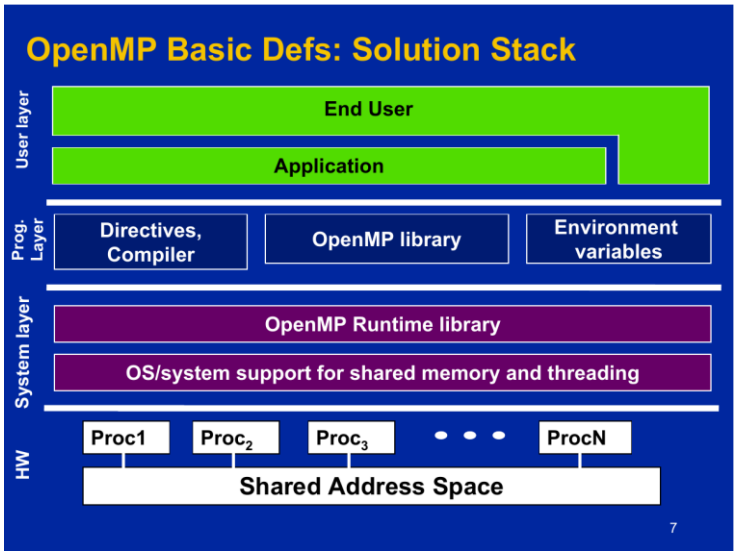
"Simple" Multithreaded Programs

A first look at
OpenMP

OpenMP

- Standard C API supports multi-threading
- *#Pragmas* define code regions that the compiler should parallelize automatically
  - Set up by the preprocessor first
- Trades some capability for ease of use

(swiped from Mattson and Meadows)



Hello world - sequential

- Build:
  - gcc -Wall -o hello-seq hello-seq.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int thread_ID = 0;  // default value

    printf("Hello-%d ... ", thread_ID);
    printf(" ... world-%d\n", thread_ID);

    return 0;
}
```

First Parallel Attempt at "Hello World"

- Build:
  - gcc -Wall -o hello-omp1 hello-omp1.c -fopenmp

```c
#include <stdio.h>
#include <omp.h>    // OpenMP function declarations

int main(int argc, char **argv)
{
    int thread_ID = 0;  // default value

    // Parallel block, with default number of threads:
    #pragma omp parallel
    {
        thread_ID = omp_get_thread_num();

        printf("Hello-%d ... ", thread_ID);
        printf(" ... world-%d\n", thread_ID);
    }

    return 0;
}
```

Did it work?

- Real time (wall-clock time) is less than user CPU time
  - So there was a speedup

```
464] time ./hello-omp1
Hello-2 ...   ... world-3
Hello-3 ...   ... world-3
Hello-1 ...   ... world-3
Hello-0 ...   ... world-3

real    0m0.006s
user    0m0.007s
sys     0m0.001s
```

- The results – are they right?
  - What happened?

OpenMP Second Try

```
#include <stdio.h>
#include <omp.h>    // OpenMP function declarations

int main(int argc, char **argv)
{
    // Parallel block, with default number of threads:
    #pragma omp parallel
    {
        int thread_ID = 0;  // thread-local variable

        thread_ID = omp_get_thread_num();

        printf("Hello-%d ... ", thread_ID);
        printf(" ... world-%d\n", thread_ID);
    }

    return 0;
}
```

Private Copy of a Variable

```
#include <stdio.h>
#include <omp.h>    // OpenMP function declarations

int main(int argc, char **argv)
{
    int thread_ID = 0;  // Shared between threads?

    // Parallel block, with default number of threads:
    #pragma omp parallel  private( thread_ID )
    {
        thread_ID = omp_get_thread_num();

        printf("Hello-%d ... ", thread_ID);
        printf(" ... world-%d\n", thread_ID);
    }

    return 0;
}
```

OpenMP and For Loops

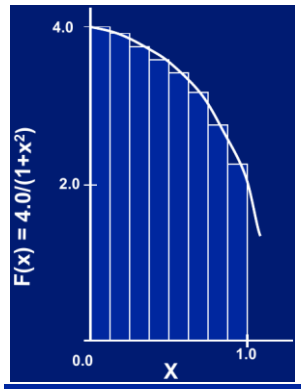
```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv)
{
#ifdef PARALLEL
#pragma omp parallel for
#endif
    for (int i = 0; i < 4 * omp_get_max_threads(); i++) {
        printf("Thread %d: loop number %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

