Shared-memory Programming with OpenMP

Advanced Computing Laboratory
Costa Rica National High Technology Center

Diego Jiménez Vargas
djimenez@cenat.ac.cr

2018

Contents

1. Introduction
   - Parallel Architectures
   - Parallel Computing
   - Directive-based Programming
   - Warm-up Exercise
2. OpenMP
   - Standard
   - Parallel Directive and Variables
   - Loop Directives
   - Additional Directives
   - Shared-memory Systems
3. Exercises

Introduction

Parallel Architecture

It's everywhere

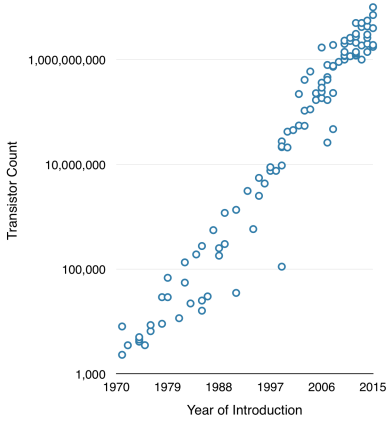


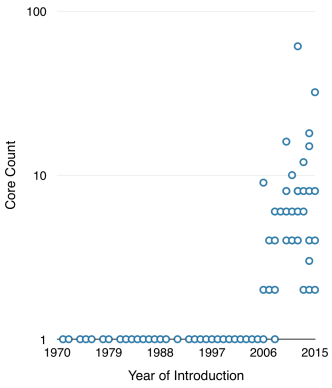





Moore's Law

Transistor density increases exponentially

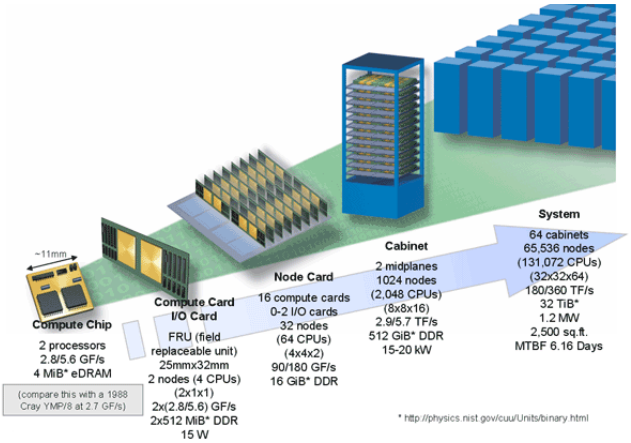




Source: `https://en.wikipedia.org/wiki/Transistor_count`

Supercomputer

A hierarchical ensemble of many parts



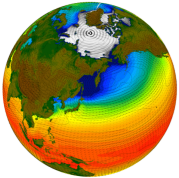
Top 500

Fastest supercomputers on Earth

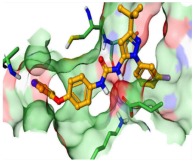
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	**Gyoukou** - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan	19,860,000	19,135.8	28,192.0	1,350
5	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209

Source: `https://www.top500.org/` (November 2017)

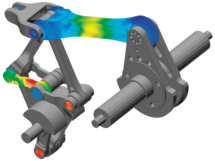
Advanced Computing
For everyone



Climatology



Medicine



Materials

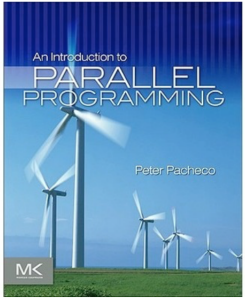
Economics

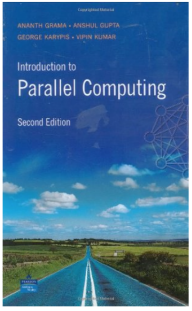
Cosmology

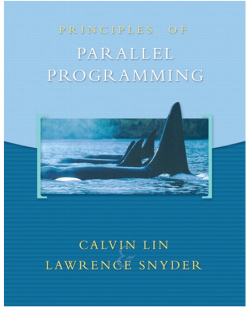
Entertainment

Parallel Computing Books

Do they look similar?

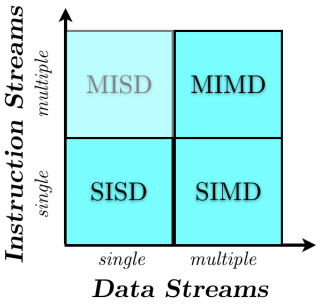






Flynn's Taxonomy

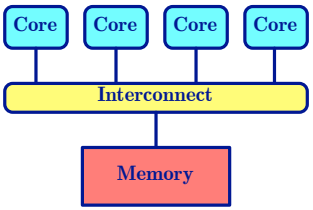
A classification of computer systems



- SISD (Single Instruction, Single Data): traditional von-Neumann model of single-processor computers
- SIMD (Single Instruction, Multiple Data): Vector processors and accelerators
- MISD (Multiple Instruction, Single Data): no system fits this description
- MIMD (Multiple Instruction, Multiple Data): standard computer clusters:
  - **Shared-memory systems**
  - Distributed-memory systems

