New Trend of HPC

- Two major trends:
  - Processor architects focus on throughput, not clock speed, to improve performance.
  - Access to widely available graphic processing units for general processing.

- The reason is heat dissipation and power consumption
Shared Memory System

- All processors share a 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).
Shared Memory System

- Symmetric Multiprocessor (SMP): memory access latency is the same for all processors.

- Also called Uniform Memory Access (UMA).

- Non-Uniform Memory Access (NUMA):
  - Different access times to memory modules.
  - Processor caches mitigate latency.
  - Improved scalability.

Distributed Memory System

- Each processor has its own private memory.
- Communication is explicit through message passing.
- Involved programming model: data distribution.
- Good scalability.
Each Titan compute node contains (1) AMD Opteron™ 6274 (Interlagos) CPU.

Each NUMA node contains a die's L3 cache and its (4) compute units (8 cores).

Each compute unit contains (2) integer cores (and their L1 cache), a shared floating point scheduler, and shared L2 cache.
A New Era of Computing

- Heterogeneous System Architecture (HAS)
  - bridges the gap between CPU and GPU cores and delivers a new innovation called compute cores.
  - This groundbreaking technology allows CPU and GPU cores to speak the same language and share workloads and the same memory to accelerate applications while delivering great performance and rich entertainment.

To illustrate, we can describe a given APU, such as the AMD A10-7850K APU, as having 12 compute cores, consisting of 4 CPU cores and 8 GPU cores.
Distributed vs. Shared Memory

- **Shared** - all processors share a global pool of memory
  - simpler to program
  - bus contention leads to poor scalability

- **Distributed** - each processor physically has its own (private) memory associated with it
  - scales well
  - memory management is more difficult
Shared Memory Parallel Programming in the Multi-Core Era

- Desktop and Laptop
  - 2, 4, 8 cores and ... ?

- A single node in distributed memory clusters
  - Cluster node: 2 → 8 → 16 cores
  - $ cat /proc/cpuinfo

- Shared memory hardware Accelerators
  - NVIDIA GeForce Titan Z: 5760 Cores, 12 GB VRAM, $3,000
  - Intel Xeon Phi 3120A: 57 Cores, 1.10GHz, $3,300

- Heterogeneous Uniform Memory
What is OpenMP?

- What does OpenMP stand for?
  - **Open** specifications for **Multi Processing** via collaborative work between interested parties from the hardware and software industry, government and academia

- Application Programming Interface (API) for multi-threaded parallelization consisting of
  - Source code directives
  - Functions
  - Environment variables

- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors
What is OpenMP?

- **Shared** Memory with **thread** based parallelism
- **Not** a new language
- OpenMP API is specified for C/C++ and Fortran
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for Fortran and pragmas for C/C++

- OpenMP website: [http://www.openmp.org](http://www.openmp.org)
Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
  - It is the de facto standard for writing shared memory programs.
  - Easy to use.
  - Incremental parallelization.
  - Flexible.

- OpenMP can be implemented incrementally, one function or even one loop at a time.
  - A nice way to get a parallel program from a sequential program.
## Comparison of Programming Models

<table>
<thead>
<tr>
<th>Feature</th>
<th>Open MP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portable</td>
<td>highly yes</td>
<td>yes</td>
</tr>
<tr>
<td>Scalable</td>
<td>less so</td>
<td>yes</td>
</tr>
<tr>
<td>Incremental Parallelization</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Fortran/C/C++ Bindings</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>High Level</td>
<td>yes</td>
<td>mid level</td>
</tr>
</tbody>
</table>
How to compile and run OpenMP programs?

- GCC 4.2 and above supports OpenMP 3.0
  - gcc -fopenmp a.c
  - g++ -fopenmp a.cpp

- To run: ‘a.out’
  - To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh)
    - export OMP_NUM_THREADS=4 (bash)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Options</th>
<th>Default behavior for # of threads (OMP_NUM_THREADS not set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU (gcc, g++, gfortran)</td>
<td>-fopenmp</td>
<td>as many threads as available cores</td>
</tr>
<tr>
<td>Intel (icc ifort)</td>
<td>-openmp</td>
<td>as many threads as available cores</td>
</tr>
<tr>
<td>Portland Group (pgcc,pgCC,pgf77,pgf90)</td>
<td>-mp</td>
<td>one thread</td>
</tr>
</tbody>
</table>
Hello World to OpenMP!

