# Day 2:
# High Performance Computing
# CMT106

David W. Walker

Professor of High Performance Computing

Cardiff University

http://www.cardiff.ac.uk/people/view/118172-walker-david

# Day 2

- 9:30 – 10:45am: Lecture on some practical aspects of OpenMP.
- 10:45 – 11:00am: Break
- 11:00 – 11:45am: Lecture (continued) on some practical aspects of OpenMP; OpenMP for nested *for* loops.
- 11:45am – 12:00pm: Discussion about the OpenMP coursework (due in Week 11).
- 12:00 – 1:30pm: Lunch break.
- 1:30 – 3:00pm: Lab session to try out some OpenMP codes presented in the lectures, or download instructions for Lab 1 from *Learning Central*.
- 3:00 – 4:15pm: Self-study, read first part of "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era" (includes 15min break).
- 4:15 – 4:30pm: Discussion about the paper.
- 4:30 – 5:00pm: Lecture on interconnection networks; network metrics.

# Topics Covered on Days 1-4

- *Day 1:* Introduction to parallelism; motivation; types of parallelism; Top500 list; classification of machines; SPMD programs; memory models; shared and distributed memory; example of summing numbers.
- *Day 2*: Shared memory programming with OpenMP; interconnection networks; network metrics.
- *Day 3*: Classification of parallel algorithms; speedup and efficiency; scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI; collective communication.
- Day 4: Integration example; regular computations and simple examples using MPI.

# Topics Covered on Days 5-7

- *Day 5*: Graphical processing units, their architecture and device memory model; programming GPUs with CUDA; the Laplace equation example code.
- *Day 6*: Dynamic communication and the molecular dynamics example; irregular computations; the WaTor simulation .
- *Day 7*: Dynamic and static load balancing techniques: orthogonal recursive bisection, hierarchical recursive bisection, block cyclic data decompositions; a cellular automaton model of a catalytic converter.

# Summing m Numbers

**Example**: summing m numbers

On a sequential computer we have,

```
sum = a[0];
for (i=1;i<m;i++) {
        sum = sum + a[i];
}
```

We would expect the running time be roughly proportional to m. We say that the running time is $\Theta(m)$.

# Summing m Numbers in Parallel

- What if we have N processors, with each calculating the m/N numbers assigned to it?
- We must add these partial sums together to get the total sum.

# Summing Using Shared Memory

The m numbers, and the global sum, are held in global shared memory.

```
global_sum = 0;
for (each processor){
        local_sum = 0;
        calculate local sum of m/N numbers
        LOCK
                global_sum = global_sum + local_sum;
        UNLOCK
}
```

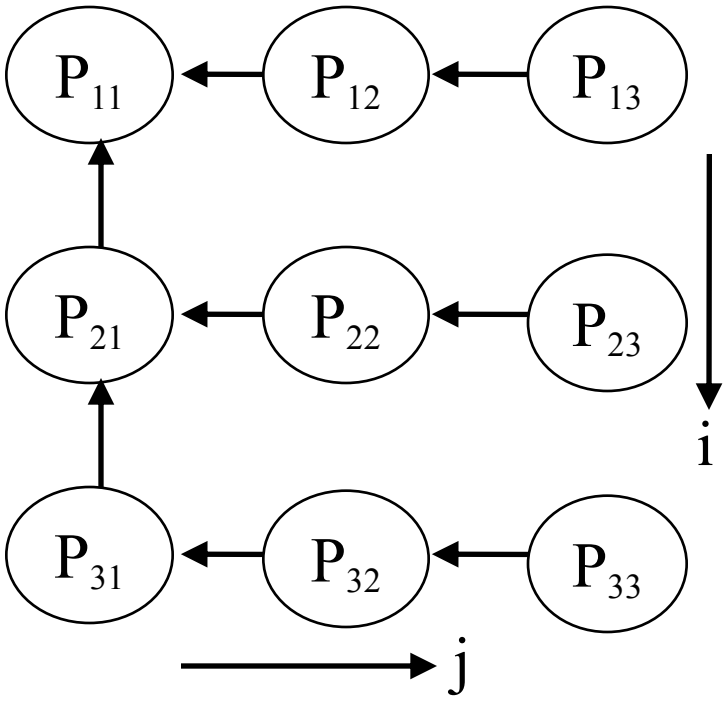
# Notes on Shared Memory Algorithm

- Since global_sum is a shared variable each processor must have mutually exclusive access to it – otherwise the final answer may be incorrect.
- The running time (or *algorithm time complexity*) is

  $\Theta(m/N) + \Theta(N)$
- where
  - m/N comes from finding the local sums in parallel
  - N comes from adding N numbers in sequence

# Summing Using Distributed Memory

Suppose we have a square mesh of N processors.




The algorithm is as follows:

1. Each processor finds the local sum of its m/N numbers
2. Each processor passes its local sum to another processor in a coordinated way
3. The global sum is finally in processor $P_{11}$.

# Distributed Memory Algorithm

The algorithm proceeds as follows:

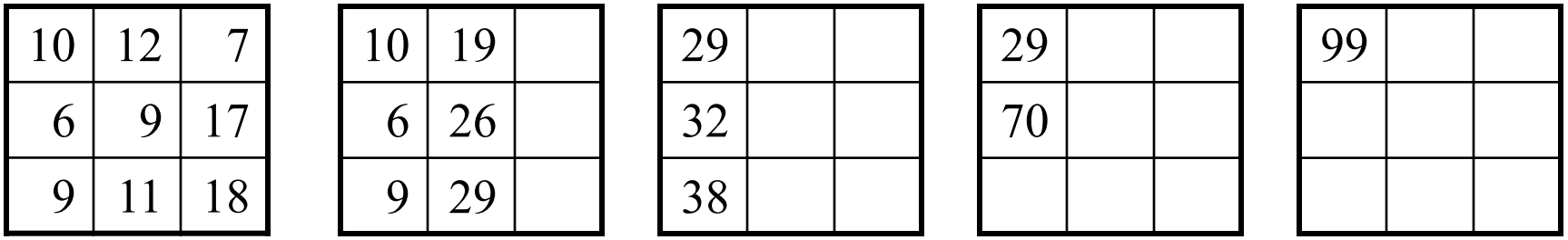
1. Each processor finds its local sum.
2. Sum along rows:
   a) If the processor is in the rightmost column it sends its local sum to the left.
   b) If the processor is not in the rightmost or leftmost column it receives the number form the processor on its right, adds it to its local sum, and sends the result to the processor to the left.
   c) If the processor is in the leftmost column it receives the number from the processor on its right and adds it to its local sum to give the row sum.
3. Leftmost column only – sum up the leftmost column:
   a) If the processor is in the last row send the row sum to the processor above
   b) If the processor is not in the last or first row receive the number from the processor below, add it to the row sum, and send result to processor above
   c) If the processor is in the first row receive the number from the processor below and add it to the row sum. This is the global sum.

# Summing Example

There are $\sqrt{N}-1$ additions and $\sqrt{N}-1$ communications in each direction, so the total time complexity is

$$\Theta(m/N) + \Theta(\sqrt{N}) + C$$

where C is the time spent communicating.



# Quick Overview of OpenMP

- OpenMP can be used to represent task and data parallelism.
- In the case of data parallelism, OpenMP is used to split loop iterations over multiple threads.
- Threads can execute different code but share the same address space.
- OpenMP is most often used on machines with support for a global address space.

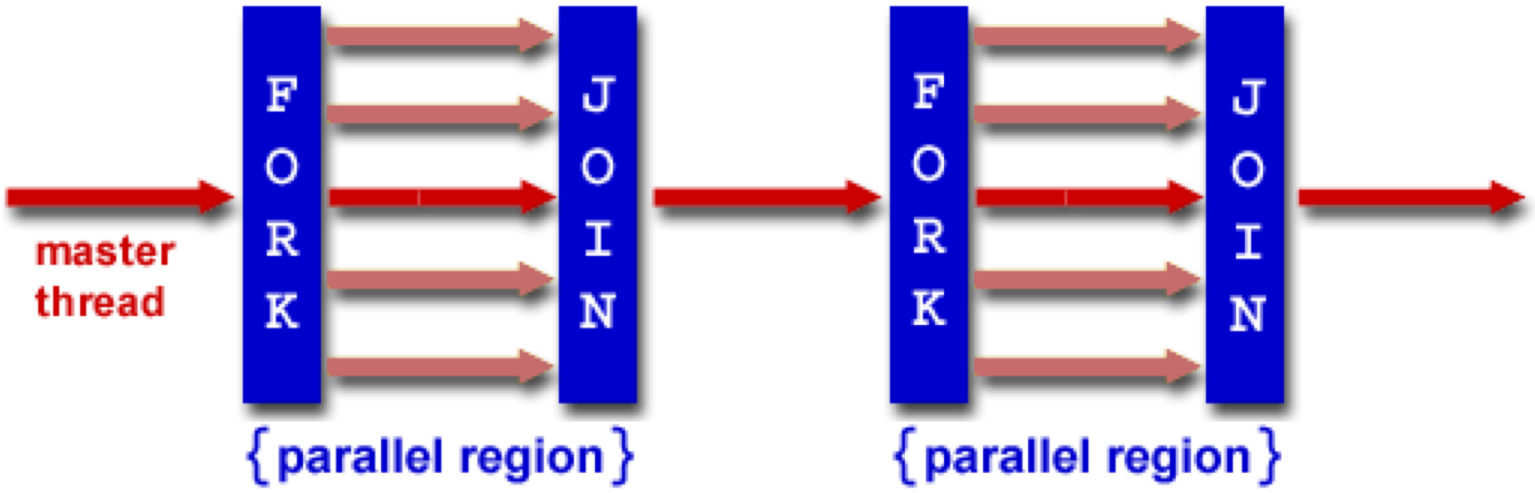
# Thread Based Parallelism

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads matches the number of machine processors/cores. However, the actual use of threads is up to the application.

# Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- In the simplest case, OpenMP is used to parallelize a serial program by inserting compiler directives.
- The OpenMP API is comprised of three distinct components.
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

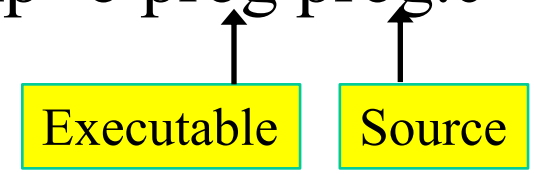
# OpenMP Fork/Join Model



# Compiling An OpenMP Program

- Most modern compilers have support for OpenMP. Check the documentation of your compiler to see which version it supports.
- gcc version 6.1 supports OpenMP 4.5
- Compile OpenMP programs as follows:

```
gcc -fopenmp -o prog prog.c
```



# Linux Lab Machines

- Named after chemical elements: hydrogen, helium, lithium, etc.
- C compiler: gcc 7.3.0
- OpenMP 4.5

# Simple Example of OpenMP

```
#include <omp.h>
#include <stdio.h>
main () {
#pragma omp parallel
{
   int n=omp_get_num_threads();
   int tid=omp_get_thread_num();
   printf("There are %d threads. Hello from thread %d\n",n,tid);
}
/* end of parallel section */
   printf("Hello from the master thread\n");
}
```

Need this header file

Create/fork threads

Call to OMP runtime library

Destroy/join threads

If there are n threads they are given unique IDs 0, 1, 2,…,n-1

There are 2 threads. Hello from thread 1

There are 2 threads. Hello from thread 0

Hello from the master thread

# Setting the Number of Threads from the Shell

- Set environment variable as follows:

```
export OMP_NUM_THREADS=8
```

- Now output is:

```
There are 8 threads. Hello from thread 2
There are 8 threads. Hello from thread 1
There are 8 threads. Hello from thread 3
There are 8 threads. Hello from thread 4
There are 8 threads. Hello from thread 6
There are 8 threads. Hello from thread 0
There are 8 threads. Hello from thread 5
There are 8 threads. Hello from thread 7
Hello from the master thread
```

# Setting the Number of Threads in the Code

```
#include <omp.h>
#include <stdio.h>
main () {
#pragma omp parallel num_threads(4)
{
   int n=omp_get_num_threads();
   int tid=omp_get_thread_num();
   printf("There are %d threads. Hello from thread %d\n",n,tid);
}
/* end of parallel section */
   printf("Hello from the master thread\n");
}
```

```
There are 4 threads. Hello from thread 1
There are 4 threads. Hello from thread 3
There are 4 threads. Hello from thread 2
There are 4 threads. Hello from thread 0
Hello from the master thread
```

# Yet Another Way

```
#include <omp.h>
#include <stdio.h>
main () {
omp_set_num_threads(6);
#pragma omp parallel
{
   int n=omp_get_num_threads();
   int tid=omp_get_thread_num();
   printf("There are %d threads. Hello from
thread %d\n",n,tid);
}
/* end of parallel section */
   printf("Hello from the master thread\n");
}
```

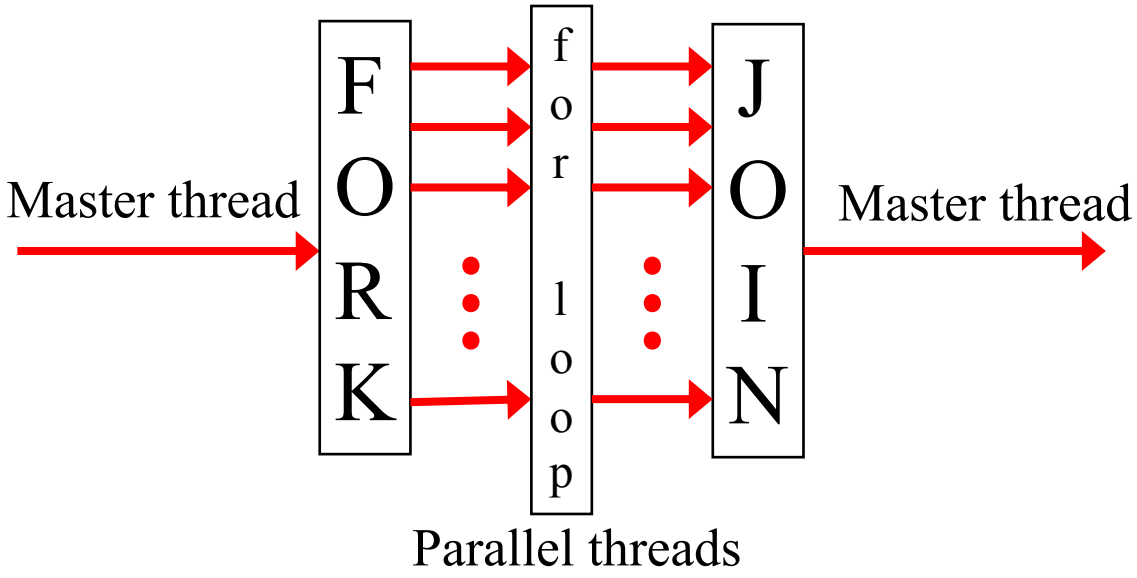
Call OMP runtime library

```
There are 6 threads. Hello from thread 1
There are 6 threads. Hello from thread 5
There are 6 threads. Hello from thread 0
There are 6 threads. Hello from thread 3
There are 6 threads. Hello from thread 4
There are 6 threads. Hello from thread 2
Hello from the master thread
```

# Work-Sharing Constructs

- **For**: shares iterations of a loop across the team of threads. Represents a type of data parallelism.
- **Task**: creates tasks that are later assigned to threads. Represents a type of task parallelism.
- **SECTIONS**: breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of task (or functional) parallelism.
- **SINGLE**: serializes a section of code.

# For Loop: Data Parallelism



# Static Loops

```
#include <omp.h>
#include <stdio.h>
#define CHUNKSIZE 100
#define N 1000
main () {
   int i, chunk, t, n;
   float a[N], b[N], c[N];
   for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i,n,t) num_threads(6)
   {
      #pragma omp for schedule(static,chunk)
      for (i=0; i < N; i++){
          c[i] = a[i] + b[i];
          n = i/chunk;
          t = omp_get_thread_num();
          if(i%chunk==0) printf("Thread %d is doing chunk %d\n",t,n);
      }
   }
}
```

Variable *i* is private so each thread has its own copy

Statically schedules successive chunk iterations to the same thread in round-robin way.

# Splitting the Loop into Chunks

- Chunk 0 is iterations i = 0 to chunk-1
- Chunk 1 is iterations i=chunk to 2*chunk-1, and so on.
- In general, chunk c is iterations c*chunk to (c+1)*chunk – 1
- For static scheduling chunk c is assigned to thread c%n, where n is the number of threads.

# Output for 6 Threads, 100 Iterations per Chunk, and i=0 to 999

Thread 1 is doing chunk 1
Thread 1 is doing chunk 7
Thread 0 is doing chunk 0
Thread 3 is doing chunk 3
Thread 2 is doing chunk 2
Thread 5 is doing chunk 5
Thread 3 is doing chunk 9
Thread 2 is doing chunk 8
Thread 0 is doing chunk 6
Thread 4 is doing chunk 4

Static Scheduling

| Chunk | First i in chunk | Last i in chunk | Thread Number |
|---|---|---|---|
| 0 | 0 | 99 | 0 |
| 1 | 100 | 199 | 1 |
| 2 | 200 | 299 | 2 |
| 3 | 300 | 399 | 3 |
| 4 | 400 | 499 | 4 |
| 5 | 500 | 599 | 5 |
| 6 | 600 | 699 | 0 |
| 7 | 700 | 799 | 1 |
| 8 | 800 | 899 | 2 |
| 9 | 900 | 999 | 3 |

# Dynamic Loops

```
#include <omp.h>
#include <stdio.h>
#define CHUNKSIZE 100
#define N 1000
main () {
   int i, chunk, t, n;
   float a[N], b[N], c[N];
   for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i,n,t) num_threads(6)
   {
      #pragma omp for schedule(dynamic,chunk)
      for (i=0; i < N; i++){
          c[i] = a[i] + b[i];
          n = i/chunk;
          t = omp_get_thread_num();
          if(i%chunk==0) printf("Thread %d is doing chunk %d\n",t,n);
      }
   }
}
```

Variable *i* is private so each thread has its own copy

Dynamically schedules chunks when a thread finishes one chunk it starts on another.