Shared-memory Systems

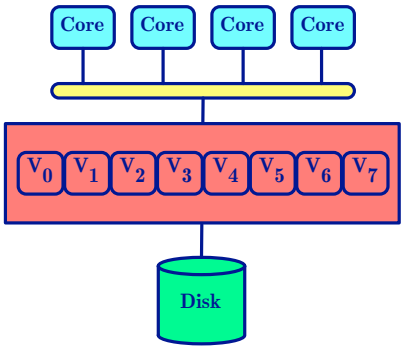
Single address space



- Communication is implicit: write and read operations on shared variables
- Simple programming model: no data distribution among processors
- Limited scalability (memory contention)
- Multicore processors fit this description
- Symmetric Multiprocessor (SMP): memory access latency is the same for all processors, also called Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA) systems are more scalable

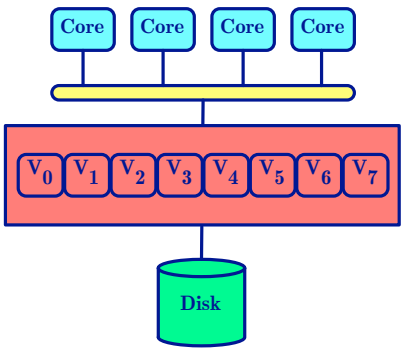
Parallel Computing

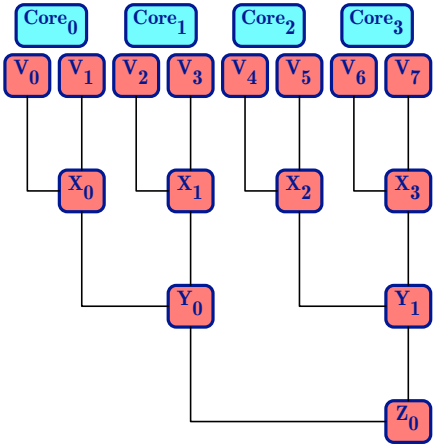
An example: parallel sum



Parallel Computing

An example: parallel sum





Speedup and Efficiency

Performance metrics

Definition (Speedup)

The ratio of the best sequential time over the parallel time with p cores:

$$S_p = \frac{T_1}{T_p}$$

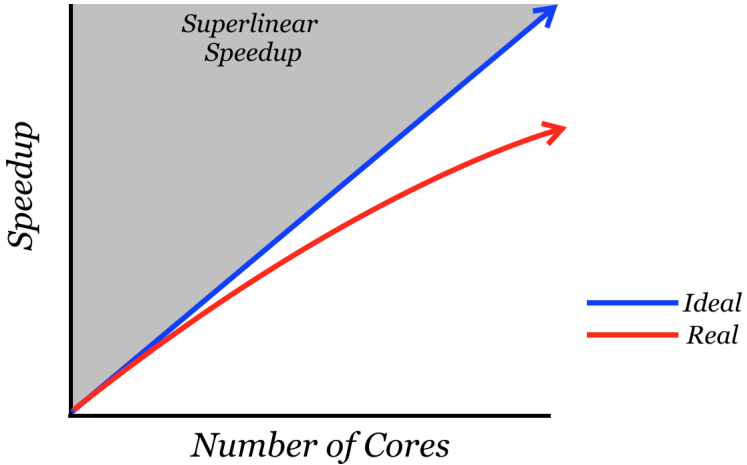
Definition (Efficiency)

The average utilization of a processor in the parallel system:

$$E_p = \frac{S_p}{p}$$

Speedup Curve

Increases with more cores until it saturates



Amdahl's Law

Bound to speedup

Theorem (Amdahl's Law)

Asume a fraction s of the work for a given problem is serial, with $0 \leq s \leq 1$, while the remaining portion $1 - s$ is p-fold parallel. Therefore,

$$T_p = sT_1 + (1 - s)\frac{T_1}{p}$$

$$S_p = \frac{T_1}{T_p} = \frac{p}{sp + (1 - s)} \qquad E_p = \frac{T_1}{pT_p} = \frac{1}{sp + (1 - s)}$$

Hence, $S_p \to \frac{1}{s}$ and $E_p \to 0$ as $p \to \infty$.
For instance, if the serial fraction exceeds 1 percent, then speedup can never exceed 100 for any p.

Scalability
Problem size importance

Definition (Scalable)

A technology is scalable if it can handle ever-increasing problem sizes.

Definition (Strong scalability)

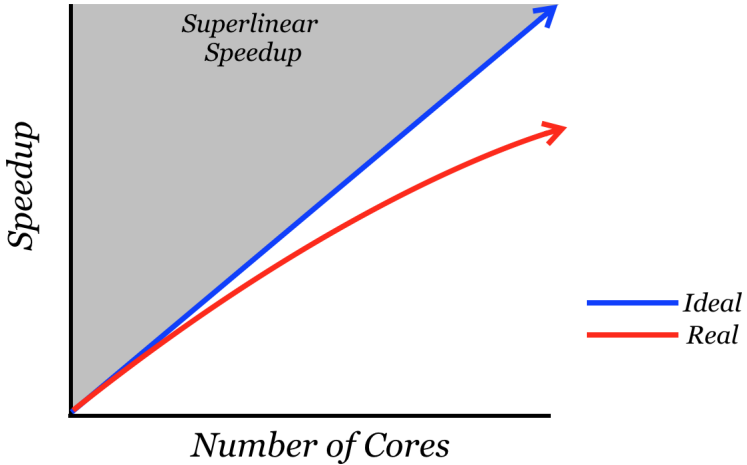
If when we increase the number of processes/threads, we can keep the efficiency fixed *without* increasing the problem size, we say a program is *strongly scalable* (Amdahl's law).

