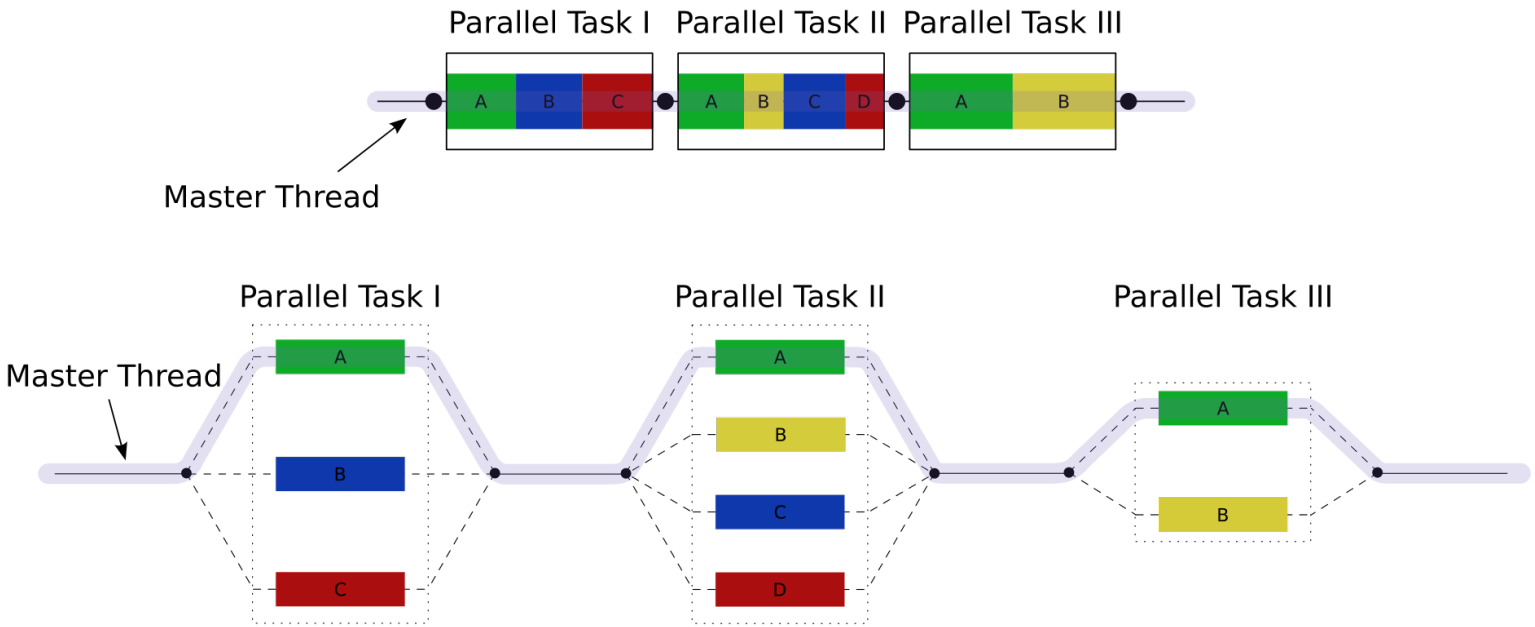
Parallel Programming with OpenMP

Introduction: OpenMP Programming Model

- Thread-based parallelism utilized on shared-memory platforms
- Parallelization is either explicit, where programmer has full control over parallelization or through using compiler directives, existing in the source code.
- Thread is a process of a code is being executed. A thread of execution is the smallest unit of processing.
- Multiple threads can exist within the same process and share resources such as memory

Introduction: OpenMP Programming Model

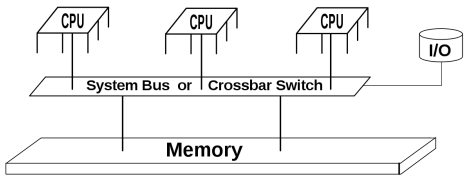
- Master thread is a single thread that runs sequentially; parallel execution occurs inside parallel regions and between two parallel regions, only the master thread executes the code. This is called the fork-join model:



Shared memory allows immediate access to all data from all processors without explicit communication.