# Output for 6 Threads, 100 Iterations per Chunk, and i=0 to 999

Thread 0 is doing chunk 1
Thread 1 is doing chunk 0
Thread 1 is doing chunk 7
Thread 1 is doing chunk 8
Thread 1 is doing chunk 9
Thread 2 is doing chunk 2
Thread 3 is doing chunk 3
Thread 4 is doing chunk 4
Thread 5 is doing chunk 5
Thread 0 is doing chunk 6

Dynamic Scheduling

| Chunk | First i in chunk | Last i in chunk | Thread Number |
|---|---|---|---|
| 0 | 0 | 99 | 1 |
| 1 | 100 | 199 | 0 |
| 2 | 200 | 299 | 2 |
| 3 | 300 | 399 | 3 |
| 4 | 400 | 499 | 4 |
| 5 | 500 | 599 | 5 |
| 6 | 600 | 699 | 0 |
| 7 | 700 | 799 | 1 |
| 8 | 800 | 899 | 1 |
| 9 | 900 | 999 | 1 |

# Guided Scheduling

- Similar to dynamic scheduling, except that the block size decreases each time a chunk of work is given to a thread.
- The size of the initial block is proportional to: number_of_iterations/number_of_threads
- Subsequent blocks are proportional to: number_of_iterations_remaining / number_of_threads
- The chunk parameter defines the minimum block size. The default chunk size is 1.

# Guided Loops

```
#include <omp.h>
#include <stdio.h>
#define CHUNKSIZE 100
#define N 1000
main () {
   int i, chunk, t,;
   float a[N], b[N], c[N];
   for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i,t) num_threads(6)
   {
      #pragma omp for schedule(guided,chunk)
      for (i=0; i < N; i++){
          c[i] = a[i] + b[i];
          t = omp_get_thread_num();
          printf("Thread %d is doing iteration %d\n",t,i);
      }
   }
}
```

Variable *i* is private so each thread has its own copy

Dynamically schedules chunks but chunk size can decrease.

To sort output in order of i: forloop3 | sort –n –k6

# Output for 6 Threads, 100 Iterations per Chunk, and i=0 to 999

Scheduling of chunks to threads may differ between runs, and the size of the chunks may differ between machines. Chunk size and scheduling depends on the runtime system.

| Chunk | First i in chunk | Last i in chunk | Thread number | Chunk size |
|---|---|---|---|---|
| 0 | 0 | 166 | 4 | 167 |
| 1 | 167 | 305 | 1 | 139 |
| 2 | 306 | 421 | 3 | 116 |
| 3 | 422 | 521 | 0 | 100 |
| 4 | 522 | 621 | 5 | 100 |
| 5 | 622 | 721 | 2 | 100 |
| 6 | 722 | 821 | 0 | 100 |
| 7 | 822 | 921 | 5 | 100 |
| 8 | 922 | 999 | 2 | 78 |

Guided Scheduling

# Runtime Scheduling

- With *schedule(runtime),* scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- Example: export OMP_SCHEDULE="static,10"
- Can also specify *schedule(auto)*, which means the compiler and/or runtime system makes all scheduling decisions.

# Runtime Loops

```
#include <omp.h>
#include <stdio.h>
#define N 1000
main () {
   int i, t,;
   float a[N], b[N], c[N];
   for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
   #pragma omp parallel shared(a,b,c) private(i,t) num_threads(6)
   {
      #pragma omp for schedule(runtime)
      for (i=0; i < N; i++){
          c[i] = a[i] + b[i];
          t = omp_get_thread_num();
          printf("Thread %d is doing iteration %d\n",t,i);
      }
   }
}
```

Schedules chunks as specified by OMP_SCHEDULE environment variable.

To sort output in order of i: forloop4 | sort –n –k6

# Parallelizing Nested Loops

```
#include <omp.h>
#include <stdio.h>
#define CHUNKSIZE 20
#define N 10
main () {
   int i, j, chunk, k[N][N];
   float a[N][N], b[N][N], c[N][N];
   for (i=0; i < N; i++)
       for (j=0;j<N; j++) {a[i][j] = (i*N+j) * 1.0; b[i][j] = (i+j*N)*1.0;}
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i,j) num_threads(5)
   {
      #pragma omp for schedule(static,chunk) nowait
         for (i=0; i < N; i++)
               for (j=0; j<N; j++){
                     c[i][j] = a[i][j] + b[i][j];
                     k[i][j] = omp_get_thread_num();
                }
   }
    for (i=0; i < N; i++)
       for (j=0;j<N; j++)
          printf("%2d%c",k[i][j],(j==(N-1)? '\n' : ' '));
}
```

What will be the output?

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

# Using the *collapse* directive

```
#include <omp.h>
#include <stdio.h>
#define CHUNKSIZE 20
#define N 10
main () {
   int i, j, chunk, k[N][N];
   float a[N][N], b[N][N], c[N][N];
   for (i=0; i < N; i++)
       for (j=0;j<N; j++) {a[i][j] = (i*N+j) * 1.0; b[i][j] = (i+j*N)*1.0;}
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i,j) num_threads(5)
   {
      #pragma omp for collapse(2) schedule(static,chunk) nowait
         for (i=0; i < N; i++)
               for (j=0; j<N; j++){
                    c[i][j] = a[i][j] + b[i][j];
                    k[i][j] = omp_get_thread_num();
                }
   }
    for (i=0; i < N; i++)
       for (j=0;j<N; j++)
          printf("%2d%c",k[i][j],(j==(N-1)? '\n' : ' '));
}
```

What will be the output?

collapse(n) means "collapse the next n nested loops into a single iteration space, and then divide into chunks