Definition (Weak scalability)

If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, we say a program is *weakly scalable* (Gustafson's law).

Speedup Curve

Increases with more cores until it saturates



Exercises

1. What is the maximum expected speedup of a system with 1024 processors?
2. What is the maximum expected efficiency of a system?
3. What is the efficiency of a 256-processor system if speedup is 64?
4. What is the speedup of a system with efficiency 0.5 and 32 processors?
5. What is the maximum speedup achievable on a parallel system if the serial portion of the program represents 20% of sequential execution time?

Compilers

Translate high-level code into low-level representation

Fortran

```
1 do i=1,n
2 C(i) = A(i) + B(i)
3 enddo
```

Assembly

```
1 Label1:
2     mov al,[si]
3     add al,[di]
4     mov [bx],al
5     inc si
6     inc di
7     inc bx
8     dec cl
9     jnz Label1
```

Compilers

Translate high-level code into low-level representation

Fortran

```
1 do i=1,n
2 C(i) = A(i) + B(i)
3 enddo
```

Assembly

```
1 Label1:
2     mov al,[si]
3     add al,[di]
4     mov [bx],al
5     inc si
6     inc di
7     inc bx
8     dec cl
9     jnz Label1
```

Compiler optimizations (gcc -O3)

-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion2
-fif-conversion
-finline-functions-called-once
-fipa-pure-const
-fipa-profile
-fipa-reference
-fmerge-constants
-fmove-loop-invariants
-fshrink-wrap
-fsplit-wide-types
-ftree-bit-ccp
-ftree-ccp
-fssa-phiopt
-ftree-ch
-ftree-copy-prop
-ftree-copyrename
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-pta
-ftree-ter
-funit-at-a-time
-ftree-builtin-call-dce
-ftree-switch-conversion
-ftree-tail-merge
-ftree-pre
-ftree-vrp
-fcrossjumping
-fcse-follow-jumps
-fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize
-fdevirtualize-speculatively
-fexpensive-optimizations
-fgcse
-fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-ra
-fipa-cp-alignment
-fipa-sra
-fipa-icf
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks
-freorder-blocks-and-partition
-freorder-functions
-frerun-cse-after-loop
-fsched-interblock
-fsched-spec
-fschedule-insns
-fschedule-insns2
-fstrict-aliasing
-fstrict-overflow
-ftree-builtin-call-dce
-ftree-switch-conversion
-ftree-tail-merge
-ftree-pre
-ftree-vrp
-fipa-ra
-finline-functions
-funswitch-loops
-fpredictive-commoning
-fgcse-after-reload
-ftree-loop-vectorize
-ftree-loop-distribute-patterns
-ftree-slp-vectorize
-fvect-cost-model
-ftree-partial-pre
-fipa-cp-clone

Parallelizing Compilers

Parallelization is just another optimization

- A compiler may automatically parallelize a sequential code
- Big quest during the 90s:
  - General parallelization is too hard for a compiler
  - Figuring out all data dependences is impossible at compile time
  - Relative success on more restrictive languages (CoArray Fortran, High Performance Fortran)
  - Vectorization of loops has become mainstream
- The programmer has to be involved in the parallelization of the code

How Do We Write Parallel Programs?

Parallel Programming Languages

During the 1990s

"C* in C	CUMULVS	Java RMI	P-RIO	Quake
ABCPL	DAGGER	javaPG	PSI	Quark
ACE	DAPPLE	JAVAR	P4-Linda	Quick Threads
ACT++	Data Parallel C	JavaSpaces	Pablo	Sage++
ADDAP	DC++	JIDL	PADE	SAM
Adl	DCE++	Joyce	PADRE	SCANDAL
Admiralty	DDD	Karma	Panda	SCHEDULE
AFAPI	DICE	Khoros	Papers	SciTL
ALWAN	DIPC	KOAN/Fortran-S	Para++	SDDA
AM	Distributed SmallTalk	LAM	Paradigm	SHMEM
AMDC	DOLIB	Legion	Parafrase2	SIMPLE
Amoeba	DOME	Lilac	Paralation	Sina
AppLeS	DOSMOS	Linda	Parallaxis	SISAL
ARTS	DRL	LiPS	Parallel Haskell	SMI
Athapascan-0b	DSM-Threads	Locust	Parallel-C++	SONiC
Aurora	Ease	Lparx	ParC	Split-C
Automap	ECO	Lucid	ParLib++	SR
bb_threads	Eilean	Maisie	ParLin	Sthreads
Blaze	Emerald	Manifold	Parlog	Strand
BlockComm	EPL	Mentat	Parmacs	SUIF
BSP	Excalibur	Meta Chaos	Parti	SuperPascal
C*	Express	Midway	pC	Synergy
C**	Falcon	Millipede	pC++	TCGMSG
C4	Filaments	Mirage	PCN	Telegraphos
CarlOS	FLASH	Modula-2*	PCP:	The FORCE
Cashmere	FM	Modula-P	PCU	Threads.h++
CC++	Fork	MOSIX	PEACE	TRAPPER
Charlotte	Fortran-M	MpC	PENNY	TreadMarks
Charm	FX	MPC++	PET	UC
Charm++	GA	MPI	PETSc	uC++
Chu	GAMMA	Multipol	PH	UNITY
Cid	Glenda	Munin	Phosphorus	V
CIlk	GLU	Nano-Threads	POET	VIC*
CM-Fortran	GUARD	NESL	Polaris	Visifold V-NUS
Code	HAsL	NetClasses++	POOL-T	VPE
Concurrent ML	HORUS	Nexus	POOMA	WinNT threads
Converse	HPC	Nimrod	PORTAL	WinPar
COOL	HPC++	NOW	PRESTO	WWWinda
CORBELATE	HPF	Objective Linda	Prospero	XENOOPS
CparPar	IMPACT	Occam	Proteus	XPC
CPS	INRIA	Omega	PSDM	Zounds
CRL	ISIS	OOF90	PSI	ZPL
CSP	JADA	Orca	PVM	
Cthreads	JADE	P++	QPC++	