- Compile for sequential running:
  - **`gcc -Wall -fopenmp -o for-s  forloop.c`**
- Compile for parallel running:
  - **`gcc -Wall -DPARALLEL -fopenmp -o for-p  forloop.c`**

Approximating Pi

- Exact integral based on arctangent of 1
- Limit of sum of deltas
  - $\Delta x = 1/N$
- Accuracy limited by computer's precision
- This is an "*embarrassingly parallelizable*" problem



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Graphics from Mattson and Meadows again

Pi Approximation – part 1

Allow cmd-line control of number of threads, fineness of delta-x approx-imation

```c
/*
* arctan-based approximation of pi
*/
#include <stdio.h>
#include <stdlib.h> // strtoul()
#include <math.h>   // M_PI
#include <omp.h>

int main(int argc, char **argv)
{
    double pi = 0.0, n, deltax;
    int nthreads;

    if (argc >= 2)
        nthreads = strtoul(argv[1], NULL, 0);
    else
        nthreads = omp_get_max_threads();
    if (argc == 3)
        n = strtod(argv[2], NULL);
    else
        n = 1.0e9;
    deltax = 1.0 / n;
```

Pi Approximation – part 2

```c
#ifdef PARALLEL
#pragma omp parallel num_threads(nthreads)
#endif
    {
        double pi_part = 0.0;
        int threadID = omp_get_thread_num();

#ifdef PARALLEL
#pragma omp for
#endif
        for (int i = 0; i < (long)n; i++) {
            double x = (i+0.5) * deltax;
            pi_part += 4.0 / (1.0 + x * x);
        }
#ifdef PARALLEL
#pragma omp atomic
#endif
        pi += deltax * pi_part;
        printf("%d: %lf, %lf\n", threadID, pi_part, pi);
    }
    printf("approx.: %.16lf\n   C   : %.16lf\n", pi, M_PI);

    return 0;
}
```

Pi Approximation - Compiling

- Build, run a *sequential* version:
  - `gcc -Wall -fopenmp piloop.c -o piloop-s`
  - `./piloop-s  1  1e10`
- Build, run a *parallel* version:
  - `gcc -Wall -fopenmp piloop.c -o piloop-p -DPARALLEL`
  - `./piloop-p  1  1e10`
  - `./piloop-p  4  1e10`
  - `./piloop-p  12  1e10`

Multiple Threads and Random Numbers

- Standard C **`random()`** function is not *thread-safe*
  - Multiple threads compromise statistical randomness
- Available **`random_r()`** function maintains separate random state for each thread
  - Additional "machinery" needed to set up the random function generators
    - array of "random_data" structures
    - **`initstate_r()`** function initializes "random_data"

Thread-safe Random Numbers – part 1

```c
/*
* Demo thread-safe random numbers
*/
#include <stdio.h>
#include <stdlib.h> // strtoul(), random_r()
#include <time.h>   // time()
#include <omp.h>

#define PRNG_BUFSZ 32   /* >= 8, the bigger the better? */

int main(int argc, char **argv)
{
    int nthreads;
    if (argc == 2)
        nthreads = strtoul(argv[1], NULL, 0);
    else
        nthreads = omp_get_max_threads();
    printf("%d threads\n", nthreads);
```

Thread-safe Random Numbers – part 2

```c
    /*-------------------------------------------------------------------*\
    | Create separate, thread-safe PRNGs - one per thread
    \*-------------------------------------------------------------------*/
    srandom( time(NULL) );

    char *rand_state_bufs = (char *)calloc(nthreads, PRNG_BUFSZ);
    struct random_data *rand_states =
        (struct random_data *)calloc(nthreads, sizeof(struct random_data));

    // Initialize a PRNG for each thread (the first argument is the seed)
    for (int t = 0; t < nthreads; t++)
        initstate_r(random(), &rand_state_bufs[t], PRNG_BUFSZ, &rand_states[t]);
    /*-------------------------------------------------------------------*/
```

Thread-safe Random Numbers – part 3

```
#pragma omp parallel num_threads(nthreads)
    {
        int threadID = omp_get_thread_num();
#pragma omp for
        for (long unsigned i = 0; i < 4 * nthreads; i++) {
            int rn;
            random_r(rand_states + threadID, &rn);
            double rf = (double)rn / (double)RAND_MAX;
            printf("Thread %*d: random number %lf\n", (threadID), threadID, rf);
        }
    }
    return 0;
}
```