Shared memory:

- multiple cpus are attached to the BUS
- all processors share the same primary memory
- the same memory address on different CPU's refer to the same memory location
- CPU-to-memory connection becomes a bottleneck: shared memory computers cannot scale very well



OpenMP versus MPI

OpenMP (Open Multi-Processing):

- easy to use; loop-level parallelism
- non-loop-level parallelism is more difficult
- limited to shared memory computers
- cannot handle very large problems

An alternative is MPI (Message Passing Interface):

- require low-level programming; more difficult programming
- scalable cost/size
- can handle very large problems

OpenMP core structure

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

int main (int argc, char *argv[])
{
int nthreads, id;

/* Fork a team of threads giving them their own copies of variables.
   Make the values of nthreads and id private to each thread */

#pragma omp parallel private(nthreads, id)
  {

  /* Obtain thread number */
  id = omp_get_thread_num();
  printf("Hello World from thread = %d\n", id);

  /* Only master thread does this */
  if (id == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */
    return 0;
}
```

OpenMP core structure

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

int main (int argc, char *argv[])
{
int nthreads, id;

/* Fork a team of threads giving them their own copies of variables.
   Make the values of nthreads and id private to each thread */

#pragma omp parallel private(nthreads, id)
  {

  /* Obtain thread number */
  id = omp_get_thread_num();
  printf("Hello World from thread = %d\n", id);

  /* Only master thread does this */
  if (id == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */
    return 0;
}
```

Beginning of the parallel region

Serial region

Fork a team of threads

Parallel region: all threads execute this

Compiling:

export OMP_NUM_THREADS=4

gcc -o hello_omp -fopenmp hello_omp.c

OpenMP C Directives

A directive has a name followed by clauses

```
#include <omp.h>

int omp_get_num_threads(void)
\* in bash, export OMP_NUM_THREADS=8 *\

#pragma omp parallel default(shared) private(beta,pi)

#pragma omp parallel for
#pragma omp critical
#pragma omp single
#pragma omp barrier


#pragma omp parallel for reduction(+:sum)


double omp_get_wtime(void);
```

OpenMP C Directives: Reduction

A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Reduction

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private ( sum )
        for (i = 0; i <= MAX ; i++)
                sum += A[i];

avg = sum/MAX;
```

Flow dependence

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp for reduction(+ : sum)
        for (i = 0; i <= MAX ; i++)
                sum += A[i];

avg = sum/MAX;
```

Dependence removed

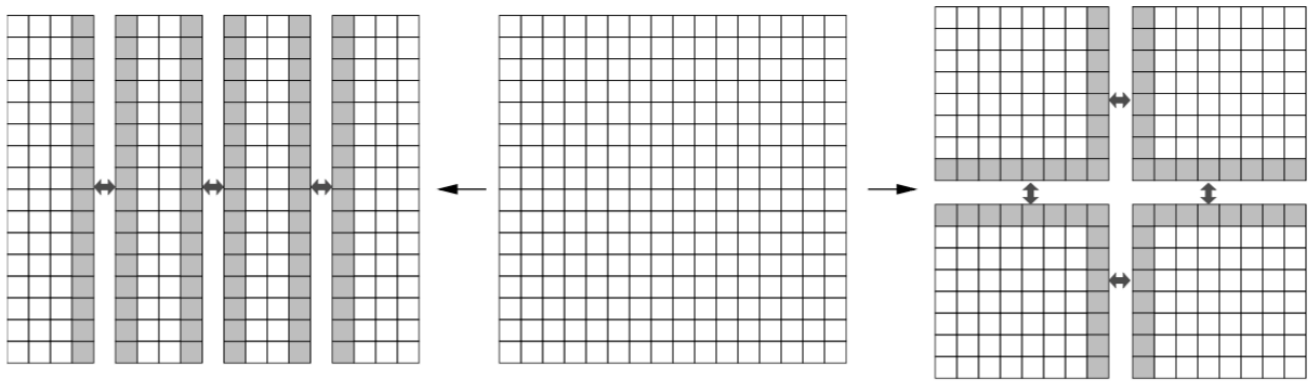
```
#pragma omp critical
        sum=sum+x
```

Matrix multiplication

```c
/* compute matrix A multiplied by matrix B; the results is stored in matrix C */
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 100                 /* number of rows in matrix A */
#define P 100                 /* number of columns in matrix A */
#define M 100                 /* number of columns in matrix B */

int main (int argc, char *argv[])
{
int tid, nthreads, i, j, k;
double  a[N][P], b[P][M], c[N][M];

/*** Create a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();
  /*** Initialize matrices ***/
  #pragma omp for
  for (i=0; i<N; i++)
    for (j=0; j<P; j++)
      a[i][j]= i+j;
  #pragma omp for
  for (i=0; i<P; i++)
    for (j=0; j<M; j++)
      b[i][j]= i*j;
  #pragma omp for
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      c[i][j]= 0;

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for
  for (i=0; i<N; i++)
    {
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<M; j++)
      for (k=0; k<P; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  }   /*** End of parallel region ***/
return(0);
}
```

Parallel scalability

- How much faster can a given problem be solved with N workers instead of one?
- How much more work can be done with N workers instead of one?
- Load imbalance hampers performance because some resources are underutilized.