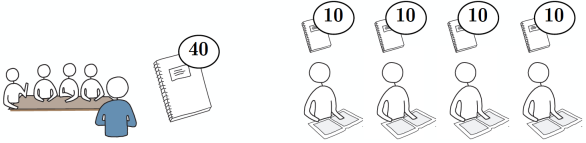
Source: `Patterns for Parallel Programming` (Tim Matson, et al, 2004)

- Based on **C/C++**, **Fortran**, **Python**
- A few dominant languages:
  - Shared-memory: **OpenMP**
  - Distributed-memory: **MPI**
  - Accelerators: **OpenACC**
- Other programming models:
  - Partitioned global address space: **UPC**, **OpenSHMEM**
  - Distributed arrays: **Charm++**, **GlobalArrays**
  - Task parallelism: **Uintah**, **Legion**
  - DARPA's High Productivity Computer Systems: **Fortress**, **Chapel**, **X10**

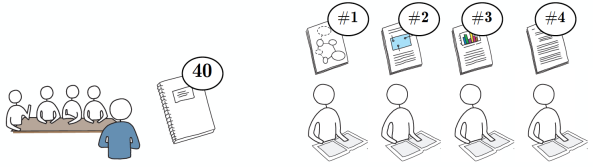
How Do We Write Parallel Programs?

Types of parallelism

- Data parallelism (SIMD): parallel execution of the same code stream on a supercomputer. Each node may process a different data portion.



- Task parallelism (MIMD): distribution of different code streams on a supercomputer. Each node may process a different data portion.



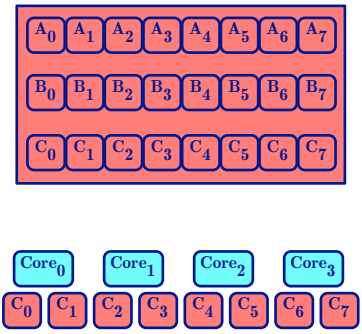
Parallel Programming

Considered harder than sequential programming

- Training in computer science is based on von Neumann architecture (sequential)
- Considerations in parallel programming:
  - Communication, sending data between processors
  - Synchronization, coordinating access to shared resources
  - Load balancing, distributing computation evenly across processor set
- Parallel machine:
  - Contains a set of p threads, numbered from 0 to $(p-1)$
  - Each thread has a unique ID

Directive-based Parallel Programming

Add pragmas to hint compiler on parallel code



Sequential Code

```
1  for(i=0; i<N; i++)
2      C[i] = A[i] + B[i];
```

Definition (Pragmas)

Directive pragmas (from *pragmatics*) is a language construct to tell a compiler how it should process an input code. Pragmas are not part of the programming language.

Parallel Code

```
1  #pragma omp parallel
2  for(i=0; i<N; i++)
3      C[i] = A[i] + B[i];
```

Parallel Hello World

A primer on parallel execution

```
1  #include <omp.h>
2  int main(){
3      #pragma omp parallel
4      {
5          printf("Hello World\n");
6      }
7  }
```

```
1  #include <omp.h>
2  int main(){
3      int id, np;
4      #pragma omp parallel private(id, np)
5      {
6          np = omp_get_num_threads();
7          id = omp_get_thread_num();
8          printf("Hello from thread %d, out of %d threads\n", id, np);
9      }
10 }
```

Parallel Hello World

A primer on parallel execution

```c
#include <omp.h>
int main(){
    #pragma omp parallel
    {
        printf("Hello World\n");
    }
}
```

```c
#include <omp.h>
int main(){
    int id, np;
    #pragma omp parallel private(id, np)
    {
        np = omp_get_num_threads();
        id = omp_get_thread_num();
        printf("Hello from thread %d, out of %d threads\n", id, np);
    }
}
```

