Part 1: Serial Code Optimization

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Let us get into GIST Linux VM

- VM4(16Cores)
- IP: 210.125.85.148

- VM3(16Cores)
- IP: 210.125.85.147

How to log-in to Linux and Update your Password

```
ai@ubuntu-20-04:~$  passwd
Changing password for ai.
Current password:
New password:
Retype new password:
passwd: password updated successfully
```

- Please check IntroToLinux.pdf that I shared at week2 schedule.
- Also, LinuxCommandCheatSheet.pdf and VIEditorCheatSheet.pdf.
- Good & free Linux/Unix tutorial: https://www.tutorialspoint.com/unix/index.htm

Construct Efficient Application

- The construction of an efficient application should adhere to the following development process:

1. Design: algorithms and data structures are designed
2. Implementation
3. Functional Testing
4. Optimization (aka Tuning)
    1. Performance testing
    2. Profiling (GDB, Valgrind, gperftools)
    3. Algorithmic optimization
    4. Platform independent optimization
    5. Software or hardware platform dependent optimization

GDB

- A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.
- GNU Debugger, which is also called gdb, is the most popular debugger for UNIX systems to debug C and C++ programs.
- GNU Debugger helps you in getting information about the following:
  - If a core dump happened, then what statement or expression did the program crash on?
  - If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
  - What are the values of program variables at a particular point during execution of the program?
  - What is the result of a particular expression in a program?
- How GDB Debugs? GDB allows you to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

Week2 Lab1: GDB Tutorial with Examples

```cpp
#include<iostream>

using namespace std;

long factorial(int n);

int main()
{
    int n(0);
    cin>>n;
    long val=factorial(n);
    cout<<val<<endl;
    cin.get();
    return 0;
}

long factorial(int n)
{
    long result(1);
    while(n--)
    {
        result*=n;
    }
    return result;
}
```

Hands on Assignment (week2_lab1)

- This has a bug.
- Where is the bug? Before you code, figure it out.
- Make your code titled “gdb_test.cpp”
- Build and Run
- Can we profile the bug using GDB?

Week2 Lab1: GDB Tutorial with Examples

```cpp
#include<iostream>

using namespace std;

long factorial(int n);

int main()
{
    int n(0);
    cin>>n;
    long val=factorial(n);
    cout<<val<<endl;
    cin.get();
    return 0;
}

long factorial(int n)
{
    long result(1);
    while(n--)
    {
        result*=n;
    }
    return result;
}
```

Into the Debugger

Now follow the commands and the outputs carefully, especially the watchpoints. What we're doing is basically:

- Setting a breakpoint just in the line of the function call
- Stepping into the function from that line
- Setting watchpoints for both the result of the calculation and the input number as it changes.
- Finally, analyzing the results from the watchpoints to find problematic behaviour

https://www.cprogramming.com/gdb.html

Development Criteria

- **Principle of diminishing returns**. Optimizations that yield big results with little effort should be applied first, as this minimizes the time needed to reach the performance goals.
- **Principle of increasing portability**. It is better to apply optimizations applicable to several platforms first, as they remain applicable on changing platform and are more understandable to other programmers.

Memory consumption vs Speed

- You could optimize your code for performance using all possible techniques, but this might generate a bigger file with bigger memory footprint.
- You might have two different optimization goals, that might sometimes conflict with each other. For example, to optimize the code for performance might conflict with optimize the code for less memory footprint and size. You might have to find a balance.
- Performance optimization is a never-ending process. Your code might never be fully optimized. There is always more room for improvement to make your code run faster.
- Sometime we can use certain programming tricks to make a code run faster at the expense of not following best practices such as coding standards, etc. Try to avoid implementing cheap tricks to make your code run faster.

Optimize Code using Appropriate Algorithm

Our Scenario: We have interval for x [-10000…10000] and interval for y [-10000…10000]. Now in these two intervals we are looking for a maximum of the function (x*x + y*y)/(y*y + b).