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

using namespace std;

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

    // Fork a team of threads giving them their own copies of variables
    #pragma omp parallel private(nthreads, tid) num_threads(8)
    {
        // Obtain thread number
        tid = omp_get_thread_num();
        cout << "Hello World from thread = " << tid << endl;

        // Only master thread does this
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            cout << "Number of threads = " << nthreads << endl;
        }
    }

    return 0;
}
```
Running OpenMP Programs

Compile:
[ahnt@hopper]$ g++ -fopenmp hello_openmp.cpp -o hello_openmp

Run:
[ahnt@hopper]$ ./hello_openmp
Hello World from thread = Hello World from thread = Hello World from thread = Hello World from thread = Hello World from thread = Hello World from thread = Hello World from thread = 60415

Hello World from thread = 3

7
Number of threads = 8

Hello World from thread = 2
# include <iostream>
# include <stdio.h>
# include <omp.h>

using namespace std;

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

    // Fork a team of threads giving them their own copies of variables
    #pragma omp parallel private(nthreads, tid) num_threads(8)
    {
        // Obtain thread number
        tid = omp_get_thread_num();
        // cout << "Hello World from thread = " << tid << endl;;

        // Only master thread does this
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            cout << "Number of threads = " << nthreads << endl;
        }
    }

    #pragma omp parallel
    printf( "Hello from thread %d of %d with printf()\n",
            omp_get_thread_num(),
            omp_get_num_threads() );

    return 0;
}
Running OpenMP Programs

Compile Again:
[ahnt@hopper]$ g++ -fopenmp hello_openmp.cpp -o hello_openmp

Run:
[ahnt@hopper]$ ./hello_openmp
Number of threads = 8
Hello from thread 5 of 8 with printf()
Hello from thread 2 of 8 with printf()
Hello from thread 4 of 8 with printf()
Hello from thread 1 of 8 with printf()
Hello from thread 6 of 8 with printf()
Hello from thread 0 of 8 with printf()
Hello from thread 3 of 8 with printf()
Hello from thread 7 of 8 with printf()
void* SayHello(void *foo) {
    printf( "Hello, world!\n" );
    return NULL;
}

int main() {
    pthread_attr_t attr;
    pthread_t threads[16];
    int tn;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], &attr, SayHello, NULL);
    }
    for(tn=0; tn<16 ; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
POSIX Threads (Pthreads)

Pthreads is a POSIX standard for describing a thread model, it specifies the API and the semantics of the calls.

- Thread API available on many OS’s
  - `#include <pthread.h>`
  - `cc myprog.c –o myprog -lpthread`

- Thread creation
  ```c
  int pthread_create(pthread_t * thread,
        pthread_attr_t * attr,
        void * (*start_routine)(void *),
        void * arg);
  ```

- Thread termination
  ```c
  void pthread_exit(void *retval);
  ```

- Waiting for Threads
  ```c
  int pthread_join(pthread_t th, void **thread_return);
  ```
Motivation

- Thread libraries are hard to use
- P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
- Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness
Motivation

Wouldn’t it be nice to write serial programs and somehow parallelize them “automatically”?

- OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence.

- OpenMP is a small API that hides cumbersome threading calls with simpler directives.
OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:
    ```
    #pragma omp construct [clause [clause]…]
    ```
  - For Fortran, the directives take one of the forms:
    ```
    C$OMP construct [clause [clause]…]
    !$OMP construct [clause [clause]…]
    !$OMP construct [clause [clause]…]
    ```
- Include files
  ```
  #include “omp.h”
  ```
How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

```c
void main()
{
    int i, k, N=1000;
    double A[N], B[N], C[N];
    for (i=0; i<N; i++)
        A[i] = B[i] + k*C[i]
}
```

```c
#include "omp.h"
void main()
{
    int i, k, N=1000;
    double A[N], B[N], C[N];
    #pragma omp parallel for
    for (i=0; i<N; i++)
        A[i] = B[i] + k*C[i];
}
```
OpenMP Fork-and-Join model

- Serial regions by default, annotate to create *parallel regions*
  - Generic parallel regions
  - Parallelized loops
  - Sectioned parallel regions

- Thread-like Fork/Join model
  - Arbitrary number of *logical* thread creation/ destruction events
int main() {
    // serial region
    printf("Hello…");
    // parallel region
    #pragma omp parallel
    {
        printf("World");
    }
    // serial again
    printf("!");
}
OpenMP Fork-and-Join model

