CS61C Summer 2013
Project 2 Extra Credit

TA: Albert Magyar

August 4, 2013

Revision History

All changes throughout the document added after the initial release are in red.

- 07/30: specification released
- 07/30: revised performance target to 25GFLOPS

1 Introduction

The second part of Project 2 involves splitting up the work of matrix multiplication across multiple threads executing in parallel on a multiprocessor workstation. While this requires a strong understanding of how the multiprocessor caching model plays into an efficient memory access patter, the details of creating new threads beyond the first and successfully causing them to work in parallel are hidden behind the "magic" of OpenMP. The goal of this extra credit assignment is to give you a taste of programming in a way that directly takes advantage of the interface provided by threads, rather than using OpenMP compiler directives to produce multi-threaded executables.

Sections 1-3 of this document are meant to give a high-level view of this assignment and clearly establish the specifications, while Section 4 is a body of reference reading that will be useful for implementing the project. Section 5 lists external resources that devote far more space to explaining the concepts touched on by this project in much greater depth than is found here.

2 Objective

Your objective is to implement a parallelized matrix multiplication algorithm that achieves an average performance of 25GFLOPS for matrices fitting the specifications for valid inputs for the second part of the Matrix Multiply project. This algorithm must be implemented in the `sgemm` function defined in a file named `sgemm-threads.c`. The prototype for this function is given below.

```
void sgemm(int m_a, int n_a, float *A, float *B, float *C);
```

3 Instructions & Rules

To get started, copy the skeleton files to your working directory with `cp ~cs61c/proj/su13_proj2-ec/* .` at the terminal. Entering the `make bench-threads` command will produce an executable, `bench-thread`,

which is run in the same way as `bench-test` for Part 2 of the project (`bench-threads` ⟨`M`⟩ ⟨`N`⟩).

To submit, run `submit proj2-ec` in a directory containing your `sgemm-threads.c` file.

Please do not post any code examples publicly on Piazza. Asking about the behavior or operation of a certain function is fine.

You are allowed to use the following tools to implement this function:

- Any built-in component of the `gnu99` C language standard
- The C Standard Library
- Any part of the `Pthreads` interface usable by including `pthread.h`
- Any Intel SSE (not AVX) Intrinsic, with the exception of aligned loads and/or stores

You are specifically disallowed from using any of the following tools:

- OpenMP compiler directives ("pragmas")
- Any function or variable declared in `omp.h`
- Aligned loads or stores
- Intel AVX Intrinsics
- Modifications to the supplied Makefile
- Any external library beyond those already mentioned

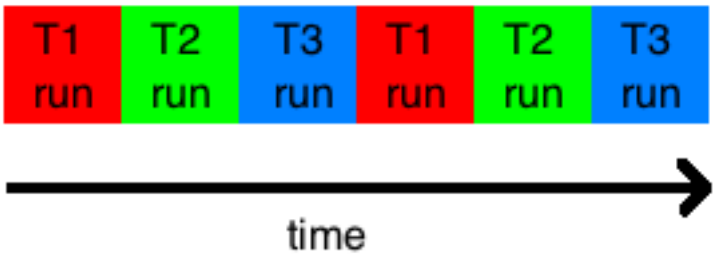
Many of the Pthreads API functions may return values indicating failure. If this happens, make a call to the `thread_api_failure` function in the `sgemm-threads.c` file; this call is reminiscent to `allocation_failed` from Project 1. Note that these calls should essentially never fail in normal conditions; much like malloc, the vast majority of cases where failure occurs reflect an underlying problem in the program's implementation.

4 Useful Information

This section is *not* a specification for any requirements of this part of the project. Any submission meeting the preceding criteria will be accepted for credit. However, this assignment involves many design decisions, most of which revolve around using the interface provided by the general model of processes and threads and the specific API used for this extra credit task, POSIX Threads (PThreads/pthreads/`pthread`). Hopefully, most of the following information will be useful for learning about the tools needed to put this project together; otherwise, the resources listed in the Additional References section provide far more detail.