- This is a function of two variables: x and y. There is one more constant which could be different and user will enter it. This constant b is always greater than 0 and also lesser than 1000. Get the value of constant b from user input.
- In our program, we will not use function pow() that is implemented in math.h (cmath) library. It would be interesting exercise to figure out which approach would create faster code.

```cpp
#include <iostream>
#include <cstdlib>

#define LEFT_MARGINE_FOR_X -10000.0
#define RIGHT_MARGINE_FOR_X 10000.0
#define LEFT_MARGINE_FOR_Y -10000.0
#define RIGHT_MARGINE_FOR_Y 10000.0

using namespace std;

int main(void)
{
    //Get the constant value
    cout<<"Enter the constant value b>0"<<endl;
    cout<<"b->"; double dB; cin>>dB;

    if(dB<=0)   return EXIT_FAILURE;
    if(dB>1000) return EXIT_FAILURE;

    //This is the potential maximum value of the function
    //and all other values could be bigger or smaller
    double dMaximumValue = (LEFT_MARGINE_FOR_X*LEFT_MARGINE_FOR_X+LEFT_MARGINE_FOR_Y*LEFT_MARGINE_FOR_Y)/ (LEFT_MARGINE_FOR_Y*LEFT_MARGINE_FOR_Y+dB);

    double dMaximumX = LEFT_MARGINE_FOR_X;
    double dMaximumY = LEFT_MARGINE_FOR_Y;

    for(double dX=LEFT_MARGINE_FOR_X; dX<=RIGHT_MARGINE_FOR_X; dX+=1.0) {
      for(double dY=LEFT_MARGINE_FOR_Y; dY<=RIGHT_MARGINE_FOR_Y; dY+=1.0) {
        if( dMaximumValue<((dX*dX+dY*dY)/(dY*dY+dB)))
        {
          dMaximumValue=((dX*dX+dY*dY)/(dY*dY+dB));
          dMaximumX=dX;
          dMaximumY=dY;
        }
      }
    }

    cout<<"Maximum value of the function is="<< dMaximumValue<<endl;
    cout<<endl<<endl;
    cout<<"Value for x="<<dMaximumX<<endl
        <<"Value for y="<<dMaximumY<<endl;

    return EXIT_SUCCESS;
}
```

Which part is not efficient?

```
20      //This is the potential maximum value of the function
21      //and all other values could be bigger or smaller
22      double dMaximumValue = (LEFT_MARGINE_FOR_X*LEFT_MARGINE_FOR_X+LEFT_MARGINE_FOR_
    Y*LEFT_MARGINE_FOR_Y)/ (LEFT_MARGINE_FOR_Y*LEFT_MARGINE_FOR_Y+dB);
23
24      double dMaximumX = LEFT_MARGINE_FOR_X;
25      double dMaximumY = LEFT_MARGINE_FOR_Y;
26
27      for(double dX=LEFT_MARGINE_FOR_X; dX<=RIGHT_MARGINE_FOR_X; dX+=1.0) {
28        for(double dY=LEFT_MARGINE_FOR_Y; dY<=RIGHT_MARGINE_FOR_Y; dY+=1.0) {
29          if( dMaximumValue<((dX*dX+dY*dY)/(dY*dY+dB)))
30          {
31            dMaximumValue=((dX*dX+dY*dY)/(dY*dY+dB));
32            dMaximumX=dX;
33            dMaximumY=dY;
34          }
35        }
36      }
```

Consider Many Things in Optimization:

- Optimize your Code using Appropriate Algorithm
- Optimize Your Code for Memory
- Input / Output files
- Using Operators
- if Condition Optimization
- Optimizing Loops
- Many more….. Read
  - https://www.thegeekstuff.com/2015/01/c-cpp-code-optimization/
  - https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf
  - https://techbeacon.com/app-dev-testing/why-unnecessary-variables-are-bad-your-code

How to get running time?