Performance with OpenMP

- The factor by which the time to solution can be improved compared to using only a single processor is called speedup.
- It is critical to parallelize the large majority of a program.
- Every time the program invokes a parallel region or loop, it incurs a certain overhead for going parallel.
- In addition to the costs of invoking the parallel loop and executing the barrier, cache and synchronization effects can greatly increase the cost.

Load balancing

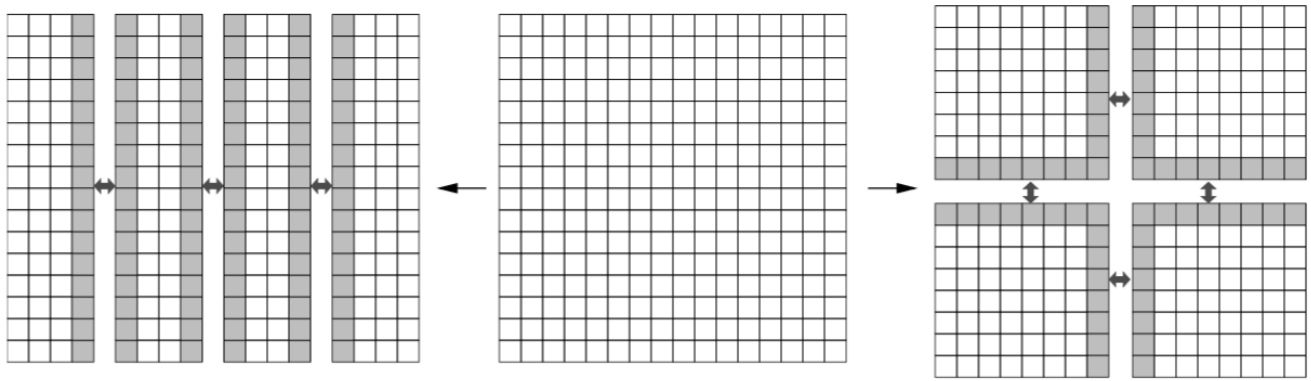
- If some threads have to do more work than others, performance will suffer.
- Consider the problem of scaling the elements of a sparse matrix; if some rows have many more non-zero terms than others, load balancing can be a problem with static schedules.
- The load balancing problem can be solved by using a dynamic schedule.
- There are also costs to using dynamic schedules.
- A static schedule can usually achieve good load distribution in situations where the amount of work per iteration is uniform or if it varies in a predictable fashion

Scaling a dense, triangular matrix.

```
for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
      a[i][j] = c * a[i][j];
    }
}
```

Scalability metrics

- Algorithmic limitations
- Serialized executions
- Startup overhead
- Communication



Scalability metrics

The overall size of the problem is $T^s = s + p$, where s is the serial (nonparallelizable) part and p is the perfectly parallelizable fraction.
Strong scaling: solving the same problem on N workers will require a runtime of

$$T^p = s + p/N$$

where N is the number of workers.
Weak scaling (time to solution is not the primary objective): scale the problem size, therefore $T^s = s + pN^\alpha$, $\alpha > 0$. We then have

$$T^p = s + pN^{\alpha-1}$$

($\alpha = 1$ is a special case, $\alpha = 0$ is the strong scaling)

Application speedup can be defined as the quotient of parallel and serial performance for fixed problem size.

Scalability metrics

Amdahl's law, parallel performance using N processors, in which s is the non-parallelizable work, is given by:

$$P^p = \frac{1}{s + \frac{1-s}{N}}$$

Scalability: So-called strong scalability is in effect the same as speedup.
A code that shows perfect speedup presents strong scalability:

$$S^p = \frac{1}{s + \frac{1-s}{N}}$$

For a fixed problem size, scalability is limited, $S^p \to \frac{1}{s}$ as $N \to \infty$.

Scalability metrics

Gustafson's law; in the case of weak scaling

$$P^p = S^p = \frac{s + (1-s) * N^\alpha}{s + (1-s) * N^{\alpha-1}}$$

($\alpha = 0$ recovers Amdahl's law)

Weak scaling allows unlimited performance, $S^p \to 1 + \frac{PN^\alpha}{s}$ as $N \to \infty$.
Parallel efficiency = speedup/N For $\alpha = 1$,

$$\epsilon = s/N + 1 - s$$

For $\alpha = 0$,

$$\epsilon = 1/(sN + (1-s))$$

Creating Parallel Regions

When a thread reaches a parallel directive (#pragma), it creates a team of threads and becomes the master of the team.

