MPI and Shared Memory

Lecture 7
March 4, 2025

To Dos

Reading for next time

Program 3 submission on Thursday

Questions about program 3?

Review: MPI Communication

- Communicator
- Point-to-Point
    - Send
    - Receive
- Collective communication
    - MPI_Reduce, MPI_Allreduce
    - MPI_Bcast
    - MPI_Scatter
    - MPI_Gather, MPI_Allgather
    - MPI_Barrier

Communication vs. Computation

- Communication **way more expensive** than local computation
- Every message has a high fixed overhead cost for sending with some additional time dependent on payload size
    - latency = fixed_overhead_1_byte + (payload-size / bandwidth)
    - 2 messages will take longer than 1 message with same data

Data Consolidation in Messages

- `count` parameter to communication functions
  - For sending of arrays
- Derived datatypes
  - Like creating a struct for the data values you need to send
  - Sequence of basic MPI datatypes with a displacement for each
    - e.g., {(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)}
- `MPI_Pack/Unpack`

Derived Datatypes

```
MPI_Type_create_struct(int count, int array_of_blocklengths[],
      MPI_Aint array_of_displacements[],
      MPI_Datatype array_of_types[],
      MPI_Datatype * new_type_p)
```

- `count` - number of elements in datatype
- `array_of_block_lengths` - number of elements in each datatype
- `array_of_displacements` - displacement in bytes from start of first data
  - Use `MPI_GetAddress(void *location_p, MPI_AInt* address_p)` to get address of data reference by `location_p`
- `array_of_types` - stores datatypes of elements
- Need to call `MPI_Type_commit(MPI_Datatype* new_type_p)` before using in communication function (just like any other MPI type)
- Need to `MPI_Type_free (MPI_Datatype *new_type_p)` when done

Parallel Performance

Collecting Timing Info

- Surround code we care about with timing collection function calls and subtract to find total time code ran
- `double MPI_Wtime(void)`
  - Returns wall clock time
- To ensure all parallel processes start at same time, have every process call `MPI_Barrier (MPI_Comm comm)` before calling `MPI_Wtime` so they start in synch
- Need to then find the largest time for the parallel processors as that is the time required by the entire set of parallel processes

Limitations to Performance Improvements

- Amdahl's Law

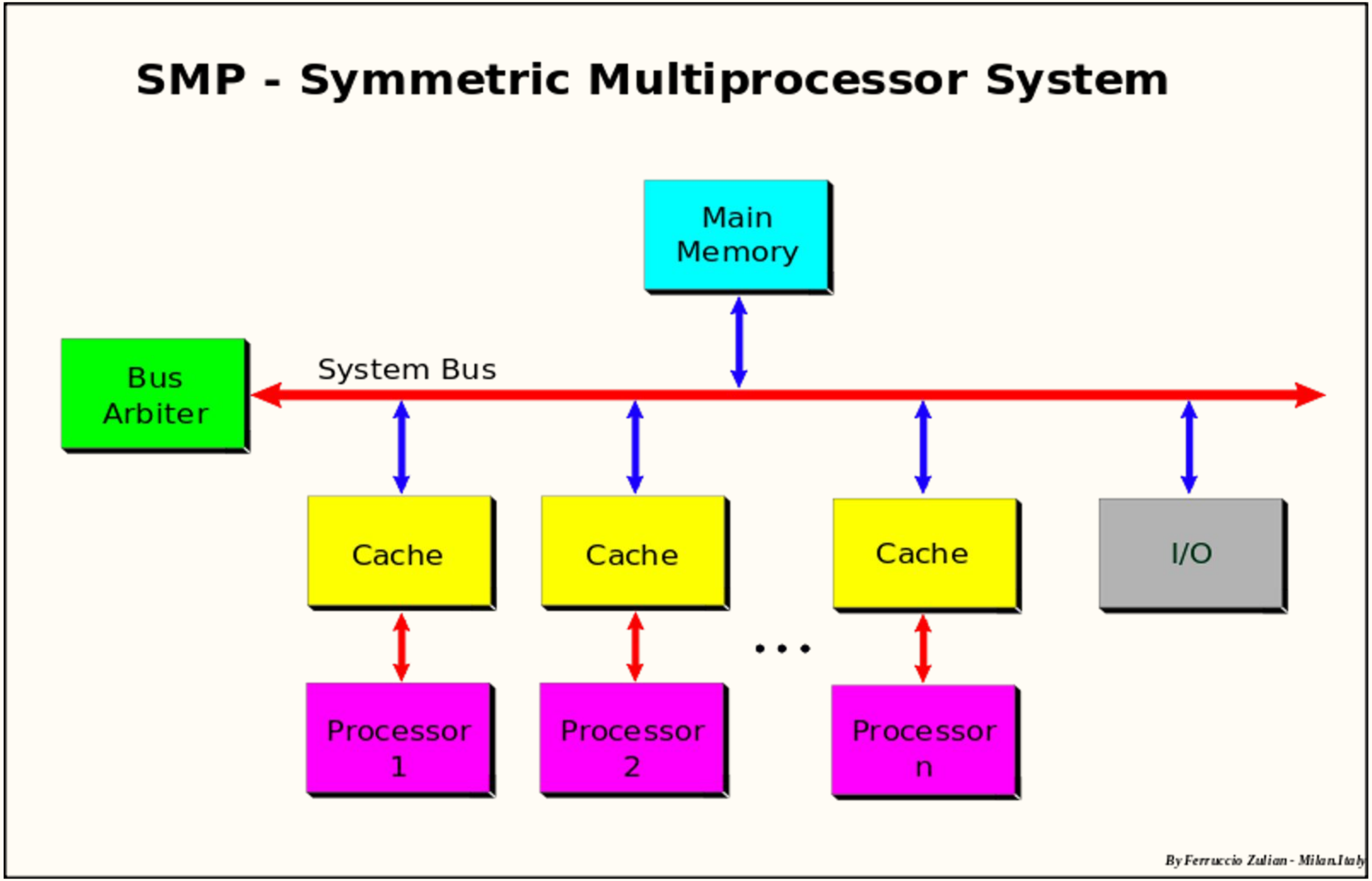
$$T_{enhanced} = (1 - fraction_{enhanced}) \times T_{unenhanced} + (\frac{fraction_{enhanced}}{Speedup_{ehnhancement}} \times T_{unenhanced})$$