Undo Parallelization

Finding a serial counterpart

- Directive-based parallel programming is based on the principle of *incremental parallelization*
- The *equivalent* serial program is always at hand:
  - To test for correctness
  - To compare performance
  - To get an more familiar serial algorithmic description

```
1  int main(){
2      #pragma omp parallel
3      BLOCK_A
4      ...
5      #pragma omp parallel
6      BLOCK_B
7      ...
8  }
```

```
1  int main(){
2      BLOCK_A
3      ...
4      BLOCK_B
5      ....
6  }
```

Exercises

1. Download the git repository and inspect the `openmp` directory:
   `git clone https://github.com/CNCA-CeNAT/CRHPCS-2018.git`
2. Get into `hello_id` directory and get familiar with the execution mechanism:
   - Type `make` to compile
   - Type `qsub hello_id.pbs` to submit
   - Open the output file to see the result
3. Get into `map` directory and implement a map program. Such program takes an array and applies a function `foo` to each member of the array. You must implement the `foo` function using any formula you want.

OpenMP

OpenMP
Open Multi-Processing

- An implementation of multithreading
- Standard API for shared-memory programming
- OpenMP Architecture Review Board (ARB) published OpenMP standard 1.0 in 1997
- Current release is OpenMP 4.5 in 2015
- Adopted by most important operating systems and supercomputer vendors
- Based on pragmas, programmer annotations for the compiler
- Embraces incremental parallelization philosophy
- Serial counterpart is always available

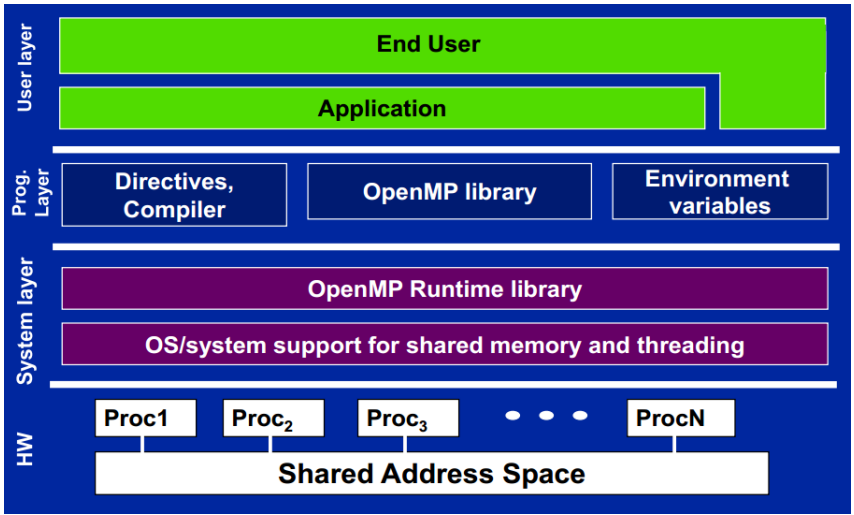
OpenMP Standard

Created by the community

- Defines an Application Program Interface (API) which provides a portable and scalable model for developers of shared memory parallel applications (instead of PThreads)
- A library with some simple functions
- A few program directives (pragmas = hints to the compiler)
- A few environment variables
- A compiler translates OpenMP functions and directives into PThread calls
- Program begins with a master thread
- Threads are forked when specified by directives

OpenMP Standard

Solution Stack



High-level representation

Translated automatically into multithreaded code

OpenMP

```
#pragma omp parallel for
for(int i=0; i<n; i++){
     A[i] = A[i] * A[i];
}
```

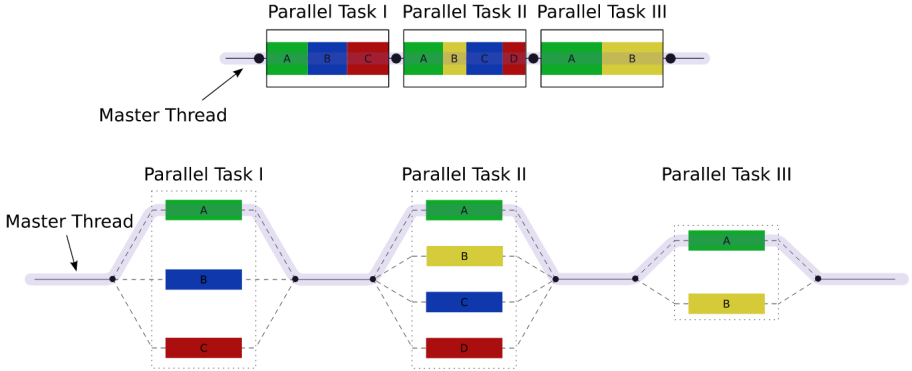
PThreads

```
struct params {
  int *A;
  int tid;
  int size;
  int nthreads;
};
void *parallel_for(void *_p){
  params *p        = (params*)_p;
  int tid                 = p->tid;
  int chunk_size = (p->size/p->nthreads);
  int start             = tid * chunk_size;
  int end              = start + chunk_size;
  for(int i = start; i < end; i++)
    A[i] = A[i] * A[i];
  return 0;
}
```

```
pthread_t threads[nthreads];
params *thread_params = (params*) ?   malloc(
     nthreads * sizeof(params));
for(int i = 0; i < nthreads; i++){
  thread_params[i].A         = A;
  thread_params[i].tid       = i;
  thread_params[i].size      = n;
  thread_params[i].nthreads = nthreads;
  pthread_create(&threads[i], NULL, ?
       parallel_for, (void*)
    &thread_params[i]);
}
for(int i = 0; i < nthreads; i++){
  pthread_join(threads[i], NULL);
}
free(thread_params);
```