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)

   structured_block
```

- At most one if clause can appear on the directive.
- At most one num_threads clause can appear on the directive.
- Jumping out of the parallel region in not permitted.

Loop-Level Parallelism

The loop iterations are distributed across the threads.

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

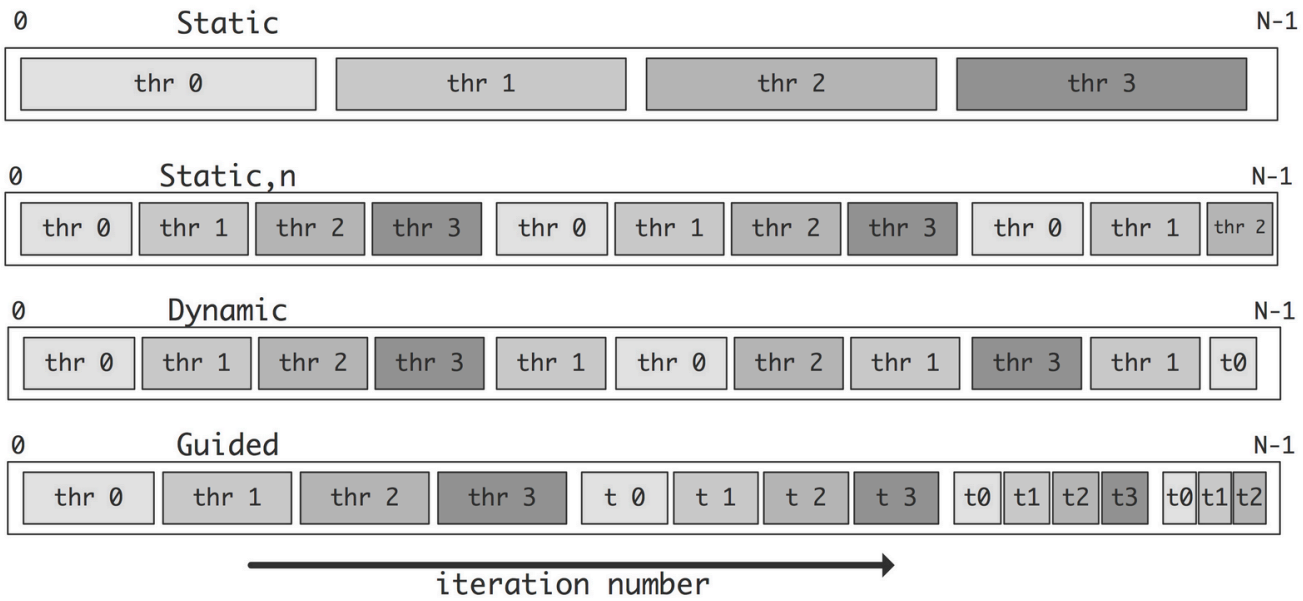
- The schedule clause specifies how the iterations in a loop are assigned to the threads.
- types are: static, dynamic, guided and runtime.
- *chunk* size can be variable.

Loop-Level Parallelism: Schedule Clause

- STATIC: Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- DYNAMIC: Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
- GUIDED Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned.
- RUNTIME The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- AUTO The scheduling decision is delegated to the compiler and/or runtime system.
- nowait: If specified, then threads do not synchronize at the

The schedule clause

```
#pragma omp parallel for schedule (...) num_threads (THREADS)
  for(i = 0; i < N; i++) {
    A[i]=0.;
  }
```



Removing Data Dependences

```
1  for(i=0;i<N;i++)
2  {
3     x = (b[i] + c[i])/2;
4     a[i] = a[i + 1] + x;
5  }
6
7
8  \*
9  Parallel version with dependences removed:
10 *\
11
12 # pragma omp parallel for shared(a,a2)
13     for(i=0;i<N-1;i++)
14       a2[i] = a[i + 1];
15
16 # pragma omp parallel for private(x) shared (a,a2)
17     for(i=0;i<N-1;i++){
18       x = (b[i] + c[i])/2;
19       a[i] = a2[i] + x;
20     }
21
```

Section Work-Sharing Constructs

Section specifies that the enclosed section(s) of code are to be divided among the threads in the team.

```
#include <omp.h>
#define N 1000

main(int argc, char *argv[]) {

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel region */

}
```

Matrix multiplication

Homework1: Matrix multiplication
Review / Compile / Run the matrix multiply example code:
Link to mm.c

- Analyze the speedup and efficiency of the parallelized code.
- Vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread.
- For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads. Compute the efficiency.
- Explain whether or not the scaling behavior is as expected.

Matrix multiplication

Homework2: Matrix multiplication

- Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup and efficiency.
- Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup and efficiency.

Jacobi Method

Homework: Parallelize the Jacobi Method
Link to jacobi.c

- Analyze the speedup and efficiency of the parallelized code.
- Vary the size of your **A** matrix and measure the runtime with one thread.
- For each matrix size, change the number of threads from 2,4,8, ... and plot the speedup versus the number of threads. Compute the efficiency.
- Explain whether or not the scaling behavior is as expected.