# The Output

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4
```

- The size of each chunk is 20 iterations.
- Chunk 0 consists of the iterations for i=0 and i=1. This gets assigned to thread 0.
- Chunk 1 consists of iterations i=2 and i=3. This gets assigned to thread 1.
- And so on.

# The Output for a Chunk Size = 5

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |

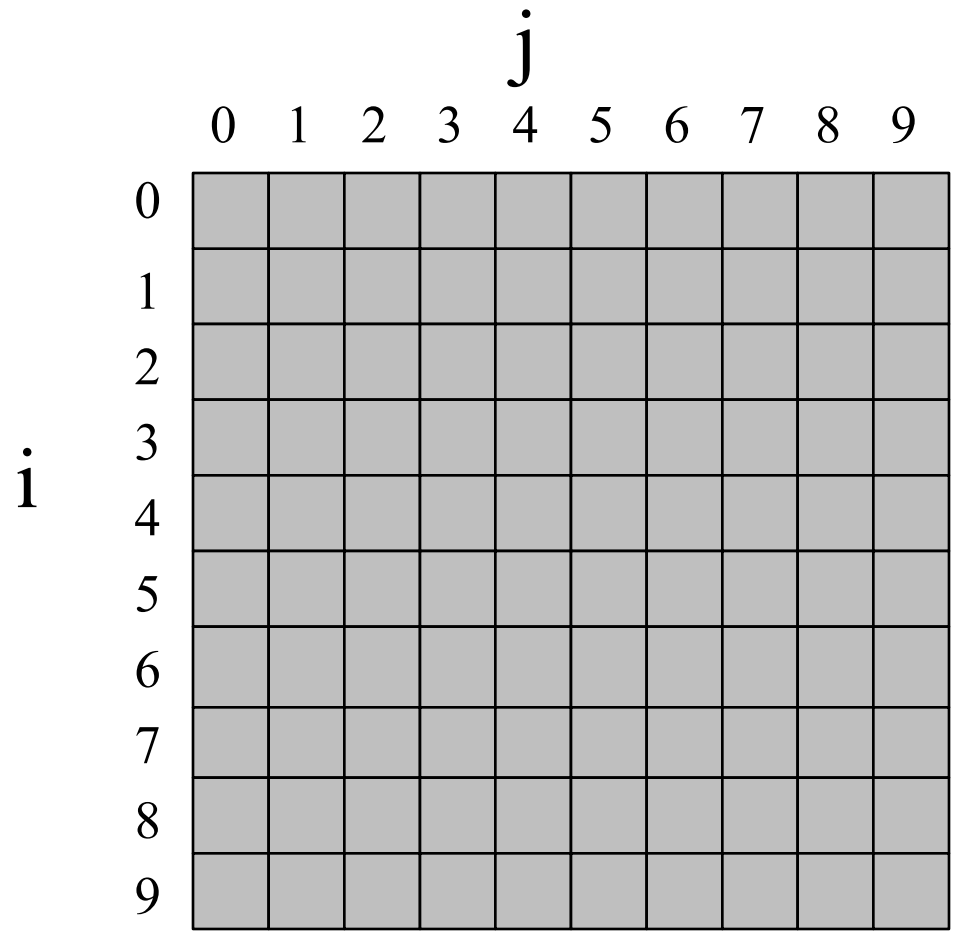
- Now set the size of each chunk to 5 iterations.
- Chunk 0 consists of the iterations for i=0 from j=0 to 4. This gets assigned to thread 0.
- Chunk 1 consists of iterations i=0 from j=5 to 9. This gets assigned to thread 1.
- And so on.

# The Output for a Chunk Size = 7

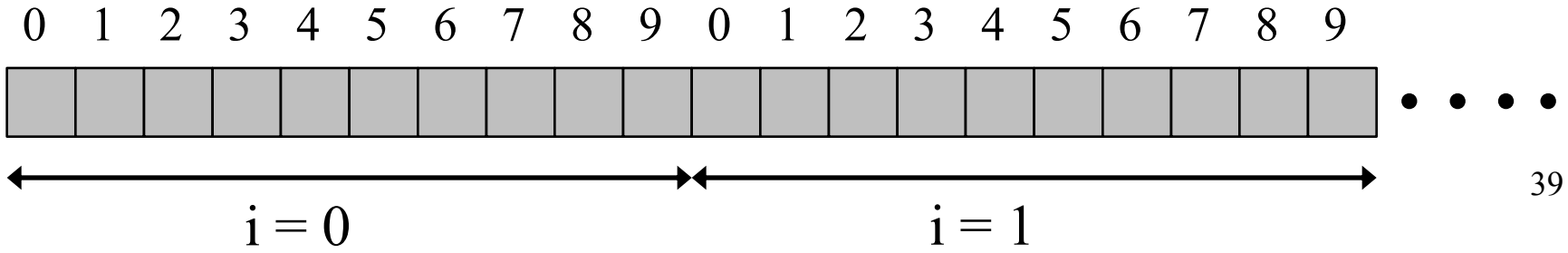
```
0 0 0 0 0 0 0 1 1 1
1 1 1 1 2 2 2 2 2 2
2 3 3 3 3 3 3 3 4 4
4 4 4 4 4 0 0 0 0 0
0 0 1 1 1 1 1 1 1 2
2 2 2 2 2 2 3 3 3 3
3 3 3 4 4 4 4 4 4 4
0 0 0 0 0 0 0 1 1 1
1 1 1 1 2 2 2 2 2 2
2 3 3 3 3 3 3 3 4 4
```

- Now set the size of each chunk to 7 iterations.
- Chunk 0 consists of the iterations for i=0 from j=0 to 6. This gets assigned to thread 0.
- Chunk 1 consists of iterations i=0 from j=7 to 9 and i=1 from 0 to 3. This gets assigned to thread 1.
- And so on.

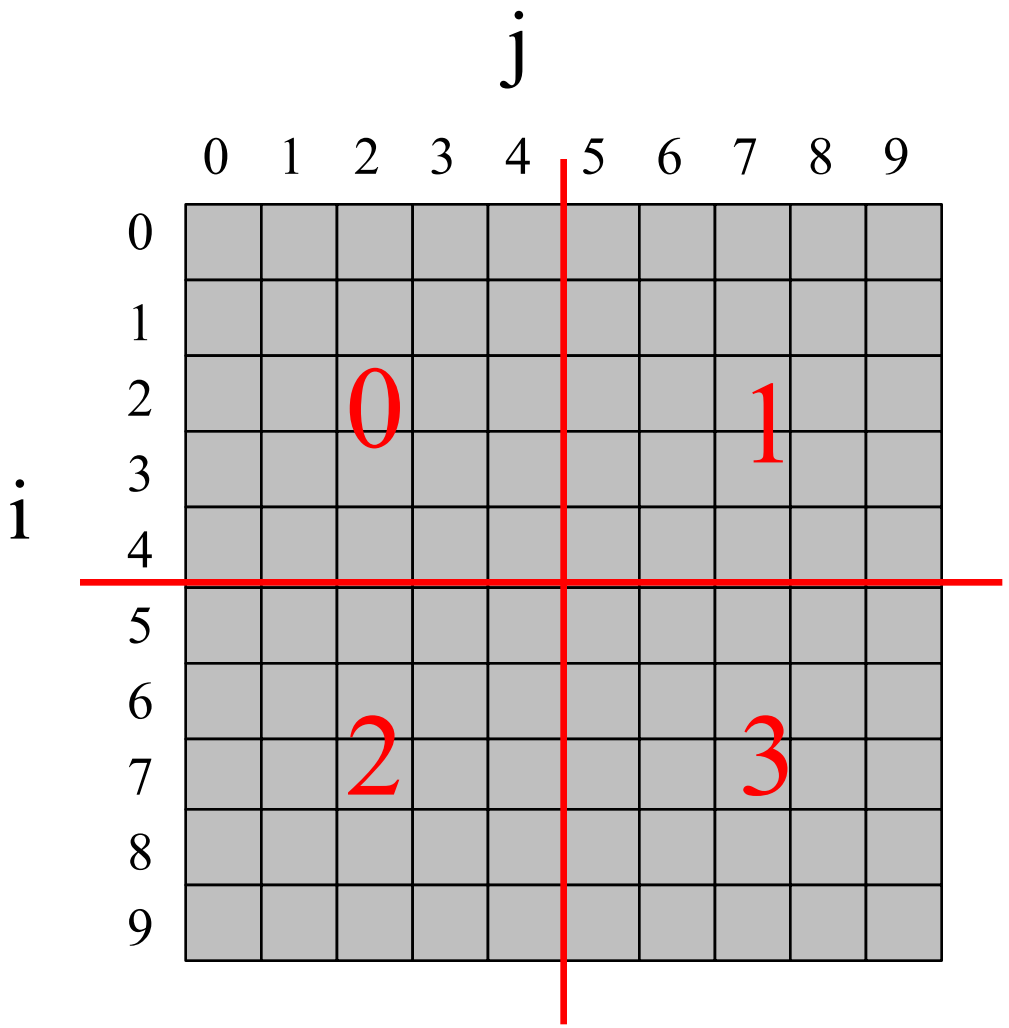
# 2D Loops ➔ 1D Iteration Space



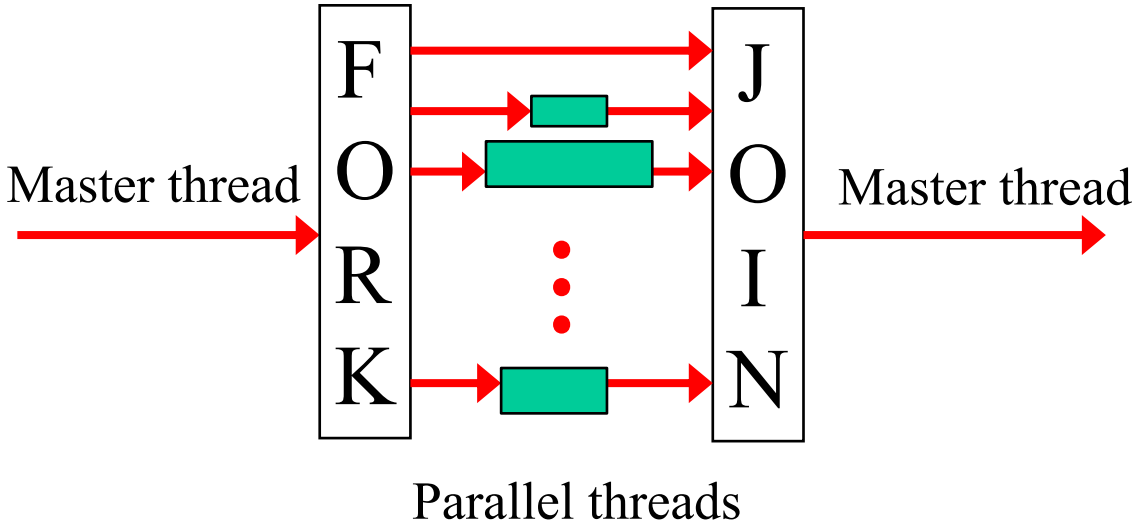

Lay the rows end to end



# Is This Partitioning Possible?



# Sections: TaskParallelism



# Sections Example

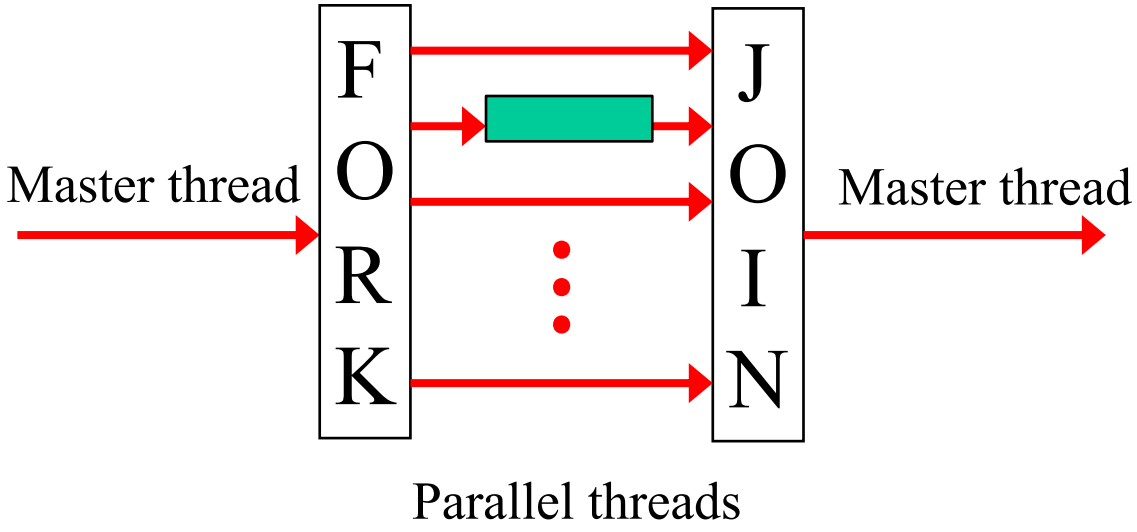
```
#include <omp.h>
#include <stdio.h>
#define N 1000
main ()
{
   int i, t;
   float a[N], b[N], c[N], d[N];
   for (i=0; i < N; i++) { a[i] = i * 1.5; b[i] = i + 22.35;}
#pragma omp parallel shared(a,b,c,d) private(i,t) num_threads(2)
  {
   t = omp_get_thread_num();
  #pragma omp sections
   {
   #pragma omp section
     {
      printf("Thread %d is doing the first section\n",t);
      for (i=0; i < N; i++) c[i] = a[i] + b[i];
     }
   #pragma omp section
     {
      printf("Thread %d is doing the second section\n",t);
      for (i=0; i < N; i++) d[i] = a[i] * b[i];
     }
   }  /* end of sections */
  }  /* end of parallel section */
}
```

Thread 0 is doing the first section
Thread 1 is doing the second section

# Notes on Sections

- If there are more threads than sections, some threads will not execute a section and some will. If there are more sections than threads, the OpenMP implementation defines how the extra sections are executed.
- It is up to the OpenMP implementation to decide which threads will execute a section and which threads will not, and it can vary from execution to execution.

# Single: Sequential Execution



# Example of Use of Single