- OS (shell) level
  - E.g. add "`time`" before the command you want to measure

```
ai@ubuntu-20-04:~/Lab/optimization$ ./gdb_test
3
0
ai@ubuntu-20-04:~/Lab/optimization$ time ./gdb_test
3
0

real    0m3.418s
user    0m0.005s
sys     0m0.001s
```

The meaning of real, user, and sys in output of Linux time command

- **real** or **total** or **elapsed** (wall clock time) is the time from start to finish of the call.
- **user** - amount of CPU time spent in user mode.
- **system** or **sys** - amount of CPU time spent in kernel mode
  - Man page: The time command runs the specified program command with the given arguments. When command finishes, time writes a message to standard output giving timing statistics about this program run. These statistics consist of (i) the elapsed real time between invocation and termination, (ii) the user CPU time (the sum of the tms_utime and tms_cutime values in a struct tms as returned by times(2)), and (iii) the system CPU time (the sum of the tms_stime and tms_cstime values in a struct tms as returned by times(2)).
  - Basically though, the user time is how long your program was running on the CPU, and the sys time was how long your program was waiting for the operating system to perform tasks for it. If you're interested in benchmarking, user + sys is a good time to use. real can be affected by other running processes, and is more inconsistent.

How to get running time?

- OS (shell) level
  - E.g. add "`time`" before the command you want to measure

- Program level
  - Include `ctime` library and use `clock()`

```
#include <ctime>

void f() {
  using namespace std;
  clock_t begin = clock();

  code_to_time();

  clock_t end = clock();
  double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
}
```

Tools from the C++ standard library

One option for measuring execution times (for a portion of C++ code) is to use some classes that are available as part of the C++11's standard library. The **high_resolution_clock** class, defined in the standard **<chrono>** header, can come in handy here. As its name suggests, this class represents a clock with the highest precision (or, equivalently, the smallest tick period) available on a given platform.

The high_resolution_clock exposes the "now" method, which returns a value corresponding to the call's point in time. You'll want to invoke this method twice: first at the beginning portion of the code, and again at the end of that portion. Using this method, we record the start and the end time of the execution.

```
#include <chrono>  // for high_resolution_clock
...


// Record start time
auto start = std::chrono::high_resolution_clock::now();

// Portion of code to be timed
...

// Record end time
auto finish = std::chrono::high_resolution_clock::now();
```

Read: https://www.pluralsight.com/blog/software-development/how-to-measure-execution-time-intervals-in-c--

Read: https://stackoverflow.com/questions/2808398/easily-measure-elapsed-time

Read: https://stackoverflow.com/questions/2962785/c-using-clock-to-measure-time-in-multi-threaded-programs/2962914#2962914

Be aware that clock( ) measures CPU time, not actual time elapsed!!

