator library is not available, the other allocators (`tbb_allocator`, `cached_aligned_allocator`, or `zero_allocator`) fall back on malloc and free.

This class is defined with `#include <tbb/scalable_allocator.h>` and is notably **not** included by the (usually) all-inclusive `tbb/tbb.h`.

tbb_allocator

The `tbb_allocator` template allocates and frees memory via the `TBBmalloc` library if it is available; otherwise, it reverts to using `malloc` and `free`. The `cache_alligned_allocator` and `zero_allocator` use `tbb_allocator`; therefore, they offer the same fall back on `malloc`, but `scalable_allocator` does not and therefore will fail if the `TBBmalloc` library is unavailable. This class is defined with `#include <tbb/tbb_allocator.h>`

zero_allocator

The `zero_allocator` allocates zeroed memory. A `zero_allocator<T,A>` can be instantiated for any class A that models the Allocator concept. The default for A is `tbb_allocator`. The `zero_allocator` forwards allocation requests to A and zeros the allocation before returning it. This class is defined with `#include <tbb/tbb_allocator.h>`.

cached_aligned_allocator

The `cached_aligned_allocator` template offers both scalability and protection against false sharing. It addresses false sharing by making sure each allocation is done on a separate cache line.

Use `cache_aligned_allocator` only if false sharing is likely to be a real problem (see Figure 7-2). The functionality of `cache_aligned_allocator` comes at some cost in space because it allocates in multiples of cache-line-size memory chunks, even for a small object. The padding is typically 128 bytes. Hence, allocating many small objects with `cache_aligned_allocator` may increase memory usage.

Trying both `tbb_allocator` and the `cache_aligned_allocator` and measuring the resulting performance for a particular application is a good idea.

Note that protection against false sharing between two objects is guaranteed only if both are allocated with `cache_aligned_allocator`. For instance, if one object is allocated by `cache_aligned_allocator<T>` and another object is allocated some other way, there is no guarantee against false sharing because `cache_aligned_allocator<T>` starts an allocation on a cache line boundary but does not necessarily allocate to the end of a cache line. If an array or structure is being allocated, since only the start of the allocation is aligned, the individual array or structure elements may land together on cache lines with other elements. An example of this, along with padding to force elements onto individual cache line, is show in Figure 7-3.

This class is defined with `#include <tbb/cache_alligned_allocator.h>`.

Memory Pool Support: `memory_pool_allocator`

Pool allocators are an extremely efficient method for providing allocation of numerous objects of fixed size *P*. Our first allocator usage is special and asks to reserve enough memory to store *T* objects of size *P*. Thereafter, when the allocator is used to provide a

chunk of memory, it returns an offset mod *P* into the allocated chunk. This is far more efficient than calling operator `new` separately for each request because it avoids the bookkeeping overhead required of a general-purpose memory allocator that services numerous requests for different-sized allocations.

The class is mainly intended to enable memory pools within STL containers. This is a "preview" feature as we write this book (likely to promote to a regular feature in the future). Use `#define TBB_PREVIEW_MEMORY_POOL 1` to enable while this is still a preview feature.

Support is provided by `tbb::memory_pool_allocator` and `tbb:: memory_pool_allocator`. These require

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include <tbb/memory_pool.h>
```

Array Allocation Support: aligned_space

This template class (`aligned_space`) occupies enough memory and is sufficiently aligned to hold an array *`T[N]`*. Elements are not constructed or destroyed by this class; the client is responsible for initializing or destroying the objects. An `aligned_space` is typically used as a local variable or field in scenarios where a block of fixed-length uninitialized memory is needed. This class is defined with `#include <tbb/aligned_space.h>`.

Replacing new and delete Selectively

There are a number of reasons one might develop custom new/delete operators, including error checking, debugging, optimization, and usage statistics gathering.

We can think of `new/delete` as coming in variations for individual objects and for arrays of objects. Additionally, C++11 defines throwing, nonthrowing, and placement versions of each of these: either the global set (`::operator new`, `::operator new[]`, `::operator delete` and `::operator delete[]`) or the class specific sets (for class `X`, we have `X::operator new`, `X::operator new[]`, `X::operator delete` and `X::operator delete[]`). Finally, C++17 adds an optional alignment parameter to all versions of new.

If we want to globally replace all the `new/delete` operators and do not have any custom needs, we would use the proxy library. This also has the benefit of replacing `malloc/free` and related C functions.

For custom needs, it is most common to overload the class-specific operators rather than the global operators. This section shows how to replace the global `new`/`delete` operators as an example which can be customized for particular needs. We show throwing and nonthrowing versions, but we did not override the placement versions since they do not actually allocate memory. We also did not implement versions with alignment (C++17) parameters. It is also possible to replace `new`/`delete` operators for individual classes using the same concepts, in which case you may choose to implement placement versions and alignment capabilities. All these are handled by TBB if the proxy library is used.

Figures 7-15 and 7-16 together show a method to replace `new` and `delete`, and Figure 7-17 demonstrates their usage. All versions of `new` and `delete` should be replaced at once, which amounts to four versions of `new` and four versions of `delete`. Of course, it is necessary to link with the scalable memory library.

Our example chooses to ignore any new handler because there are thread-safety issues, and it always throws `std::bad_alloc()`. The variation of the basic signature includes the additional parameter `const std::nothrow_t&` that means that this operator will not throw an exception but will return `NULL` if the allocation fails. These four nonthrowing exception operators can be used for C runtime libraries.

We do not have to initialize the task scheduler to be able to use the memory allocator. We do initialize it in this example because it uses `parallel_for` in order to demonstrate the use of memory allocation and deallocation in multiple tasks. Similarly, the only header file that is required for the memory allocator is `tbb/tbb_allocator.h`.