4.1 What Is a Thread and What is a Process?

A thread is the basic "unit of execution" of code on a computer system. From the discussion of thread-level parallelism in CS61C so far, it is clear that multiple threads may execute at once on a computer, each with its own instruction stream. In fact, the number of threads that may execute on a computer system at once is completely decoupled from the number of processors or cores it has! A single-processor CPU accomplishes "simultaneous" execution of an arbitrary number of threads by *multiplexing in time*, or spending short, interleaved *time slices* executing the instruction stream of each thread, as shown below.



However, the execution of an instruction stream depends on properly maintaining all of its state, regardless of what else is happening on the computer. Therefore, the system must be sure to maintain each thread's state independently such that it may be "swapped in" when needed. For an example of the kind of necessary state that must he held independently per thread, consider the following set of instruction sequences that illustrates the separate actions of two threads.

```
# Thread 1
addiu $t0, $0, -8 # $t0 holds -8
...               # continues executing
```

```
# Thread 2
addiu $t0, $0, -4 # $t0 holds -4
...               # continues executing
```

Clearly, we can't just forbid multiple threads running on the same computer from using the same registers – how would function calls happen at all, if only one thread could use `$ra`? Since the two threads execute independently, even if they run the same code from the same program, they necessarily must have a different idea of what `$t0` contains, as if threads shared the same contents for the architectural set of registers, they could never successfully operate at the same time! Therefore, it makes sense that different threads have different *copies* of `$t0`, along with different copies of the other 31 MIPS architectural registers and different values for their program counters.

A process is a particular invocation of a computer program. If, for example, two users run Emacs at the same time, two *processes* will be running the same *program*. Within a process is all the code to run that particular program along with all of the state information used in executing it. Each process may consist of one thread, or each may have many threads. Although any two threads, whether from the same process or not, must have *separate* copies of their execution state. One example of information that must be independent between threads is the set of register values, discussed above. Near the end of this section, the boundaries of what is shared between different threads in the same process and what is independently maintained by each thread will be clarified. However, the group of all the threads that are part of the same process also share all the resources that are allocated *per-process* among themselves.

From the perspective of CS61C, the most important thing allocated to a process is an *address space*. Although this material has not yet been covered in lecture, each process does not see the same address space as other processes running concurrently on the same system. For example, in the following code sequences, part of the instruction stream for each of two processes is shown in MIPS assembly code. Assume for this example that each process has only one thread, and that the instruction streams below show the execution of that "main" thread for each process.

```
# Process 1
lw $t0, 16384($0) # loads from 0x00004000
sw $t0, 16388($0) # stores to 0x00004004
...               # continues executing
```

```
# Process 2
lw $t1, 16384($0) # loads from 0x00004000
sw $t1, 16388($0) # stores to 0x00004004
...               # continues executing
```

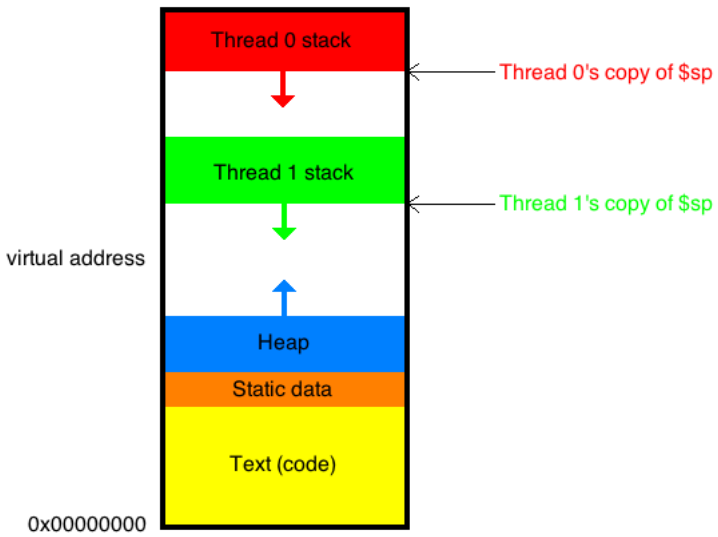
Uh-oh. Each process writes different information to the same address! One potential fix is to prevent programs from writing to the same address; however, beyond being generally impractical, this will instantly break if two processes are separate invocations of the same program. When building an executable a given program, the linker replaces immediates in expanded `la` pseudoinstructions with the addresses of statically-allocated data. If a user runs two copies of the same program that uses such statically-allocated space, each process will necessarily contain accesses to the same address, some of which may be writes!