Fork-join Model

Parallelism shrinks and expands



Source: `http://www.wikipedia.org`

Parallel Regions

Defined with `#pragma omp parallel`

```
1 #pragma omp parallel
2 {
3     printf("Hello World\n");
4 }
```

The number of spawned threads can be specified through:

- Using the clause:

  `num_threads(thread_count)`

- An API function:

  `void omp_set_num_threads(int number);`

- An environment variable:

  `export OMP_NUM_THREADS=32`

```
1 #pragma omp parallel num_threads(32)
2 {
3     printf("Hello World\n");
4 }
```

```
1 omp_set_num_threads(32);
2 #pragma omp parallel
3 {
4     printf("Hello World\n");
5 }
```

Number of Threads

Define the level of concurrency

- The `num_threads(thread_count)` clause can be added to a parallel directive
- It allows the programmer to specify the number of threads that should execute the following block
- There may be system-defined limitations on the number of threads that a program can start
- The OpenMP standard does not guarantee that this will actually start `thread_count` threads
- Most current systems can start hundreds or even thousands of threads
- Unless we are trying to start a lot of threads, we will almost always get the desired number of threads

Embarrassingly Parallel

A problem that has a trivial parallel solution

- Simple distribution of work is enough to parallelize
- No significant communication complexity
- No significant synchronization complexity
- Should we feel embarrassed to come up with an easy parallel solution? Of course not!
- Embarrassingly parallel $\rightarrow$ Delightfully parallel

Critical Sections and Barriers

Synchronizing threads

- A critical section is executed by one thread at a time; (name) is optional
- Critical sections with different names can be executed simultaneously

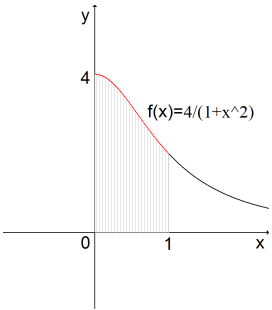
```
1 #pragma omp parallel
2 {
3     value = foo();
4     #pragma omp critical(name)
5     {
6         sum = sum + value;
7     }
8 }
```

- All the threads in the active team must reach the barrier point before any can continue:

```
1 #pragma omp parallel
2 {
3     value = foo();
4     #pragma omp barrier
5     value = goo();
6 }
```

Exercise
Trapezoidal rule

- Approximating pi via numerical integration



- Divide interval up into subintervals.
- Assign subintervals to processes
- Each process calculates partial sum
- **Add all the partial sums together to get pi.**

Interacting with OpenMP Library

Useful functions and tips

- Compiler flag:
  `icc -qopenmp pgm.c`
- Include file:
  `#include <omp.h>`
- Environment variable for number of threads:
  `OMP_NUM_THREADS`

- Useful functions:
  - `omp_get_num_threads()`
    returns the number of active threads
  - `omp_get_thread_num()`
    returns the thread ID
  - `omp_set_num_threads(int t)`
    sets number of active threads
  - `omp_get_wtime()`
    gets current wall clock time in seconds
  - `omp_in_parallel()`
    returns true if program is running in parallel; false otherwise

Data Scoping
Types of variables

- `shared`: accesible by all threads (default)
- `private`: one copy for each thread
- `firstprivate`: private but initialized to its value before entering the region
- `lastprivate`: private but on exit, value in master is last value in thread
- `default`: default type for all variables in the thread
- `reduction`: variables declared in enclosing context, are private in the parallel sections, but reduced upon exit using the specified operator (e.g. $+, *, -, \&, |, \hat{}, \&\&, ||$)

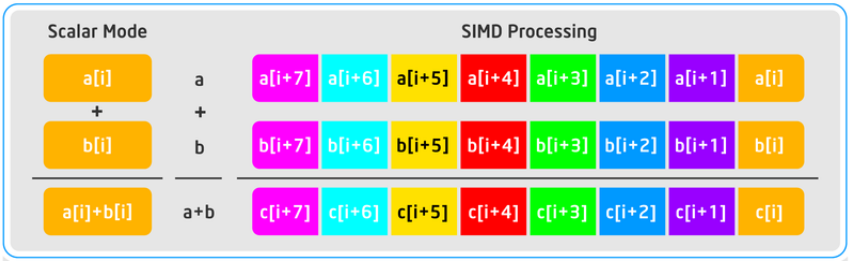
```
1 #pragma omp parallel private(id, np)
2 {
3     np = omp_get_num_threads();
4     id = omp_get_thread_num();
5     printf("Hello from thread %d, out of %d threads\n", id, np);
6 }
```

Vectorized Loops

Defined with `#pragma omp simd`

- Enforces vectorization of for loops
- The compiler may not automatically vectorize
- Several clauses and extensions

```
1  #pragma omp simd
2  for(i = 0; i < n; i++) {
3        A[i] = A[i] + K;
4  }
```