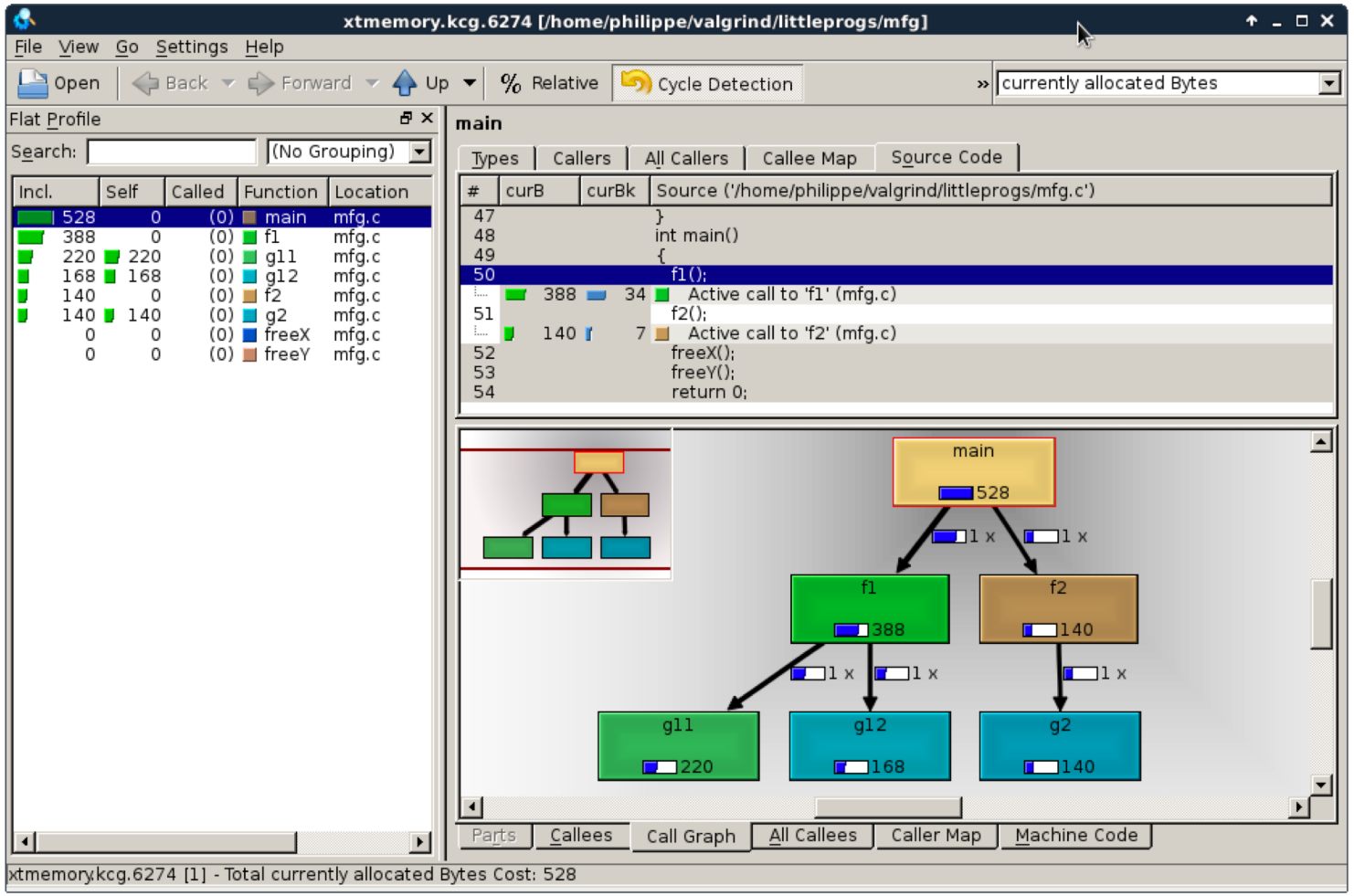
Profiler tool

- GNU “gprof”
- Compile with `-pg` option
- Run `$ gprof`

Hands on Lab using gdb_test.cpp by reference (https://www.thegeekstuff.com/2012/08/gprof-tutorial/)