To address this issue, computer system designers have come up with the concept of *virtual memory*. Fundamentally, we like to portray memory as an array, and accessing a particular address as looking at the corresponding index of an array. In a virtual memory system, different processes are presented with entirely different arrays! More specificially, when threads from the same process attempt to access memory, the address of each access is used to find a location within a common `virtual address space` or logical "array" of memory. When threads from different processes attempt to access memory, the hardware and operating system work together to ensure that the address for each of their requests gets translated into a location in a different virtual address space. The end result is that each process is presented with the view (or abstraction) that it has its own "copy" of memory, in which it may do whatever it pleases with any address where space has been allocated (say with `malloc`), without the interference of other processes.

With all these different memory arrays floating around, it's hard to imagine where they all get stored, as the computer only has a fixed amount of "real", or *physical* memory. Indeed, the way in which the system figures out where to store the contents of each view of memory is a little bit complicated. Fortunately, it's outside the scope of this assignment; unfortunately, it's coming up in lecture very soon!

For this project, you will be writing code that executes on multiple threads within a single process. Therefore, it is okay to have only the basic facts about what resources one process is allocated independently from all others, as your concern will remain entirely within one process. However, it is important to realize that how virtual address spaces are allocated at the process level, rather than at the thread level. If one thread in the process running your multithreaded matrix multiplication program performs a store to a given address, all other threads in that process will see the results.

On the other hand, it is also important to know what is allocated separately for each thread. Since each thread executes independently, it must have its own copy of every architectural register. Since `$sp` is one of these registers, and the structure of the call stack reflects which function a thread is executing at any given moment, it makes sense that different threads have different stacks pointed to by different stack pointers. But how does this work when all the threads share the same address space?



Here, we see that the threads have different stacks residing at different ranges of addresses within the same virtual address space. Therefore, when one thread enters a function and encounters a statement defining a local variable, it will allocate space by decrementing *its own stack pointer*. While this is independent of any copies of that local variable on other threads' stacks (and also independent of copies of the variable in other frames from other calls of the same function on the same stack), if another thread somehow got hold of the address of or a pointer to this variable, it would be able to modify it, and the thread owning that stack would see the results of the modification, as they share address spaces! When the CPU switches between threads in an operation known as a *context switch*, it brings in all the appropriate copies of the architectural registers for that thread. Therefore, the stack pointer register always points at the right stack for the currently running thread.

Within one process, all threads share a common heap to use as a pool for dynamic memory allocation. This means that blocks allocated with `malloc` are often used as shared variables within the shared address space of the process. However, it is important to note that most pointers to blocks allocated by `malloc` used in code from CS61C so far are local variables themselves! This often necessitates designing your program so that a copy of all relevant pointers to shared data (say, `A`, `B`, or `C`) makes it to each thread. Mechanisms for doing this are discussed in the next section.

In OpenMP, the model of shared and private data is such that any variable name declared outside a parallel block is within a conceptual scope that is shared across threads, while any variable name declared within a parallel block is private to a thread, unless other clauses are used to change this behavior. While this is nice and consistent, it violates the perspective of the normal thread model, as many variables shared between OpenMP threads are declared in a local scope, and therefore would reside on different stacks in a normal multithreaded program!

In summary, the following things are shared among threads within a process:

- An address space
- A heap segment within that address space
- Variables in global or file scope

While the following things are not:

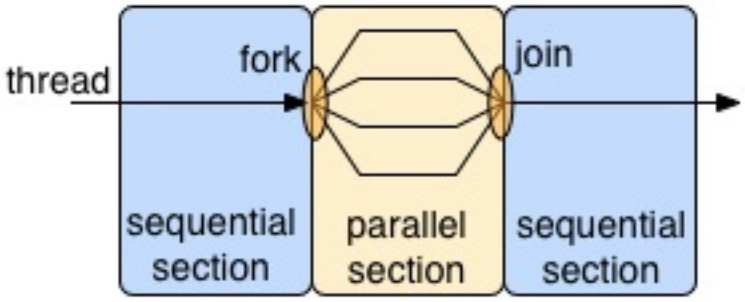
- The stack of each thread

- Local variables *including pointers to blocks on the heap*
- Registers

4.2 The Fork-Join Model

As discussed in the introduction for Lab 8, parallel sections in programs using OpenMP follow a fork-join model. In these programs, a master thread creates several worker threads which split up the computationally-intensive task in parallel. This is the *fork* section, which refers to creating a group of new worker threads. An important note is that `fork`, a standard C system function, relates to creating a new process, not a new thread. Don't get confused by these complementary meanings of fork, and don't use the `fork` function in your code!

After creating these threads, the master thread *joins* each of them. The semantics of one thread joining another is that the joiner will not resume executing past its join statement until the joined thread has completed its execution. Therefore, the diagram below (from the lab documentation) illustrates the flow of the computation very well, as many threads are created at one point, and execution of the master thread resumes after all the other worker threads have terminated (finished).



The high-level outline for an algorithm employing the fork-join model depends on the semantics of the thread creation and joining interface offered by the particular API. Here, the fork-join model will be sketched out in terms of POSIX Threads.

4.3 Creating And Running New Threads

While there are many different ways to think about creating and starting the execution of new threads, the POSIX Threads interface is built in a way that takes advantage of the procedural nature of the C programming language. When a process is first created, it by default has one thread, which we will refer to as the *master thread*. Just as this thread always begins execution by entering the `main` function, we will similarly provide each additional thread we create with a function to execute upon creation. You may think about this as being analogous to the `main` of that particular thread.

To create an additional or *child* thread, the creating thread supplies the thread creation function with a pointer `f_pointer` to the function that the child thread should begin executing upon creation, along with a list of arguments `arg_list`. The return values of the thread creation function are `status`, a numeric code indicating whether an error occurred, along with a numeric value `thread_id` that other threads may later use to refer to the thread that was created.

A child thread terminates when it returns from the function it executes upon creation, just as the main thread terminates when it returns from `main`. Here, it is important to note that returning successfully from the thread creation function means that the child thread has been created, not that it has terminated; the second meaning would actually stifle our attempts at parallelism by forcing sequential execution!

Upon successful return of the thread creation function, the child thread is already active; there is no need to explicitly start it after creation. Working with function pointers, translating argument lists to a single parameter, and emulating multiple return values in C all take a little bit of work, so for now the examples will be in pseudocode. The following code is the start of a multithreaded "Hello, world!" program. Note that the child thread prints out `Hello`, while the master thread prints `world!`, though not necessarily in that order.

```
print_hello():
    print "Hello, ";
    return;

main(argc, argv):
    arg_list = { };                                                  // no arguments
    (status, thread_id) = create_thread(print_hello, arg_list); // 2 return values
    print "world!";                                                  // "world! Hello"?
    return;
```

4.4 Now Comes The Join

The above code is very limited as a "Hello, world!" program, as there is no guarantee on the *scheduling* of the two threads. In other words, while both threads are active, scheduling of the two threads can result in either thread executing before the other, interleaved execution, or even simultaneous execution. Note that this is exactly what makes thread-level parallelism useful for performance programming!

Imagine your main thread has just entered a subroutine `long_function` where it has a computationally challenging operation to perform. It wants to split the task between multiple threads, but it doesn't want to return control to the calling subroutine (the subroutine running in the main thread that called `long_function`) until *after* all of the threads have finished.

The first task is splitting up the task. The programmer has determined that one call to `long_function(a,b)` can be split up into two halves. By passing these parameters, along with some cues to indicate which half of the computation to perform, the main thread can delegate work effectively to the child thread. The (incorrect) code produced after the programmer's first stab at the problem is shown below.

```
short_function(a, b, chunk_number):
    ...  // Do half the work

long_function(a, b):
    arg_list_1 = { a, b, 1 };  // Cue: do the first half
    arg_list_2 = { a, b, 2 };  // Cue: do the second half
    (status_1, thread_id_1) = create_thread(short_function, arg_list_1);
    (status_2, thread_id_2) = create_thread(short_function, arg_list_2);
    return;  // Are they done yet?

main(argc, argv):
    a = { 1, 1, 2, 3 };
    b = { 0, 1, 0, 1};
    long_function(a,b);
```

Unfortunately, there is no guarantee that the two child threads have completed their work by the time the return statement of `long_function` is reached. The only thing that is necessarily true is that both have been created and are ready to be executed. Therefore, `long_function` may return before its work is

done!