```
#include <omp.h>
#include <stdio.h>
main () {
#pragma omp parallel
{
  int tid=omp_get_thread_num();
#pragma omp single
  {
    int n=omp_get_num_threads();
    printf("There are %d threads. This was output by thread %d\n",n,tid);
  }
  printf("Hello from thread %d\n",tid);
}
printf("Hello from the master thread\n");
}
```

Variables declared within a parallel section are automatically private.

Only one thread does this

# Example Output For 8 Threads

```
There are 8 threads. This was output by thread 1
Hello from thread 6
Hello from thread 1
Hello from thread 4
Hello from thread 5
Hello from thread 2
Hello from thread 3
Hello from thread 0
Hello from thread 7
Hello from the master thread
```

# Reduction Operations

```
#include <omp.h>
#include <stdio.h>
main () {
   int i, n, chunk;
   float a[100], result;
   n = 100;
   chunk = 10;
   for (i=0; i < n; i++) a[i] = i * 1.0;
    result = 0.0;
   #pragma omp parallel for default(shared) private(i) schedule(static,chunk) reduction(+:result)
   for (i=0; i < n; i++) result = result + a[i] ;
   printf("Average = %f\n",result/(float)n);
}
```

Equivalent to:

**#pragma omp parallel default(shared) private(i)**

**#pragma omp for schedule(static,chunk) reduction(+:result)**

Don't need braces around the **for** because the parallel section is only one line long.

printf is done by master thread only.

Average = 49.500000

# Multiple Reduction Operations

```
#include <omp.h>
#include <stdio.h>
main () {
   int i, n, chunk, result3=0;
   float a[100], b[100], result1, result2;
   n = 100;
   chunk = 10;
   result1 = result2 = 0.0;
   for (i=0; i < n; i++) {
      a[i] = i * 1.0;
      b[i] = i * 2.0;
   }
   #pragma omp parallel for default(shared) private(i) schedule(static,chunk)   \
              reduction(+:result1, result2)  reduction (|:result3)
   for (i=0; i < n; i++) {
      result1 = result1 + a[i] ;
      result2 = result2 + b[i];
      result3 = result3|i;
  }
   printf("Finally, result1 = %f, result2 = %f, and result3 = %d\n",result1,result2,result3);
}
```

Continues line.

Bitwise inclusive *or* operation.

Finally, result1 = 4950.000000, result2 = 9900.000000, and result3 = 127

# Min/Max Reduction Operations

- Unlike Fortran, C/C++ doesn't have intrinsic functions for min and max.
- Thus, in C we can't use the OpenMP *reduction* clause to find the minimum or maximum of an array.
- Instead we use the *critical* directive.

# Use of Critical Directive

```
#include <omp.h>
#include <stdio.h>
#include <float.h>
int main (int argc, const char * argv[]) {
        int i, n, chunk;
        float a[100], b[100], prod, minval;
        /* Some initializations */
        n = 100;
        chunk = 10;
        for (i=0; i < n; i++) {
                a[i] = i * 1.0;
                b[i] = i * 2.0;
        }
        minval = FLT_MAX;
#pragma omp parallel for default(shared) private(i,prod) schedule(static,chunk)
        for (i=0; i < n; i++) {
                prod = a[i]*b[i];
#pragma omp critical
                minval = (prod < minval ? prod : minval);
        }
        printf("Minimum value = %f\n",minval);
}
```

Initialize to largest possible float value

This line is a critical section

Minimum value = 0.000000

# Synchronization Constructs

- **Barrier**: No thread can proceed past the barrier until all the threads reach it.
- **Critical**: Provides mutually exclusive access to a section of code.
- **Atomic**: specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it.

# Barrier Directive

```
#include <omp.h>
#include <stdio.h>
main()
{
   int k, x[10], y=0, z=0;
#pragma omp parallel shared(x,y,z) private(k) num_threads(2)
{
   int t = omp_get_thread_num();
   if (t==0) y = 10;
   else z = 20;
   #pragma omp barrier
   #pragma omp for
   for (k=0;k<10;k++) x[k] = y + z + k;
}
   printf("x = ");
   for(k=0;k<10;k++) printf("%d, ",x[k]);
   printf("\n");
}
```

If this barrier is removed thread 0 may not set its value of y before thread 1 executes its part of the for loop. Or thread 1 may not set its value of z before thread 0 executes its part of the for loop

The output (with barrier):

x = 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,

# Critical Directive

```
#include <omp.h>
#include <stdio.h>
main()
{
  int x=0, y=0;
#pragma omp parallel shared(x,y) num_threads(6)
  {
    int t = omp_get_thread_num();
    #pragma omp critical
    {
      x = x + t;
      y = y + 1;
    }
  }
  printf("x = %d, y = %d\n",x,y);
}
```

Only one thread at a time can enter the critical section.

The output:

x = 15, y = 6

# Atomic Directive

- The *atomic* directive applies only to a single, immediately following statement.
- Can only be applied to simple operations. Cannot be applied to function calls or array indexing
- It is like a mini-critical section.
- In general, using *atomic* is more efficient than using *critical*, especially if the hardware supports atomic updates.

# Atomic Directive

```
#include <omp.h>
#include <stdio.h>
main()
{
  int x=0;
#pragma omp parallel shared(x) num_threads(6)
  {
    int t = omp_get_thread_num();
    #pragma omp atomic
      x += t;
  }
  printf("x = %d\n",x);
}
```

x is updated atomically. i.e., as a single uninterruptable action

The output:

x = 15

# Data Scope Attribute Clauses

- We have already met *default*, *shared* and *private* clauses.
- In addition there are 3 more:
  - *firstprivate*
  - *lastprivate*
  - *copyin*
- Note that when a private variable is created for each thread in a parallel section, it should not be assumed that it is initialized.

# Without *firstprivate* Clause

```
#include <omp.h>
#include <stdio.h>
main () {
   int x = 10;
#pragma omp parallel private(x) num_threads(4)
{
   int t  = omp_get_thread_num();
   int tx = x+t;
   printf("tx = %d on thread %d\n",tx,t);
}
}
```

Typical output:

tx = 0 on thread 0
tx = 1391744771 on thread 3
tx = 2 on thread 2
tx = 1 on thread 1

# With *firstprivate* Clause

```
#include <omp.h>
#include <stdio.h>
main () {
   int x = 10;
#pragma omp parallel firstprivate(x) num_threads(4)
{
   int t  = omp_get_thread_num();
   int tx = x+t;
   printf("tx = %d on thread %d\n",tx,t);
}
}
```

Output:

tx = 10 on thread 0
tx = 13 on thread 3
tx = 12 on thread 2
tx = 11 on thread 1

# Without *lastprivate* Clause