Parallel Loops

Defined with `#pragma omp for`

- Automatically distributes iterations in the for-loop to active threads

```
1 #pragma omp for
2 for(i=0; i<N; i++){
3     C[i] = A[i] + B[i];
4 }
```

- Usually used inside a parallel region

```
1 #pragma omp parallel
2 {
3     #pragma omp for
4     for(i=0; i<N; i++)
5         C[i] = A[i] + B[i];
6 }
```

- Pragmas `parallel` and `for` can be coalesced:

```
1 #pragma omp parallel for
2 for(i=0; i<N; i++)
3     C[i] = A[i] + B[i];
```

- Iteration index (variable `i` above) gets automatically privatized
- Reduction variables are useful often times

```
1 #pragma omp parallel for reduction
      (+:sum)
2 for(int i=0; i<N; i++){
3     sum += vector[i];
4 }
```

Matrix Multiplication

Avoiding concurrency overhead

```
1  for (i=0; i<N; i++)
2      for (j=0; j<N; j++)
3          for (k=0; k<N; k++)
4              C[i][j] += A[i][k]*B[k][j];
```

Outer-loop parallelization

```
1  #pragma omp parallel for
2  for (i=0; i<N; i++)
3      for (j=0; j<N; j++)
4          for (k=0; k<N; k++)
5              C[i][j] += A[i][k]*B[k][j];
```

Inner-loop parallelization

```
1  for (i=0; i<N; i++)
2      #pragma omp parallel for
3      for (j=0; j<N; j++)
4          for (k=0; k<N; k++)
5              C[i][j] += A[i][k]*B[k][j];
```

Data Dependencies

Programmers must be careful to avoid race conditions

Fibonacci Sequence

$$f_n = f_{n-1} + f_{n-2}$$

Sequential Code

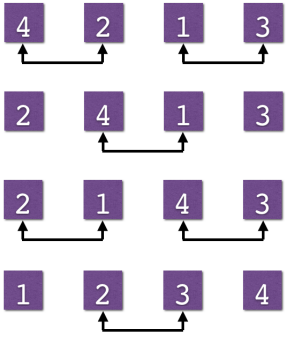
```
1  fibo[0] = fibo[1] = 1;
2  for (i=2; i<10; i++)
3       fibo[i] = fibo[i-1] + fibo[i-2];
```

(Incorrect) Parallel Code

```
1  fibo[0] = fibo[1] = 1;
2  omp_set_num_threads(2);
3  #pragma omp parallel for
4  for (i=2; i<10; i++)
5       fibo[i] = fibo[i-1] + fibo[i-2];
```

Exercise

1. Implement the odd-even parallel sort algorithm in the `odd_even_sort` directory. It is the parallel equivalent to *bubble sort*. When sorting n numbers with n processors, it alternates between $\frac{n}{2}$ odd and even phases:
   - Each processor holds one number
   - Odd phase: odd-numbered processors compare and swap their number with neighbor
   - Even phase: even-numbered processors compare and swap their number with neighbor



Scheduling loops
Avoiding load imbalance

- Assignment of loops is system dependent
  - Normally block partitioning

```
1  sum = 0.0
2  for (i = 0; i <= n; i++)
3      sum += f(i)
```

- Good assignment of iterations to threads can have a very significant effect on performance.
- Assigning iterations to threads is called **scheduling**

Scheduling

Used to specify distribution of iterations to threads

- Syntax:
  `#pragma omp parallel for schedule(type,chunksize)`
- Parameter `type` can be:
  - *Static*: iterations are distributed among threads before the loop is executed
  - *Dynamic or guided*: iterations are assigned to threads during the execution of the loop
  - *Auto*: the compiler or runtime system determines the proper schedule
  - *Runtime*: the environment variable `OMP_SCHEDULE` is used to determine schedule at runtime
- Parameter `chunksize` is a positive integer

Static Scheduling

Default distribution

- The iteration space is divided into chunks and distributed to threads in a round-robin fashion (default chunk size is 1)

- Example: 24 iterations with 4 threads

Chunksize	T1	T2	T3	T4
1	0,4,8,12,16,20	1,5,9,13,17,21	2,6,10,14,18,22	3,7,11,15,19,23
2	0,1,8,9,16,17	2,3,9,10,18,19	4,5,11,12,20,21	6,7,13,14,22,23
3	0,1,2,12,13,14	3,4,5,15,16,17	6,7,8,18,19,20	9,10,11,21,22,23
4	0,1,2,3,16,17,18,19	4,5,6,7,20,21,22,23	8,9,10,11	12,13,14,15

Dynamic Scheduling

Adapts to remaining work

- Iterations are divided into chunks and assigned to threads dynamically

- Each thread executes a chunk, and when done, requests another chunk from the runtime system
- Although each chunk contains the same number of iterations, chunks may have different execution times

Guided Scheduling

Big chunks first

- Similar to dynamic scheduling, except that chunk size decreases as the execution progresses

- At the dynamic scheduling decision, the chunk size is equal to $\frac{1}{p}$ of the remaining iterations, where p is the number of threads
- Can specify the smallest chunksize (except possibly the last)
- The default smallest chunksize is 1

Exercise

1. What would be the assignment of iterations to threads for a guided schedule in a parallel for loop with 1000 iterations and 2 threads?
   The sequence in which threads 0 and 1 request more work is given by:
   0,1,1,1,0,1,0,1,1,1,1.