```c
printf("program begin\n");
N = 1000;

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

M = 500;
#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("program done\n");
```

Serial

Parallel

Serial

Parallel

Serial
OpenMP Constructs

OpenMP’s constructs:

- Parallel Regions
- Worksharing (for/DO, sections, …)
- Data Environment (shared, private, …)
- Synchronization (barrier, flush, …)
- Runtime functions/environment variables (omp_get_num_threads(), …)
OpenMP API Overview

Three Components:

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)

- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

- Implementations differ in their support of all API components.
Compiler Directives

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag.

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

- For example:

<table>
<thead>
<tr>
<th>Fortran</th>
<th>!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA, PI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#pragma omp parallel default(shared) private(beta, pi)</td>
</tr>
</tbody>
</table>
Compiler Directives

- private (list), shared (list)
- firstprivate (list), lastprivate (list)
- reduction (operator: list)
- schedule (method [, chunk_size])
- nowait
- if (scalar_expression)
- num_thread (num)
- threadprivate(list), copyin (list)
- ordered
- collapse (n)
- tie, untie
- And more …
Run-time Library Routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution

- For example:

<table>
<thead>
<tr>
<th>Fortran</th>
<th>INTEGER FUNCTION OMP_GET_NUM_THREADS()</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#include &lt;omp.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int omp_get_num_threads(void)</td>
</tr>
</tbody>
</table>
Run-time Library Routines

- Number of threads: `omp_{set,get}_num_threads`
- Thread ID: `omp_get_thread_num`
- Scheduling: `omp_{set,get}_dynamic`
- Nested parallelism: `omp_in_parallel`
- Locking: `omp_{init,set,unset}_lock`
- Active levels: `omp_get_thread_limit`
- Wallclock Timer: `omp_get_wtime`
- thread private
- call function twice, use difference between end time and start time
- And more …
Environment Variables

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

- For example,

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>csh/tcsh</td>
<td>setenv OMP_NUM_THREADS 8</td>
</tr>
<tr>
<td>sh/bash</td>
<td>export OMP_NUM_THREADS=8</td>
</tr>
</tbody>
</table>
Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- And more …
Create threads

- Compiler Directives

```c
#pragma omp parallel num_threads(8)
```

- Run-time Library

```c
omp_set_num_threads(8);
```

- Environment Variables

```bash
export OMP_NUM_THREADS=8
```
As of June 2015, the documentation for LC's default compilers claims the following OpenMP support:

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Supports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel C/C++, Fortran</td>
<td>14.0.3</td>
<td>OpenMP 3.1</td>
</tr>
<tr>
<td>GNU C/C++, Fortran</td>
<td>4.4.7</td>
<td>OpenMP 3.0</td>
</tr>
<tr>
<td>PGI C/C++, Fortran</td>
<td>8.0.1</td>
<td>OpenMP 3.0</td>
</tr>
<tr>
<td>IBM Blue Gene C/C++</td>
<td>12.1</td>
<td>OpenMP 3.1</td>
</tr>
<tr>
<td>IBM Blue Gene Fortran</td>
<td>14.1</td>
<td>OpenMP 3.1</td>
</tr>
<tr>
<td>IBM Blue Gene GNU C/C++, Fortran</td>
<td>4.4.6</td>
<td>OpenMP 3.0</td>
</tr>
</tbody>
</table>
## Compiling

<table>
<thead>
<tr>
<th>Compiler / Platform</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Linux Opteron/Xeon</td>
<td>icc, icpc, ifort</td>
<td>-openmp</td>
</tr>
<tr>
<td>PGI Linux Opteron/Xeon</td>
<td>pgcc, pgCC, pgf77, pgf90</td>
<td>-mp</td>
</tr>
<tr>
<td>GNU Linux Opteron/Xeon IBM Blue Gene</td>
<td>gcc, g++, g77, gfortran</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>IBM Blue Gene</td>
<td>bgxlc_r, bgcc_r, bgxlc_r, bgxlc++_r, bgxlc89_r, bgxlc99_r, bgxlf_r, bgxlf90_r, bgxlf95_r, bgxlf2003_r</td>
<td>-qsmp=omp</td>
</tr>
</tbody>
</table>

*Be sure to use a thread-safe compiler - its name ends with \_r*