A brief introduction to parallel programming models

Eduard Ayguadé
Computational Sciences Dep., Barcelona Supercomputing Center
Computer Architecture Dep., Technical University of Catalunya

Scalable programming models

- Shared-memory programming models
  - Threading model
  - Memory model (consistency)
  - Work creation and distribution
  - Synchronization

- Distributed-memory programming models
  - Process model
  - Data distribution
  - Data movement
  - Synchronization

- Chip-level programming models
  - Multicore, SMT, ...
  - Streaming
  - Local memories, DMA transfers, ...

The Grid

- Future Petascale systems

Hardware DSM
- Thread affinity
- Page migration: first/next touch

Software DSM
- Paging shared memory
- Overlapped data movement with computation

Hardware DSM
- Page migration: first/next touch

Software DSM
- Paging shared memory
- Overlapped data movement with computation

Grid programming models
- Granularity issues
- Heterogeneous systems
- Dynamicity in availability of resources

Future Petascale systems

Large cluster systems

Small DMM

On-chip SMP

Chip
Hybrid programming models

Most modern high-performance computing systems are clusters of SMP nodes with multicore chips

- Interconnection Network
- Memory
- SMP

Programming models allow to specify:
- How computation is distributed?
- How data is distributed and how is it accessed?
- How to avoid data races?

Hybrid vs. single programming model

Shared address space

Programmer needs:
- Distribute work
- All threads can access data, heap and stacks
- Use synchronization mechanisms to avoid data races
Example: heat distribution

\[ x_i(t) = \frac{(x_{i-1}(t-1) + x_{i+1}(t-1))}{2} \]

```
int t; /* time step */
int i; /* array index */
float x[n+1], y[n+1]; /* temperatures */
x[0] = y[0] = 20;
x[n+1] = y[n+1] = 100;
for(i=1; i<n+1; i++)
x[i] = 0;
for(t=1; t<=100; t++)
{
    for(i=1; i<n+1; i++)
       y[i] = 0.5 * (x[i-1] + x[i+1]);
    swap(x, y);
}
```

Heat distribution: sequential program
int t /* time step */, i;
int id; /* my processor id */
shared float x[1002]; /* temperatures */
int leftmost, rightmost; /* processor boundary points */
float x_leftmost, x_rightmost; /* their temperatures */

id=GetID();
leftmost=id*100+1; rightmost=(id+1)*100;
if(id==0){x[0]=20; x[1001]=100;}
for(i=leftmost; i<=rightmost; i++) x[i]=0;

for(t=1; t<=100; t++)
    barrier(n);
    x_leftmost=0.5*(x[leftmost-1]+x[leftmost+1]);
    x_rightmost=0.5*(x[rightmost-1]+x[rightmost+1]);
    barrier(n);
    for(i=leftmost+1; i<rightmost; i++)
        x[i]=0.5*(x[i-1]+x[i+1]);
    x[leftmost]=x_leftmost;
    x[rightmost]=x_rightmost;

barrier(n);
Altix 4700: C2 blade

- Each blade contains 2 sockets for Itanium2 Montecito dual-core

- SHub 2.0 with FSB at 533 MHz (1), 12 DDR2 memory sockets and 2 NumaLink4 channels (6.4 GB/s each)

(1) 16 bytes x 533 MHz = 8.5 GB/s
Altix 4700: 32 and 64 processor system

6 x 6.4 = 38.4 GB/s = 3.84 GB/s/blade

8 x 6.4 = 51.2 GB/s = 2.56 GB/s/blade

Altix 4700: 128 processor system

- Systems larger than 32 blades require the use of additional dense router modules, with 4 8x8 crossbar ASIC up to 256 compute blades
- Systems larger than 256 compute blades require additional Shubs that create 1D or 2D meshes (up to 4096 compute blades)

8 x 6.4 = 51.2 GB/s = 1.28 GB/s/blade
**Altix 4700: up to 256 compute blades**

- Single process sequential
  - Code
  - Data + Heap
  - Stack

- Single process multithreaded
  - Code
  - Data + Heap
  - Stack

**Shared address space**

- **Programmer needs:**
  - Distribute work
  - All threads can access data, heap and stacks
  - Use synchronization mechanisms to avoid data races

- **But memory is not flat …**
  - Perform work to benefit from spatial and temporal locality
- 20 SGI® Altix™ 3700 superclusters, each with 512 Itanium2 processors (1.5 GHz, 6 MB cache)
- Infiniband network to connect superclusters

**Programmer needs:**
- Distribute work
- Distribute data
- Use communication mechanisms to share data explicitly
- Use synchronization mechanisms to avoid data races
Message passing programming model

- Send specifies buffer to be transmitted and receiving process
- Receive specifies sending process and application storage to receive into
- Optional tag on send and matching rule on receive
- Implicit synchronization (e.g. non-blocking send and blocking receive)
- Many overheads: copying, buffer management, protection

Heat distribution: message passing

```c
int t /* time step */, i;
int id; /* my processor id */
int num_points; /* number of points per processor */
float x[-1:(1000/P)]; /* temperatures, including shadow points */

num_points = 1000/P;
id=GetID();
if (id == 0) x[-1]=20;
if (id == (P-1)) x[num_points]=100;
for(i=0; i<num_points; i++) x[i]=0;
for(t=1;t<=100;t++){
    if(id < P) receive(id+1,&x[num_points]);
    if(id > 0) receive(id-1,&x[-1]);
    for(i=0; i<num_points; i++)
        x[i]=0.5*(x[i-1]+x[i+1]);
    if(id > 0) send(id-1,x[0]);
    if(id < P) send(id+1,x[num_points-1]);
}  
```