Exercise

1. What would be the assignment of iterations to threads for a guided schedule in a parallel for loop with 1000 iterations and 2 threads?
   The sequence in which threads 0 and 1 request more work is given by: 0,1,1,1,0,1,0,1,1,1,1.

Chunk	Thread
[1-500]	0
[501-750]	1
[751-875]	1
[876-937]	1
[938-968]	0
[969-984]	1
[985-993]	0
[994-996]	1
[997-998]	1
999	1
1000	1

Sections

Pieces of codes that can run concurrently

- Define independent pieces of code
- May combine the `parallel` and `sections` pragmas (as with `for` pragma)
- If there are more threads than sections, then some are idle
- If there are fewer threads than sections, then some sections are serialized

```
1  #define N 1000
2  main(){
3      int i;
4      float a[N], b[N], c[N];
5      for (i=0; i < N; i++)
6          a[i] = b[i] = ... ;
7          #pragma omp parallel shared(a,b,c) private(i)
8          { ...
9              #pragma omp sections
10             {
11                 #pragma omp section
12                 {
13                     for (i=0; i < N/2; i++)
14                         c[i] = a[i] + b[i];
15                 }
16                 #pragma omp section
17                 {
18                     for (i=N/2; i < N; i++)
19                         c[i] = a[i] + b[i];
20                 }
21             } /* end of sections */
22             ...
23         } /* end of parallel section */
24 }
```

Single and Master

Define sections of code run by only one thread

- The single directive serializes a section of code within a parallel region
  - More convenient and efficient than terminating a parallel region and starting it later
  - Typically used to serialize a small section of the code that's not thread safe
  - Threads in the team that do not execute the single directive, wait at the end of the enclosed code block, unless a `nowait` clause is specified
- The master directive is the same as the single directive except that:
  - The serial code is executed by the master thread
  - The other threads skip the master section, but do not wait for the master thread to finish executing it

```
1 #pragma omp parallel num_threads(32)
2 {
3     printf("Hello World\n");
4     #pragma omp single
5     printf("Only one thread\n");
6     printf("Goodbye World\n");
7 }
```

Task Parallelism

Creating independent pieces of work

- Pragma `task` spawns a task (non-blocking)
- Pragma `taskwait` waits for all spawned tasks
- Pragmas for task parallelism must be used inside a parallel section
- **Taskloop construct**: loop iterations are partitioned into tasks and executed as tasks by the runtime system.

```
1  int fibo(int n){
2    int fn1, fn2;
3    if (n<2)
4      return n;
5    else {
6      #pragma omp task
7      fn1 = fibo(n-1);
8      #pragma omp task
9      fn2 = fibo(n-2);
10     #pragma omp taskwait
11     return fn1 + fn2;
12   }
13 }
14 int main() {
15   #pragma omp parallel shared(n)
16   {
17     #pragma omp single
18     fibo(n);
19   }
20 }
```

Shared-memory Supercomputers

Powerful easy-to-program machines

CRAY THE SUPERCOMPUTER COMPANY

sgi

NUMASCALE BIGGER DATA ANALYTICS

- Cray XE1
- Multi streaming processors
- Up to 32 TB memory
- Up to 8192 processors

- SGI UV-2000
- Up to 64 TB memory
- Up to 256 processor sockets

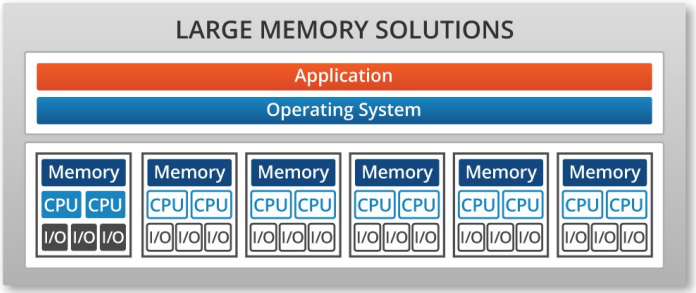
- Numaconnect
- Up to 256 TB memory
- Up to 4096 nodes

Virtual Shared Memory

Single address space via software

ScaleMP
CREATING THE POWER OF ONE

- Hypervisor or virtual machine monitor for HPC
- *Virtualization through aggregation*: combine multiple servers to create a symmetric multiprocessing system



Source: `http://www.hpcwire.com`

Using OpenMP
The Next Step

- The latest standard provides a way of controlling thread affinity (OpenMP threads placement)
  - Memory bandwidth
  - Latency
  - Cache utilization
  - First Touch Data Placement
- Heterogeneous Architectures
  - *Device Constructs*
  - GPUs, DSPs, FPGAs...
  - Threads cannot migrate across devices

Exercises

Exercise 1

Implement the parallel shear-sort algorithm.
Get into the `shear_sort` directory and read the description of the problem in file `shear_sort.pdf`.

Exercise 2

Implement the Strassen algorithm for parallel matrix multiplication.
Get into the `strassen` directory and read the description of the problem in file `strassen.pdf`.

Exercise 3

Think of any algorithm you may find relevant to your research interests and try to parallelize it using OpenMP pragmas.

Acknowledgements

Thank you!