- Inherently sequential parts of code
- Overhead in parallel parts of code
  - Communication of data between processes
  - Load balancing
  - Synchronization

Speedup and Efficiency

- Speedup(n, p) = $T_{serial(n)} / T_{parallel(n,p)}$
  - Linear speedup = p
- Efficiency (n, p) = S(n,p) / p = $T_{serial(n)} / (p * T_{parallel(n,p)})$
  - Linear speedup corresponds to Efficiency = p/p = 1
- Strongly scalable - maintain constant efficiency w/out increasing problem size
- Weakly scalable - maintain constant efficiency if problem size increases at same rate as number of processes

Shared Memory



https://commons.wikimedia.org/wiki/File:SMP_-_Symmetric_Multiprocessor_System.svg

Shared Memory Programming (aka Multithreaded Programming)

Processes

- Instance of running program
    - One process per instance
- Memory is private to process
- Associated resources (e.g. files)
- Access permissions
- Includes state for at least one thread of execution
    - HW registers, runnable/blocked

Threads

- Contained within a process
- 1+ threads per process
- All memory shared among threads
    - Each thread has own stack memory
- All resources and permissions defined by process
- Each thread has own state of execution
    - HW registers, runnable/blocked

POSIX Threads (aka Pthreads)

- Standard for Unix-like OSes
- Specifies API for multithreaded programming
- Like MPI, just a library linked with C programs

Basic Pthreads program setup

- `#include <pthread.h>`
- When compiling, link in pthreads library

```
gcc -g -o hello hello.c -lpthread
```

- When running, just run as normal

```
./hello
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int num_threads;
void *Hello(void *rank)
{
  printf("Hello from thread %ld out of %d\n", (long)rank, num_threads);
  return NULL;
}

int main(int argc, char *argv[])
{
  long pthread;
  num_threads= 4;
  pthread_t ids[num_threads];

  for(long i = 0; i < num_threads; i++){
    pthread_create(&ids[i], NULL, Hello, (void*)i);
  }
  printf("Hello from main!\n");
  for(int i = 0; i < num_threads; i++){
    pthread_join(ids[i], NULL);
  }

  return 0;
}
```

Some key features

- All non-stack variables shared by all threads
  - Where the challenges of multithreaded programming comes from!
- Variables allocated on a thread's stack are considered private
  - Technically, they can be accessed by other threads, but that's not good practice
- Running the program multiple times may result in different outcomes

Creating a Thread

```
int pthread_create(pthread_t* thread_p, const
    pthread_attr_t* attr_p, void* (*start_routine)(void*),
    void* arg_p)
```

- `pthread_t*` :
  - Stores thread-specific info that uniquely identifies thread
  - User program cannot access the contents
- `pthread_attr_p*` :
  - Specifies new threads attributes
  - Set stack, set stack size, set scheduling policy, set affinity, etc.
  - NULL indicates use of default values

Creating a Thread

```
int pthread_create(pthread_t* thread_p, const
    pthread_attr_t* attr_p, void* (*start_routine)(void*),
    void* arg_p)
```

- `void* (*start_routine)(void*)` :
  - Function pointer for code thread should execute
  - Function must take single void* argument and return void *
- `void* arg_p` :
  - Arguments to be passed to function
  - Can be single value cast to void*
  - Often is array or other complex data structure that contains 1+ values

K&R 5.11 talks about pointers to functions

Waiting for a Thread

```
int pthread_join(pthread_t thread, void **ret_val_p)
```

- Waits for thread specified to finish execution
  - If multiple threads call join on same thread, undefined action
  - Unjoined threads are considered to be zombie threads
- `void** ret_val_p:`
  - Pointer to location of the item returned by thread's return statement
  - Can be NULL