```
#include <omp.h>
#include <stdio.h>
main () {
   int i = 10;
#pragma omp parallel private(i) num_threads(4)
{
   int x = 1;
   #pragma omp for
   for (i=0;i<100;i++) x++;
}
   printf("i = %d\n",i);
}
```

Value of *i* in parallel section is not copied back to master thread, so *i* does not change in this case.

Output:

i = 10

# With *lastprivate* Clause

```
#include <omp.h>
#include <stdio.h>
main () {
   int i = 10;
#pragma omp parallel num_threads(4)
{
   int x = 1;
   #pragma omp for lastprivate(i)
   for (i=0;i<100;i++) x++;
}
   printf("i = %d\n",i);
}
```

Output:

i = 100

# Threadprivate Variables

- A *private* variable is local to a parallel region.
- A *threadprivate* variable persist across parallel regions (depending on some restrictions). The master thread uses the original variable, all other threads make a private copy of the original variable.

# Threadprivate Example

```
#include <omp.h>
#include <stdio.h>
int a, b, tid;
#pragma omp threadprivate(a)
main(int argc, char *argv[]) {
   omp_set_dynamic(0);
   printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
{
   tid = omp_get_thread_num();
   a = tid; b = tid;
   printf("Thread %d: a,b= %d %d %f\n",tid,a,b);
} /* end of parallel region */
   printf("Master thread doing serial work here\n");
   printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
{
   tid = omp_get_thread_num();
   printf("Thread %d: a,b= %d %d\n",tid,a,b);
} /* end of parallel region */
}
```

Varial

Varial

```
1st Parallel Region:
Thread 3: a,b= 3 3
Thread 1: a,b= 1 1
Thread 2: a,b= 2 2
Thread 0: a,b= 0 0
Thread 5: a,b= 5 5
Thread 7: a,b= 7 7
Thread 6: a,b= 6 6
Thread 4: a,b= 4 4
Master thread doing serial work
2nd Parallel Region:
Thread 3: a,b= 3 0
Thread 1: a,b= 1 0
Thread 0: a,b= 0 0
Thread 5: a,b= 5 0
Thread 7: a,b= 7 0
Thread 2: a,b= 2 0
Thread 6: a,b= 6 0
Thread 4: a,b= 4 0
```

# Copyin Clause

- The *copyin* clause can be used in a *parallel* directive to initialize a *threadprivate* variable in each thread to the value in the master thread.
- Similar to *firstprivate*.

# Copyin Example

```
#include <omp.h>
#include <stdio.h>
int a, tid;
#pragma omp threadprivate(a)
main(int argc, char *argv[]) {
   omp_set_dynamic(0);
   a = 10;
   printf("1st Parallel Region:\n");
#pragma omp parallel private(tid) copyin(a)
{
   tid = omp_get_thread_num();
   a += tid;
   printf("Thread %d: a = %d\n",tid,a);
} /* end of parallel region */
   printf("Master thread doing serial work: a=%d\n",a);
   printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid) copyin(a)
{
   tid = omp_get_thread_num();
   printf("Thread %d: a= %d\n",tid,a);
} /* end of parallel region */
}
```

Variable *a* is threadprivate

Copyin variable *a*

# Copyin Example Output

Output with *copyin* for 2nd parallel region

```
1st Parallel Region:
Thread 1: a = 11
Thread 4: a = 14
Thread 3: a = 13
Thread 6: a = 16
Thread 0: a = 10
Thread 2: a = 12
Thread 5: a = 15
Thread 7: a = 17
Master thread doing serial work: a=10
2nd Parallel Region:
Thread 1: a= 10
Thread 3: a= 10
Thread 2: a= 10
Thread 6: a= 10
Thread 5: a= 10
Thread 4: a= 10
Thread 7: a= 10
Thread 0: a= 10
```

Output with no *copyin* for 2nd parallel region

```
1st Parallel Region:
Thread 1: a = 11
Thread 5: a = 15
Thread 6: a = 16
Thread 2: a = 12
Thread 0: a = 10
Thread 7: a = 17
Thread 3: a = 13
Thread 4: a = 14
Master thread doing serial work: a=10
2nd Parallel Region:
Thread 2: a= 12
Thread 6: a= 16
Thread 1: a= 11
Thread 7: a= 17
Thread 5: a= 15
Thread 3: a= 13
Thread 4: a= 14
Thread 0: a= 10
```

# Copyprivate Clause

- The *copyprivate* clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

# Copyprivate Example

```
#include <stdio.h>
main () {
#pragma omp parallel num_threads(6)
{
   int tid = omp_get_thread_num();
   int a = tid;
   int x = 17;
#pragma omp single copyprivate(x)
{
   a = 10;
   x = 20;
}
   printf("Thread %d: a=%d, x=%d\n",tid,a,x);
}
}
```

Variables *a, x* and *tid* are private

Value of *x* will be broadcast to all threads

```
Thread 1: a=1, x=20
Thread 0: a=10, x=20
Thread 3: a=3, x=20
Thread 4: a=4, x=20
Thread 2: a=2, x=20
Thread 5: a=5, x=20
```

# Copyprivate Example 2

```
#include <stdio.h>
main () {
#pragma omp parallel num_threads(6)
{
   int tid = omp_get_thread_num();
   int a = tid;
   int x = 17;
#pragma omp single copyprivate(a, x)
{
   a = 10;
   x = 20;
}
   printf("Thread %d: a=%d, x=%d\n",tid,a,x);
}
}
```

Variables *a, x* and *tid* are private

Value of *a* and x will be broadcast to all threads