```
#include <tbb/parallel_for.h>
#include <tbb/tbb_allocator.h>

// No retry loop because we assume that
// scalable_malloc does all it takes to allocate
// the memory, so calling it repeatedly
// will not improve the situation at all
//
// No use of std::new_handler because it cannot be
// done in portable and thread-safe way
//
// We throw std::bad_alloc() when scalable_malloc
// returns NULL (we return NULL if it is a no-throw
// implementation)

void* operator new (size_t size) throw (std::bad_alloc)
{
    if (size == 0) size = 1;
    if (void* ptr = scalable_malloc (size))
        return ptr;
    throw std::bad_alloc (  );
}

void* operator new[] (size_t size) throw (std::bad_alloc)
{
    return operator new (size);
}

void* operator new (size_t size, const std::nothrow_t&) throw ()
{
    if (size == 0) size = 1;
    if (void* ptr = scalable_malloc (size))
        return ptr;
    return NULL;
}

void* operator new[] (size_t size, const std::nothrow_t&) throw ()
{
    return operator new (size, std::nothrow);
}
```

Figure 7-15. *Demonstration of replacement of new operators (`tbb_nd.cpp`)*

```
void operator delete (void* ptr) throw ()
{
    if (ptr != 0) scalable_free (ptr);
}

void operator delete[] (void* ptr) throw ()
{
    operator delete (ptr);
}

void operator delete (void* ptr, const std::nothrow_t&) throw ()
{
    if (ptr != 0) scalable_free (ptr);
}

void operator delete[] (void* ptr, const std::nothrow_t&) throw ()
{
    operator delete (ptr, std::nothrow);
}
```

Figure 7-16. *Continuation from the previous figure, replacement of delete operators*

```
int main (int argc, char** argv)
{
  const size_t size = 1000;
  const size_t chunk = 100;

  // scalable_malloc will be called to allocate
  // the memory for this array of integers
  int *p = new int[size];

  tbb::parallel_for (size_t{0}, size, [=](size_t chunk)
                     {
                       // scalable_malloc will be called
                       // to allocate the memory
                       // for this array of integers
                       int *p = new int [chunk];

                       // scalable_free will be called to
                       // deallocate the memory
                       // for this array of integers
                       delete[] p;
                     } );