In order to fix this, we will employ the `join` operation provided by the POSIX Threads API. For our purposes, the `join` function takes the form of a function with one parameter, which is the ID of the thread to join. As explained in the intro to the fork-join model, thread that invokes the `join(target_thread_id)` function on another thread will not return from the call to `join` until the target thread terminates.

```
short_function(a, b, chunk_number):
    ...  // Do half the work

long_function(a, b):
    arg_list_1 = { a, b, 1 };  // Cue: do the first half
    arg_list_2 = { a, b, 2 };  // Cue: do the second half
    (status_1, thread_id_1) = create_thread(short_function, arg_list_1);
    (status_2, thread_id_2) = create_thread(short_function, arg_list_2);
    join(thread_id_1);
    join(thread_id_2);
    return;  // They are done!

main(argc, argv):
    a = { 1, 1, 2, 3 };
    b = { 0, 1, 0, 1};
    long_function(a,b);
```

There is something quite interesting about the use of multiple joins. Since the call to `join(thread_id_1)` will not return until after the first child thread has terminated, and since there is no guarantee that the first child thread will finish before the second child thread, there is a chance that the call to `join(thread_id_1)` will not return until after `both` child threads have terminated. Therefore, it is convenient that a call to `join` that is given the ID of a thread that has already finished (as the second thread may have) will return instantly. However, it is nonsensical to attempt to join the same thread twice, as it is only possible to wait for the thread to finish *one time*.

4.5 The Pthreads API: Man Page Spark Notes

For more information on any of these functions, enter the command `man` ⟨`name_of_function`⟩ into the console of a Unix, Linux, or Mac machine to read the man (manual) page describing that function. Almost all of the components of the Pthreads API are very well documented in this manner; for example, `man pthread_join` brings up everything you need to know about the syntax and semantics of joins for Pthreads! Once in a man page, use the up- and down-arrow keys to scroll, and press `q` at any time to exit.

Below are descriptions of a subset of the types and functions declared in `pthread.h`; while there are many more available, this section gives you enough information to get started on (and likely finish) a working implementation of this assignment. **Much of the complexity of programming with Pthreads is in the pervasive use of `void *` to move data. Sorry to put you through that! Many of the Pthreads API functions may fail.** If this happens, make a call to the `thread_api_failure` function in the `sgemm-threads.c` file; this call is reminiscent to `allocation_failed` from Project 1. Again, full documentation for every part is available in the man page for `pthread`.

A large number of more-complicated C functions have a return values indicating whether they succeeded, while other information is stored at the addresses pointed to by pointers passed in as arguments. For example, the following (fake) example of a random-number-generating function returns its status code

– traditionally 0 for success – while writing the random number that is generated to the address pointed to by the float pointer passed as a parameter.

```
int success;
float random_number;
success = get_random_number(&random_number); // pass in (float *)
if (success) {
    printf("Random number is: %f\n", random_number);
}
```

4.5.1 Types

- **`pthread_t`**
  This is the type for thread IDs. See `pthread_create`, `pthread_self`, and `pthread_equal`.

- **`void *`**
  This type is a pointer that points to data of unspecified type. This may be used *with great care* to move a pointer to an arbitrary but agreed-upon data object (region of memory, in the C sense of the word) into a function that takes only a `void *` parameter, as shown below.

```
void print_string(void *s) {
    char *str = (char *) s;  // s had better point to a block of chars!
    printf("%s", str);
}

int main(int argc, char **argv) {
    char *hello = "Hello, World!";
    void *copy = (void *) hello;
    print_string(copy);
    return 0;
}
```

- **`void *(* function_name) (void *)`**
  Function pointer types are specific to the signature of the functions to which they point. Here, this type describes a function pointer to a function fitting the prototype `void *function_name(void *)`.