```
Thread 1: a=10, x=20
Thread 4: a=10, x=20
Thread 2: a=10, x=20
Thread 0: a=10, x=20
Thread 3: a=10, x=20
Thread 5: a=10, x=20
```

# Coursework 1: OpenMP

- Download the code blur.c and the file David.ps from Learning Central.
- The code repetitively blurs the image in David.ps by replacing the RGB pixel values by the average of their 4 neighbours.
- Your tasks are to:
  - Use OpenMP to parallelize the code.
  - Perform timing runs for differing numbers of threads, scheduling methods, and chunk sizes.
  - Write a short report on your findings.

# Interconnection Networks

- Parallel computers with many processors do not use shared memory hardware.
- Instead each processor has its own local memory and data communication takes place via message passing over an *interconnection network*.
- The characteristics of the interconnection network are important in determining the performance of a multicomputer.
- If network is too slow for an application, processors may have to wait for data to arrive.

# Examples of Networks

Important networks include:

- fully connected or all-to-all
- mesh
- ring
- hypercube
- shuffle-exchange
- butterfly
- cube-connected cycles

# Network Metrics

A number of metrics can be used to evaluate and compare interconnection networks.

- **Network connectivity** is the minimum number of nodes or links that must fail to partition the network into two or more disjoint networks.
- Network connectivity measures the resiliency of a network, and its ability to continue operation despite disabled components. When components fail we would like the network to continue operation with reduced capacity.

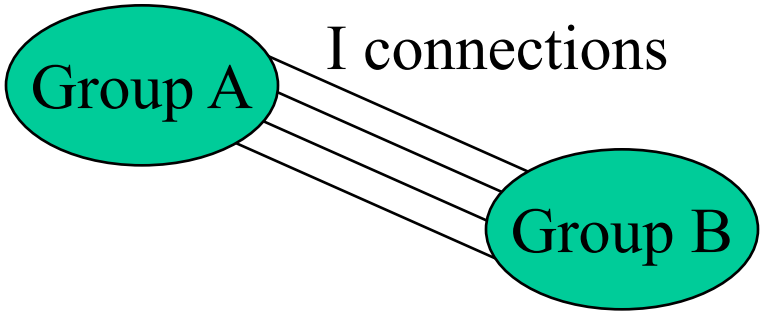
# Network Metrics 2

- **Bisection width** is the minimum number of links that must be cut to partition the network into two equal halves (to within one). The *bisection bandwidth* is the bisection width multiplied by the data transfer rate of each link.
- **Network diameter** is the maximum internode distance, i.e., the maximum number of links that must be traversed to send a message to any node along the shortest path. The lower the network diameter the shorter the time to send messages to distant nodes.

# Network Metrics 3

**Network narrowness** measures congestion in a network.

- Partition the network into two groups A and B, containing NA and NB nodes, respectively, with NB ≤ NA.
- Let I be the number of connections between nodes in A and nodes in B. The narrowness of the network is the maximum value of NB/I for all partitionings of the network.



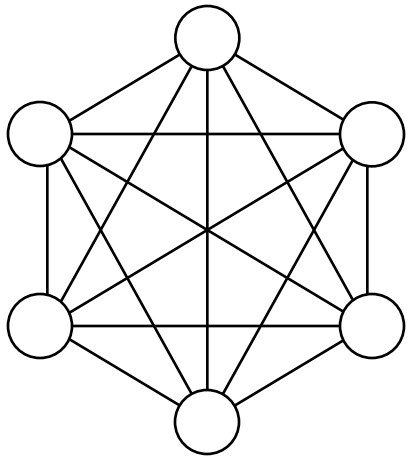

- If the narrowness is high (NB>I) then if the group B nodes want to communicate with the group A nodes congestion in the network will be high.

# Network Metrics 4

- **Network Expansion Increment** is the minimum number of nodes by which the network can be expanded.
  - A network should be expandable to create larger and more powerful parallel systems by simply adding more nodes to the network.
  - For reasons of cost it is better to have the option of small increments since this allows you to upgrade your machine to the required size.
- **Number of edges per node**. If this is independent of the size of the network then it is easier to expand the system.

# Fully Connected Network

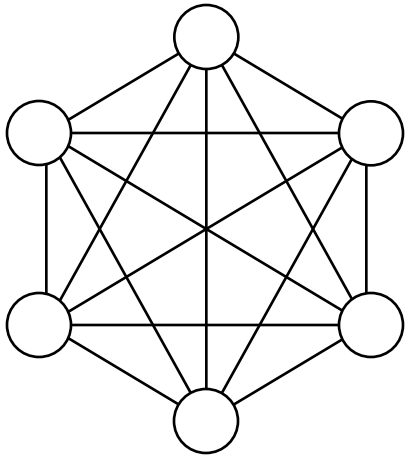
- In the fully connected, or all-to-all, network each node is connected directly to all other nodes.
- This is the most general and powerful interconnection network, but it can be implemented for only a small number of nodes.



# Fully Connected Network 2

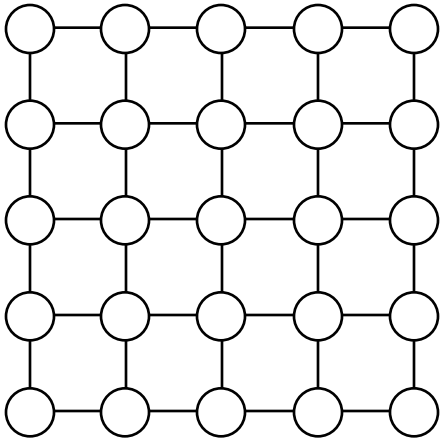
For n even:

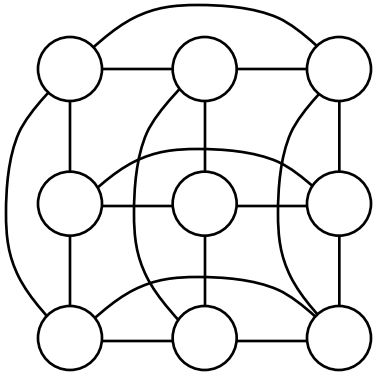
- Network connectivity = n - 1
- Network diameter = 1
- Network narrowness = 2/n
- Bisection width = $n^2/4$
- Expansion Increment = 1
- Edges per node = n - 1



# Mesh Networks

- In a mesh network nodes are arranged as a q-dimensional lattice, and communication is allowed only between neighboring nodes.
- In a *periodic mesh*, nodes on the edge of the mesh have wrap-around connections to nodes on the other side. This is sometimes called a *toroidal mesh*.

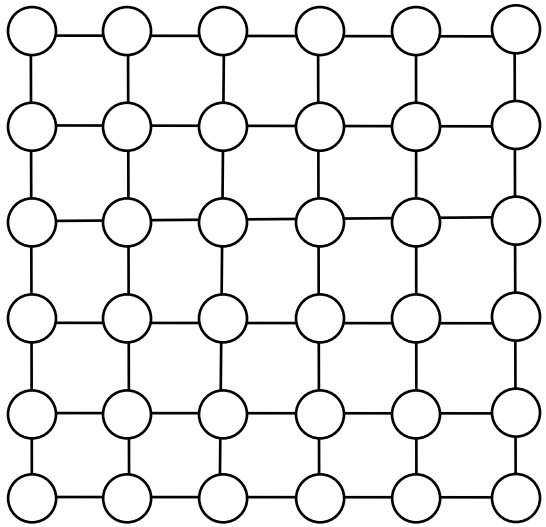




# Mesh Metrics

For a q-dimensional non-periodic lattice with $k^q$ nodes:

- Network connectivity = q
- Network diameter = q(k-1)
- Network narrowness = k/2
- Bisection width = $k^{q-1}$
- Expansion Increment = $k^{q-1}$
- Edges per node = 2q



# Ring Networks

A simple ring network is just a 1D periodic mesh.

- Network connectivity = 2
- Network diameter = n/2
- Network narrowness = n/4
- Bisection width = 2
- Expansion Increment = 1
- Edges per node = 2

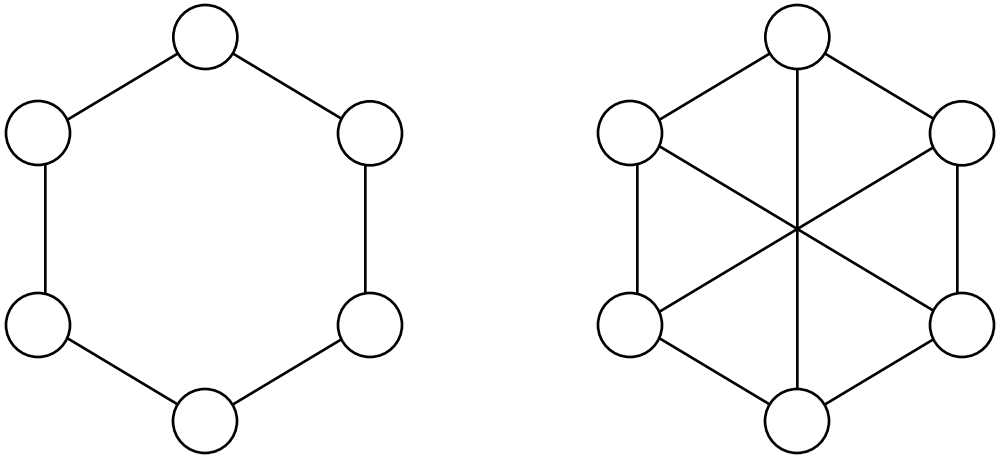