Memory Footprint

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.
- https://valgrind.org/



https://valgrind.org/docs/manual/manual-core.html

Memory Footprint

3. Running your program under Memcheck

If you normally run your program like this:

```
myprog arg1 arg2
```

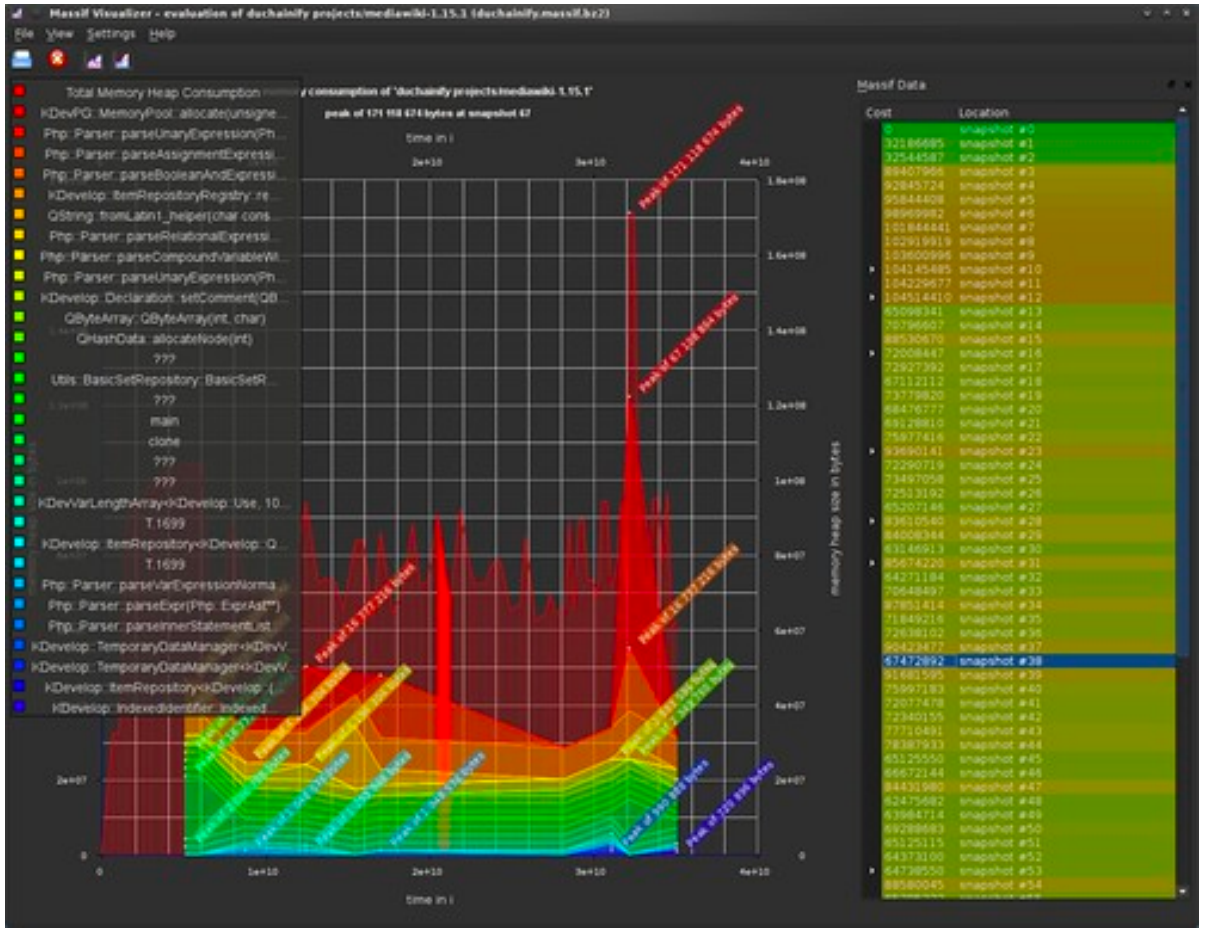
Use this command line:

```
valgrind --leak-check=yes myprog arg1 arg2
```

Memcheck is the default tool. The `--leak-check` option turns on the detailed memory leak detector.

Massif: Heap Profiler

- Maybe it's because C and C++ programmers must think that they know where the memory is being allocated. After all, you can see all the calls to malloc() and new and new[ ], right?
- But, in a big program, do you really know which heap allocations are being executed, how many times, and how large each allocation is?
- Can you give even a vague estimate of the memory footprint for your program?
- Do you know this for all the libraries your program uses?
- Use Massif that is a heap profiler and visualize it.
- Read: https://courses.cs.washington.edu/courses/cse326/05wi/valgrind-doc/ms_main.html



Problems with Functions

For example, if you have a code like this, it might be a bad thing.

```
for(int i=1; i<=10; ++i)
     DoSomething(i);
```

Why? As soon as you code something like this you will have to call DoSomething 10 times, and function calls could be expensive.

To implement this better, you could do it like this, and implement that for repetition in your function.

```
DoSomething(n);
```

Try to reduce unnecessary variables

- Declare the local variable as close to where it is used and remove un-necessary once.
- Just return it. Don’t store.
- Remove functions and variables which you do not use.
- Read: https://techbeacon.com/app-dev-testing/why-unnecessary-variables-are-bad-your-code

Do less work

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6   endif
7 enddo
```

Do less work

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6     exit
7   endif
8 enddo
```

Tips for Optimizing C/C++ Code

Code for correctness first, then optimize!

- This does not mean write a fully functional ray tracer for 8 weeks, then optimize for 8 weeks!
- Perform optimizations on your ray tracer in multiple steps.
- Write for correctness, then if you know the function will be called frequently, perform obvious optimizations.
- Then profile to find bottlenecks, and remove the bottlenecks (by optimization or by improving the algorithm). Often improving the algorithm drastically changes the bottleneck – perhaps to a function you might not expect. This is a good reason to perform obvious optimizations on all functions you know will be frequently used.