4.5.2 Functions

- **`pthread_create`**

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

  Creates a thread that executes the routine `start_routine` upon with argument `arg` upon creation, and returns 0 if the thread creation was successful. Even through the `void *(*start_routine) (void *)` part looks complicated, it just means that the function pointer passed in as the third parameter to `pthread_create` must take a single `void *` argument and have return type `void *`. In order to use a particular function as the start routine, simply type its name as the parameter.

  When the invocation of `start_routine` corresponding with the execution of the new thread begins, it receives `arg` as its only argument of type `void *`. This might seem a little restrictive, but consider the fact that you can define struct that hold all the parameters you want and have the main thread pack all the child threads' arguments up nicely in structs before using the address-of operator (`&`) and

an implicit cast to `void *` to pass them through the `arg` parameter. The prologue of the function executed by the child threads would have to reverse this to unpack the needed parameters.

The ID of the created thread is written to the `pthread_t` location pointed to by the `thread` parameter. This is an example of using extra parameters to emulate multiple return values!

The `attr` parameter controls the properties of the created thread; for our purposes, simply pass in `NULL` to create a default thread.

Example:

```
void *find_nth_fibonacci(void *arg) {
    int n = *( (int *) arg);
    int result = recursive_fib(n);
    int *retval = malloc(sizeof(int));
    *retval = result;
    return (void *) retval;  return pointer to value stored on shared heap
}

int main(int argc, char** argv) {
    ...
    pthread_t child_thread_id;
    int n = 4;
    void *child_thread_retval;
    int create_status = pthread_create(child_id, NULL, find_nth_fibonacci,
        (void *) &n);
    // child_thread_retval will hold return value
    int join_status = pthread_join(child_id, &child_thread_retval);
    int my_fib_n = *((int *) child_thread_retval);
    ...
    free(child_thread_retval);  // clean up the block left by the child thread
    return 0;
}
```

Retrieving the return values of the function is accomplished through `pthread_join`. As you can see, the process of moving data into threads at their creation and out of threads at their termination is a little bit involved, and the use of `void *` might come to make your head hurt. Remember that the main thread now has to free the space allocated by the child thread to hold the value pointed to by returned pointer.

- **`pthread_join`**

```
int pthread_join(pthread_t thread, void **retval);
```

Joins the calling thread to the thread whose ID is specified by `thread`. Returns 0 for success or a non-zero value for error, but only after the target thread has finished. Calls to `pthread_join` that target a thread that has already terminated return immediately.

By creating a pointer, here referred to as `ptr`, of type `void *` in the calling scope and passing a pointer to that pointer as the `retval` parameter, `ptr` will hold the `void *` pointer returned by the joined thread's starting function, which is its return value. The starting function may return a pointer to heap data that is cast to `void *` before being returned, and restored to its original type by the joiner.

Example:

```
void *ret;
int pthread_join(child_ID, &ret);  // ret holds returned pointer
interesting_data_t *child_data = (interesting_data_t *) ret;  // can use a cast
```

- **`pthread_self`**

  ```
  pthread_t pthread_self(void);
  ```

  Returns the ID of the running thread, which is conceptually the same as the thread that makes the call to `pthread_self`. The ID has type `pthread_t`, which should not be compared using the `==` operator – see `pthread_equal`.

  Note: this is *not* to be used as a direct replacement for `omp_get_thread_num`. Many of the idioms you have seen for dividing work between threads divide the iterations in a for loop using an expression involving the return value of `omp_get_thread_num`. However, these assume that the ID numbers of the worker threads fall exactly in the range $[0, N_{threads})$. In general, your program might not provide that guarantee, so if you want to give each thread a number in that range that, say, parametrizes which section of the matrix to work on, it might be better to provide that explicitly though the structure of the parameters passed to its starting function!

- **`pthread_equal`**

  ```
  int pthread_equal(pthread_t t1, pthread_t t2);
  ```

  Returns a nonzero value if the two `pthread_t` thread IDs refer to the same thread.

5 Further Reading

- POSIX Threads Explained. IBM.
  http://www.ibm.com/developerworks/library/l-posix1/index.html
- A Pthreads Tutorial, New Mexico State University
  http://www.cs.nmsu.edu/ jcook/Tools/pthreads/pthreads.html