The problem for a simple ring is its large diameter.

# Chordal Ring Networks

- A *chordal ring* uses extra chordal links to reduce the diameter.
- For a ring with extra diametric links we have (for n even)
  - Network connectivity = 3
  - Network diameter = ceiling(n/4)
  - Network narrowness = n/(n+4)
  - Bisection width = 2+n/2
  - Expansion Increment = 2
  - Edges per node = 3

# Examples of Ring Networks

- Here are a simple ring and a chordal ring with diametric links, each of size 6 nodes.

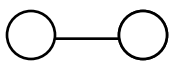


# Hypercube Networks

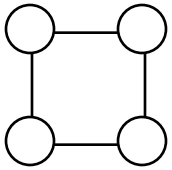
- A hypercube network consists of $n=2^k$ nodes arranged as a k-dimensional hypercube. Sometimes called a *binary n-cube*.
- Nodes are numbered 0, 1,…,n-1, and two nodes are connected if their node numbers differ in exactly one bit.
  - Network connectivity = k
  - Network diameter = k
  - Network narrowness = 1
  - Bisection width = $2^{k-1}$
  - Expansion increment = $2^k$
  - Edges per node = k

# Examples of Hypercubes

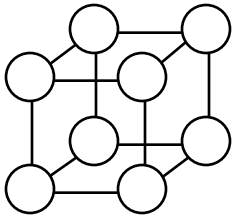
See http://en.wikipedia.org/wiki/Hypercube



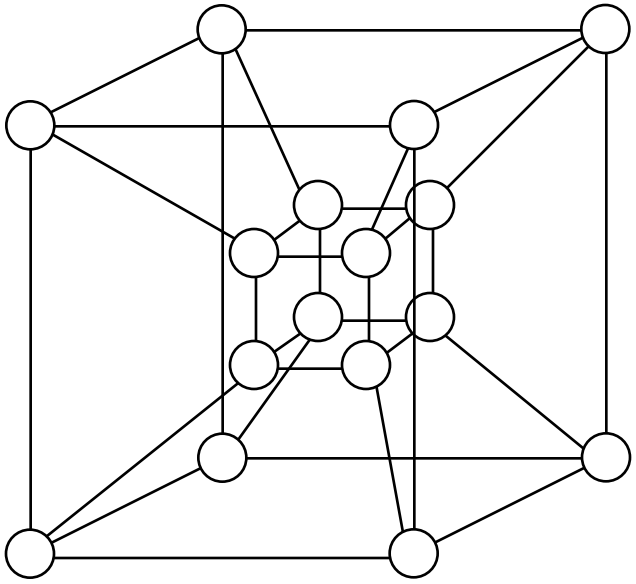
1D



2D

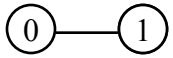


3D

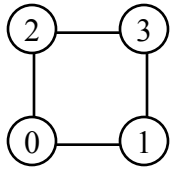


4D

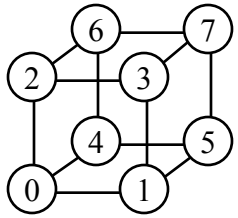
# Numbering Hypercube nodes



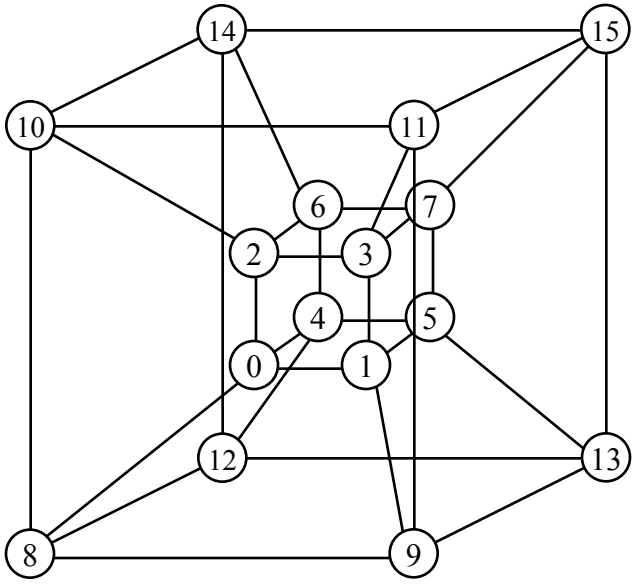

1D




2D

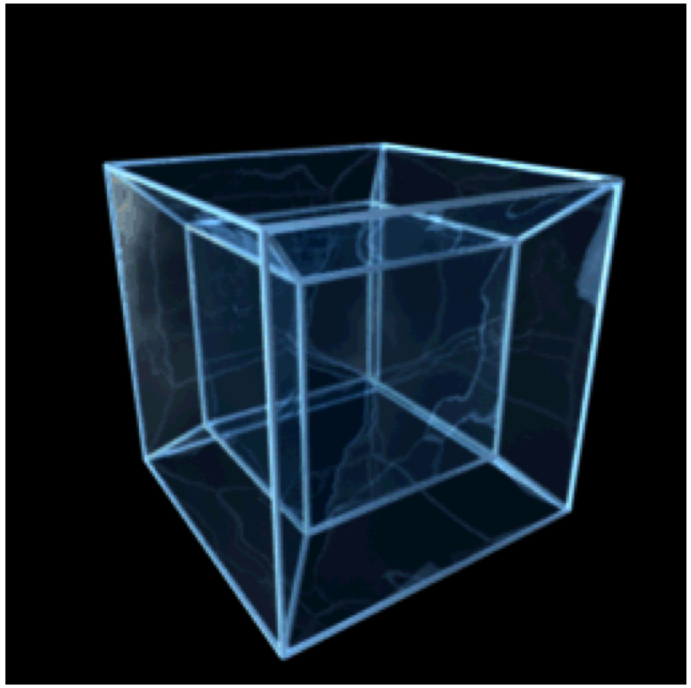



3D




4D

# Rotating 4D Hypercube



# Mapping Grids to Hypercubes

- In the example in which we summed a set of numbers over a square mesh of processors each processor needs to know where it is in the mesh.
- We need to be able to map node numbers to locations in the process mesh
  - Given node number k what is its location (i,j) in the processor mesh?
  - Given a location (i,j) in the processor mesh what is the node number, k, of the processor at that location?
  - We want to choose a mapping such that neighbouring processes in the mesh are also neighbours in the hypercube. This ensures that when neighbouring processes in the mesh communicate, this entails communication between neighbouring processes in the hypercube.

# Binary Gray Codes

- Consider just one dimension – a periodic processor mesh in this case is just a ring.
- Let G(i) be the node number of the processor at position i in the ring, where $0 \leq i < n$. The mapping G must satisfy the following,
  - It must be unique, i.e., $G(i) = G(j) \Leftrightarrow i = j$.
  - G(i) and G(i-1) must differ in exactly one bit for all i, $0 \leq i < n-1$.
  - G(n-1) and G(0) must differ in exactly one bit.

# Binary Gray Codes 2

- A class of mappings known as *binary Gray codes* satisfy these requirements. There are several n-bit Gray codes. Binary Gray codes can be defined recursively as follows:

  Given a d-bit Gray code, a (d+1)-bit Gray code can be constructed by listing the d-bit Gray code with the prefix 0, followed by the d-bit Gray code in reverse order with prefix 1.

# Example of a Gray Code

- Start with the Gray code G(0)=0, G(1)=1.
- Then the 2-bit Gray code is given in Table 1, and the 3-bit Gray code is given in Table 2.

| i | $[G(i)]_2$ | G(i) |
|---|---|---|
| 0 | 00 | 0 |
| 1 | 01 | 1 |
| 2 | 11 | 3 |
| 3 | 10 | 2 |

Table 1: A 2-bit Gray code

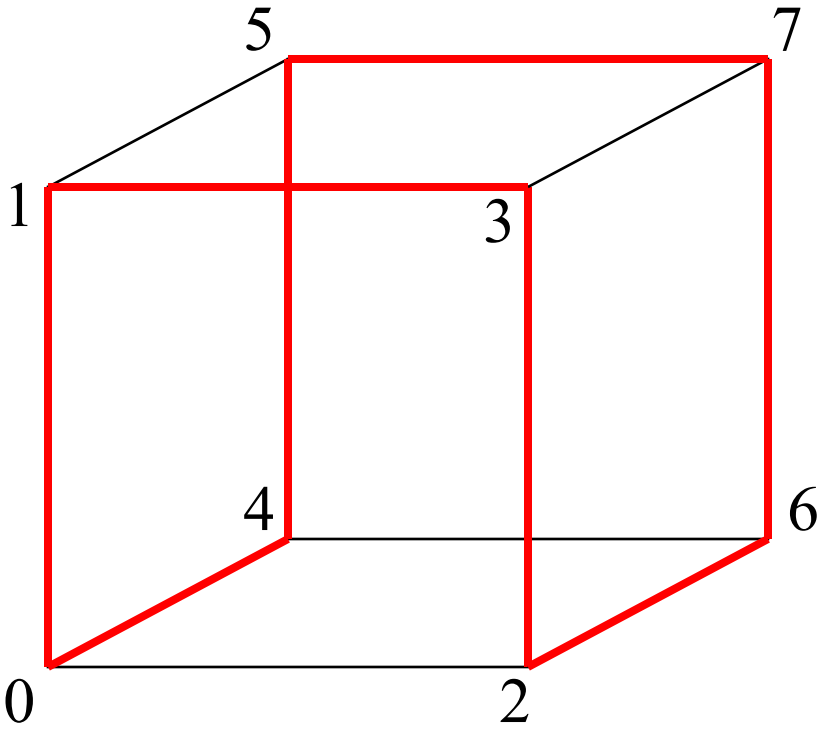
# Example of a Gray Code 2

| i | $[G(i)]_2$ | G(i) |
|---|---|---|
| 0 | 000 | 0 |
| 1 | 001 | 1 |
| 2 | 011 | 3 |
| 3 | 010 | 2 |
| 4 | 110 | 6 |
| 5 | 111 | 7 |
| 6 | 101 | 5 |
| 7 | 100 | 4 |

Table 2: A 3-bit Gray code

# Example of a Gray Code 3

- A ring can be embedded in a hypercube as follows:



# Multi-Dimensional Gray Codes

- To map a multidimensional mesh of processors to a hypercube we require that the number of processors in each direction of the mesh be a power of 2. So

  $2^{d_{r-1}} \times 2^{d_{r-2}} \times \ldots \times 2^{d_0}$

  is an r-dimensional mesh and if d is the hypercube dimension then:

  $d_0 + d_1 + \ldots + d_{r-1} = d$

# Multi-Dimensional Gray Codes 2

- We partition the bits of the node number and assign them to each dimension of the mesh. The first $d_0$ go to dimension 0, the next $d_1$ bits go to dimension 1, and so on. Then we apply separate inverse Gray code mappings to each group of bits.

# Mapping a $2 \times 4$ Mesh to a Hypercube

| K | $[k_1]_2,[k_0]_2$ | $[G^{-1}(k_1)]_2,[G^{-1}(k_0)]_2$ | (i,j) |
|---|---|---|---|
| 0 | 0, 00 | 0, 00 | (0,0) |
| 1 | 0, 01 | 0, 01 | (0,1) |
| 2 | 0, 10 | 0, 11 | (0,3) |
| 3 | 0, 11 | 0, 10 | (0,2) |
| 4 | 1, 00 | 1, 00 | (1,0) |
| 5 | 1, 01 | 1, 01 | (1,1) |
| 6 | 1, 10 | 1, 11 | (1,3) |
| 7 | 1, 11 | 1, 10 | (1,2) |

# Mapping a 2 × 4 Mesh to a Hypercube 2

- A 2 × 4 mesh is embedded into a 3D hypercube as follows: