Computing on Graphs – An Overview

Lecture 2

CSCI 4974/6971

1 Sep 2016

Today's Learning

1. Computations of Graphs
2. OpenMP refresher
3. Hands-on: Breadth-First Search

Computations of Graphs

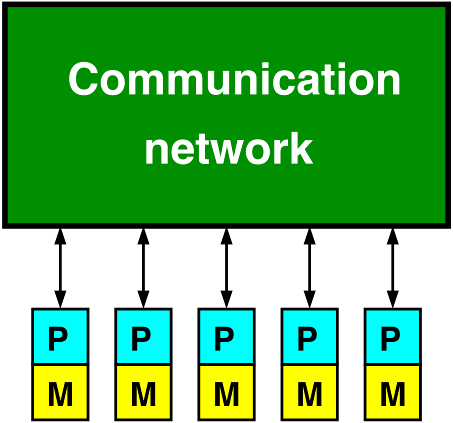
Overview

- Vertex-centric Model
- Bulk-Synchronous Parallization
- Push vs. Pull updating
- Storing graphs in memory

Bulk Synchronous Parallel Model

Slides from Rob Bisseling

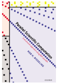
Parallel computer: abstract model



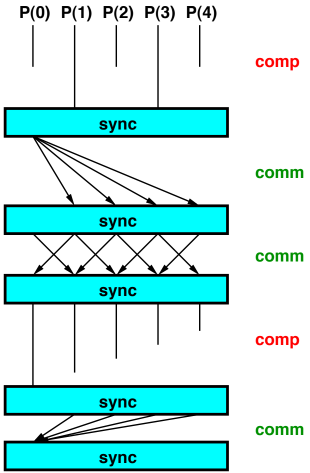
Bulk synchronous parallel (BSP) computer.
Proposed by Leslie Valiant, 1989.

BSP computer

- A BSP computer consists of a collection of processors, each with its own memory. It is a distributed-memory computer.
- Access to own memory is fast, to remote memory slower.
- Uniform-time access to all remote memories.
- No need to open the black box of the communication network. Algorithm designers should not worry about network details, only about global performance.
- Algorithms designed for a BSP computer are portable: they can be run efficiently on many different parallel computers.

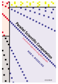


Parallel algorithm: supersteps



BSP algorithm

- A BSP algorithm consists of a sequence of supersteps.
- A computation superstep consists of many small steps, such as the floating-point operations (flops) addition, subtraction, multiplication, division. In scientific computing, flops are the common unit for expressing computation cost.
- A communication superstep consists of many basic communication operations, each transferring a data word such as a real or integer from one processor to another.
- In our theoretical algorithms, we distinguish between the two types of supersteps. This helps in the design and analysis of parallel algorithms.
- In our practical programs, we drop the distinction and mix computation and communication freely in each superstep.



Vertex-centric Model

Slides from Wenfei Fan, QSX: Advanced Topics in Databases

Vertex-centric models

Bulk Synchronous Parallel Model (BSP)

- Leslie G. Valiant: A Bridging Model for Parallel Computation. Commun. ACM 33 (8): 103-111 (1990)
- Processing: a series of supersteps
- Vertex: computation is defined to run on each vertex
- Superstep S: *all vertices compute in parallel; each vertex v may*
  - receive messages sent to v from superstep S – 1;
  - perform some computation: modify its states and the states of its outgoing edges
  - Send messages to other vertices (to be received in the next superstep)

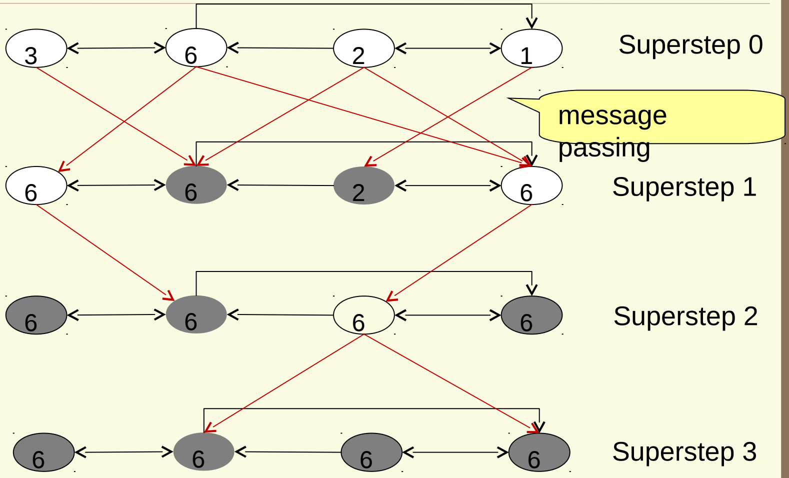
Vertex-centric, message passing

Pregel: think like a vertex

- Input: a directed graph G
  - Each vertex v: a node id, and a value
  - Edges: contain values (associated with vertices)
- Vertex: modify its state/edge state/edge sets (topology)
- Supersteps: within each, *all vertices compute in parallel*
- Termination:
  - Each vertex votes to halt
  - When all vertices are inactive and no messages in transit
- Synchronization: supersteps

Asynchronous: all vertices within each superstep

Example: maximum value



Shaded vertices: voted to halt

Pushing vs. Pulling

Push vs. Pull

General idea

- We have a graph structure we want to compute on
- We have a algorithm we want to run
- That algorithm utilizes stored per-vertex data
- We iteratively update that data with a vertex-centric computation
- We can update that data by having vertices *push* data updates to their neighbors or *pull* in data updates
  - Either the vertices' own data gets updated or the neighbors' data gets updated

Push vs. Pull

Pushing

Pushing:

- Information is pushed – a vertex updates its neighbor's data
- **The Good:**
  - Can be work-optimal – only push needed updates
- **The Bad:**
  - Synchronization concerns – race-conditions updating neighbor's data
- **The Algorithms:**
  - Standard breadth-first search – push "discovery" to neighbors and update distance/level/parent data
  - Color Propagation connectivity algorithm – push colors to neighbors

Push vs. Pull

Pulling

Pulling:

- Each vertex pulls in information from neighbors to update their own value
- **The Good:**
  - Minimal synchronization concerns, only updating own value
  - Easier to parallelize – can often get better scaling
- **The Bad:**
  - Not necessarily work-optimal – but there exist ways to make it close
- **The Algorithms:**
  - Standard PageRank – pull in neighbors' PageRanks, update own value
  - Label Propagation – find max label count among neighbors, update own value to it

An Introduction to OpenMP

Ruud van der Pas

An Introduction Into OpenMP

Ruud van der Pas

Senior Staff Engineer
Scalable Systems Group
Sun Microsystems

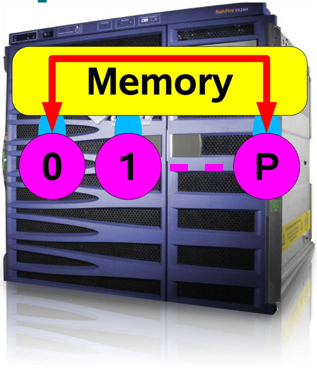
IWOMP 2005
University of Oregon
Eugene, Oregon, USA
June 1-4, 2005

Outline

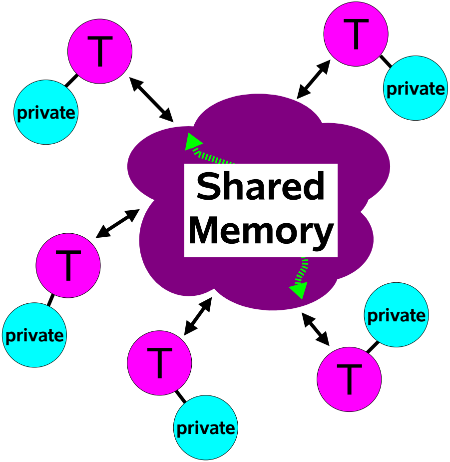
- The OpenMP Programming Model
- OpenMP Guided Tour
- OpenMP Overview
  - Clauses
  - Worksharing constructs
  - Synchronization constructs
  - Environment variables
  - Global Data
  - Runtime functions
- Wrap-up

The OpenMP Programming Model

OpenMP™



Shared Memory Model



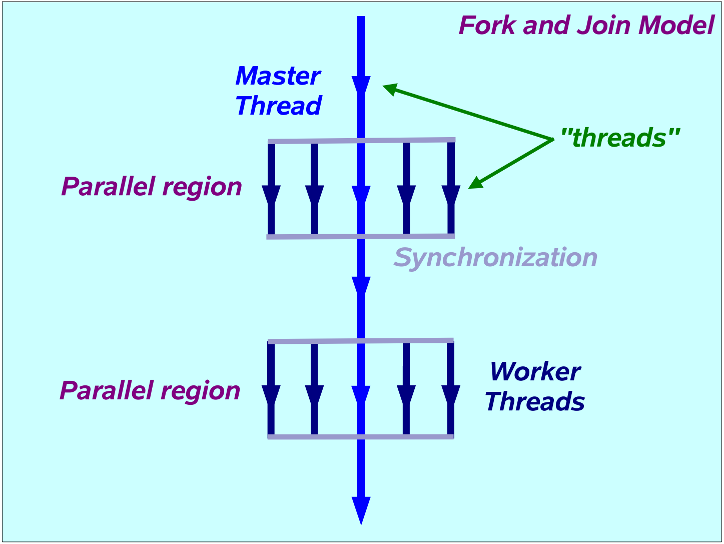
Programming Model

- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can be accessed only by the threads that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place, but it is mostly implicit*

About Data

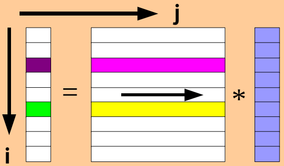
- ***In a shared memory parallel program variables have a "label" attached to them:***
  - ***Labelled "Private" ⇨ Visible to one thread only***
    - ✔ ***Change made in local data, is not seen by others***
    - ✔ ***Example - Local variables in a function that is executed in parallel***
  - ***Labelled "Shared" ⇨ Visible to all threads***
    - ✔ ***Change made in global data, is seen by all others***
    - ✔ ***Example - Global data***

The OpenMP execution model



Example - Matrix times vector

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
   sum = Σ b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum = Σ b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
   sum = Σ b[i=5][j]*c[j]
   a[5] = sum
 i = 6
   sum = Σ b[i=6][j]*c[j]
   a[6] = sum
```

... etc ...

OpenMP Guided Tour

http://www.openmp.org

http://www.compunity.org

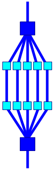
When to consider using OpenMP?

- ❑ *The compiler may not be able to do the parallelization in the way you like to see it:*
  - *A loop is not parallelized*
    - ✔ *The data dependency analysis is not able to determine whether it is safe to parallelize or not*
  - *The granularity is not high enough*
    - ✔ *The compiler lacks information to parallelize at the highest possible level*
- ❑ *This is when explicit parallelization through OpenMP directives and functions comes into the picture*

About OpenMP

- *The OpenMP programming model is a powerful, yet compact, de-facto standard for Shared Memory Programming*
- *Languages supported: Fortran and C/C++*
- *Current release of the standard: 2.5*
  - *Specifications released May 2005*
- *We will now present an overview of OpenMP*
- *Many details will be left out*
- *For specific information, we refer to the OpenMP language reference manual (http://www.openmp.org)*

Terminology



- *OpenMP Team := Master + Workers*
- *A Parallel Region is a block of code executed by all threads simultaneously*
  - *The master thread always has thread ID 0*
  - *Thread adjustment (if enabled) is only done before entering a parallel region*
  - *Parallel regions can be nested, but support for this is implementation dependent*
  - *An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially*
- *A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work*

A loop parallelized with OpenMP

```
#pragma omp parallel default(none) \
            shared(n,x,y) private(i)
{
 #pragma omp for
   for (i=0; i<n; i++)
       x[i] += y[i];
} /*-- End of parallel region --*/
```

clauses

```
!$omp parallel default(none)        &
!$omp shared(n,x,y) private(i)
!$omp do
      do i = 1, n
         x(i) = x(i) + y(i)
      end do
!$omp end do
!$omp end parallel
```

Components of OpenMP

Directives

- Parallel regions
- Work sharing
- Synchronization
- Data scope attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism

Runtime environment

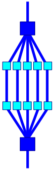
- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Timers
- API for locking

Directive format

- ❑ *C: directives are case sensitive*
  - *Syntax:* **#pragma omp directive [clause [clause] ...]**
- ❑ *Continuation: use \ in pragma*
- ❑ *Conditional compilation:* **_OPENMP** *macro is set*

- ❑ *Fortran: directives are case insensitive*
  - *Syntax:* **sentinel directive [clause [[,] clause]...]**
  - *The sentinel is one of the following:*
    - ✔ *!$OMP or C$OMP or *$OMP* *(fixed format)*
    - ✔ *!$OMP* *(free format)*
- ❑ *Continuation: follows the language syntax*
- ❑ *Conditional compilation:* **!$** *or* **C$ -> 2 spaces**

A more elaborate example



```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];


#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];


#pragma omp barrier
          ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
          ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop (work will be distributed)

parallel loop (work will be distributed)

synchronization

Statement is executed by all threads

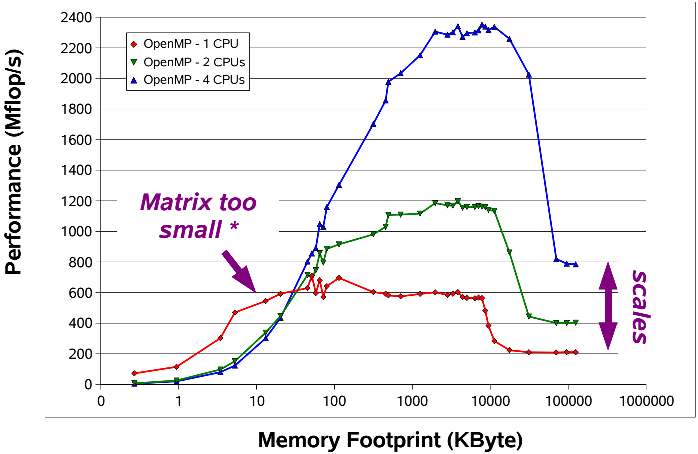
parallel region

Another OpenMP example

```
 1 void mxv_row(int m,int n,double *a,double *b,double *c)
 2 {
 3  int i, j;
 4  double sum;
 5
 6 #pragma omp parallel for default(none) \
 7                  private(i,j,sum) shared(m,n,a,b,c)
 8  for (i=0; i<m; i++)
 9  {
10     sum = 0.0;
11     for (j=0; j<n; j++)
12       sum += b[i*n+j]*c[j];
13      a[i] = sum;
14  } /*-- End of parallel for --*/
15 }
```

```
% cc -c -fast -xrestrict -xopenmp -xloopinfo mxv_row.c
"mxv_row.c", line  8: PARALLELIZED, user pragma used
"mxv_row.c", line 11: not parallelized
```

OpenMP performance



SunFire 6800
UltraSPARC III Cu @ 900 MHz
8 MB L2-cache

***) With the IF-clause in OpenMP this performance degradation can be avoided**

Some OpenMP Clauses

About OpenMP clauses

- *Many OpenMP directives support clauses*
- *These clauses are used to specify additional information with the directive*
- *For example,* ***private(a)*** *is a clause to the for directive:*
  - **#pragma omp for private(a)**
- *Before we present an overview of all the directives, we discuss several of the OpenMP clauses first*
- *The specific clause(s) that can be used, depends on the directive*

The if/private/shared clauses

if (scalar expression)

- ✔ *Only execute in parallel if expression evaluates to true*
- ✔ *Otherwise, execute serially*

```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

private (list)

- ✔ *No storage association with original object*
- ✔ *All references are to the local object*
- ✔ *Values are undefined on entry and exit*

shared (list)

- ✔ *Data is accessible by all threads in the team*
- ✔ *All threads access the same address space*

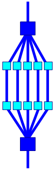
About storage association

- ***Private variables are undefined on entry and exit of the parallel region***
- ***The value of the original variable (before the parallel region) is undefined after the parallel region !***
- ***A private variable within a parallel region has no storage association with the same variable outside of the region***
- ***Use the first/last private clause to override this behaviour***
- ***We will illustrate these concepts with an example***

Example private variables

```
main()
{
  A = 10;

 #pragma omp parallel
 {
   #pragma omp for private(i) firstprivate(A) lastprivate(B)...
   for (i=0; i<n; i++)
   {
        ....
        B = A + i;          /*-- A undefined, unless declared
                                 firstprivate --*/

        ....
   }

   C = B;                   /*-- B undefined, unless declared
                                 lastprivate --*/


 } /*-- End of OpenMP parallel region --*/

}
```

The first/last private clauses



firstprivate (list)

- *All variables in the list are initialized with the value the original object had before entering the parallel construct*

lastprivate (list)

- *The thread that executes the sequentially last iteration or section updates the value of the objects in the list*

The default clause

Sun microsystems

default (none | shared | private) — *Fortran*

default (none | shared) — *C/C++*

Note: default(private) is not supported in C/C++

none

- ✔ *No implicit defaults*
- ✔ *Have to scope all variables explicitly*

shared

- ✔ *All variables are shared*
- ✔ *The default in absence of an explicit "default" clause*

private

- ✔ *All variables are private to the thread*
- ✔ *Includes common block data, unless THREADPRIVATE*

The reduction clause - example

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

Variable SUM is a shared variable

- *Care needs to be taken when updating shared variable SUM*
- *With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided*

The reduction clause

reduction ([operator | intrinsic]) : list)	*Fortran*
reduction (operator : list)	*C/C++*

- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is defined as:*

Check the docs for details

Fortran

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr_list)
x = intrinsic (expr_list, x)
```

C/C++

```
x = x operator expr
x = expr operator x
x++, ++x, x--, --x
x <binop> = expr
```

- ✔ *Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed*
- ✔ *The reduction can be hidden in a function call*

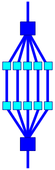
The nowait clause

- *To minimize synchronization, some OpenMP directives/pragmas support the optional* ***nowait*** *clause*
- *If present, threads will not synchronize/wait at the end of that particular construct*
- *In Fortran the nowait is appended at the closing part of the construct*
- *In C, it is one of the clauses on the pragma*

```
#pragma omp for nowait
{
        :
}
```

```
!$omp do
          :
          :
!$omp end do nowait
```

The parallel region



A parallel region is a block of code executed by multiple threads simultaneously

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this will be executed in parallel"

} (implied barrier)
```

```
!$omp parallel [clause[[,] clause] ...]

    "this will be executed in parallel"

!$omp end parallel (implied barrier)
```

The parallel region - clauses

A parallel region supports the following clauses:

if	*(scalar expression)*	
private	*(list)*	
shared	*(list)*	
default	*(none|shared)*	*(C/C++)*
default	*(none|shared|private)*	*(Fortran)*
reduction	*(operator: list)*	
copyin	*(list)*	
firstprivate	*(list)*	
num_threads	*(scalar_int_expr)*	

Worksharing Directives

Work-sharing constructs

The OpenMP work-sharing constructs

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$OMP END SECTIONS
```

```
#pragma omp single
{
        ....
}

!$OMP SINGLE
        ....
!$OMP END SINGLE
```

- *The work is distributed over the threads*
- *Must be enclosed in a parallel region*
- *Must be encountered by all threads in the team, or none at all*
- *No implied barrier on entry; implied barrier on exit (unless nowait is specified)*
- *A work-sharing construct does not launch any new threads*

The WORKSHARE construct

Fortran has a fourth worksharing construct:

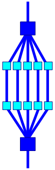
```
!$OMP WORKSHARE

    <array syntax>

!$OMP END WORKSHARE [NOWAIT]
```

Example:

```
!$OMP WORKSHARE
    A(1:M) = A(1:M) + B(1:M)
!$OMP END WORKSHARE NOWAIT
```

The omp for/do directive

The iterations of the loop are distributed over the threads



```
#pragma omp for [clause[[,] clause] ...]
   <original for-loop>
```

```
!$omp do [clause[[,] clause] ...]
         <original do-loop>
!$omp end do [nowait]
```

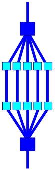
Clauses supported:

private	**firstprivate**
lastprivate	**reduction**
ordered*	***schedule***
nowait	

← *covered later* (applies to *ordered** and *schedule*)

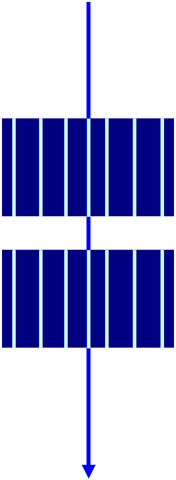
***) Required if ordered sections are in the dynamic extent of this construct**

The omp for directive - example



```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait

     for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait

     for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)



Load balancing

- *Load balancing is an important aspect of performance*
- *For regular operations (e.g. a vector addition), load balancing is not an issue*
- *For less regular workloads, care needs to be taken in distributing the work over the threads*
- *Examples of irregular worloads:*
  - *Transposing a matrix*
  - *Multiplication of triangular matrices*
  - *Parallel searches in a linked list*
- *For these irregular situations, the* schedule *clause supports various iteration scheduling algorithms*

The schedule clause/1

schedule (static | dynamic | guided [, chunk])
schedule (runtime)

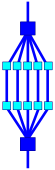
static [, chunk]

- ✔ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✔ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*

Example: Loop of length 16, 4 threads:

TID	0	1	2	3
no chunk	1-4	5-8	9-12	13-16
chunk = 2	1-2	3-4	5-6	7-8
	9-10	11-12	13-14	15-16

The schedule clause/2



dynamic [, chunk]

- ✔ *Fixed portions of work; size is controlled by the value of chunk*
- ✔ *When a thread finishes, it starts on the next portion of work*

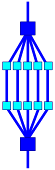
guided [, chunk]

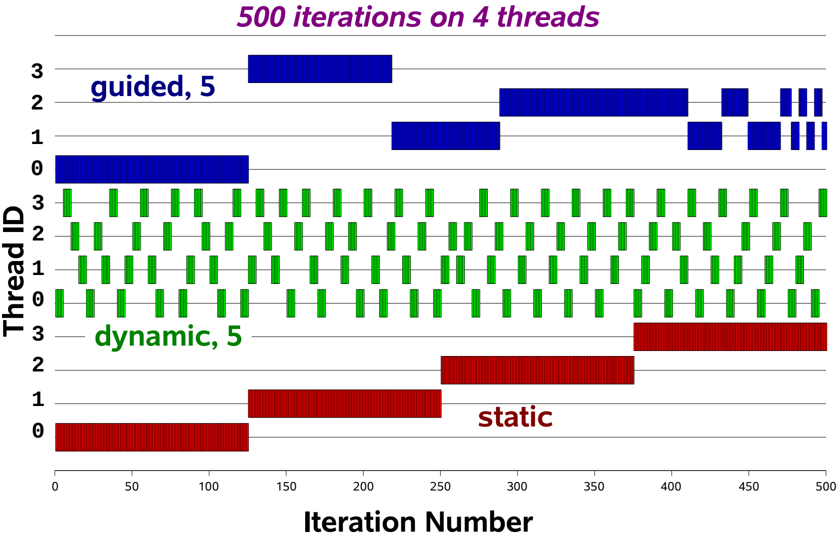
- ✔ *Same dynamic behaviour as "dynamic", but size of the portion of work decreases exponentially*

runtime

- ✔ *Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE*

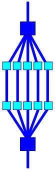
The experiment





Synchronization Controls

Barrier/1



Suppose we run each of these two loops in parallel over i:

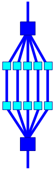
```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day)

Why ?

Barrier/2



We need to have <u>updated all of a[ ]</u> first, before using a[ ]

```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

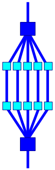
wait !

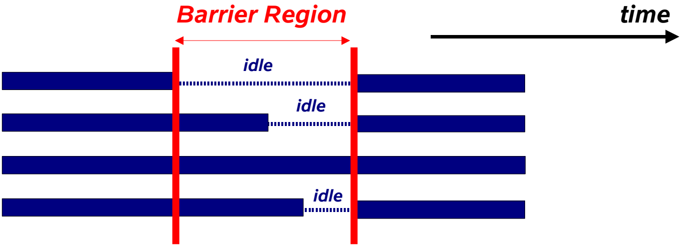
barrier

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point

Barrier/3





Each thread waits until all others have reached this point:

```
#pragma omp barrier
```

```
!$omp barrier
```

When to use barriers ?

- *When data is updated asynchronously and the data integrity is at risk*
- *Examples:*
  - *Between parts in the code that read and write the same section of memory*
  - *After one timestep/iteration in a solver*
- *Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors*
- *Therefore, use them with care*

Critical region/1

If sum is a shared variable, this loop can not be run in parallel

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

We can use a critical region for this:

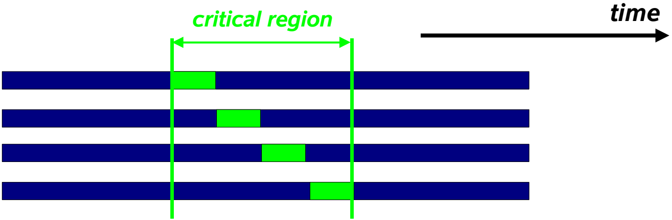
```
for (i=0; i < N; i++){
      .....
   sum += a[i];
      .....
}
```

one at a time can proceed

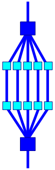
next in line, please

Critical region/2

- *Useful to avoid a race condition, or to perform I/O (but which still will have random order)*
- *Be aware that your parallel computation may be serialized and so this could introduce a scalability bottleneck (Amdahl's law)*



Critical region/3



All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit !

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

```
#pragma omp atomic
    a[indx[i]] += b[i];
```

Single processor region/1

This construct is ideally suited for I/O or initialization

Serial

```
for (i=0; i < N; i++)
{
    .....
  "read a[0..N-1]";
    .....
}
```

Parallel

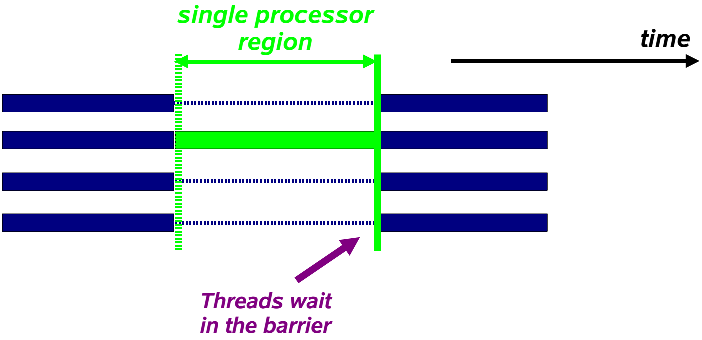
"declare A to be be shared"

```
#pragma omp parallel for
for (i=0; i < N; i++)
{
    .....
    one volunteer requested
  "read a[0..N-1]";
    thanks, we're done
    .....
}
```

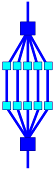
May have to insert a barrier here

Single processor region/2

- Usually, there is a barrier needed after this region
- Might therefore be a scalability bottleneck (Amdahl's law)



SINGLE and MASTER construct



Only one thread in the team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

```
!$omp single [clause[[,] clause] ...]
   <code-block>
!$omp end single [nowait]
```

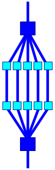
Only the master thread executes the code block:

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

More synchronization directives



The enclosed block of code is executed in the order in which iterations would be executed sequentially:

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
        <code-block>
!$omp end ordered
```

Expensive !

Ensure that all threads in a team have a consistent view of certain objects in memory:

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

In the absence of a list, all visible variables are flushed

Summary

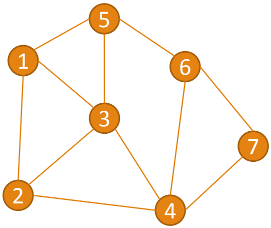
- *OpenMP provides for a compact, but yet powerful, programming model for shared memory programming*
- *OpenMP supports Fortran, C and C++*
- *OpenMP programs are portable to a wide range of systems*
- *An OpenMP program can be written such that the sequential version is still "built-in"*

Graph Representations, Computing for Data Analytics: Methods and Tools

Da KuangG, Polo Chau

Sparse matrix: Graph adjacency matrix

How to represent a graph?



	1	2	3	4	5	6	7
1		1	1		1		
2	1		1	1			
3	1	1		1	1		
4		1	1			1	1
5	1		1			1	
6				1	1		1
7				1		1	

A node in a graph is typically connected to only a small fraction of nodes.

Source: www.cs.umn.edu/~metis

Sparse matrix is often very sparse

Term-document matrix for 4.5M English Wikipedia articles:

0.05% nonzeros

DBLP co-authorship network for 300,000 academic authors:

0.0007% nonzeros

→ We need efficient storage for sparse matrices.

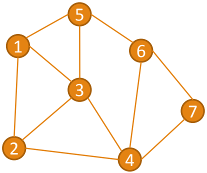
Storage of a sparse matrix

We store only the nonzeros and their positions

- (row, column, value)-triplet

Use the same example:

(1, 2, 1) (1, 3, 1) (1, 5, 1)
(2, 1, 1) (2, 3, 1) (2, 4, 1)
(3, 1, 1) (3, 2, 1) (3, 4, 1) (3, 5, 1)
(4, 2, 1) (4, 3, 1) (4, 6, 1) (4, 7, 1)
(5, 1, 1) (5, 3, 1) (5, 6, 1)
(6, 4, 1) (6, 5, 1) (6, 7, 1)
(7, 4, 1) (7, 6, 1)



	1	2	3	4	5	6	7
1		1	1		1		
2	1		1	1			
3	1	1		1	1		
4		1	1			1	1
5	1		1			1	
6				1	1		1
7				1		1	

This is the "edge list" format; in this case, an array of tuples of length 3.

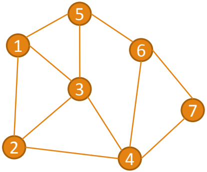
Viewing indices of the matrix as graph nodes, these triplets are edges.

Symmetric sparse matrix ($A = A^T$) $\Leftrightarrow$ Undirected graph

What about the adjacency matrix of **directed graph**?
And **Bipartite graph**?

Coordinate list (COO) format

The triplets can be stored as 3 arrays: rows, cols, values.



	0	1	2	3	4	5	6
0		1	1		1		
1	1		1	1			
2	1	1		1	1		
3		1	1			1	1
4	1		1			1	
5				1	1		1
6				1		1	

```
  rows = [0,0,0,1,1,1,2,2,2,2,3,3,3,3,4,4,4,5,5,5,6,6]
  cols = [1,2,4,0,2,3,0,1,3,4,1,2,5,6,0,2,5,3,4,6,3,5]
values = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

Note: 0-based arrays

Compressed sparse row (CSR) format

Suppose a sparse matrix has `nnz` nonzero entries.

`rows` = [0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6]

`cols` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5]

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

The COO format needs `3nnz` elements to store the matrix. Can we do better?

When the nonzeros are stored row by row (and row IDs start at 0), we can compress the above storage:

`rowptr` = [0, 3, 6, 10, 14, 17, 20, 22] — Row pointer

`colind` = [1, 2, 4, 0, 2, 3, 0, 1, 3, 4, 1, 2, 5, 6, 0, 2, 5, 3, 4, 6, 3, 5] — Column index

`values` = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] — Values

This CSR format needs `2nnz+n` elements to store the matrix.

Similarly, we have compressed sparse column (CSC) format.

Breadth-First Search

Overview

- General Algorithm
- “Pushing”
- “Pulling”
- `C++` demonstration

Breadth-First Search

Algorithm

Why BFS? Prototypical graph algorithm, high memory access/communication to computation ratio. Has been used as an example for extreme optimization (Graph500.org)

- We select a root
- We want to figure out the number of hops/distance of every vertex reachable from the root
- Naturally iterative – one level/hop from the root at a time
- Algorithm concludes when no new vertices are found on a level

Breadth-first search - pushing

```
procedure BFS(G(V, E), root)
    for all v ∈ V do
        Levels(v) ← −1                              ▷ Initialize levels
    level ← 0
    Q ← root
    Levels(root) ← level
    while Q ≠ ∅ do                                  ▷ Finishing when queue is empty
        level ← level + 1
        for all v ∈ Q do
            for all ⟨v, u⟩ ∈ E do
                if Level(u) < 0 then                ▷ Have we discovered u?
                    Level(u) ← level                ▷ v pushes update to u
                    Q_next ← u
        Swap(Q, Q_next)
        Q_next ← ∅
```

Breadth-first search - pulling

```
procedure BFS(G(V, E), root)
    for all v ∈ V do
        Levels(v) ← −1
    level ← 0
    Q ← root
    Levels(root) ← level
    size = 1
    while size > 0 do                    ▷ Instead of a queue, just track level size
        level ← level + 1
        size ← 0
        for all v ∈ V do
            if level(v) < 0 then         ▷ We haven't discovered v yet
                for all ⟨v, u⟩ ∈ E do
                    if Level(u) = level − 1 then
                        Level(v) ← level        ▷ v pulls update from u
                        size ← size + 1
                        break                   ▷ No need to go further
```

C++ Demonstration – Blank code and data available on website

www.cs.rpi.edu/~slotag/classes/FA16/index.html