  return 0;
}
```

Figure 7-17. *Driver program to demonstrate the new/delete replacements*

Performance Tuning: Some Control Knobs

TBB offers some special controls regarding allocations from the OS, huge page support, and flushing of internal buffers. Each of these is provided to fine-tune performance.

Huge pages (*large pages* on Windows) are used to improve the performance for programs that utilize a very large amount of memory. In order to use huge pages, we need a processor with support, an operating system with support, and then we need to do something so our application takes advantage of huge pages. Fortunately, most systems have all this available, and TBB includes support.

What Are Huge Pages?

In most cases, a processor allocates memory 4K bytes at a time in what are commonly called pages. Virtual memory systems use page tables to map addresses to actual memory locations. Without diving in too deep, suffice to say that the more pages of memory that an application uses, the more page descriptors are needed, and having a lot of page descriptors flying around causes performance issues for a variety of reasons. To help with this issue, modern processors support additional page sizes that are much larger than 4K (e.g., 4 MB). For a program using 2 GB of memory, 524,288 page descriptions are needed to describe the 2 GB of memory with 4K pages. Only 512 page descriptions are needed using 4 MB descriptors and only two if 1 GB descriptors are available.

TBB Support for Huge Pages

To use huge pages with TBB memory allocation, it should be explicitly enabled by calling `scalable_allocation_mode( TBBMALLOC_USE_HUGE_PAGES,1)`, or by setting the `TBB_MALLOC_USE_HUGE_PAGES` environment variable to `1`. The environment variable is useful when substituting the standard malloc routines with the `tbbmalloc_proxy` library.

These provide ways to tweak the algorithms used for all usages of the TBB scalable memory allocator (regardless of the method of usage: proxy library, C functions, or C++ classes). The functions take precedence over any environment variable settings. These are definitely not for casual use, they are here for self-proclaimed “control freaks” and offer great ways to optimize performance for particular needs. We recommend careful evaluation of the performance impact on an application, in the target environment, when using these features.

Of course, both methods assume that the system/kernel is configured to allocate huge pages. The TBB memory allocator also supports pre-allocated and transparent huge pages, which are automatically allocated by the Linux kernel when suitable. Huge pages are not a panacea; they can have negative impact on performance if their usage is not well considered.

The functions, as listed in Figure 7-18, are defined with `#include <tbb/tbb_allocator.h>`.

`int scalable_allocation_mode (int mode, intptr_t value)`
`mode =`
`TBBMALLOC_USE_HUGE_PAGES or`
`TBBMALLOC_SET_SOFT_HEAP_LIMIT`
	Set TBB allocator-specific allocation modes.

`Environment variable:` `TBB_MALLOC_USE_HUGE_PAGES`	A value of "1" (one) will enable the use of huge pages by the allocator if supported by the operating system.
`int scalable_allocation_command (int cmd, void *reserved)` `reserved should be zero`	Call TBB allocator-specific commands.

Figure 7-18. *Ways to refine TBB scalable memory allocator behaviors*

scalable_allocation_mode(int mode, intptr_t value)

The `scalable_allocation_mode` function may be used to adjust the behavior of the scalable memory allocator. The arguments, described in the following two paragraphs, control aspects of behavior of the TBB allocators. The function returns `TBBMALLOC_OK` if the operation succeeded, `TBBMALLOC_INVALID_PARAM` if mode is not one of those described in the following subsections, or if value is not valid for the given mode. A return value of `TBBMALLOC_NO_EFFECT` is possible for conditions described when they apply (see explanation of each function).

TBBMALLOC_USE_HUGE_PAGES

`scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES,1)`

This function enables the use of huge pages by the allocator if supported by the operating system; a zero as the second parameter disables it. Setting the TBB_MALLOC_USE_HUGE_PAGES environment variable to one has the same effect as calling `scalable_allocation_mode to` enable this mode. The mode set with `scalable_allocation_mode`

takes priority over the environment variable. The function will return `TBBMALLOC_NO_EFFECT` if huge pages are not supported on the platform.

TBBMALLOC_SET_SOFT_HEAP_LIMIT

```
scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)
```

This function sets a threshold of `size` bytes on the amount of memory the allocator takes from the operating systems. Exceeding the threshold will urge the allocator to release memory from its internal buffers; however, it does not prevent the TBB scalable memory allocator from requesting more memory when needed.

int scalable_allocation_command(int cmd, void *param)

The `scalable_allocation_command` function may be used to command the scalable memory allocator to perform an action specified by the first parameter. The second parameter is reserved and must be set to zero. The function will return `TBBMALLOC_OK` if the operation succeeded, `TBBMALLOC_INVALID_PARAM` if `reserved` is not equal to zero, or if `cmd` is not a defined command (`TBBMALLOC_CLEAN_ALL_BUFFERS` or `TBBMALLOC_CLEAN_THREAD_BUFFERS`). A return value of `TBBMALLOC_NO_EFFECT` is possible as we describe next.

TBBMALLOC_CLEAN_ALL_BUFFERS

```
scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)
```

This function cleans internal memory buffers of the allocator and possibly reduces memory footprint. It may result in increased time for subsequent memory allocation requests. The command is not designed for frequent use, and careful evaluation of the performance impact is recommended. The function will return `TBBMALLOC_NO_EFFECT` if no buffers were released.

TBBMALLOC_CLEAN_THREAD_BUFFERS

```
scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)
```

This function cleans internal memory buffers but only for the calling thread. It may result in increased time for subsequent memory allocation requests; careful evaluation of the

performance impact is recommended. The function will return `TBBMALLOC_NO_EFFECT` if no buffers were released.

Summary

Using a scalable memory allocator is an essential element in any parallel program. The performance benefits can be very significant. Without a scalable memory allocator, serious performance issues often arise due to contention for allocation, false sharing, and other useless cache to cache transfers. The TBB scalable memory allocation (`TBBmalloc`) capabilities include use of `new` as well as explicit calls to `malloc`, and so on, all of which can be used directly or they can all be automatically replaced via the proxy library capability of TBB. The scalable memory allocation in TBB can be used regardless of whether we use any other part of TBB; the rest of TBB can be used regardless of which memory allocator is used (`TBBmalloc, tcmalloc, jemalloc, malloc`, etc.). The `TBBmalloc` library remains extremely popular today and is definitely one of the best scalable memory allocators available.