Tips for Optimizing C/C++ Code

Jumps/branches are expensive. Minimize their use whenever possible.

- Function calls require two jumps, in addition to stack memory manipulation.
- Prefer iteration over recursion.
- Use inline functions for short functions to eliminate function overhead.
- Move loops inside function calls.
- Long if...else if...else if...else if... chains require lots of jumps for cases near the end of the chain. Convert it to switch.

Tips for Optimizing C/C++ Code

Avoid/reduce the number of local variables

- Local variables are normally stored on the stack. However if there are few enough, they can instead be stored in registers. In this case, the function not only gets the benefit of the faster memory access of data stored in registers, but the function avoids the overhead of setting up a stack frame.
- (Do not, however, switch wholesale to global variables!)

Tips for Optimizing C/C++ Code

- Think about the order of array indices
- Pass structures by reference, not by value.
- For most classes, use the operators += , -= , *= , and /= , instead of the operators + , - , * , and / .
- Avoid dynamic memory allocation during computation.
- Avoid unnecessary data initialization.
- Try to early loop termination and early function returns.
- Simplify your equations on paper!
- Consider ways of rephrasing your math to eliminate expensive operations.

Tips for Optimizing C/C++ Code

- Read: https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf

Part 2: OpenMP Basic

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

What is OpenMP?

- What does OpenMP stands for?
  - **Open** specifications for **Multi Processing (OpenMP)**
- Application Programming Interface (API) for multi-threaded parallelization consisting of
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

What is OpenMP?

- Shared Memory with thread based parallelism
- Not a new language
- OpenMP API is specified for C/C++ and Fortran
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for Fortran and pragmas for C/C++
- OpenMP website: http://www.openmp.org
- OpenMP tutorial: https://hpc.llnl.gov/tuts/openMP/

Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
  - It is the de facto standard for writing shared memory programs.
  - Easy to use.
  - Incremental parallelization.
  - **Portability**.

- OpenMP can be implemented incrementally, one function or even one loop at a time.
  - A nice way to get a parallel program from a sequential program.

POSIX Threads (Pthreads)

Pthreads is a POSIX standard for describing a thread model, it specifies the API and the semantics of the calls.

- Thread API available on many OS's
  - #include <pthread.h>
  - cc myprog.c –o myprog -lpthread
- Thread creation
  - int pthread_create(pthread_t * thread,
    pthread_attr_t * attr,
    void * (*start_routine)(void *),
    void * arg);
- Thread termination
  - void pthread_exit(void *retval);
- Waiting for Threads
  - int pthread_join(pthread_t th, void **thread_return);

Motivation

- Thread libraries are hard to use
- P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
- Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness

Motivation

- Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?
  - **OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence**
  - **OpenMP is a small API that hides cumbersome threading calls with simpler *directives***

OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

    #pragma omp construct [clause [clause]…]
  - For Fortran, the directives take one of the forms:

    C$OMP construct [clause [clause]…]

    !$OMP construct [clause [clause]…]

    *$OMP construct [clause [clause]…]

- Include files

  #include “omp.h”

How is OpenMP typically used?

- OpenMP is usually used to parallelize loops very easily:
  - Find your most time consuming loops.
  - Split them up between threads.

Sequential Program

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

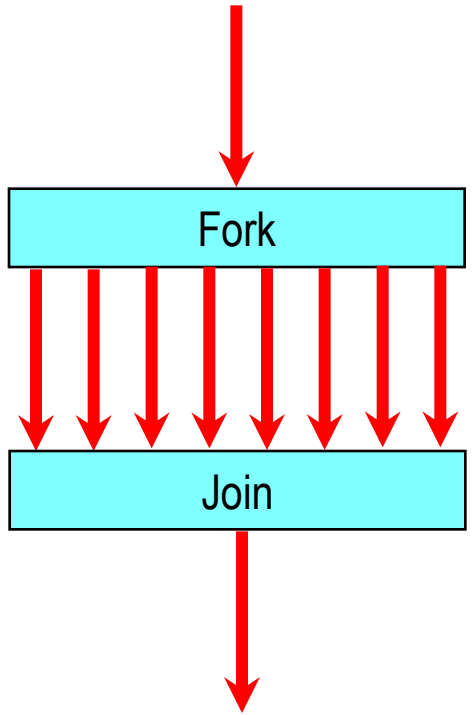
Parallel Program

```
#include “omp.h”
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

OpenMP Fork-and-Join model

- Serial regions by default, annotate to create *parallel regions*
  - Generic parallel regions
  - Parallelized loops
  - Sectioned parallel regions

- Thread-like Fork/Join model
  - Arbitrary number of *logical* thread creation/ destruction events



First OpenMP Program

- Always, starts with “Hello World!”

helloworld_omp.cpp

- Open a terminal
- Open a file with a filename “helloworld_omp.cpp”
- Write a general C++ code to print out “Hello World” as below!

```
#include <iostream>

using namespace std;

int main()
{
        cout << "Hello World!\n";

        return 0;
}
```

helloworld_omp.cpp

- Build and run!

```cpp
#include <iostream>

using namespace std;

int main()
{
        cout << "Hello World!\n";

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World!
ai@ubuntu-20-04:~/Lab/OpenMP$
```

helloworld_omp.cpp

- Add “omp.h” library
- Build!

```
#include <iostream>
#include <omp.h>

using namespace std;

int main()
{
        cout << "Hello World!\n";

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World!
ai@ubuntu-20-04:~/Lab/OpenMP$
```

helloworld_omp.cpp

- Add “int id = omp_get_thread_num();” and update cout line.
- Build. You will get an error.
- Build it with “-fopenmp” option.

```
#include <iostream>
#include <omp.h>

using namespace std;

int main()
{
        int id = omp_get_thread_num();
        cout << "Hello World!\n";

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ helloworld_omp.cpp -o helloworld_omp
/usr/bin/ld: /tmp/ccc9HL1C.o: in function `main':
helloworld_omp.cpp:(.text+0xd): undefined reference to `omp_get_thread_num'
collect2: error: ld returned 1 exit status
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World!
```

helloworld_omp.cpp

- Add “#pragma omp parallel”
- Print out ID together.
- Build with “-fopenmp” option.
- You will get the build error

```cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main()
{
        #pragma omp parallel
        int id = omp_get_thread_num();
        cout << "Hello World from ID=" << id << "!\n";

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
helloworld_omp.cpp: In function ‘int main()’:
helloworld_omp.cpp:10:36: error: ‘id’ was not declared in this scope
   10 |  cout << "Hello World from ID=" << id << "!\n";
      |                                    ^~
```

helloworld_omp.cpp

- Add bracket ({ }) as right.
- Build with “-fopenmp” option.
- Run the binary!
- Yeah, the first OpenMP results!

```
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=Hello World from ID=7!
Hello World from ID=12!
Hello World from ID=8!
Hello World from ID=Hello World from ID=9!
Hello World from ID=1!
Hello World from ID=2!
Hello World from ID=5!
Hello World from ID=13!
Hello World from ID=4!
Hello World from ID=3!
Hello World from ID=15!
Hello World from ID=11!
14!
Hello World from ID=6!
Hello World from ID=0!
10!
```

```
#include <iostream>
#include <omp.h>

using namespace std;

int main()
{
        #pragma omp parallel
        {
        int id = omp_get_thread_num();
        cout << "Hello World from ID=" << id << "!\n";
        }

        return 0;
}
```

helloworld_omp.cpp

- Let us change from cout to printf!
- Build and run!

```
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
        #pragma omp parallel
        {
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n",id);
        }

        return 0;
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=15!
Hello World from ID=0!
Hello World from ID=12!
Hello World from ID=14!
Hello World from ID=11!
Hello World from ID=9!
Hello World from ID=3!
Hello World from ID=13!
Hello World from ID=8!
Hello World from ID=7!
Hello World from ID=1!
Hello World from ID=2!
Hello World from ID=4!
Hello World from ID=5!
Hello World from ID=10!
Hello World from ID=6!
```

helloworld_omp.cpp

- Don’t change source code.
- $ export OMP_NUM_THREADS=4 as below.
- Do not build.
- Run!

```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
        #pragma omp parallel
        {
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n",id);
        }

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NUM_THREADS=4
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=2!
Hello World from ID=1!
Hello World from ID=3!
```

helloworld_omp.cpp

- What if we move the omp_get_thread_num() line before the pragma directive?

```
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
        int id = omp_get_thread_num();
        #pragma omp parallel
        {
        printf("Hello World from ID=%d!\n", id);
        }

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ g++ -fopenmp helloworld_omp.cpp -o helloworld_omp
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=0!
Hello World from ID=0!
Hello World from ID=0!
```

Compiling

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	icc icpc ifort	-openmp
PGI Linux Opteron/Xeon	pgcc pgCC pgf77 pgf90	-mp
GNU Linux Opteron/Xeon IBM Blue Gene	gcc g++ g77 gfortran	-fopenmp
IBM Blue Gene	bgxlc_r, bgcc_r bgxlC_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r *Be sure to use a thread-safe compiler - its name ends with _r	-qsmp=omp

How to compile and run OpenMP programs?

- gcc 4.2 and above supports OpenMP 3.0
  - `$ gcc -fopenmp a.c`
  - `$ g++ -fopenmp a.cpp`

Compiler	Compiler Options	Default behavior for # of threads (OMP_NUM_THREADS not set)
GNU (gcc, g++, gfortran)	-fopenmp	as many threads as available cores
Intel (icc ifort)	-openmp	as many threads as available cores
Portland Group (pgcc,pgCC,pgf77,pgf90)	-mp	one thread

- To run: 'a.out'
  - To change the number of threads:
    - For Bash Shell

      `$ export OMP_NUM_THREADS=4`

helloworld_omp.cpp

- Don’t change source code.
- $ export OMP_NUM_THREADS=4 as below.
- Do not build.
- Run!

```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main()
{
        #pragma omp parallel
        {
        int id = omp_get_thread_num();
        printf("Hello World from ID=%d!\n",id);
        }

        return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/OpenMP$ export OMP_NUM_THREADS=4
ai@ubuntu-20-04:~/Lab/OpenMP$ ./helloworld_omp
Hello World from ID=0!
Hello World from ID=2!
Hello World from ID=1!
Hello World from ID=3!
```

Set the number of threads in various ways

- Compiler Directives

```
#pragma omp parallel num_threads(4)
```

- Run-time Library

```
omp_set_num_threads(4);
```

- Environment Variables

```
export OMP_NUM_THREADS=4
```

OpenMP API Overview

Three Components:

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)
- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.
- Implementations differ in their support of all API components.

Compiler Directives

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag.
- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- For example:

Fortran	`!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)`
C/C++	`#pragma omp parallel default(shared) private(beta,pi)`

Compiler Directives

- private (list), shared (list)
- firstprivate (list), lastprivate (list)
- reduction (operator: list)
- schedule (method [, chunk_size])
- nowait
- if (scalar_expression)
- num_thread (num)
- threadprivate(list), copyin (list)
- ordered
- collapse (n)
- tie, untie
- And more …

Run-time Library Routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- For example:

Fortran	`INTEGER FUNCTION OMP_GET_NUM_THREADS()`
C/C++	`#include <omp.h>` `int omp_get_num_threads(void)`

Run-time Library Routines

- Number of threads: omp_{set,get}_num_threads
- Thread ID: omp_get_thread_num
- Scheduling: omp_{set,get}_dynamic
- Nested parallelism: omp_in_parallel
- Locking: omp_{init,set,unset}_lock
- Active levels: omp_get_thread_limit
- Wallclock Timer: omp_get_wtime
- thread private
- call function twice, use difference between end time and start time
- And more …

Environment Variables

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy
- For example,

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- And more …

OpenMP Constructs (to be continued)

OpenMP's constructs:

- Parallel Regions
- Worksharing (for/DO, sections, …)
- Data Environment (shared, private, …)
- Synchronization (barrier, flush, …)
- Runtime functions/environment variables (omp_get_num_threads(), …)